SC26-3985-2
File No. S370-25

Program Product

VS FORTRAN Application Programming: Guide

Program Numbers 5748-FO3 (Compiler and Library)
5748-LM3 (Library Only)

Release 1.1

IBM

This publication was produced using the
IBM Document Composition Facility
(program number 5748-XX9) and
the master was printed on the IBM 3800 Printing Subsystem.

Second Edition (January 1982)

This is a major revision of, and makes obsolete, SC26-3985-1, and its technical newsletters, SN26-0806 and SN26-0865.

This edition applies to Release 1.1 of VS FORTRAN, Program Products 5748-F03 (Compiler and Library) and 5748-LM3 (Library only), and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 and 4300 Processors Bibilography, GC20-0001, for the editions that are applicable and current.

ABOUT THIS BOOK

This manual describes how to use VS FORTRAN, together with the supported operating systems, to design, develop, test, and run programs written in VS FORTRAN at the 1978 language level. It is designed as a guide for using VS FORTRAN at the current language level; it is not intended to be used as a reference manual.

MANUAL ORGANIZATION

This manual is designed for FORTRAN application developers who use VS FORTRAN in two different ways:

- Engineers and scientists who use FORTRAN as a tool in mathematical problem solving
- Application programmers who use all of the FORTRAN features to code FORTRAN programs for their own use or for others

Because of these differences in FORTRAN usage, this manual is organized into sections:

"Part 1—Simplified FORTRAN Programming" describes how to develop and run mathematical problem-solving FORTRAN programs that have relatively little system interaction.

"Part 2—Advanced FORTRAN Programming" describes all aspects of VS FORTRAN program development, including sophisticated use of the language and of the operating systems.

Both of these sections are organized in the same way; they present, in chronological order, the steps you follow in developing a FORTRAN application program:

1. Designing
2. Coding
3. Compiling
4. Fixing compile-time errors
5. Link-editing
6. Executing (in test mode)
7. Fixing execution-time errors
8. Executing (in production mode)

The rest of the manual describes particular self-contained VS FORTRAN features:

"Part 3—FORTRAN Special Features" describes how to use VS FORTRAN input/output facilities, the facilities for calling and called programs, and VM/370-CMS with VS FORTRAN. There's also a brief description of the execution-time library.

Appendixes give useful supplementary information: information on the hardware devices that can be used, and considerations for assembler language subroutines.

INDUSTRY STANDARDS

The VS FORTRAN Compiler and Library program product is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of June, 1980:

American National Standard Programming Language FORTRAN, ANSI X3.9-1978 (also known as FORTRAN 77).

International Organization for Standardization ISO 1539-1980 Programming Languages-FORTRAN.

These two standards are technically equivalent. In this manual, references to the current standard are references to these two standards.

American Standard FORTRAN, X3.9-1966.

International Organization for Standardization ISO R 1539-1972 Programming Languages-FORTRAN.

These two standards are technically equivalent. In this manual, references to the old standard are references to these two standards.

For both current and old standard language, a number of IBM language extensions are also included in VS FORTRAN. In this book, references to current FORTRAN or current language are references to the current standard plus the IBM extensions valid with it; references to old FORTRAN or old language are references to the old standard plus the IBM extensions valid with it.

RELATED PUBLICATIONS

This manual is designed as a guide on how to use VS FORTRAN at the current language level. It is not intended to be used as a reference manual.

Reference documentation for VS FORTRAN is given in the following publications:

VS FORTRAN Application Programming:

Language Reference manual, GC26-3986—which describes each syntactic element available when you're using the current language

System Services Reference Supplement, SC26-3988—which provides FORTRAN-specific system information on running VS FORTRAN programs under VM/370-CMS, OS/VS, and DOS/VSE

Library Reference manual, SC26-3989—which describes each mathematical and service subprogram contained in the execution-time library: its algorithm, accuracy, range, and error conditions

Source-Time Reference Summary, SX26-3731—which is a pocket-sized reference card containing formats for current source language and brief descriptions of the compiler options

IBM System/360 and System/370 FORTRAN IV Language, GC28-6515—which describes the source language available when you're using the old language.

FORTRAN Coding Form, GX28-7327—useful for coding fixed-form FORTRAN programs.

The previously listed publications give documentation only for application programming.

If you need information on VS FORTRAN installation or customization, see:

VS FORTRAN Installation and Customization, SC26-3987—which gives information on installation planning, on the compiler and library installation macros, storage requirements, and administrative information about controlling input/output, and how to create and alter the option table

The Program Directory for the system you're operating under, shipped with the product, gives details on installing VS FORTRAN.

RELATED SYSTEMS PUBLICATIONS

Detailed system information is not included in the VS FORTRAN publications. Therefore, in order to use this set of manuals properly, you should be sure you have on hand one of the following sets of manuals, depending on the system you're using.

For information on developing algorithms for direct files, see the _Introduction to IBM Direct Access Storgage Devices and Organization Methods_, GC20-1649.

VM/370-CMS Systems Publications

The following publications contain guide and reference documentation you'll need:

IBM Virtual Machine Facility/370:

CP Command Reference for General Users, GC20-1820

CMS User's Guide, GC20-1819

CMS Command and Macro Reference, GC20-1818

Terminal User's Guide, GC20-1810

OS/VS Systems Publications

OS/VS Linkage Editor and Loader, GC26-3813

OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide, GC26-3838

OS/VS Tape Labels, GC26-3795

MVS PUBLICATIONS: If you're using MVS, you'll need the following additional publications:

OS/VS2 MVS Data Management Services Guide, GC26-3875

OS/VS2 Access Method Services, GC26-3841

OS/VS2 MVS JCL, GC28-0692

OS/VS2 Debugging Guide, GT28-0632

OS/VS2 TSO Terminal User's Guide, GC28-0645

OS/VS2 TSO Command Language Reference, GC28-0646

TSO-3270 Structured Programming Facility (SPF) Program Reference Manual, SH20-1730

OS/VS1 PUBLICATIONS: If you're using OS/VS1, you'll need the following additional publications:

OS/VS1 Data Management Services Guide, GC26-3874

OS/VS1 Access Method Services, GC26-3840

OS/VS1 JCL Services, GC24-5100

OS/VS1 JCL Reference, GC24-5099

OS/VS1 Debugging Guide, GC24-5093

DOS/VSE Publications

The following publications contain guide and reference documentation you'll need:

DOS/VSE System Management Guide, GC33-5371

DOS/VSE Data Management Concepts, GC24-5138

DOS/VSE Serviceability Aids and Debugging Procedures, GC33-5380

VSE/VSAM Programmer's Reference, SC24-5145

Using VSE/VSAM Commands and Macros, SC24-5144

Using the VSE/VSAM Space Management for SAM Feature, SC24-5192

DOS/VSE Tape Labels, GC33-5374

DOS/VSE DASD Labels, GC33-5375

SUMMARY OF AMENDMENTS

RELEASE 1.1, JANUARY 1982

FORTRAN INTERACTIVE DEBUG

FORTRAN Interactive Debug Release 2.2 runs with VS FORTRAN-compiled programs.

SERVICE CHANGES

In addition to minor corrections to examples, the following areas have been changed:

- INCLUDE statement
- DATA statement
- Block IF statement
- Additional coding errors to avoid
- @PROCESS statement
- OS/VS JOB and DD statements
- Optional OS/VS compile-time data sets
- TSO LINK command
- Command procedures for foreground processing
- MAP option
- External filename
- BACKSPACE statement
- Subprogram references in FORTRAN
- Differences between VS FORTRAN and current implementations
- Statement number limits

Two major sections: "Using VM/370-CMS with VS FORTRAN" and "Using OS/VS2-TSO with VS FORTRAN" have been moved from Part 3 to Part 2.

RELEASE 1, JUNE 1981

MISCELLANEOUS CHANGES

- Page numbers have been changed in the list of figures.
- Description of the TERMINAL option has been added.
- Examples have been added and changed.
- A DOS/VSE load module logical units chart has been added.
- The order of variables in the COMMON statement has been corrected.

- Two appendixes have been added:
  - Differences Between VS FORTRAN and Current Implementations
  - Internal Limits in VS FORTRAN
- The glossary and index have been corrected.

RELEASE 1, APRIL 1981

OPTIMIZE COMPILER OPTION

The compiler option, OPTIMIZE(0), is available with Release 1.0 of VS FORTRAN. Additional optimization features (OPTIMIZE (1/2/3)), discussed in this manual, are planned for availability at a later time.

CONTENTS

VS FORTRAN Overview 1
VS FORTRAN—Thumbnail Description 1
VS FORTRAN Publications 2
IBM Extension Documentation 3

Part 1—Simplified FORTRAN Programming 5

Designing Your Program—Simplified Programming 6

Coding Your Program—Simplified Programming 7
Before You Begin 7
Retrieving Data—The READ Statement 7
  Converting Character Data—Internal READ Statement 8
Defining Data 8
  VS FORTRAN Data Types 9
  Defining Data Items 9
    Predetermined Data Type Definition 9
    Explicit Data Type Definition 10
  Defining Program Constants 11
    Using Program Constants Directly 11
    Using Names for Program Constants 11
    Program Reference Points—Using Statement Numbers . . 12
  Defining Arrays and Subscripts 12
    Defining and Referring to One-Dimensional Arrays . . . 12
    Defining and Referring to Multidimensional Arrays . . 13
Using the Assignment Statement 13
  Initializing Variables—Assignment Statement 14
  Using Arithmetic Expressions in Assignment Statements . 14
  Using Intrinsic Functions 15
Controlling Program Flow 16
  Programming Alternative Execution Paths—Arithmetic IF Statement 16
  Programming Alternative Execution Paths—Logical IF Statement 16
  Executing Procedures Repetitively—DO Statement 17
    Processing One-Dimensional Arrays—DO Statement . . . 17
    Processing Multidimensional Arrays—Nested DO Statement 17
Obtaining Results—Using the WRITE Statement 18
Ending Your Program—END Statement 19
Coding Errors to Avoid 19

Compiling Your Program—Simplified Programming 20
Entering Your Source Program 20
Requesting Compilation 20
  Requesting Compilation Only—OS/VS 20
  Requesting Compilation Only—DOS/VSE 21
Compiler Output 21

Fixing Compile-time Errors—Simplified Programming 22
    Using the Messages 23

Link-Editing Your Program—Simplified Programming 24
Requesting Link-Editing—OS/VS 24
Requesting Link-Editing—DOS/VSE 24
  Using the Link-Edited Program 25

Executing Your Program—Simplified Programming 26
Executing Your Program—OS/VS 26
  Setting Up Test Data Files—OS/VS 26
  Compile, Link-Edit, and Execute—OS/VS 27
Executing Your Program—DOS/VSE 27
  Setting Up Test Data Files—DOS/VSE 27
  Compile, Link-Edit, and Execute—DOS/VSE 28
Execution Output 28

Fixing Execution-Time Errors—Simplified Programming . . . 29

Finding Errors 29
Execution-Time Error Messages 29
Endless Loops or Waits 29
Unexpected Output 30
Using Debugging Packets 30
Fixing Errors 30

A Sample Program—Simplified Programming 31

Part 2—Advanced FORTRAN Programming 33

Designing Your Program—Advanced Programming 34
Top-Down Design and Development 34
Using Top-Down Design in VS FORTRAN 36
Top-Down Design with a Single Object Module—INCLUDE Statement 36
Top-Down Design with Multiple Object Modules—Using Subprograms 37

Coding Your Program—Advanced Programming 38
Using Fixed-Form Input—FORTRAN Coding Form 38
Using Free-Form Input 38
Defining Data 39
Predetermined Data Type Definition 40
Explicit Data Type Definition 40
Typing Groups of Data Items—IMPLICIT Statement 40
Typing for Specific Data Items—Explicit Type Statements 41
Data Classifications 42
Variables 42
Constants 42
Defining Constants by Value 42
Defining Constants by Name—PARAMETER Statement . . . 46
Arrays and Subscripts 46
One-Dimensional Arrays 47
Multidimensional Arrays 47
Arrays—Implicit Lower Bounds 47
Arrays—Explicit Lower Bounds 48
Arrays—Signed Subscripts 48
Arrays—Programming Considerations 49
Substrings of Character Items 49
Using Data Efficiently 50
Initializing Data—DATA Statement 50
Initializing Arrays—DATA Statement 50
Managing Data Storage—EQUIVALENCE Statement 51
Execution-Time Efficiency Using Equivalence 52
Defining and Using Expressions 53
Arithmetic Expressions 53
Character Expressions 55
Relational Expressions 56
Relational Expressions—Character Operands 57
Logical Expressions 57
Assigning Values to Data—Assignment Statement 58
Arithmetic Assignments 59
Character Assignments 59
Logical Assignments 60
Saving Coding Effort with Statement Functions 60
Controlling Program Flow 61
Using Structured Programming—Block IF Statement 61
Using the CONTINUE Statement 64
Programming Loops—DO Statement 64
Using Program Switches—Assigned GO TO Statement 66
Using Conditional Transfers—Computed GO TO Statement . 66
Suspending Program Execution—PAUSE and STOP Statements 67
Suspending Execution Temporarily—PAUSE Statement . . 67
Stopping Programs Permanently—STOP Statement 67
Ending Your Program—END Statement 68
Use Pre-Written Source Code 68
Avoiding Coding Errors 68

Compiling Your Program—Advanced Programming 71
Automatic Cross Compilation 71
Overall Job Control Considerations 71

Syntax for Job Control Statements 71
Using the Compile-Time Options 72
Modifying PROC Statements—OS/VS 82
Modifying EXEC Statements—OS/VS 82
Modifying DD Statements—OS/VS 83
Job Control Considerations—DOS/VSE 84
Using DOS/VSE Job Control Statements 85
Identifying a Job—DOS/VSE JOB Statement 85
Specifying Linkage-Editor Options—DOS/VSE OPTION Statement 85
Defining Files—DOS/VSE ASSGN Statement 85
Defining Files on Direct Access Devices—DOS/VSE 86
DLBL Statement—DOS/VSE 86
EXTENT Statement—DOS/VSE 86
Requesting Execution—DOS/VSE EXEC Statement 86
Requesting Compilation—DOS/VSE 87
Compiling a Single Source Program—DOS/VSE 87
Batch Compilation of More Than One Source Program—DOS/VSE 87
Compile-Time Files—DOS/VSE 88
Writing and Modifying Cataloged Procedures—DOS/VSE . . . 88
Retrieving Cataloged Procedures—DOS/VSE 88
Temporarily Modifying Cataloged Procedures—DOS/VSE . . 88
Compiler Output 89
Compiler Output with Default Options 89
Output with Explicit Compiler Options 90
Cataloging Your Source Program 90
Cataloging Your Source—OS/VS 90
Cataloging Your Source—DOS/VSE 91
Cataloging Your Object Module—DOS/VSE 92
Cataloging Your Object Module—OS/VS 92
Cataloging Your Object Module—DOS/VSE 92

Using VM/370-CMS with VS FORTRAN 93
The CP and CMS Commands 93
Using Your Terminal with CMS 93
Creating Your Source Program—CMS EDIT Command 93
Using the FORTRAN INCLUDE Statement—CMS 95
Compiling Your Program—Using CMS 95
Compiler Output—CMS 96
The LISTING File—CMS 96
The TEXT File—CMS 96
Making Libraries Available—CMS GLOBAL Command 96
Loading and Executing Your Program under CMS 97
Using the LOAD and INCLUDE Commands—CMS 97
Creating a Nonrelocatable Load Module—GENMOD Command 97
Defining Sequential and Direct Data Files—CMS 98
Specifying a File Identifier—CMS 98
Using the FILEDEF Command—CMS 98
Defining Sequential and Direct Disk Files—CMS 98
Defining Tape Files—CMS 98
Defining Terminal Files—CMS 99
Defining Unit Record Files—CMS 99
Defining VSAM Sequential and Direct Files—CMS 99
Defining a VSAM File to CMS 100
Using the DLBL Command—CMS 100
Creating a VSAM DEFINE Command—CMS EDIT Command . . . 100
Making the Catalog Entry—AMSERV Command 100
Creating and Processing VSAM Files—CMS 101

Using OS/VS2-TSO with VS FORTRAN 102
Using the TSO Commands 102
Using Your Terminal with TSO 102
Creating Your Source Program—TSO EDIT Command 102
Compiling Your Program—TSO ALLOCATE and CALL Commands . . 104
Allocating Compilation Data Sets—TSO ALLOCATE Command 104
Requesting Compilation—CALL Command 104
Compiler Output—TSO 105
The LIST File—TSO 105
The OBJ File—TSO 105
Link-Editing and Executing Your Program under TSO 105
Link-Editing Your Program—TSO LINK Command 106

Linkage Editor Listings—TSO LINK Command 106
Executing Your Program—TSO 106
Using the CALL Command—TSO Load Module Execution . . . 106
Using the Loader—TSO LOADGO Command 107
Fixing Execution Errors Under TSO 107
Requesting Command Procedure Processing Under TSO 108
Command Procedures For Foreground Processing 108
Command Procedures for Background Execution 108
TSO File Naming Conventions 109
File Identification—TSO ALLOCATE Command 110
Defining VSAM Sequential and Direct Files—TSO 111
System Considerations Under OS/VS2-TSO 111

Fixing Compile-Time Errors—Advanced Programming 112
Using the Compiler Output Listing 112
Compilation Identification 112
Source Program Listing—SOURCE Option 112
Diagnostic Message Listing—FLAG Option 113
OS/VS Cataloged Procedures and Compiler Message Codes 115
DOS/VSE Message Code Considerations 115
Using the MAP and XREF Options 115
Source Program Map—MAP Option 116
NAME Column 116
MODE Column 116
TAG Column 117
Address Column 117
COMMON Block Maps—MAP Option 117
Statement Label Map—MAP Option 118
Source Program Cross-Reference Dictionary—XREF Option 118
Data Item Dictionary—XREF Option 119
Statement Label Dictionary—XREF Option 119
End of Compilation Message 120
Using the Standard Language Flagger—FIPS Option 120

Link-Editing Your Program—Advanced Programming 122
Automatic Cross-System Support 122
Linkage Editor Input 122
Object Module as Link-Edit Data Set—DECK Option 122
ESD Record 123
TXT Record 123
RLD Record 123
END Record 124
Producing a Load Module—OBJECT Option 124
Link-Editing for Immediate Execution 124
Link-Editing Your Program—OS/VS 125
Cataloging Your Load Module—OS/VS 125
Executing a Link-Edit—OS/VS 125
Using the Linkage Editor—OS/VS 125
Using the Loader—OS/VS 126
Link-Editing Your Program—DOS/VSE 127
Cataloging Your Load Module—DOS/VSE 127
Executing a Link-Edit—DOS/VSE 128
Logical Units Used for Link-Editing—DOS/VSE 128
Linkage Editor Output 129

Executing Your Program—Advanced Programming 130
Executing Your Program—OS/VS 130
Using Load Module Data Sets—OS/VS 130
Using Cataloged Load Modules—OS/VS 131
Executing the Load Module—OS/VS 131
Execute Only—OS/VS 131
Link-Edit and Execute—OS/VS 131
Compile, Link-Edit, and Execute—OS/VS 131
Executing Your Program—DOS/VSE 132
Load Module Logical Units—DOS/VSE 132
Executing the Load Module—DOS/VSE 133
Execute Only—DOS/VSE 133
Link-Edit and Execute—DOS/VSE 133
Compile, Link-Edit, and Execute—DOS/VSE 133
Load Module Execution-Time Output 134
Execution without Error 134
Execution with Errors 134

Fixing Execution-Time Errors—Advanced Programming 135
Execution-Time Messages 135
Library Diagnostic Messages 135
Using the Optional Traceback Map 135
LIST Compiler Option and Traceback Maps 137
Program Interrupt Messages 137
Operator Messages 138
Using Debug Packets 139
Using Extended Error Handling 140
Extended Error Handling By Default 140
Controlling Extended Error Handling—Call Statements 142
Extended Error Handling—DOS/VSE Considerations . . . 143
Object Module Listing—LIST Option 143
Requesting Dumps 149
Requesting Dynamic Dumps—CALL Statement 149
Dynamic Dumps—DOS/VSE Considerations 149
Requesting an Abnormal Termination Dump 150
Requesting a Dump—OS/VS 150
Requesting a Dump—DOS/VSE 150

Part 3—FORTRAN Special Features 151

Programming Input and Output 152
Using VS FORTRAN Input/Output Statements 152
Using Common Options For Input/Output 152
Connecting to a File—OPEN Statement 153
Creating File Records—WRITE Statement 155
Retrieving File Records—READ Statement 155
Obtaining File Information—INQUIRE Statement 156
Disconnecting a File—CLOSE Statement 156
Using Unformatted and Formatted I/O 157
Formatting FORTRAN Records—FORMAT Statement 157
Group FORMAT Specifications 159
Using Specifications Repeatedly—FORMAT Control . . . 160
Using One FORMAT Statement with Variable Formats . . . 160
Using Sequential Files 161
Source Program Considerations 161
Using the OPEN Statement—Sequential Files 161
Using the WRITE Statement—Sequential Files 161
Using the READ Statement—Sequential Files 162
Using the ENDFILE Statement—Sequential Files 163
Using the REWIND Statement—Sequential Files 163
Using the BACKSPACE Statement—Sequential Files . . . 163
Using the CLOSE Statement—Sequential Files 164
Using Asynchronous Input/Output 164
Using the Asynchronous WRITE Statement 164
Using the Asynchronous READ Statement 165
Using the Asynchronous WAIT Statement 165
Using List-Directed Input/Output 166
Input Data—List-Directed I/O 166
List-Directed READ Statement 167
List-Directed WRITE Statement 168
Using Internal Files 168
Using the READ Statement—Internal Files 169
Using the WRITE Statement—Internal Files 169
System Considerations—Sequential Files 170
OS/VS Considerations—Sequential Files 170
DOS/VSE Considerations—Sequential Files 170
Using Direct Files 170
Source Program Considerations 171
Using the OPEN Statement—Direct Files 171
Using the WRITE Statement—Direct Files 171
Using the READ Statement—Direct Files 172
Using the CLOSE Statement—Direct Files 173
System Considerations—Direct Files 173
OS/VS Considerations—Direct Files 173
DOS/VSE Considerations—Direct Files 174
Using VSAM Files 174
Source Language Considerations—VSAM Files 175
Processing VSAM Sequential Files 176
Processing VSAM Direct Files 177
Processing VSAM-Managed Sequential Files 179
Obtaining the VSAM Return Code—IOSTAT Option 179

Defining a VSAM File 179
Defining a VSAM Sequential File 180
Defining a VSAM Direct File 180
Defining a DOS/VSE VSAM-Managed Sequential File . . . 181
Processing DEFINE Commands 181
Processing DEFINE Commands—OS/VS 181
Processing DEFINE Commands—DOS/VSE 182
Creating and Processing VSAM Files 182
Creating and Processing VSAM Files—OS/VS 182
Creating and Processing VSAM Files—DOS/VSE 182
System Considerations—Input/Output 183
Using Input/Output Labels 183
Magnetic Tape Labels 183
OS/VS Tape Label Considerations 184
DOS/VSE Tape Label Considerations 184
Direct Access Device Labels 184
Volume Labels 184
Processing Direct Access Device Labels 185
OS/VS Direct Access Label Considerations 185
DOS/VSE Direct Access Label Considerations 185
Defining FORTRAN Records—System Considerations 185
Record Formats—OS/VS 185
Defining Records—OS/VS 186
Record Formats—DOS/VSE 187

Coding Calling and Called Programs 189
Kinds of Called Programs 189
Sharing Data Between Programs 190
Passing Arguments Between Programs 190
Passing Arguments to a FUNCTION Subprogram 191
Passing Arguments to a SUBROUTINE Subprogram 191
General Rules for Arguments 192
Assigning Argument Values 192
Sharing Data Storage—COMMON Statement 193
Data Item Order—COMMON Statement 193
Type and Length Considerations—COMMON Statement . . . 194
Efficient Arrangement of Variables—COMMON Statement 195
EQUIVALENCE Considerations—COMMON Statement 197
Blank and Named Common 198
Blank Common 199
Named Common 199
Using Blank Common and Named Common 199
Using Blank and Named Common—Example 200
Coding Main Programs—Calling Programs 201
Naming Your Main Program—PROGRAM Statement 201
Invoking FORTRAN-Supplied Functions 201
Using FORTRAN-Supplied Functions as Arguments—INTRINSIC Statement 202
Comparing Character Operands—FORTRAN-Supplied Functions 202
Invoking FUNCTION and SUBROUTINE Subprograms 203
Invoking FUNCTION Subprograms 203
Invoking SUBROUTINE Subprograms—CALL Statement . . . 204
Coding Subprograms—Called Programs 205
Coding FUNCTION Subprograms 205
Coding SUBROUTINE Subprograms 206
Specifying Alternative Entry Points—ENTRY Statement 206
Alternative Entry Points in FUNCTION Subprograms . . . 206
Alternative Entry Points in SUBROUTINE Subprograms . . 207
Specifying Alternative Return Points—RETURN Statement 207
Retaining Subprogram Values—SAVE Statement 208
Initializing Named Common—BLOCK DATA Subprograms . . . 208
System Considerations 209
Overlaying Programs in Storage 210
Specifying OS/VS Overlays 210
Specifying DOS/VSE Overlays 211

Using the Optimization Feature 213
Select the Higher Optimization Levels 213
Write Programs of Efficient Size 213
Use Unformatted I/O 213
Use Array Operands for I/O Transfers 213
Use LOGICAL Variables of Length 4 214

Use INTEGER Variables of Length 4 214
Eliminate EQUIVALENCE Statements 214
Initialize Large Arrays During Execution 214
Use COMMON Blocks Efficiently 215
Pass Subroutine Arguments in COMMON Blocks 215
Don't Use Variably Dimensioned Arrays 215
Write Critical Loops Inline 216
Ensure Recognition of Duplicate Computations 216
Ensure Recognition of Constant Computations 217
Ensure Recognition of Constant Operands 217
Eliminate Scaling Computations 217
Define Arrays with Identical Dimensions 217
Define Arrays with Identical Element Specifications . . 217
Use Critical Variables Carefully 218
Avoid Unneeded Fixed/Float Conversions 218
Minimize Conversions Between Single and Double Precision 218
Use Scalar Variables as Accumulators 218
Use Efficient Arithmetic Constructions 218
Use IF Statements Efficiently 219
Use the Object Program Listing 219
Source Considerations with OPTIMIZE(3) 219
Common Expression Elimination—OPTIMIZE(3) 220
Computational Reordering—OPTIMIZE(3) 220
Instruction Elimination—OPTIMIZE(3) 220

Using The Execution-Time Library 222
Mathematical and Character Functions 222
Service Routines 222
Error-Handling Routines 223
Automatic Error Handling 223
Error Handling Under Your Control 223

Appendixes 225

Appendix A. Device Information 226
Minimum and Maximum Block Size Values 226
Direct Access Device Capacities 227

Appendix B. Assembler Language Considerations 228
Subprogram References in FORTRAN 228
The Argument List 228
The Save Area 229
The Calling Sequence 229
Linkage in Assembler Subprograms 229
Called Assembler Subprograms 229
Called and Calling Assembler Subprograms 230
Main Programs 230
Using FORTRAN Data in Assembler Subprograms 230
Using COMMON Data in Assembler Subprograms 230
Using Blank Common Data in Assembler Programs 231
Using Named Common Data in Assembler Programs 231
Retrieving Arguments in an Assembler Program 231
Retrieving Variables from the Argument List 231
Retrieving Arrays and Array Elements from the Argument List 232
Returning to Alternative Return Points 232
Internal Representation of FORTRAN Data 232
CHARACTER Items in Internal Storage 232
LOGICAL Items in Internal Storage 232
INTEGER Items in Internal Storage 233
REAL Items in Internal Storage 233
COMPLEX Items in Internal Storage 234

Appendix C. Differences Between VS FORTRAN and Current Implementations 236

Appendix D. Internal Limits in VS FORTRAN 238
Nested DO Loops 238
Expression Evaluation 238
Nested Statement Function References 238
Nested INCLUDE Statements 238
Nested Block IF Statements 238
Character Constants 238

Referenced Variables 238
Parentheses Groups 238
Statement Labels 239

Glossary 240

Index 245

FIGURES

1. VS FORTRAN Application Programming Publications 3
2. IBM Extensions in Examples and Figures 3
3. Two-Dimensional Array—Physical Layout in Storage . . 13
4. Arithmetic Operators and Operations 15
5. VS FORTRAN Error Message Example 22
6. Example of Top-Down Program Design 36
7. Data Types and Valid Lengths 39
8. Three-Dimensional Array—Implicit and Explicit Lower Bounds 48
9. Arrays—Effect of Negative Lower Bounds 49
10. Sharing Storage Between Arrays—EQUIVALENCE Statement 52
11. Storage Efficiency and the EQUIVALENCE Statement . . 54
12. Arithmetic Operators and Operations 55
13. Relational Operators and their Meanings 56
14. Logical Operators and their Meanings 57
15. Block IF Statement—Valid Forms 63
16. Cataloged Procedure FORTVC, OS/VS 78
17. Cataloged Procedure FORTVCL, OS/VS 78
18. Cataloged Procedure FORTVCLG, OS/VS 79
19. Cataloged Procedure FORTVLG, OS/VS 80
20. Cataloged Procedure FORTVG, OS/VS 80
21. Cataloged Procedure FORTVCG, OS/VS 81
22. Cataloged Procedure FORTVL, OS/VS 81
23. Compiler Output Using Explicit Options 91
24. VM/370-CMS Commands Often Used with VS FORTRAN . . . 94
25. OS/VS2-TSO Commands Often Used with VS FORTRAN . . . 103
26. Allocating TSO Compilation Data Sets 104
27. Source Program Listing Example—SOURCE Option 113
28. Examples of Compile-Time Messages—FLAG Option . . . 114
29. Example of a Storage Map—MAP Option 116
30. Module Identification Codes 118
31. Example of a Cross-Reference Dictionary—XREF Option 120
32. Object Module Structure 124
33. Load Module Data Sets—OS/VS 130
34. Load Module Logical Units—DOS/VSE 132
35. Operator Message Format 139
36. Using Batch Symbolic Debugging Statements 141
37. Library Names for DOS/VSE Error Handling and Service Routines 143
38. Object Module Listing Example—LIST Compiler Option 145
39. Some Codes Used with the FORMAT Statement 158
40. Display for FORMAT E14.5E2 159
41. FORTRAN Statements Valid with VSAM Files 175
42. DOS/VSE Logical Units and Devices Allowed 187
43. Example of Shared Data Areas—COMMON Statement . . . 193
44. Transmitting Values Between Common Areas 194
45. Blank and Named Common Storage Sharing 201
46. CALL Statement Execution 204
47. VS FORTRAN Devices—Minimum and Maximum Block Sizes 226
48. Direct Access Device Capacities 227

VS FORTRAN OVERVIEW

FORTRAN (FORmula TRANslator) is a programming language especially useful for applications involving mathematical computations and other manipulations of numeric data. It's particularly suited to scientific and engineering applications.

FORTRAN looks and reads much like mathematical equations, so that you can use conventional mathematical symbols and constructions to control computer operations. FORTRAN is problem-oriented and relatively machine-independent; this frees you from machine restrictions and lets you concentrate on the logical aspects of your data processing problems.

Compared with machine-oriented languages, FORTRAN gives you easy program development, decreased debugging effort, and overall greater data processing efficiency.

Source programs written in VS FORTRAN consist of statements you write to solve your problem; the statements must conform to the VS FORTRAN programming rules.

The VS FORTRAN compiler then analyzes your source program statements and translates them into machine language, which is suitable for execution on a computer system. The VS FORTRAN compiler also produces other output to help you debug your source and object programs.

The VS FORTRAN compiler generates object programs that use the services of the VS FORTRAN execution-time library, and of the supporting operating systems. It depends upon them for the programming services it must use.

The VS FORTRAN compiler operates under control of one of the following operating systems: VM/370-CMS, OS/VS2 MVS with or without TSO, OS/VS1, or DOS/VSE. You can compile your program under any one of these systems and then link-edit the program and its subroutines to execute under any of the others.

VS FORTRAN—THUMBNAIL DESCRIPTION

VS FORTRAN is a program product that runs under VM/370-CMS, under MVS or OS/VS1, or under DOS/VSE. It's compatible in language, because it accepts two language levels:

Current FORTRAN—1978 American National Standard FORTRAN (technically equivalent to ISO FORTRAN 1980), plus IBM extensions. (The language features described in this manual are current FORTRAN.)

Old FORTRAN—1966 American National Standard FORTRAN (technically equivalent to ISO FORTRAN 1972), plus IBM extensions. (Old FORTRAN language is documented in IBM System/360 and System/370: FORTRAN IV Language.)

The current FORTRAN language has a number of features never before available in System/370 FORTRAN:

Added Control Over Input/Output—through the OPEN, CLOSE, and INQUIRE statements

VSAM Sequential and Direct File Processing—through VSAM ESDS and RRDS data sets.

CHARACTER data type—gives you more flexible and direct control of character variables and arrays; this is useful when you're using READ and WRITE statements to process internal files.

Internal File Processing—lets you transfer data from one internal storage area to another. The READ statement converts the data from character to internal data types; the WRITE statement converts the data from internal data types to character data.

Structured Programming Aids—the block IF statement, plus the INCLUDE and CONTINUE statements, makes structured code sequences easy to implement.

Constant Names—let you name constants and define their values once at the beginning of a program; later in the program, a reference to the constant name is a reference to that value.

In addition, VS FORTRAN lets you write your source programs in either free or fixed format, lets you use the standard language flagger to identify nonstandard source language elements in your programs, and gives you diagnostics that are more informative than ever before.

VS FORTRAN PUBLICATIONS

The VS FORTRAN publications are designed to help you develop your programs with a minimum of wasted effort.

This book, VS FORTRAN Application Programming: Guide, gives guidance information on designing, coding, debugging, testing, and executing VS FORTRAN programs written at the current language level. It is not intended to be used as a reference manual.

A series of related publications give you detailed reference documentation you can use when you're actually performing the tasks this manual describes:

VS FORTRAN Application Programming:

Language Reference manual, GC26-3986—gives you the semantic rules for coding VS FORTRAN programs when you're using current FORTRAN. This manual also documents the differences between the old FORTRAN language and the current FORTRAN language.

Library Reference manual, SC26-3989—gives you detailed information about the execution-time library subroutines.

System Services Reference Supplement, SC26-3988—gives you FORTRAN-specific reference documentation for the system you're operating under.

Source-Time Reference Summary, SX26-3731—is a pocket-sized reference card containing current language formats and brief descriptions of the compiler options.

System/360 and System/370 FORTRAN IV Language manual, GC28-6515—gives you the rules for writing VS FORTRAN programs when you're using old FORTRAN.

Figure 1 shows how these manuals should be used together.

Application Programming Publications

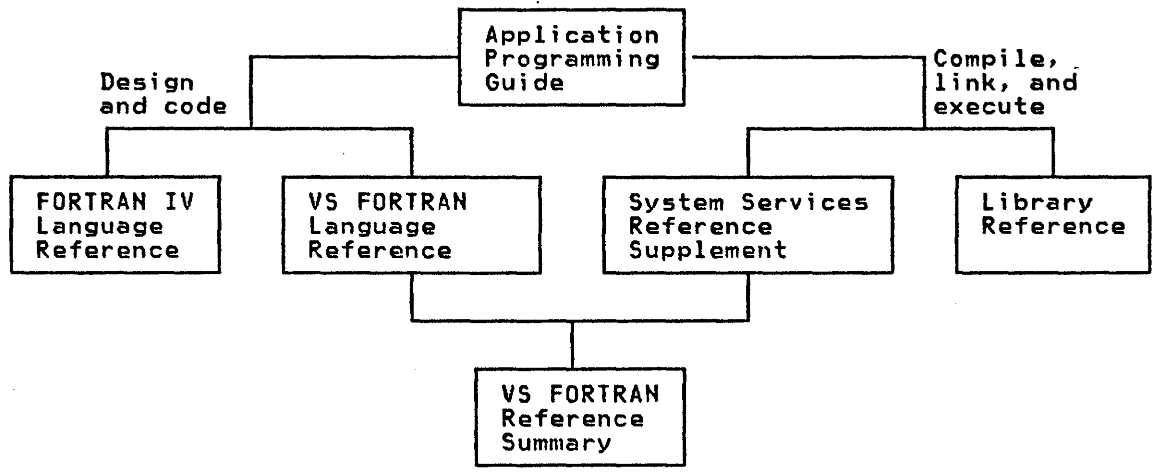
Application Programming Guide
Design and code
Compile, link, and execute
FORTRAN IV Language Reference
VS FORTRAN Language Reference
System Services Reference Supplement
Library Reference
VS FORTRAN Reference Summary

Figure 1. VS FORTRAN Application Programming Publications

IBM EXTENSION DOCUMENTATION

Sections of this manual describe VS FORTRAN source language usage, both usage of standard language and of IBM language extensions. The IBM extensions are indicated in the following ways:

IBM EXTENSION

In text, the IBM source language extensions are documented as this paragraph is shown.

END OF IBM EXTENSION

In examples and figures, IBM extensions are boxed, as shown in Figure 2.

Data Type	Valid Storage Lengths	Default Length
Integer	$\boxed{2}$ or 4	4

Figure 2. IBM Extensions in Examples and Figures

PART 1—SIMPLIFIED FORTRAN PROGRAMMING

If you are a student, or a scientist, engineer, or other professional who uses FORTRAN only as a tool for problem solving, this part of this manual is meant for you. It gives you the simpler ways of using current VS FORTRAN, without documenting data processing details you don't need.

This part is divided into seven sections, which guide you through the seven steps to follow when developing any VS FORTRAN program:

1. "Designing Your Program—Simplified Programming"

2. "Coding Your Program—Simplified Programming"

3. "Compiling Your Program—Simplified Programming"

4. "Fixing Compile-time Errors—Simplified Programming"

5. "Link-Editing Your Program—Simplified Programming"

6. "Executing Your Program—Simplified Programming"

7. "Fixing Execution-Time Errors—Simplified Programming"

If you find errors in your program at step 4 or 7, you must repeat the earlier steps. If you don't find any errors, you can of course omit steps 4 and 7.

The sample program at the end of this part illustrates how you can use VS FORTRAN to program a problem solution.

Most of the examples in the text for this part are taken from this sample program.

Before you begin coding any FORTRAN program, you should first consider what you want your program to do and the best way to do it. Every program has a purpose, and that purpose should guide its design.

Typically, a program reads some data, processes it, and then writes out the results of the processing. Thus, it's convenient to think of your program's structure in these three blocks and, in your logic structure, to take account of each block:

1. Reading in the data—know where the data will come from and the form it will have once you retrieve it.

2. Processing the data—know what you want to do with it, and the order in which you must do it, to get the results you're looking for. Consider the calculations you must code and their order, tests, loops, assignments, and so forth.

3. Writing out the results—decide where to print or record them.

Once you've decided the purpose, you must consider how your FORTRAN program can best be structured to meet the purpose you have in mind. After you have a general idea of how your program will be structured, it's a good idea to write a step-by-step picture of its logic: possibly through a brief outline or a flow chart.

Now you're ready to go to the next step, "Coding Your Program—Simplified Programming."

CODING YOUR PROGRAM—SIMPLIFIED PROGRAMMING

Once you've drawn up an overall design for your program, you can use VS FORTRAN to implement your design, statement by statement. The result is a VS FORTRAN program, a logical solution to your problem; this program is called the source program—because you write it in a source language, in this case FORTRAN.

Every FORTRAN program is made up of three elements: data, expressions, and statements:

> Data is a collection of factual items. In FORTRAN, these data items are represented by variables, constants, and arrays.
>
> Expressions are written representations of data relationships. The simplest form of an expression is the name of a single data item; through the use of operators (for example, arithmetic symbols), you can express more complicated forms of data relationships.
>
> Statements use data and expressions to tell the FORTRAN compiler what to do. There are two kinds of statements:
>
> > Nonexecutable Statements specify the nature of data you want to process or define the characteristics of your source program.
> >
> > Executable Statements cause operations to be performed.

All of these FORTRAN source program elements are explained in more detail later in this chapter.

In coding your program, you must follow the rules of the level of VS FORTRAN you're using:

- If you're coding a new VS FORTRAN program, use the VS FORTRAN Application Programming: Language Reference manual.
- If you're updating an existing FORTRAN IV program, use the IBM System/360 and System/370 FORTRAN IV Language manual.

BEFORE YOU BEGIN

The easiest way to code FORTRAN programs is by using the preprinted FORTRAN Coding Form; it's specially designed to help guide you in program coding.

The rules for writing VS FORTRAN fixed-form programs are easy to follow when you use the coding form.

IBM EXTENSION

In VS FORTRAN, you can also use free-form input— which frees you of many of the restrictions imposed by fixed form.

END OF IBM EXTENSION

Programming rules for coding VS FORTRAN source language statements are given in the VS FORTRAN Application Programming: Language Reference manual.

RETRIEVING DATA—THE READ STATEMENT

Your program probably retrieves data that it needs for processing; if it does, your program must use a READ statement.

In the READ statement, you tell the system the unit from which you want the data retrieved, and the data items in which you want the data placed. For example:

```
READ(UNIT=5)CHARIO
```

The meaning of each part of this READ statement is:

The word **READ** tells the compiler that you want to retrieve some data.

UNIT=5 identifies the unit that contains the data you want to retrieve. (The unit number 5 is only an example of a valid unit number; check with your system administrator for unit numbers that you can use with READ statements.)

CHARIO is a variable you've defined within your program and, in which you want the retrieved data placed. (Variables are data items that can change in value during program execution.)

When your READ statement is executed, the next item of data on the input device is placed in variable CHARIO.

CONVERTING CHARACTER DATA—INTERNAL READ STATEMENT

The preceding external READ statement transfers data into your storage without any data conversions.

If your input data comes from your terminal, or from the system card reader, you must then convert the external data (which is in character format—one byte per character) into an internal format your program can use.

You can use an internal READ statement to perform the conversion. For example:

```
READ (UNIT=CHARIO, FMT='(F3.1)') DELTBS
```

The meaning of each part of this READ statement is:

The word **READ** tells the compiler that you want to retrieve some data.

UNIT=CHARIO identifies the data item CHARIO as the unit that contains the character data you want to convert.

FMT='(F3.1)' gives the compiler the following information:

FMT= the following codes refer to formatting information.

F the input data is to be converted to real internal format; that is, a real number in floating-point notation, four bytes in length.

3.1 the input character data is contained in three bytes, with one place as a decimal fraction.

DELTBS tells the compiler to store the converted data in the real data item named DELTBS in which you want the retrieved data placed.

Programming rules for coding the READ statement are given in the VS FORTRAN Application Programming: Language Reference manual.

DEFINING DATA

All the data you use in a VS FORTRAN program—like the variables defined in the preceding READ statements, or program constants, or arrays—must be defined to the program. The following paragraphs tell you how to define all of them.

VS FORTRAN DATA TYPES

When you're writing a VS FORTRAN program, you must define all the data you use—its type and its organization. Your definitions can use default values or can be explicit.

The data types you'll most often use are:

> Integer items—made up of whole numbers. They can be signed or unsigned.
>
> Real items—numbers that contain either a decimal point or an exponent. They can be fractional; they can be signed or unsigned.
>
> Complex items—a pair of integer or real items, written within parentheses and separated by a comma. The first item is the real part of the complex number; the second item is the imaginary part. Either item can be signed or unsigned.
>
> Character items—made up of any characters in the computer's character set.

Use integer, real, and complex items in mathematical or relational expressions. Use character items in character expressions.

In storage, integer items are represented as binary fixed-point numbers; real and complex items are represented as floating-point numbers; character items are represented as one byte for each character in the item.

Reference documentation for these data types is given in the VS FORTRAN Application Programming: Language Reference manual.

DEFINING DATA ITEMS

You must define every data item you use in your VS FORTRAN program, such as the variable DELTBS in the example above, either through predetermined definitions or through explicit definitions.

Predetermined Data Type Definition

You can define data items (such as the DELTBS variable) by simply coding them in your READ statement (or any other executable statement).

FORTRAN then assigns a data type, predetermined by the initial character of the name, as follows:

- Names beginning with characters I through N are predefined as integer items of length 4.
- Names beginning with any other alphabetic character are predefined as real items of length 4.

IBM EXTENSION

- Names beginning with the currency symbol ($) are predefined as real items of length 4.

END OF IBM EXTENSION

(In the preceding example, the VS FORTRAN compiler assumes you want the variable DELTBS defined as a real item of length 4.)

These are the only data types that FORTRAN can predetermine for you.

Explicit Data Type Definition

You can define data items explicitly through the IMPLICIT statement and through explicit type statements.

USING THE IMPLICIT STATEMENT: Use the IMPLICIT statement to type groups of items, according to the initial character of their names. For example:

```
      IMPLICIT CHARACTER*15 (C)
```

specifies that any item whose initial character is C is a CHARACTER item of length 15, with one exception. The exception is that explicitly typed items are not included; that is, explicit typing overrides both IMPLICIT type definitions and predetermined type definitions.

USING EXPLICIT TYPE STATEMENTS: Use explicit type statements to define the data type for specific data items. For example:

IBM EXTENSION

```
      REAL*8 S(50),T(50),W(50),CM2
```

This statement defines the variables as follows:

- REAL*8 is a type definition that specifies CM2 as a real variable of length 8, and S, T, and W as real arrays, each element of which is of length 8. (Arrays and array elements are explained later in this chapter.)

END OF IBM EXTENSION

There are several other data types you can explicitly define:

- REAL defines data items as real items of length 4.
- INTEGER defines data items as integer variables of length 4.
- CHARACTER defines data items as character variables.
- DOUBLE PRECISION defines data items as items of length 8.
- COMPLEX defines data items as complex numbers.

If you include the preceding IMPLICIT statement and this explicit type statement in one program, the program data items are typed as follows:

Names	**Data Type**	**Length**
S, T, W	REAL	8
Other names beginning with S,T,W	REAL	4
Names beginning with A,B	REAL	4
CM2	REAL	8
Other names beginning with C	CHARACTER	15
Names beginning with D thru H	REAL	4
Names beginning with I thru N	INTEGER	4
Names beginning with O thru Z	REAL	4

IBM EXTENSION

Names beginning with $	REAL	4

END OF IBM EXTENSION

Programming rules for explicit type statements and the IMPLICIT statement are given in the VS FORTRAN Application Programming: Language Reference manual.

Note: In this manual, unless the example description explicitly states otherwise, examples use the predetermined type definitions for all data items.

DEFINING PROGRAM CONSTANTS

There are two ways to define constants in your program. You can state their values directly, or you can define them through names.

Using Program Constants Directly

You can define program constants directly by using their values within the program. For example, if you code:

```
TZ = I - 1
```

you've defined the value 1 as an integer constant of length 4.

However, if you code:

```
TZ = I - 1.0
```

you define the value 1.0 as a real constant of length 4. (Because your statement uses integer data, it's better if you define the constant as an integer constant; however, it isn't necessary.)

You can also define character constants—a group of characters, enclosed in apostrophes, that the program treats exactly as you've specified. For example:

```
' TC11 FAILED   '
```

is treated by the program as:

```
bTC11bFAILEDbbb
```

(Note that the program does not include the apostrophes as part of the character constant; however, it does include the trailing blanks enclosed within the apostrophes. In this example the blanks are shown as a series of 'b's—however, in actual program output they would appear as spaces.)

Programming rules for program constants, are given in the VS FORTRAN Application Programming: Language Reference manual.

Using Names for Program Constants

Suppose that your program must refer frequently to a particular constant value. In such a case, it may be more convenient to give the value a name and then use the name to refer to the value throughout the rest of the program.

In VS FORTRAN, you can use implicit or explicit type definitions and the PARAMETER statement for this purpose. For example:

```
PARAMETER(CM2=0.0001D0)
```

The CM2 in this example has been previously defined as a real item of length 8. This PARAMETER statement now defines CM2 as a constant with the value 0.0001. The D in the constant specifies that the value has the precision of a REAL item of length 8.

Once you've defined CM2 as the name of a constant, you can use the name CM2 throughout the program in references to the value 0.0001; for example:

```
21     IF (W(J-1) - W(J) - CM2)  23,16,16
```

The compiler treats this statement as exactly equivalent to:

```
21     IF (W(J-1) - W(J) - 0.0001)  23,16,16
```

You can use the PARAMETER statement to define constants of the following data types: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER.

IBM EXTENSION

You can also use the PARAMETER statement to define constants of the following data types: REAL*8, REAL*16, COMPLEX*32, INTEGER*2, or LOGICAL*1.

END OF IBM EXTENSION

Programming rules for program constants and for the PARAMETER statement are given in the VS FORTRAN Application Programming: Language Reference manual.

Program Reference Points—Using Statement Numbers

The "21" in the previous examples is a statement number. It serves as a reference point for other statements in the program—such as the logical IF statement described later in this chapter.

DEFINING ARRAYS AND SUBSCRIPTS

In FORTRAN, an array is a named set of data items, called array elements. Each array element in the set is the same size and has the same data characteristics as all the others; each element can be referred to in your program through subscripts:

```
7      DELS = S(1) - P
```

The (1) is a subscript, and tells the program you're referring to the first array element in the array named S.

You use names to identify arrays; you can implicitly or explicitly define the data type of an array in the same way as variables or program constants. For example:

IBM EXTENSION

```
REAL*8 S(50),T(50),W(50),CM2
```

The preceding statement defines S, T, and W as REAL arrays, each containing 50 array elements; each array element is a REAL item of length 8. (CM2 is the program constant previously described.)

END OF IBM EXTENSION

Defining and Referring to One-Dimensional Arrays

A one-dimensional array has a series of elements, each contiguous with the others and each the same size as the others. For example, suppose that in ARRAY1 there are four elements, containing the values 5, 10, 15, and 20, respectively.

You can define this array through a DIMENSION statement:

```
DIMENSION ARRAY1(4)
```

In this DIMENSION statement, you've defined ARRAY1 as a one-dimensional array containing four elements, each implicitly defined as a real item of length 4.

Your program can then refer to the elements in ARRAY1, using subscripts (integers enclosed in parentheses), as follows:

```
ARRAY1 (1)        refers to the element with value 5
ARRAY1 (2)        refers to the element with value 10
ARRAY1 (3)        refers to the element with value 15
ARRAY1 (4)        refers to the element with value 20
```

(Note that if you refer to ARRAY1 without specifying any subscripts you are making a collective reference to all four array elements. For most program references, this isn't valid—see the VS FORTRAN Application Programming: Language Reference manual for restrictions.)

In VS FORTRAN, you can use an integer variable as a subscript; in this way, you can place a new value into the variable each time your program must make a reference to the array. For example, if you place the value 2 in the variable ISUB1 and then specify:

```
ARRAY1(ISUB1)
```

you're making a reference to ARRAY1(2).

In this example, you've implicitly defined subscript ISUB1 as an integer data item of length 4.

Defining and Referring to Multidimensional Arrays

In VS FORTRAN, you can define arrays of up to seven dimensions, in which case, in order to refer to a specific array element, you must specify as many subscripts as there are dimensions. For example, the following DIMENSION statement defines an array of two dimensions:

```
DIMENSION ARRAY2(4,5)
```

In this statement, you've defined ARRAY2 as a two-dimenional array containing 20 elements, each implicitly defined as a real item of length 4.

Figure 3 shows how this array is logically laid out, with values in each sequential element increasing from 5 by 5.

```
 1,1   2,1   3,1   4,1   1,2   2,2   ...  4,4   1,5   2,5   3,5   4,5
------------------------------------      -----------------------------
|  5 |  10 |  15 |  20 |  25 |  30 | ...  | 80 |  85 |  90 |  95 | 100|
------------------------------------      -----------------------------
```

Figure 3. Two-Dimensional Array—Physical Layout in Storage

As with one-dimensional arrays, you can use integer variables as subscripts.

For example, if you place the value 2 in subscript ISUB1 and the value 5 in the subscript ISUB2, then the following reference:

```
ARRAY2 (ISUB1,ISUB2).
```

is a reference to ARRAY2(2,5)—the element containing the value 90.

When you must make sequential references to one array element after another, the value of the first subscript increases most rapidly, and the value of the last subscript increases least rapidly.

Programming rules for arrays and subscripts are given in the VS FORTRAN Application Programming: Language Reference manual.

USING THE ASSIGNMENT STATEMENT

Once your program defines the data it will access, you can use the assignment statement to manipulate it.

INITIALIZING VARIABLES—ASSIGNMENT STATEMENT

Before you use any item in your program, you must place a valid value in it. The READ statement in "Retrieving Data—The READ Statement" does so for the variable DELTBS; the PARAMETER statement does so for named constants. For variables, you can also use the assignment statement to initialize values.

For example, the following assignment statements

```
      CHAR1 = ' TC11 FAILED    '
      CHAR2 = ' TC11 COMPLETED'
```

initialize CHAR1 to the character value " TC11 FAILED   " and CHAR2 to the character value " TC11 COMPLETED".

Note that the blanks shown are included as part of the character values.

USING ARITHMETIC EXPRESSIONS IN ASSIGNMENT STATEMENTS

You can also use the assignment statement to evaluate arithmetic expressions:

```
10    DELT= 0.1 * DELT
```

When you use the assignment statement in this manner, you're using it much as you would any mathematical equation.

However, the assignment statement always tells the compiler to evaluate the expression to the right of the equal sign and to place the result in the item to its left.

Thus, in this example it's valid to multiply the current value of DELT by 0.1 and to place the result back into DELT.

The preceding rule also ensures that the two following statements are not equivalent:

```
      DELT= 0.1 * DELT     (valid FORTRAN statement)

      DELT * 0.1 = DELT    (statement in error—
                           cannot be compiled)
```

In addition, you must specify all computations explicitly. That is, if you specify the two variables A and B, as follows:

```
      AB
```

the two variables are not multiplied; instead, they're considered one variable, with the name AB. To multiply A and B, you must specify:

```
      A * B
      or
      B * A
```

The arithmetic operators you can use in mathematical expressions, and the order in which they're evaluated, are shown in Figure 4.

Operation	Arithmetic Operator	Order of Evaluation
Evaluation of Functions	(none)	First
Exponentiation	**	Second
Multiplication	*	Third
Division	/	Third
Addition	+	Fourth
Subtraction	-	Fourth

Figure 4. Arithmetic Operators and Operations

The VS FORTRAN compiler always evaluates mathematical expressions in this order, with the following additional considerations (important to remember because otherwise you may get results you don't expect):

- Within the exponentiation evaluation level, operations are performed right to left; within all other evaluation levels, operations are performed left to right.
- Parentheses are evaluated as they are in mathematics; they specify the order in which operations are to be performed. That is, expressions within parentheses are evaluated before the result is used. For example, the expression ((A-B)+C)*E is evaluated as follows:
  1. A-B is evaluated, giving result1
  2. result1+C is evaluated, giving result2
  3. result2*E is evaluated, giving the final result

Don't attempt to write two arithmetic operators consecutively in the same expression. The following expression is invalid:

```
A * -B
```

because the compiler cannot evaluate it properly. If you want to multiply A by -B, you can write:

```
A * (-B)
```

then -B is evaluated and the result is multiplied by A.

Programming rules for the assignment statement are given in the VS FORTRAN Application Programming: Language Reference manual.

USING INTRINSIC FUNCTIONS

The arithmetic operators allow you to code simple arithmetic operations easily. More complex arithmetic operations would be difficult to code.

However, VS FORTRAN has a set of mathematical functions you can use to perform specific operations. For example, to obtain and use the square root of real item ALG in an equation, you can write:

```
REL = INT1 + SQRT(ALG)
```

This statement causes VS FORTRAN to obtain the square root of ALG, add it to INT1, and place the result in REL.

Mathematical functions you can use include logarithmic, exponential, trigonometric, and hyberbolic functions, as well as functions that determine maximum and minimum values.

The mathematical functions are fully described in the VS FORTRAN Application Programming: Library Reference manual.

CONTROLLING PROGRAM FLOW

In a VS FORTRAN program, statements are ordinarily executed one after another, just as they appear in the program. However, VS FORTRAN lets you change the order of execution through FORTRAN control statements—which include the arithmetic and logical IF statements and the DO statement.

PROGRAMMING ALTERNATIVE EXECUTION PATHS—ARITHMETIC IF STATEMENT

Programs you write must often decide on alternative paths of execution, depending on the results of some previous action.

The arithmetic IF statement provides this function for your programs. For example, the following IF statement:

```
IF (J - 2) 20,21,20
```

tells the FORTRAN compiler to:

1. Evaluate the expression (J - 2). (The expression can be any arithmetic expression.)
2. If the result is less than zero, transfer control to statement number 20.
3. If the result is equal to zero, transfer control to statement number 21.
4. If the result is greater than zero, transfer control to statement number 20.

PROGRAMMING ALTERNATIVE EXECUTION PATHS—LOGICAL IF STATEMENT

Another way to program alternative paths of execution, in this case depending on the evaluation of some condition, is through the logical IF statement. For example, the following IF statement:

```
IF (TZ.EQ.5.0) GO TO 25
```

tells the FORTRAN compiler to:

1. Evaluate the expression

   ```
   (TZ.EQ.5.0)
   ```

   which is a relational expression telling FORTRAN to test whether or not the value in variable TZ is equal to 5.0. If TZ is equal to 5.0, the expression is "true"; if TZ is not equal to 5.0, the expression is "false."

   (The expression can be any relational or logical expression.)

2. If the result of the evaluation is "true," transfer control to statement 25.
3. If the result of the evaluation is "false," transfer control to the next executable statement following this IF statement.

Programming rules for the IF statement are given in the VS FORTRAN Application Programming: Language Reference manual.

EXECUTING PROCEDURES REPETITIVELY—DO STATEMENT

Another powerful control statement—the DO statement—lets you repetitively execute a whole series of statements a given number of times and then exit to continue sequential processing.

The DO statement is particularly useful when you want to initialize all of the items in an array.

Processing One-Dimensional Arrays—DO Statement

For example, you want to read a specific set of values into ARRAY0, depending on the current contents of ARRAY1. The DO statement in the following example sets up the loop to process each array element in turn:

```
      DOUBLE PRECISION ARRAY0
      DIMENSION ARRAY0(4), ARRAY1(4)
        .
        .
        .
      DO 40 INT=1,4,1

           (code to set values in ARRAY0,
           using values from ARRAY1)

40    CONTINUE
        .
        .
        .
```

This DO statement tells FORTRAN to execute the code to set values in ARRAY0 (that is, the code intervening between the DO statement and statement number 40) 4 times, as follows:

1. The program is to set INT to the value 1 (INT=1 tells it this).
2. The program is to execute the code sequence until INT equals 4 (INT=1,4 tells it this).
3. Each time the code sequence is executed, the value of INT is to be incremented by 1 (INT=1,4,1 tells it this).

Thus, the code sequence will be executed four times, and then the next statement after statement number 40 will be executed.

Processing Multidimensional Arrays—Nested DO Statement

You can include DO statements completely within the range of a DO loop—that is, among the statements within the range that the initial DO statement controls. You can use this FORTRAN feature to initialize multidimensional arrays. The DO statements in the following example show how it's done:

```
      DIMENSION ARRAY2 (4,5)
      VALU = 0.0
          .
          .
          .
      DO 40 ISUB2=1,5,1
      DO 40 ISUB1=1,4,1
      ARRAY2(ISUB1,ISUB2)=VALU
      VALU=VALU + 1.0
40    CONTINUE
          .
          .
          .
```

The first DO statement varies the second subscript from its minimum to its maximum value. The second DO statement varies the first subscript from its minimum to its maximum value as the first subscript is set to each of its specified values. When you specify these two DO statements in this order, your program places ascending values in each array element in sequence.

For a four-dimensional table, you'd specify

1. A DO statement controlling references to the fourth subscript.

2. Contained in that DO statement's range would be a second DO statement controlling references to the third subscript.

3. Contained in that DO statement's range would be a third DO statement controlling references to the second subscript.

4. Contained in that DO statement's range would be a last DO statement controlling references to the first subscript.

Programming rules for the DO statement are given in the VS FORTRAN Application Programming: Language Reference manual.

OBTAINING RESULTS—USING THE WRITE STATEMENT

To print or display the results of your program, use the WRITE statement.

For example, your program develops two values, TZ and V, which it then uses to test how the program plots a curve. At the end of execution, you'll want to save these values. The following example shows how you can do this:

```
      WRITE (UNIT=8) TZ,V
```

The meaning of each part of this WRITE statement is:

WRITE tells FORTRAN that your program is to output some information, in this case on an external device.

(UNIT=8) specifies that the information produced by the write statement will be sent to the output device identified by this unit number. (The unit number 8 is only an example; check with your system administrator for valid unit numbers you can use with the WRITE statement.)

TZ and **V** are the data items to be written to the output device.

The WRITE statement sends the data contained in each item listed, and in the order the items are specified.

Reference documentation for the WRITE statement is given in the VS FORTRAN Application Programming: Language Reference manual.

ENDING YOUR PROGRAM—END STATEMENT

Once your program has completed all the processing you want done, you must tell the VS FORTRAN compiler that there are no further statements to process.

You do this through the END statement, which is an executable statement and which must be the last statement on the last line in your program. For example:

```
27    END
```

The statement number (27) on this statement allows you to transfer control to this END statement from various parts of the program.

When the END statement is executed, program execution is ended, and control is returned to the system.

Programming rules for the END statement are given in the VS FORTRAN Application Programming: Language Reference manual.

CODING ERRORS TO AVOID

While you're coding your program, be careful not to make these common programming errors:

1. Misspelling FORTRAN words and data names.
2. Omitting required punctuation.
3. Not observing FORTRAN formatting rules.
4. Forgetting to assign values to variables and arrays before you use them.
5. Moving data into an item that's too small for it. (This causes truncation.)
6. Branching into DO loops from other areas in the program.

Errors like these can either prevent your program from executing at all or may cause your program to produce erroneous output. For many of these errors, you'll get compilation messages explaining what's wrong; for others, you may not get any error messages, because the error won't become evident until you actually execute the program.

You can quite often detect misspellings and uninitialized items by examining the source program map and the cross-reference listing.

Once your program is coded, you use the VS FORTRAN compiler to compile it, that is, to translate it into machine code. This is explained in "Compiling Your Program—Simplified Programming."

COMPILING YOUR PROGRAM—SIMPLIFIED PROGRAMMING

Your VS FORTRAN source program is meaningful to you, but it means nothing to the computer, which understands only a language known as machine code.

For this reason, you must compile your program. That is, you must request the VS FORTRAN compiler program to translate your FORTRAN source statements into an "object module"—a machine code translation of your source program.

ENTERING YOUR SOURCE PROGRAM

Your first step is to enter your source program into the system. You can, for example, enter the program from a terminal as a CMS or TSO file, or you can key it onto a diskette, or into a punched card deck.

For the method your organization uses, see your system administrator.

Whatever method your organization uses, you must submit the source program as a file with 80-character records; each record must follow VS FORTRAN formatting rules.

REQUESTING COMPILATION

Once you've entered your source program into the system, you can ask the VS FORTRAN compiler to process it. When you request compilation, the VS FORTRAN compiler reads and analyzes your source program and translates it into machine code.

You can request compilation along with the other steps needed for program development—link-editing and execution.

For early debugging, however, it's usually better to request a compile-only run. That way, the compiler can find syntax errors in your program that would prevent a successful execution, and you don't waste machine time and computer printout paper.

See "Using VM/370-CMS with VS FORTRAN" for information on compiling programs under CMS.

See "Using OS/VS2-TSO with VS FORTRAN" for information on compiling programs under TSO.

REQUESTING COMPILATION ONLY—OS/VS

The easiest way to request compilation under OS/VS is to use the VS FORTRAN cataloged procedure for compilation only.

Use the following job control statements to execute the compile-only procedure:

```
//jobname   JOB
//            EXEC FORTVC
//FORT.SYSIN DD *
(source program)
/*
//
```

where:

jobname
is the name you're giving this compilation-only job.

See the VS FORTRAN Application Programming: System Services Reference Supplement for more information on these job control statements.

REQUESTING COMPILATION ONLY—DOS/VSE

The easiest way to request compilation under DOS/VSE is to use the following job control statements:

```
//          JOB  jobname
//          EXEC VFORTRAN
            (source program)
/*
/&
```

where the source program is on SYSIPT, and:

jobname
is the name you're giving to this compilation-only job.

See the VS FORTRAN Application Programming: System Services Reference Supplement for more information on the DOS/VSE JOB statement.

COMPILER OUTPUT

The VS FORTRAN compiler gives you some or all of the following output, depending on the options in effect for your organization:

- The Source Program Listing—as you entered it, but with compiler-generated internal sequence numbers prefixed at the left; the sequence numbers identify the line-numbers referred to in compiler messages.
- An object module—a translation of your program in machine code.
- Messages about the results of the compilation.
- Other listings helpful in debugging.

These listings are described fully in "Fixing Compile-Time Errors—Advanced Programming" and "Fixing Execution-Time Errors—Advanced Programming" in Part 2; examples of output for each feature are also given there.

If your compilation was completed without error messages, you can proceed to "Link-Editing Your Program—Simplified Programming."

If your compilation caused error messages, you may have to fix up your errors, as described in "Fixing Compile-time Errors—Simplified Programming."

For each error it finds in your source program, the compiler gives you a self-explanatory error message; an example of these messages is shown in Figure 5.

```
*** VS FORTRAN ERROR MESSAGES ***

IFX1027I   RPLC   12(S)    27    NON-SUBSCRIPTED ARRAY NAME APPEARS AS LEFT-OF-EQUAL
                                 SIGN VARIABLE. SPECIFY A SUBSCRIPTED ARRAY NAME OR
                                 A VARIABLE NAME.
```

Figure 5. VS FORTRAN Error Message Example

The way the messages are printed is fixed. Each message begins with an 8-character identifier and a 4-character pointer, followed by a level code, followed by the internal sequence number, followed by the message text. Each part is explained in the following paragraphs.

MESSAGE IDENTIFIER: Each message begins with an 8-character identifier:

IFX
: identifies the message as one from the VS FORTRAN compiler. (If the message has some other prefix, it was sent by some other part of the system; in this case, see your system administrator.)

nnnnI
: uniquely identifies this message.

4-CHARACTER POINTER: The message identifier is followed by a 4-character pointer, which is sometimes needed by system programmers for detailed error diagnosis.

LEVEL CODE: The 4-character pointer is followed by a code, which tells you the severity level of the message:

16(U)
: for unrecoverable error. The compilation was stopped before it was complete.

12(S)
: for severe error. The compiler cannot guarantee a compilation that will execute correctly. The statement in error was not processed. (Errors after this one in this same statement couldn't be found in this compilation.)

8(E)
: for error. The compiler found an error and attempted to correct it; the program may or may not execute correctly.

4(W)
: for warning. The compiler detected a possible error in your program.

0(I)
: for informational. A note giving you information about compiler-detected conditions during compilation.

You must always correct U-level, S-level, or E-level errors before attempting another compilation.

You should review W-level messages to see if they'll let your program execute correctly; if they won't, you must correct them.

I-level messages do not necessarily indicate an error in your program. Therefore, you may not need to make any corrections because of them.

INTERNAL SEQUENCE NUMBER (ISN): This is a compiler-generated number showing the statement at which the error occurred. It isn't printed if the error isn't connected with one particular statement. (This is the sequence number prefixed to each statement in the source program listing.)

Logical IF statements consist of two parts (heading and trailing). Normally, an ISN is assigned is assigned to each part of the statement. However, if the trailer is an unconditional GOTO, it will not be assigned an ISN, because the effect of this statement is contained in the evaluation of the header.

MESSAGE TEXT: The self-explanatory text of the message.

Using the Messages

With the information the compiler messages give you, you can go back, correct your source program as indicated, and then recompile your program.

When your program compiles correctly (without any message levels higher than I) you can go on to the next step, and link-edit it, as described in "Link-Editing Your Program—Simplified Programming."

LINK-EDITING YOUR PROGRAM—SIMPLIFIED PROGRAMMING

Before you can execute your program, you must link-edit it—even if you have only one program and want to "link" it only with the system.

During program development when you're testing one version after another, or if your program is one you're expecting to execute only a few times, you can combine the link-editing step with the execution step; see "Executing Your Program—Simplified Programming" on how to do this.

However, if you will be executing this program many times, you should compile and link-edit it, and then file it in a library for future reference.

For information on link-editing under CMS, see "Using VM/370-CMS with VS FORTRAN."

For information on link-editing under TSO, see "Using OS/VS2-TSO with VS FORTRAN."

REQUESTING LINK-EDITING—OS/VS

The simplest way to link-edit your program under OS/VS is to use the cataloged procedure, which compiles and link-edits your program all in one step. The job control statements you use are:

```
//jobname    JOB
//           EXEC FORTVCL
//FORT.SYSIN DD *
(source program)
/*
//
```

where:

jobname
is the name you're giving this compile-and-link-edit job.

See the VS FORTRAN Application Programming: System Services Reference Supplement for more information on these job control statements.

REQUESTING LINK-EDITING—DOS/VSE

The easiest way to request link-editing under DOS/VSE is to use the following job control statements:

```
// JOB jobname
// OPTION CATAL
   PHASE FIRST,*
// EXEC VFORTRAN,SIZE=800K
     (source program)
/*
// EXEC LINKEDT
/&
```

where the source program is on SYSIPT, and:

jobname
is the name you're giving to this compile-and-link-edit job.

OPTION CATAL
causes the system to catalog the relocatable phase (object module) in the core image library.

PHASE FIRST,*
gives the cataloged phase the name FIRST.

See the VS FORTRAN Application Programming: System Services Reference Supplement for more information on the DOS/VSE JOB statement.

USING THE LINK-EDITED PROGRAM

You can catalog the output of your link-edit job and then execute the link-edited module any time you want.

How you execute your program is described in "Executing Your Program—Simplified Programming."

EXECUTING YOUR PROGRAM—SIMPLIFIED PROGRAMMING

Once you've compiled your program correctly and link-edited it, you can execute it.

The easiest way to execute it is to combine all three steps into one: compilation, link-editing, and execution.

During testing, if your program doesn't change existing files of data, you may want to execute it as it stands, right away. However, if it alters valuable data files, you should develop alternative data files for testing, and then try out program execution, using the alternative test data files. In this way, if there are execution errors, you won't introduce bad data into the real files.

See "Using VM/370-CMS with VS FORTRAN" for information on executing your program under CMS.

See "Using OS/VS2-TSO with VS FORTRAN" for information on executing your program under TSO.

EXECUTING YOUR PROGRAM—OS/VS

The simplest way to execute your program under OS/VS is to use one of the cataloged procedures:

- Set up any test data files that you'll need, and then
- Compile, link-edit and execute your source program all in one job

The following sections outline the job control statements to use.

SETTING UP TEST DATA FILES—OS/VS

The easiest way to set up data files for test execution is to use standard system files for them. The standard system files, such as SYSIN, SYSOUT, and SYSPRINT, are predefined, so that the job control statements are very simple.

Under OS/VS, the simplest way to prepare test data files is to develop them as 80-character records and then include them in the input stream. You use the following job control statements:

```
//FTxxF001 DD *
        (your data records)
/*
```

where

xx
: is the FORTRAN unit number for your test data file.

  Note that this is not a standard system file; the standard system files for input are named either FT05F001 or SYSIN.

COMPILE, LINK-EDIT, AND EXECUTE—OS/VS

The job control statements you use are:

```
//jobname JOB
          EXEC FORTVCLG
//FORT.SYSIN DD *
  (VS FORTRAN source program)
/*
  (compilation and link-edit steps executed)
//GO.SYSIN DD *
  (data)
/*
  (program execution step executed)
//
```

where:

jobname
is the name you're giving this compile, link-edit, and execute job.

In this example, both your source program and your test data are on the system input device (SYSIN).

For reference documentation on these job control statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

EXECUTING YOUR PROGRAM—DOS/VSE

Under DOS/VSE, the simplest way to execute your program is to request compilation, link-editing, and execution outlined in the following section:

- Set up your test data files, and then
- Compile, link-edit and execute your source program all in one job

The following sections outline the job control statements to use.

SETTING UP TEST DATA FILES—DOS/VSE

The easiest way to set up data files for test execution is to use standard system files for them.

Under DOS/VSE, the simplest way to prepare test data files is to develop them in card image form, and then include them in the input stream, following the EXEC statement. See the DOS/VSE example in the next section.

COMPILE, LINK-EDIT, AND EXECUTE—DOS/VSE

To request compilation, link-editing, and execution under DOS/VSE, you use the following job control statements:

```
// JOB jobname
// OPTION LINK    (to execute without saving the phase)
   or
// OPTION CATAL   (to execute and save the phase)
// EXEC VFORTRAN,SIZE=800K
   (source program)
/*
   (compilation step executed)
// EXEC LINKEDT
   (link-edit step executed)
// EXEC
   (data)
/*
   (program execution step executed)
/&
```

where the source program and the test data are on the system input device (SYSIPT), and

jobname
is the name you're giving to this job.

For reference documentation on these job control statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

EXECUTION OUTPUT

When your program executes without errors and with the results you expect, you've completed your job of program development.

If you find errors in your program output, and you may, you'll have to go on to the next step and fix up your errors, as described in "Fixing Execution-Time Errors—Simplified Programming."

FIXING EXECUTION-TIME ERRORS—SIMPLIFIED PROGRAMMING

Your program may run the first time without any errors at all, in which case, you've completed your job of program development. However, you'll often discover that it contains errors, which you must correct. The following sections outline what you must do.

FINDING ERRORS

If your program has errors, you'll find it out in one of three ways:

- Your program ends prematurely, and you receive an error message.
- Your program keeps running but never finishes execution.
- The output you're getting is not what you expect.

EXECUTION-TIME ERROR MESSAGES

The messages you get at execution time may be from VS FORTRAN or from some other part of the system.

Execution-time messages from the VS FORTRAN library have message identifiers beginning with IFY; they're presented in a format similar to that of the compiler messages (see "Fixing Compile-time Errors—Simplified Programming").

The text of the message usually explains what the problem is; if you need supplementary information, you can find it in the _VS FORTRAN Application Programming: Library Reference_ manual, which lists all the execution-time messages.

If the message has some other prefix, it comes from another part of the system. In this case, ask your system administrator for assistance.

ENDLESS LOOPS OR WAITS

If your program runs on and on and never finishes, it's caught either in a closed loop or a wait:

- A _closed loop_ is a series of statements that repeat themselves endlessly.
- A _wait_ is a suspension of execution while the program waits for an action to occur (which may never happen).

When either result occurs, check the logic of your program:

- For Endless Loops—check for any unintentional loops; check to make sure that every intentional loop has an exit—that it either has a finite number of repetitions or a feasible condition that causes execution to branch out of it.

  You can also recompile and run the program again, inserting WRITE statements to indicate when your program is entering a specific loop, and when it is exiting from that loop.

- For Endless Waits—check that a PAUSE statement is receiving the expected operator response, or that the operator has provided the system resources your program expects, for example, that a tape was mounted properly.

If you can't find the cause, even after checking these possibilities, ask your system administrator for assistance.

UNEXPECTED OUTPUT

If you're getting unexpected output, you may have one or more of the following problems in your program:

- Your input or output statements (READ or WRITE) are in error.
- Your logic flow is incorrect. For example, a WRITE statement may not be executed or may be executed at the wrong time, or a calculation is being skipped over or executed too often.
- Your mathematical calculations themselves are in error, so that the data is wrong when you write it.
- The input you expect is itself incorrect, so that the program itself is executing correctly, but appears to be in error.
- Your program constants may inadvertently be changed during execution.

Sometimes the nature of the erroneous output tells you what is wrong. If it doesn't, double check your source program, looking for the kinds of errors outlined above.

Sometimes, by inspecting the source program map and the cross-reference listing, you can find inconsistencies in the way your program uses variables.

If you can't find the cause, even after checking these possibilities, ask your system administrator for assistance.

USING DEBUGGING PACKETS

VS FORTRAN also has a number of source statements that let you define debugging packets at the beginning of your source program; a debugging packet lets you obtain information about the conditions existing during program execution and may help you pinpoint where your errors are. See "Fixing Execution-Time Errors—Advanced Programming" for further information.

If, even after using a debugging packet, you still can't find what's wrong, ask your system administrator for assistance.

FIXING ERRORS

If you find errors in your program, you must go back and recode the erroneous statements. Before you try to execute it to obtain the desired results, you must then recompile and relink-edit it.

A SAMPLE PROGRAM—SIMPLIFIED PROGRAMMING

The following sample program illustrates most of the programming capabilities discussed in the previous chapters.

```
      PROGRAM SAMPLE
C GENERALIZED TEST CASE
C TESTS NESTED DO, GO TO AND IF WITHIN DO, VARIABLE SUBSCRIPTS,
C MULTIPLE IFS IN SUCCESSION, PARAMETER AND IMPLICIT STATEMENTS.
C THE ROUTINE ITSELF GENERATES VARIOUS SLOPES.  THE TEST
C CASE IS SELF CHECKING.  SLOPE IS CALCULATED AT THE 5 POINTS ON THE
C CURVE, 1.,2.,3.,4., AND 5.
      IMPLICIT CHARACTER*14 (C)
      REAL*8 S(50),T(50),W(50),CM2
      PARAMETER(CM2=0.0001D0)
      CHARIO='0.1           '
      CHAR1 = ' TC11 FAILED   '
      CHAR2 = ' TC11 COMPLETED'
      READ(UNIT=5) CHARIO
      READ(UNIT=CHARIO,FMT='(F3.1)') DELTBS
  1   DO 25 I=1,6
  2   TZ = I - 1
  3   P = (TZ + 1.0) * (TZ *TZ)
  4   DELT = DELTBS
  5   T (1) = TZ + DELT
  6   S (1) = (T(1) + 1.0)*(T(1)*T(1))
  7   DELS = S(1) - P
  8   W(1) = DELS/DELT
  9   DO 16 J =2,50
 10   DELT= 0.1 * DELT
 11   T(J) = TZ + DELT
 12   S(J) = (T(J) +1.0)*(T(J)*T(J))
 13   DELS = S(J) - P
 14   W(J) = DELS/DELT
 19   IF (J - 2) 20,21,20
 20   A = W(J-1) - W(J)
      B = W(J-2) - W(J-1)
      IF (A - B) 21,22,22
 21   IF (W(J-1) - W(J) - CM2)  23,16,16
 16   CONTINUE
 22   V = W(J-1)
      GO TO 24
 23   V = W(J)
 24   IF (TZ.EQ.0.0.AND.V-0.0.GT.0.1) GO TO 26
      IF (TZ.EQ.0.0) GO TO 25
      IF (TZ.EQ.1.0.AND.V-5.0.GT.0.1) GO TO 26
      IF (TZ.EQ.1.0) GO TO 25
      IF (TZ.EQ.2.0.AND.V-16.0.GT.0.1) GO TO 26
      IF (TZ.EQ.2.0) GO TO 25
      IF (TZ.EQ.3.0.AND.V-33.0.GT.0.1) GO TO 26
      IF (TZ.EQ.3.0) GO TO 25
      IF (TZ.EQ.4.0.AND.V-56.0.GT.0.2) GO TO 26
      IF (TZ.EQ.4.0) GO TO 25
      IF (TZ.EQ.5.0.AND.V-85..GT..2) GO TO 26
      IF (TZ.EQ.5.0) GO TO 25
      GO TO 25
 26   WRITE (FMT=100,UNIT=6) CHAR1,TZ,V
C   WRITE DATA TO DISK FILE, UNFORMATTED
      WRITE (UNIT=8) TZ,V
 25   CONTINUE
      WRITE (6,101) CHAR2
 100  FORMAT (A15,' WITH TZ AND V =, RESPECTIVELY,',F4.1,F12.4)
 101  FORMAT (A15)
      STOP
 27   END
```

PART 2—ADVANCED FORTRAN PROGRAMMING

If you are a programmer who uses the entire range of FORTRAN language capabilities in coding your programs, this part of this manual is meant for you. It tells you how to take advantage of the current VS FORTRAN language and processing capabilities in order to create efficient VS FORTRAN programs.

This part is divided into seven sections, to guide you through the seven steps you follow when developing any VS FORTRAN program:

1. "Designing Your Program—Advanced Programming"
2. "Coding Your Program—Advanced Programming"
3. "Compiling Your Program—Advanced Programming"
4. "Fixing Compile-Time Errors—Advanced Programming"
5. "Link-Editing Your Program—Advanced Programming"
6. "Executing Your Program—Advanced Programming"
7. "Fixing Execution-Time Errors—Advanced Programming"

If you find errors in your program at step 4 or 7, you must repeat the earlier steps. If you don't find any errors, you can of course omit steps 4 and 7.

Program development considerations for VS FORTRAN mainline processing are documented in this part.

Programming considerations for input and output, for calling and called programs, for the optimization feature, for the execution-time library, for VM-370/CMS, and for OS/VS2-TSO are given in Part 3.

DESIGNING YOUR PROGRAM—ADVANCED PROGRAMMING

Program design is your first step in application development—when you design your program, you're setting up the framework for a proper solution.

In your thinking, of course, you must take into account the hardware and software available to you; for example, the processing unit, disk drives, diskettes, tape drives, and terminals you can use, plus the system you operate under, the program products available, other application programs your program can use, and sequences of code you can use in common with other applications.

Typically, a program reads data, processes it, and then writes out the results in one form or another. Thus, it's usually convenient if you think of your program structure in these three blocks:

1. Reading the data. Understand where the data will come from, and what it will look like.

2. Process the data. Know that you want to do with it—what results you are looking for. Consider the manipulations and repetitions to achieve them: mathematical calculations, tests, loops, moves, and so forth.

3. Write the results. Decide where to print or record them, and how they should look (if they're for general use, possibly add explanatory text).

Almost every program is made up of these three basic read, process, and write elements. However, if the application you're developing is large and complex, you may find it desirable to split up one or more of these basic elements into smaller, more manageable units of processing logic.

If you begin your program design with these most inclusive program elements, and then develop the logic of each successively more detailed level of the program design, you're actually using top-down design—one of the more powerful design tools available to you—to solve your application problem.

TOP-DOWN DESIGN AND DEVELOPMENT

When you're using top-down design and development, you can make your program design—which usually maps out the larger relationships between program elements—go hand in hand with the coding effort. Used as a conscious strategy, it can make your program development faster and more efficient. You'll do less backtracking to fill in missing code and remove redundancies in the program logic.

When you use top-down design and development, you code the most inclusive logic portions of your program first, and test and debug them. Then, you code the next successively lower logic portions and add them to the portions that are already up and running. At each level, you can code in stubs to represent the next lower logic level.

A stub is a place holder for an unwritten sequence of code—for example, something as simple as a comment stating what the code will do, or something as elaborate as a complete description of its processing logic together with portions of completed code.

In this way, when you add new modules of code, you're adding them to code that you've already tested and debugged. You'll usually find errors only within the new code itself or in the interconnections between the new and the already existing code.

You'll find a number of advantages in using this development strategy:

- You'll detect design logic errors very early. This is because you're integrating the program's functions as you develop it.
- You'll detect and locate coding errors easily. Any error that appears in a newly developed sequence of code is usually within the newly developed sequence itself, or in the interconnections between the new sequence and the old ones.
- Your program testing begins earlier in the development cycle, since you test each new module as you add it.
- You don't need temporary drivers to test uncompleted code. (A driver is a temporary control module not used in the completed program.) Your control modules are at the highest logic level, and they are the first modules you'll complete and test.

By breaking up a large program into smaller logic modules, you get other added advantages. You make the overall logic of the program more apparent, so that it's easier to read, easier to understand, easier to implement, and easier to change.

Figure 6 illustrates the top-down design and development strategy. In the figure, the following points apply:

1. Module A is the highest-level module of all. It controls modules B, C, and D.
2. Modules A, B, C, E, and G have been developed and tested.
3. Module B is related to lower-level modules E (fully-developed) and F (which is an undeveloped stub).
4. Module C is related to lower-level module G (fully-developed).
5. Stub module D is related to stub proposed submodules H and I. Stub module D has been written as a stub; stub modules H and I are provided for in the program design but haven't yet been written, even as stubs.
6. Even though it's functionally incomplete, this program will execute to its logical end.

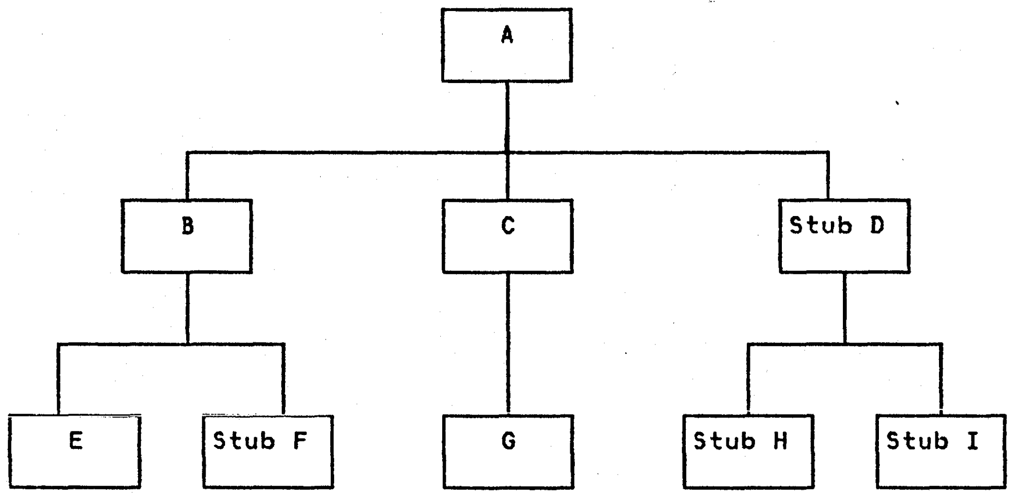

Figure 6. Example of Top-Down Program Design

Once you have a design and program development strategy worked out, you're ready to proceed to the next task in application development: writing your source program.

USING TOP-DOWN DESIGN IN VS FORTRAN

To help you use top-down design and development, VS FORTRAN provides:

IBM EXTENSION

- The nested INCLUDE statement

END OF IBM EXTENSION

- SUBROUTINE and FUNCTION Subprograms

IBM EXTENSION

TOP-DOWN DESIGN WITH A SINGLE OBJECT MODULE—INCLUDE STATEMENT

In VS FORTRAN, you can nest INCLUDE statements to as many as 16. This lets you take advantage of top-down design in a program that you want to compile as one object module. This is how you do it:

Write the highest-level sequences of code in your program first; these sequences contain INCLUDE statements for the lower-level code sequences.

During early program development, these INCLUDE statements can point to undeveloped stubs that stand in for the lower-level code.

Later in program development, these same INCLUDE statements can point to the completed code sequence.

Because INCLUDE statements can be nested, you can code INCLUDE statements at every code sequence level in your program. For example, in the program shown in Figure 6, you could code the following INCLUDE statements:

1. Code Sequence A could contain:

   INCLUDE (B) (B contains tested and debugged code)

   INCLUDE (C) (C contains tested and debugged code)

   INCLUDE (D) (D is a stub, to be replaced later)

2. Code Sequence B could contain:

   INCLUDE (E) (E contains tested and debugged code)

   INCLUDE (F) (F is a stub, to be replaced later)

3. Code Sequence C could contain:

   INCLUDE (G) (G contains tested and debugged code)

4. Code Sequence D is a stub; later it could contain:

   INCLUDE (H) (H is not yet written, even as a stub)

   INCLUDE (I) (I is not yet written, even as a stub)

You get two advantages when you use the INCLUDE statement in this way:

- You can produce a program structure that's syntactically correct and that you can compile correctly, even before you've completed all the coding.
- You avoid the execution overhead that CALL statement linkages entail (although there is some extra compile-time overhead).

END OF IBM EXTENSION

TOP-DOWN DESIGN WITH MULTIPLE OBJECT MODULES—USING SUBPROGRAMS

For large or complex programs, you can use SUBROUTINE or FUNCTION subprograms to separately code logically distinct portions of your program. Each subprogram is a complete program within itself, each subprogram communicating with the others through the parameters they pass and through COMMON data areas.

During program development, you'd develop and code the highest level control modules first. In the control modules you can code statements that invoke the next lower level subprograms. During early development, these subprograms could be stubs; later, they could be completed subprograms.

Using this method, you can produce a program structure that's syntactically correct and that you can compile and link-edit correctly, even at the earliest stages of program development.

Once you've developed all the programs, you can link-edit them together into one load module.

CODING YOUR PROGRAM—ADVANCED PROGRAMMING

Once you've completed your program design, you can use the VS FORTRAN language to create a source program, a logical solution to your problem that follows your program design.

Every FORTRAN program is made up of three elements: data, expressions, and statements:

> Data is a collection of factual items. In FORTRAN, these data items are represented by variables, constants, and arrays.
>
> Expressions are written representations of data relationships. The simplest form of an expression is the name of a single data item; through the use of operators (for example, arithmetic symbols), you can express more complicated forms of data relationships.
>
> Statements use data and expressions to tell the FORTRAN compiler what the object program must do. There are two kinds of statements:
>
> > Nonexecutable Statements specify the nature of data you want to process or define the characteristics of your source program or define the way in which data is to be read or written.
> >
> > Executable Statements cause operations to be performed.

All of these FORTRAN source program elements are explained in more detail later in this chapter.

In coding your program, you must follow the rules of the level of VS FORTRAN you're using:

- If you're coding a new VS FORTRAN program, use the VS FORTRAN Application Programming: Language Reference manual.
- If you're updating an existing FORTRAN IV program, use the IBM System/360 and System/370 FORTRAN IV Language manual.

USING FIXED-FORM INPUT—FORTRAN CODING FORM

Fixed-form input is the traditional way to code FORTRAN programs; the FORTRAN Coding Form is designed to help guide you in fixed-form program preparation.

For reference documentation about VS FORTRAN fixed-form input, see the VS FORTRAN Application Programming: Language Reference manual.

IBM EXTENSION

USING FREE-FORM INPUT

In VS FORTRAN, you can also use free-form input— which gets rid of many restrictions imposed by fixed-format input.

Free-form input is particularly useful if you're coding your programs at a terminal; there's no artifical dependence on 80-character card image format. Although all your source program records are 80 characters in length, you can break each statement (and line of your program) at any convenient logical point.

Reference documentation for VS FORTRAN free-form input is given in the VS FORTRAN Application Programming: Language Reference manual.

END OF IBM EXTENSION

DEFINING DATA

When you're writing a VS FORTRAN program, you must define all the data you use—its type and its organization. Your definitions can depend upon predetermined definitions or they can be explicit.

The data type can be integer, real, complex, logical, or character. In your source program, you code them as described in the following paragraphs.

Integer items are made up of whole numbers. They can be signed or unsigned.

Real items are numbers that must contain either a decimal point or an exponent. They can be fractional; they can be signed or unsigned.

Complex items are represented by a pair of items (either of which can be real or integer) written within parentheses and separated by a comma. The first item is the real part of the complex number; the second item is the imaginary part. Either item can be signed or unsigned. A complex item is converted to a pair of real items of the appropriate length.

You use integer, real, and complex items in mathematical or relational expressions.

Logical items have a value of either "true" or "false." You use logical items only in logical expressions.

Character items can be made up of any characters in the computer's character set. You use character items in character and relational expressions.

Figure 7 shows the lengths valid for each data type.

Reference documentation for these data types is given in the VS FORTRAN Application Programming: Language Reference manual.

Data Type	**Valid Storage Lengths**	**Default Length**
Integer	$\boxed{\text{2 or}}$ 4	4
Real	4, 8 $\boxed{\text{or 16}}$	4
Complex	8, 16 $\boxed{\text{or 32}}$	8
Character	1 through 500	1
Logical	$\boxed{\text{1 or}}$ 4	4

Figure 7. Data Types and Valid Lengths

You define the data type of an item either through implicit naming conventions or through explicit definitions.

PREDETERMINED DATA TYPE DEFINITION

In VS FORTRAN, if you don't otherwise define a named item, it's given a data type, depending on the initial letter of its name:

Items whose names begin with I through N are integer items of length 4.

Items whose names begin with any other letter are real items of length 4.

IBM EXTENSION

Items whose names begin with the currency symbol ($) are real items of length 4.

END OF IBM EXTENSION

No other data types have a predetermined definition.

EXPLICIT DATA TYPE DEFINITION

There are two ways you can define data items explicitly—using the IMPLICIT statement or using explicit type statements.

Typing Groups of Data Items—IMPLICIT Statement

Using the IMPLICIT statement, you can explicitly specify the data types for items whose names begin with specific letters. For example, if you specify:

```
     IMPLICIT DOUBLE PRECISION (A-C, F),
              LOGICAL (E,L),CHARACTER(D,G,H)
```

your program will treat data items as shown below.

Names Beginning with	Have Data Type	Have Length
A through C, and F	DOUBLE PRECISION	8
E and L	LOGICAL	4
D, G, and H	CHARACTER	1
I through K, M, N	INTEGER	4 (default)
O through Z and $	REAL	4 (default)

IBM EXTENSION

If you specify an IMPLICIT statement with the following initial letters:

```
     (Y - B)
```

The compiler performs a "wraparound" scan to find the beginning initial (Y), and the ending initial (B)—which is lower in the FORTRAN collating sequence than Y. That is, you are implicitly typing all items with names beginning with Y, Z, $, A, and B. You'll get a warning message when this situation occurs; however, your program will compile and execute.

END OF IBM EXTENSION

Reference documentation for the IMPLICIT statement is given in the VS FORTRAN Application Programming: Language Reference manual.

Typing for Specific Data Items—Explicit Type Statements

Explicit type statements define the data type for specific data items that you name in your program. For such items you can specify the data type and the length, and, optionally, initial values for data items and dimension information for arrays.

For example, you can specify:

```
DOUBLE PRECISION   MEDNUM
CHARACTER *80      INREC
```

IBM EXTENSION

```
INTEGER *2         COUNTR
REAL*16            BIGNUM, ARRAY2*4(5,5)
```

As an alternative for MEDNUM, you can specify:

```
REAL *8            MEDNUM
```

END OF IBM EXTENSION

These statements specify that:

IBM EXTENSION

COUNTR is an integer item of length 2.

BIGNUM is a real item of length 16.

END OF IBM EXTENSION

ARRAY2 is a two-dimensional array, with elements of length 4 (specified by *4). There are five elements in each dimension (specified by (5,5)). (Arrays are explained in "Arrays and Subscripts".)

INREC is a character item of length 80.

MEDNUM is a real item of length 8.

If you specify the preceding IMPLICIT statement in the same program as these explicit type statements, the explicit type statements override the IMPLICIT statement specifications, and in your program:

INREC is a character item of length 80.

MEDNUM is a real item of length 8.

All other items with names beginning with I or M are integer items of length 4.

IBM EXTENSION

BIGNUM is a real item of length 16.

ARRAY2 is a two-dimensional array, with elements of length 4 (specified by *4). There are five elements in each dimension (specified by (5,5)).

COUNTR is an integer item of length 2.

END OF IBM EXTENSION

All other data items whose names begin with B or C are real items of length 8.

Reference documentation for explicit type statements is given in the VS FORTRAN Application Programming: Language Reference manual.

DATA CLASSIFICATIONS

All the data you use in your program—whether it's data you've retrieved from an external device or data your program develops internally—must be constants, or be contained in variables or arrays. Data in any of these classifications can be any of the data types previously described. Data classifications are discussed in the following sections.

VARIABLES

A variable is a named unit of data, occupying a storage area. The value of a variable can change during program execution.

The name of a variable can determine its data type, as described in "Predetermined Data Type Definition," or you can define its data type explicitly, as described in "Explicit Data Type Definition."

The value contained in a variable is always the current value stored there. Before you've assigned a value to a variable, its contents are undefined. You can initialize values using the DATA statement; alternatively, your first executable statement referring to it—for example, a READ statement or an assignment statement—can assign a value to it.

Reference documentation for variables is given in the VS FORTRAN Application Programming: Language Reference manual.

CONSTANTS

A constant is a data item in a program that has a fixed, unvarying value. You can refer to a constant by its value, or you can name the constant and use the name in all program references.

The constants you can use are:

Arithmetic (integer, real, or complex)—use arithmetic constants for arithmetic operations, and to initialize integer, real, or complex variables, and as arguments for subroutines, and so forth.

Logical—use logical constants in logical expressions, and to initialize logical variables, and as arguments for subroutines, and so forth.

Character—use character constants in character and relational expressions, and to initialize character variables, and as arguments for subroutines, and so forth.

Hollerith—use Hollerith constants to initialize data items in a FORMAT statement.

IBM EXTENSION

Literal (old FORTRAN only)—similar in usage to character constants.

Hexadecimal—use hexadecimal constants to initialize items.

END OF IBM EXTENSION

Defining Constants by Value

You can use constants in your program by simply specifying their values. For example:

```
    CIRC =2*PI*RAD
```

or

```
CIRC =2.0*PI*RAD
```

where the value 2 represents an integer constant of that value, and where the value 2.0 represents a real constant of that value.

You can specify all types of constants in this way:

Arithmetic Constants—integer, real, or complex.

- **Integer Constant**—written as an optional sign followed by a string of digits. For example:

```
-12345
 12345
```

- **Real Constant**—can take three forms:

  1. Basic Real Constant—written as an optional sign, followed by an integer part (made up of digits), followed by a decimal point, followed by a fraction part (made up of digits). Either the integer or fraction part can be omitted. For example:

```
+123.45
0.12345
```

  2. Integer Constant With Real Exponent—written as an integer constant followed by a real exponent in the form of a letter, followed by a 1- or 2-digit integer constant. Optionally, the exponent can be signed. For example:

+12345E+2	E exponent (which occupies four storage positions and has the value +1,234,500; the precision is approximately 7.2 decimal digits)
-12300D-03	D exponent (which occupies eight storage positions and has the value -12.3; the precision is approximately 16.8 decimal digits)

IBM EXTENSION

12345Q03	Q exponent (which occupies 16 storage positions and has the value +12,345,000; the precision is approximately 35 decimal digits)

END OF IBM EXTENSION

  3. Basic Real Constant With Real Exponent—written as a basic real constant followed by a real exponent; the real exponent is written as one of the letters D, E, or Q, followed by a 1- or 2-digit integer constant. Optionally, the exponent can be signed. For example:

0.12345E+2	E exponent (which occupies four storage positions and has the value +12.345; the precision is approximately 7.2 decimal digits)
0.12345D-03	D exponent (which occupies eight storage positions and has the value

	+0.00012345; the precision is approximately 16.8 decimal digits)

IBM EXTENSION

-1234.5Q03	Q exponent (which occupies 16 storage positions and has the value -1,234,500; the precision is approximately 35 decimal digits)

END OF IBM EXTENSION

- **Complex Constant**—written as a left parenthesis, followed by a pair of integer constants or real constants separated by a comma, followed by a right parenthesis.

  The first integer or real constant represents the real part of the complex number; the second integer or real constant represents the imaginary part of the complex number. The real and imaginary parts need not be of the same size; the smaller part is made the same size as the larger part. For example:

(123.45,-123.45E2)	(has the value +123.45, -12345i; both the real and imaginary parts have a length of 4)

IBM EXTENSION

(123.45,-123.45D2)	(has the value +123.45, -12345i; the real part has a length of 4, the imaginary part a length of 8)

(The real part (a real constant) is converted to a real constant of length 8.)

(12345,-123.45Q2)	(has the value +12345, -12345i; the real part has a length of 4, the imaginary part a length of 16)

(The real part (an integer constant) is converted to a real constant of length 16.)

END OF IBM EXTENSION

Note: In these examples, the character _i_ has the value of the square root of -1.

Logical Constant—written as .TRUE. or .FALSE. in expressions. (In input/output statements you can use T or F as abbreviations.)

IBM EXTENSION

(In the DATA initialization statement, you can also use T or F as abbreviations.)

END OF IBM EXTENSION

For a logical item named COMP, you can specify, for example:

```
LOGICAL COMP
COMP=.FALSE.
```

This sets the logical item COMP to the value "false."

Character Constant—written as an apostrophe, followed by a string of characters, followed by an apostrophe. The character string can contain any characters in the computer's character set. For example:

```
'PARAMETER = '

'THE ANSWER IS:'

'''TWAS BRILLIG AND THE SLITHY TOVES'
```

Note: If you want to include an apostrophe within the character constant, you code two adjacent apostrophes, as shown in the last example, which is displayed as:

```
'TWAS BRILLIG AND THE SLITHY TOVES
```

Hollerith Constant—valid only in a FORMAT statement. It is written as an integer constant followed by the letter H, followed by a string of characters. The character string can contain any characters in the computer's character set. For example:

```
FORMAT(I3,11H = THE NORM)

FORMAT(2D8.6, 18H ARE THE 2 ANSWERS)
```

IBM EXTENSION

In old FORTRAN, the literal constant performs functions similar to the current FORTRAN character constant, and the current Hollerith constant. (Reference documentation is given in the IBM System/360 and System/370 FORTRAN IV Language manual.)

Hexadecimal Constant—written as the character Z, followed by a hexadecimal number, made up of the digits 0 through 9 and the letters A through F. You write a hexadecimal constant as 2 hexadecimal digits for each byte.

You can use hexadecimal constants only in a DATA statement to initialize data items of all types except character.

```
REAL *4 TEMP
DATA TEMP/ZC1C2C3C4/
```

END OF IBM EXTENSION

Reference documentation for program constants is given in the VS FORTRAN Application Programming: Language Reference manual.

Defining Constants by Name—PARAMETER Statement

If your program uses one constant frequently, you can use the PARAMETER statement to name the constant and assign it a value. You can do this once, before your program first uses the constant, and then refer to the constant, wherever it's used, by its name.

There are two advantages in handling constants this way:

- The name for the constant can be a meaningful name—which makes the logic of the program easier to understand when the time comes for maintenance updates.
- If for some reason the value of the constant must be changed, you can change it once, in the PARAMETER statement, and all references throughout the program are updated.

You use the PARAMETER statement to assign names and values to constants. For example:

```
    CHARACTER *5 C1,C2
    PARAMETER (C1='DATE ',C2='TIME ',RATE=2*1.414)
```

The CHARACTER explicit type statement defines items C1 and C2 as character items of length 5. The PARAMETER statement then defines these items as named program constants:

C1 has the value "DATE ". The constant is five characters long; the blank following the word DATE is part of the constant.

C2 has the value "TIME ". The constant is five characters long; the blank following the word TIME is part of the constant.

RATE is defined implicitly as a REAL *4 item. Therefore, it's a real constant, four storage positions long, with a value of 2 times 1.414 or 2.828.

You'll note that RATE is defined through the expression 2*1.414; when you define a constant using an expression in this way, the expression you specify must be a constant expression.

In the PARAMETER statement, the value you assign to the constant must be consistent with its data type; that is, C1 and C2 must contain character data, and RATE must contain real data. If any data conversions must be performed, they are made according to the rules for the assignment statement. (See "Assigning Values to Data—Assignment Statement.")

The value you assign through a PARAMETER statement to a character constant must contain no more than 255 characters.

Reference documentation for the PARAMETER statement is given in the VS FORTRAN Application Programming: Language Reference manual.

ARRAYS AND SUBSCRIPTS

A VS FORTRAN array is a set of consecutive data items, each of which is the same data type and length as the other items in the set.

In FORTRAN, you can give a name to the entire array and then refer to each of the individual items. The individual items are called array elements, and you can refer to any individual element by specifying its position within the array through one or more subscripts, depending upon the number of dimensions in the array.

You can define an array by using the DIMENSION statement, the explicit type statement, or the COMMON statement.

One-Dimensional Arrays

To define a one-dimensional array, you specify only one dimension declarator. For example, you want to define a one-dimensional array, named ARRAY1, that contains five array elements. You can do so through the following DIMENSION declaration:

```
      DIMENSION ARRAY1(5)
```

In this case, you've defined ARRAY1 as an array containing five elements, each implicitly defined as a REAL item of length 4.

SUBSCRIPT REFERENCES: Program references to ARRAY1 take the form of subscripts:

- As integer constants:

  ```
  ARRAY1(2)
  ```

  In this example, the subscript specifies a reference to the second array element.

- As integer variables:

  ```
  ARRAY1(NUM)
  ```

  where (NUM) represents the integer variable subscript, into which you can place the values 1 through 5. (For ARRAY1, any other values produce invalid array references.)

IBM EXTENSION

You can also specify subscript references as real constants, variables, or expressions; the compiler converts the real value to an integer value.

END OF IBM EXTENSION

Reference documentation for the DIMENSION statement and for subscript references is given in the VS FORTRAN Application Programming: Language Reference manual.

Multidimensional Arrays

In VS FORTRAN, arrays can have up to seven dimensions; that is, you may need to specify up to seven dimension declarators to define the array, and up to seven subscripts to identify a specific array element. (The number of subscripts you specify must always equal the number of dimensions in the array.)

Multidimensional arrays are stored in column-major order. That is, the first subscript always varies most rapidly, and the last subscript always varies least rapidly.

For example, if you define the 3-dimensional array ARR3(2,2,2), it's placed in storage in the order shown in Figure 8. In this example, the lower bounds of the subscripts are 1; therefore, the first array element is (1,1,1); you'd refer to the second array element as ARR3(2,1,1), and you'd refer to the seventh array element as ARR3(1,2,2).

Arrays—Implicit Lower Bounds

In the preceding examples, the subscripts are shown as having a range from 1 through the upper bound for each dimension of the array; that is, in ARRAY3 the implicit lower bound for each dimension is 1, and the explicit upper bound for each dimension is 2.

Arrays—Explicit Lower Bounds

In VS FORTRAN, you can also explicitly state both the lower and upper bounds for any array. For example, for ARR3A you could specify:

```
DIMENSION ARR3A(4:5,2:3,1:2)
```

The layout in storage—as shown in Figure 8—is exactly the same as for ARR3; however, valid array references would range from ARR3A(4,2,1) through ARR3A(5,3,2).

```
ARR3—Implicit Lower Bounds
-----------------------------------------------------------------------
| 1,1,1 | 2,1,1 | 1,2,1 | 2,2,1 | 1,1,2 | 2,1,2 | 1,2,2 | 2,2,2 |
-----------------------------------------------------------------------

ARR3A—Explicit Lower Bounds
-----------------------------------------------------------------------
| 4,2,1 | 5,2,1 | 4,3,1 | 5,3,1 | 4,2,2 | 5,2,2 | 4,3,2 | 5,3,2 |
-----------------------------------------------------------------------
```

Figure 8. Three-Dimensional Array—Implicit and Explicit Lower Bounds

Arrays—Signed Subscripts

In VS FORTRAN, your array declaration can specify positive or negative signed declarators for either the lower or the upper bounds. This can make a difference in the number of array elements the array contains.

For example, if you define ARR2 and ARR2S as follows:

```
DIMENSION ARR2(4,2), ARR2S (-2:2,2)
```

the two arrays are laid out in storage as shown in Figure 9:

Valid array references for ARR2 range from ARR2(1,1) through ARR2(4,2), and there are eight array elements.

Valid array references for ARR2S range from ARR2S(-2,1) through ARR2S(2,2) (with ARR2S(0,1) and ARR2S(0,2) included), and there are ten array elements.

Because a zero subscript is valid for ARR2S, there are two more array elements in ARR2S than in ARR2.

```
ARR2(4,2)—is arranged in storage like this:
-----------------------------------------------------
| 1,1 |  2,1 | 3,1 | 4,1 | 1,2 |  2,2 |  3,2 | 4,2 |
-----------------------------------------------------

ARR2S(-2:2,2)—is arranged in storage like this:
------------------------------------------------------------------
| -2,1 | -1,1 | 0,1 | 1,1 | 2,1 | -2,2 | -1,2 | 0,2 | 1,2 | 2,2 |
------------------------------------------------------------------
```

Figure 9. Arrays—Effect of Negative Lower Bounds

Arrays—Programming Considerations

Wherever possible, you should specify arrays as one-dimensional rather than as multidimensional. The fewer the dimensions in an array, the faster your array references execute.

Always make sure that subscript values refer to elements within the bounds of the array; if they don't, you can possibly destroy data or instructions.

SUBSTRINGS OF CHARACTER ITEMS

For character arrays and character variables, you can make substring references (that is, references to only a portion of the item) using substring notation.

You specify substring references by naming the array element or variable and then adding the substring reference—a left parenthesis, the lower bound, a colon, the upper bound, and a right parenthesis, in that order.

For example:

VAR1(2:4) means that the substring consists of the second through fourth characters in the character variable VAR1;

ARR1(2)(1:4) specifies that the substring consists of the first through fourth characters in the second array element of the character array ARR1.

You can omit the lower bound of the substring reference if it is equal to 1; that is, ARR(2)(:4) is exactly equivalent to ARR(2)(1:4).

You can use character substrings in program references and in assignment statements. For example, if you define a variable and an array as follows:

```
CHARACTER*10 SUVAR,SUARR(3)
SUVAR='ABCDEFGHIJ'
```

and you specify the following assignment:

```
SUARR(2)(:5)=SUVAR(6:10)
```

then, when the assignment statement is executed, the last five characters of SUVAR (that is, FGHIJ) are placed in the first five characters of the second array element of SUARR; the last five characters of that array element are unchanged.

Reference documentation for character substrings is given in the <u>VS FORTRAN Application Programming: Language Reference</u> manual.

USING DATA EFFICIENTLY

How efficiently your program uses the system depends, in part, on how you define and use the data in your program. The choices you make also depend upon the results you want to achieve.

This section discusses how to initialize data and how to reuse storage for different data items in the same program.

INITIALIZING DATA—DATA STATEMENT

You can use the DATA statement to initialize variables and arrays. You must place it after any specification statement or IMPLICIT statement that refers to the items you're initializing. For example, your program could contain the following statements:

```
CHARACTER *4 CARL,CELS*2
DATA DEG,CELS,CARL/10.2,'DG','SURD'/,AVCH/.1515/
```

and the data items would be initialized to the following values:

```
DEG    (real constant)         initialized to    10.2
CELS   (character constant)    initialized to    DG
CARL   (character constant)    initialized to    SURD
AVCH   (real constant)         initialized to   .1515
```

You can also use named constants to initialize data items:

```
PARAMETER (DEGI=10.2)
DATA DEG/DEGI/
```

which initializes the real variable DEG to the value 10.2.

Reference documentation for the DATA statement is given in the VS FORTRAN Application Programming: Language Reference manual.

Initializing Arrays—DATA Statement

There are special considerations when you initialize arrays with the DATA statement, as follows:

INITIALIZING ARRAY ELEMENTS: You can initialize any element of an array by subscripting the array name. Only one element is initialized. The following example shows how to initialize individual array elements:

```
DIMENSION A(10)
DATA A(1),A(2),A(4),A(5)/1.0,2.0,4.0,5.0/
```

the array elements are initialized as follows:

```
A(1)     initialized to     1.0
A(2)     initialized to     2.0
A(3)     is not initialized
A(4)     initialized to     4.0
A(5)     initialized to     5.0
A(6) through A(10) are not initialized.
```

INITIALIZING CHARACTER ARRAY ELEMENTS: In a character array, it isn't necessary to specify the constant as the same length as the character array element:

If the character constant is shorter than the character array element, the array element is padded at the right with blanks.

If the character constant is longer than the character array element, the constant is truncated at the right.

For example, if you specify the following statements:

```
      CHARACTER *4 CARRAY(4)
      DATA CARRAY(1),CARRAY(4)/'ABC','EFGHI'/
```

the CARRAY array is initialized as follows:

```
      CARRAY(1)   initialized to   ABC   (fourth character is blank)
      CARRAY(2) and CARRAY(3) are not initialized
      CARRAY(4)   initialized to   EFGH  (I is truncated)
```

INITIALIZING ARRAYS—IMPLIED DO LISTS: You can use implied DO lists to initialize parts or all of an array. You use the implied DO list to specify the values the subscripts should assume.

Initializing an Entire Array—Implied DO List: You can initialize an entire array to the value 0.0, as follows:

```
      DIMENSION ARRAYE(10,10)
      DATA ((ARRAYE(I,J),I=1,10),J=1,10)/100*0.0/
```

This DATA statement tells the compiler to:

1. Vary the subscript I from 1 to 10 each time the subscript J is incremented, and vary the subscript J from 1 to 10; the implied increment for both I and J is 1.

2. Place 100 repetitions of the value 0.0 in the 100 array elements; the repetition factor is specified by the 100*.

Initializing an Identity Matrix—Implied DO Lists: You're allowed to nest implied DO lists in a DATA statement. In this way you can initialize an identity matrix, using one DATA statement:

```
      DIMENSION ARRAYI(10,10)
      DATA ((ARRAYI(I,J),I=1,J-1),J=2,10)/45*0.0/,
           ((ARRAYI(I,J),J=1,I-1),I=2,10)/45*0.0/,
           (ARRAYI(I,I),I=1,10)/10*1.0/
```

This DATA statement tells the compiler to:

1. Vary the subscript I from 1 to 1 less than the value of J, each time the subscript J is incremented; subscript J is incremented from 2 to 10. This fills the upper right 45 array elements with 0.0.

2. Vary the subscript J from 1 to one less than the value of I each time the subscript I is incremented; subscript I is incremented from 2 to 10. This fills the lower left 45 array elements with 0.0.

3. Use the value of I for both subscripts (I,I), and vary I from 1 to 10. This fills the principal diagonal with the value 1.0.

You can also use the DATA statement to initialize the entire array to zeros, and then specify a DO statement and an assignment statement to initialize the principal diagonal. For example:

```
      DIMENSION ARRAYD(10,10)
      DATA ((ARRAYD(I,J),I=1,10),J=1,10) /100*0.0/
      DO 30 I=1,10,1
      ARRAYD(I,I)=1.0
 30   CONTINUE
```

MANAGING DATA STORAGE—EQUIVALENCE STATEMENT

You can control storage allocation within your program by using the EQUIVALENCE statement.

When your program's logic permits it, you can use this statement to specify that one storage area is to be shared by two or more data items. These items can be variables or arrays, and they can be of the same or of differing data types. There's one restriction: CHARACTER items cannot be equivalenced with other data types.

Note that only the storage itself is equivalent (shared); mathematical equivalence is implied only when the sharing items are of the same type, when they share exactly the same storage, and when the value assigned to the shared area is of that type.

The EQUIVALENCE statement is particularly useful when you use it with the COMMON statement; this kind of usage is described in "EQUIVALENCE Considerations—COMMON Statement" in Part 3.

When you use the EQUIVALENCE statement with array elements, you can implicitly specify the storage sharing of other array elements within the same array; this is because arrays are stored in a predetermined order. For example, if you write an EQUIVALENCE statement referring to ARR3 (illustrated in Figure 8), as follows:

```
      DIMENSION ARR3(2,2,2)
      INTEGER *4 ARR3B(4)
      EQUIVALENCE (ARR3(2,2,1),ARR3B(1))
```

then the array elements of ARR3 and ARR3B share storage as shown in Figure 10, with the displacement for the array elements shown in the right-hand column.

```
ARR3 Storage         ARR3B Storage       Displacement

-------------        -------------       0
 ARR3(1,1,1)
-------------        -------------       4
 ARR3(2,1,1)
-------------        -------------       8
 ARR3(1,2,1)
-------------        -------------       12
 ARR3(2,2,1)           ARR3B(1)
-------------        -------------       16
 ARR3(1,1,2)           ARR3B(2)
-------------        -------------       20
 ARR3(2,1,2)           ARR3B(3)
-------------        -------------       24
 ARR3(1,2,2)           ARR3B(4)
-------------        -------------       28
 ARR3(2,2,2)
-------------        -------------       32
```

Figure 10. Sharing Storage Between Arrays—EQUIVALENCE Statement

Reference documentation for the EQUIVALENCE statement is given in the VS FORTRAN Application Programming: Language Reference manual.

Execution-Time Efficiency Using Equivalence

When you make items equivalent, you can specify them in any order in your EQUIVALENCE statement. However, unless you ensure that all the arithmetic items in the group have the proper boundary alignment, you can lose object-time efficiency.

When all the items in an equivalence group are defined as beginning at the same storage address, there is no problem.

There can be problems, however, when some items begin at a displacement from the beginning address. In this case, you can ensure proper boundary alignment by defining each equivalent arithmetic item as starting at a displacement from the beginning of the group that can be evenly divided by its length, or, if it's COMPLEX, half its length. The two examples in Figure 11 illustrate this concept.

DEFINING AND USING EXPRESSIONS

Expressions are combinations of data items and operators that represent a value. You can use them for arithmetic operations, character operations, logical operations, or in relational operations.

The simplest form of an expression is simply the name of a data item, or the value or named value of a constant. You can specify more complicated expressions by using operators to combine data items. The kind of operators you can use depends upon the type of expression you're specifying:

Arithmetic operators—for arithmetic expressions

Character operators—for character expressions

Relational operators—for relational expressions

Logical operators—for logical expressions

The precedence of one type of operator over another is in the order given above.

ARITHMETIC EXPRESSIONS

You can use arithmetic expressions to specify mathematical relationships of various kinds. The valid arithmetic expressions, and how you can combine them, are shown in Figure 12.

Specify all desired computations explicitly, and make certain that no two arithmetic operators appear consecutively.

In addition, be aware that the compiler evaluates arithmetic expressions as follows:

- Operands are evaluated in their order of precedence from highest to lowest.

- Within the exponentiation precedence level, operations are performed right to left. For example, A**B**C is evaluated as (A**(B**C)).

- Within all other precedence levels, operations are performed left to right. For example, A+B+C is evaluated as ((A+B)+C).

- Parentheses are evaluated as they are in mathematics; they specify the order in which operations are to be performed. That is, expressions within parentheses are evaluated before the result is used. For example, the expression ((A-B)+C)*E is evaluated as follows:

  1. A-B is evaluated, giving result1

  2. result1+C is evaluated, giving result2

  3. result2*E is evaluated, giving the final result

- You can use the + and - operators as signs for an item; when you use them this way, they're evaluated as if they were addition or subtraction operators. That is, A=-B+C is evaluated as though written A=-(B)+C.

The following EQUIVALENCE statement (where A is REAL*4, I is INTEGER*4, and A2 is DOUBLE PRECISION):

```
      DIMENSION A(10),I(16),A2(5)
      EQUIVALENCE (A(1),I(7),A2(1))
```

causes the items to be laid out in storage as follows:

```
Displacement:
0   4   8   12  16  20  24  28  32  36  40  44  48  52  56  60  64
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
-----------------------------------------------------------------

I array                  I(7)
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

                         A(1)
A array                 |---|---|---|---|---|---|---|---|---|---|

                         A2(1)
A2 array                |-------|-------|-------|-------|-------|
```

(Efficiently laid out—A and A2 begin at a displacement of 24 storage positions from the beginning of the equivalence group)

The following EQUIVALENCE statement (using the same items):

```
      DIMENSION A(10),I(16),A2(5)
      EQUIVALENCE (A(1),I(6),A2(1))
```

causes the items to be laid out in storage as follows:

```
Displacement:
0   4   8   12  16  20  24  28  32  36  40  44  48  52  56  60  64
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
-----------------------------------------------------------------

I array              I(6)
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

                     A(1)
A array             |---|---|---|---|---|---|---|---|---|---|

                     A2(1)
A2 array            |-------|-------|-------|-------|-------|
```

(Inefficiently laid out—A2 begins at a displacement of 20—not divisible by 8—from the beginning of the equivalence group)

Figure 11. Storage Efficiency and the EQUIVALENCE Statement

Operation	Arithmetic Operator	Precedence
Function Evaluation	(none)	Highest
Exponentiation	**	Second Highest
Multiplication	*	Third Highest
Division	/	
Addition	+	Lowest
Subtraction	-	

Figure 12. Arithmetic Operators and Operations

In arithmetic expressions, you can specify the operands as any mix of integer, real, or complex items.

However, you should be careful, when you're specifying arithmetic expressions, that you define the operands so that you get the precision you want in the result. For example, if you specify:

```
      DOUBLE PRECISION RESULT
      RESULT = AR3*AR1
```

AR3 and AR1 (both REAL *4 numbers) are multiplied together, the REAL *4 result is padded with zeros and placed in RESULT. Thus, while the RESULT is the length of a DOUBLE PRECISION item, it has only the precision obtainable using REAL*4 operands (approximately 7.2 decimal digits).

If you're dividing one operand by another, you should define the operands and the result as REAL items.

If you divide one integer by another, any remainder is ignored, and you get an integer quotient:

```
      DATA I1/10/,I2/15/
      RESULT=I2/I1
```

In this case, the expression on the right of the equal sign is evaluated to the integer 1; then the result is converted to a floating point number and stored in RESULT.

You can also perform more complex mathematical operations by using the intrinsic functions that VS FORTRAN provides; see "Saving Coding Effort with Statement Functions" for details.

Reference documentation for arithmetic expressions is given in the VS FORTRAN Application Programming: Language Reference manual.

CHARACTER EXPRESSIONS

You specify character expressions by combinations of character items, the character concatenation operator, and optional parentheses.

The simplest form of character expression is simply a character item itself.

You can combine character operands by using the concatenation operator (//) to join one operand to the next. For example:

```
      CHARACTER *12 CHAR
      CHARACTER *6 CHAR1,CHAR2
      DATA CHAR1/'ABCDEF'/,CHAR2/'GHIJKL'/
      CHAR = CHAR1//CHAR2
```

The concatenation operator (//) specifies that the contents of CHAR2 are to be joined to those of CHAR1 to form the character string:

```
      ABCDEFGHIJKL
```

which is then assigned to the character variable CHAR.

Reference documentation for character expressions is given in the VS FORTRAN Application Programming: Language Reference manual.

RELATIONAL EXPRESSIONS

You can form relational expressions by combining two arithmetic expressions with a relational operator, or by combining two character expressions with a relational operator.

The relational operators you can use are listed in Figure 13.

Relational Operator	Meaning
.GT.	greater than
.LT.	less than
.GE.	greater than or equal to
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

Figure 13. Relational Operators and their Meanings

You can combine expressions as shown in the following examples:

```
      A.GE.B
```

If A is greater than or equal to B, this relational expression is evaluated as "true"; otherwise, it is evaluated as "false."

```
      (A+B).LT.(C-B)
```

If the result of A+B is less than the result of C-B, this relational expression is evaluated as "true"; otherwise, it is evaluated as "false."

```
      COMPLEX CMPLX1,CMPLX2
      (CMPLX1-2).EQ.(CMPLX2+2)
```

If the result of CMPLX1-1 is equal to the result of CMPLX2+2, this relational expression is evaluated as "true"; otherwise, it is evaluated as "false." (The only relational operators you can use with complex arithmetic operands are .EQ. and .NE.)

```
      CHARACTER *5 CHAR4, CHAR5, CHAR6*8
      (CHAR4//CHAR5).GT.CHAR6
```

In this expression, CHAR6 is extended 2 characters to the right with blank characters; CHAR6 is then compared with the concatenation of CHAR4 and CHAR5, according to the EBCDIC collating sequence. If the concatenation of CHAR4 and CHAR5 evaluates as greater than CHAR6, this relational expression is evaluated as "true"; otherwise, it is evaluated as "false."

Relational Expressions—Character Operands

When you use a relational expression to compare character operands, the comparison is made using the EBCDIC collating sequence.

For example, if character items C1 (containing '3AB') and C2 (containing 'XYZ') are compared, as follows:

```
      L = C1.GT.C2
```

C1.GT.C2 evaluates as "true."

However, if you use the intrinsic functions (LLT, LGT, LLE, and LGE) to compare character operands, the comparison is made using the ASCII collating sequence. See "Comparing Character Operands—FORTRAN-Supplied Functions" in Part 3 for additional information.

Reference documentation for relational expressions and for the EBCDIC and ASCII collating sequences is given in the VS FORTRAN Application Programming: Language Reference manual.

LOGICAL EXPRESSIONS

You use logical expressions to combine logical operands—a logical constant, logical variable or array element, logical function references, and logical or relational expressions—optionally enclosed in parentheses.

The logical operators you can use are shown in Figure 14.

Logical Operator	Meaning	Precedence
.NOT.	Logical negation	Highest
.AND.	If both operands are "true," the expression is "true"; otherwise the expression is "false"	
.OR.	If either operand is "true," the expression is "true"; otherwise the expression is "false"	
.EQV.	If both operands are "true" or if both operands are "false," the expression is "true"; otherwise the expression is "false"	Lowest
.NEQV.	If both operands are "true" or if both operands are "false," the expression is "false"; otherwise the expression is "true"	Lowest

Figure 14. Logical Operators and their Meanings

The following examples show some of the ways you can use logical expressions:

1. A.GT.B.OR.A.EQ.C

   This logical expression is "true" if <u>one</u> of the following is "true":

   A is greater than B, or

   A is equal to C

   otherwise, it is "false."

2. A.GT.B.AND.A.EQ.C

   This logical expression is "true" only if <u>both</u> the following are "true":

   A is greater than B, and also

   A is equal to C

   otherwise, it is "false."

3. A.GT.B.AND..NOT.A.EQ.C

   This logical expression is "true" only if both the following are "true":

   A is greater than B, and also

   A is not equal to C

   otherwise, it is "false."

4. A.GT.B.OR.A.EQ.C.AND.B.LT.D

   This logical expression is evaluated in the following order:

   a. A.GT.B is evaluated, giving a truth value <u>v</u>.

   b. A.EQ.C is evaluated, giving a truth value <u>w</u>.

   c. B.LT.D is evaluated, giving a truth value <u>x</u>.

   d. <u>w</u>.AND.<u>x</u> is evaluated, giving a truth value <u>y</u>.

   e. <u>v</u>.OR.<u>y</u> is evaluated, giving the final truth value <u>z</u>.

   The expression is "true" if either <u>v</u> or <u>y</u> evaluates as "true."

Reference documentation for logical expressions is given in the <u>VS FORTRAN Application Programming: Language Reference</u> manual.

<u>ASSIGNING VALUES TO DATA—ASSIGNMENT STATEMENT</u>

In a VS FORTRAN program, the assignment statement lets you assign values to data.

The assignment statement closely resembles a conventional algebraic equation, except that the value to the right of the equal sign replaces (is assigned to) the value to the left.

You can use the assignment statement to assign one constant, variable, or array element to another variable or array element.

You can specify arithmetic, character, and logical operands and expressions to the right of the equal sign.

ARITHMETIC ASSIGNMENTS

You can assign arithmetic operands and expressions to other arithmetic operands; the item(s) to the right of the equal sign don't have to be the same type or length as the item to the left of the equal sign.

For example (assuming default naming conventions), if you make the following assignments, you'll get the indicated results:

PI=3.14159
: assigns the real constant 3.14159 to the REAL variable of length 4 named PI.

ARRAY3(NUM) = DIFF
: assigns the value currently contained in the REAL variable of length 4 to the REAL array element ARRAY3(NUM) of length 4

INTR = DIFF
: the value of DIFF is converted to an INTEGER value of length 4 (that is, the largest integer in the real item is used, without rounding) and placed in INTR.

DIFF = INTR
: the value of INTR is converted to a REAL value of length 4 and placed in the variable DIFF.

DIFF=INTR+DIFF
: the value of INTR is converted to a REAL value of length 4 and added to the current value of DIFF; the result is the new value of DIFF.

You can use and combine all the arithmetic operators, as shown in Figure 12.

Reference documentation for arithmetic assignments is given in the <u>VS FORTRAN Application Programming: Language Reference</u> manual.

CHARACTER ASSIGNMENTS

You can use character assignments to initialize character items. For example:

```
      CHARACTER*10 SUVAR
      SUVAR='ABCDEFGHIJ'
```

which assigns the value ABCDEFGHIJ to the character variable SUVAR.

When the operands are character items or expressions, you can specify the item to the left and the item(s) to the right of the equal sign with differing lengths. When you execute the assignment, the item to the left is either padded at the right with blanks or the data is truncated to fit into the item at the left:

```
      CHARACTER *5 A,B,C,E *13
      DATA A/'WHICH'/,B/' DOG '/,C/'BITES'/
      E=A//B//C
```

In the assignment statement, the concatenation symbols (//) place the contents of A, B, and C one after another into E.

After the assignment statement is executed, the character variable E contains:

```
      WHICH DOG BIT
```

(Note that the characters ES are truncated.)

You can also define character items on either side of the equal sign as substrings, in which case only the substring portion of the item is acted upon:

```
      A(4:5)=C(3:4)
```

After this assignment statement is executed, A contains the characters "WHITE".

There's one restriction upon character assignment statements: the item to the left of the equal sign, and those to the right must not overlap in any way; that is, they must not refer to the same character positions, completely or in part. If they do, you can get unpredictable results, but you won't get an error message.

Reference documentation for character assignments is given in the VS FORTRAN Application Programming: Language Reference manual.

LOGICAL ASSIGNMENTS

When the operand to the left of the equal sign is a logical item, the operands or expressions to the right must evaluate to a logical value of either "true" or "false."

In a logical assignment, the righthand operands and expressions can be logical items, and logical or relational expressions. Within the relational expressions, you can use arithmetic or character operands.

For example, you can use arithmetic operands, as follows:

```
      LOGICAL *4 LOGOP
      DOUBLE PRECISION AR1/1.1/,AR2/2.2/,AR3/3.3/,AR4
      LOGOP = (AR4.GT.AR1).OR.(AR2.EQ.AR3)
```

"true" is placed in LOGOP when AR4 is greater than AR1; otherwise AR2 and AR3 must be tested. Then if AR2 is equal to AR3, "true" is placed in LOGOP; otherwise "false" is placed in LOGOP.

For example, you can use character operands, as follows:

```
      LOGICAL *4 LOGOP
      CHARACTER *6 CHAR1, CHAR2, CHAR3
      DATA CHAR1/'ABCDEF'/ CHAR2/'GHIJKL'/
      LOGOP = (CHAR2.EQ.CHAR3).AND.(CHAR1.LT.CHAR2)
```

"true" is placed in LOGOP when CHAR2 and CHAR3 are equal and CHAR1 is less than CHAR2; otherwise, "false" is placed in LOGOP. (Unless their values change during execution, CHAR1 always evaluates as less than CHAR2.)

Reference documentation for logical assignments is given in the VS FORTRAN Application Programming: Language Reference manual.

SAVING CODING EFFORT WITH STATEMENT FUNCTIONS

If your program makes the same complex calculation a number of times, you can define a statement function and then, whenever the program must make the given calculation, refer to that function.

For example,

```
      WORK(A,B,C,D,E) = 3.274*A + 7.477*B - C/D + (X+Y+Z)/E
```

defines the statement function WORK, where WORK is the function name and A, B, C, D, and E are the dummy arguments.

The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

All statement function definitions must precede the first executable statement of the program.

The function reference might appear in a statement as follows:

```
      W = WORK(GAS,OIL,TIRES,BRAKES,PLUGS) - V
```

This is equivalent to:

```
      W = 3.274*GAS + 7.477*OIL - TIRES/BRAKES + (X+Y+Z)/PLUGS - V
```

Note the correspondence between the dummy arguments A, B, C, D, and E in the function definition and the actual arguments GAS, OIL, TIRES, BRAKES, and PLUGS in the function reference.

For reference documentation about statement functions, see the VS FORTRAN Application Programming: Language Reference manual.

CONTROLLING PROGRAM FLOW

Unless you explicitly change the flow of control, VS FORTRAN programs execute one statement after another sequentially. In VS FORTRAN, you can alter the sequence of control, using the ASSIGN, DO, GO TO, IF, PAUSE, STOP, and END statements.

In VS FORTRAN, the arithmetic and logical IF statements and the unconditional GO TO statement work in the same way they've done in older IBM FORTRAN implementations.

However, there are new VS FORTRAN programming options available—such as the block IF statement—and changed programming rules for the DO statement, assigned and computed GO TO statements, and the PAUSE, STOP, and END statements. Programming considerations for these new language features are reviewed in the following sections.

USING STRUCTURED PROGRAMMING—BLOCK IF STATEMENT

The VS FORTRAN structured programming statements—the block IF statement and its associated ELSE IF, ELSE, and END IF statements—help you create programs that conform to structured programming rules:

1. Write all code in control structures.

2. Construct each control structure so that it has only one entrance and only one exit.

3. Make each control structure nestable.

4. Control program flow along paths that define the structure itself.

5. Indent the source code to reflect the logic flow between control structures.

These rules make programs simpler and easier to understand; each structure is small and self-contained, and the structure of the program reflects its logic.

The block IF statement—together with its subsidiary ELSE IF, ELSE, and END IF statements—helps you write VS FORTRAN programs that conform to these structured programming rules. They let you nest IF procedures in a simple straightforward way.

Two terms are needed to explain the concepts of the block IF statement, **IF-level** and **IF-block**:

IF-level The number of IF-levels in a program unit is determined by the number of sets of block-IF statements (IF THEN and END IF statements).

The IF-level of a particular statement is determined with the formula:

n1 - n2

where:

n1 is the number of block IF statements from the beginning of the program unit up to and including this statement.

n2 is the number of END IF statements in the program unit up to, but not including, this statement.

Thus, in the following example:

```
10    IF (A .EQ. B) THEN
11          .
            .
            .
20       IF (C+D .EQ. 0) THEN
21          .
            .
29          .
30       END IF
            .
            .
39          .
40    END IF
```

The block IF at number 10 is at IF level 1, and the block IF statement at statement 20 is at IF level 2.

IF-block An IF-block begins with the first statement after the block IF statement (IF THEN), and ends with the statement preceding the next ELSE IF, ELSE, or END IF statement at the same IF-level as the block IF statement. The IF-block includes all the executable statements in between.

Thus, in the preceding example, the IF blocks are:

- Statements 11-39 inclusive.
- Statements 21-29 inclusive.

In Figure 15, the IF blocks for each example are:

Example 1: The IF block ends with the statement preceding END IF.

Example 2: The IF block ends with the statement preceding ELSE.

Example 3: The IF block ends with the statement preceding ELSE IF.

You can code an empty IF-block; that is, you can code an IF-block that has no executable statements in it.

You must not transfer control into an IF-block from outside the IF-block.

There are three forms a block IF statement can take, as shown in Figure 15.

```
Example 1:     IF (expression) THEN
                    (code executed if expression is "true")
               END IF

Example 2:     IF (expression) THEN
                    (code executed if expression is "true")
                  ELSE
                    (code executed if expression is "false")
               END IF

Example 3:     IF (expression1) THEN
                    (code executed if expression1 is "true")
               ELSE IF (expression2) THEN
                    (code executed if expression1 is
                    "false" and expression2 is "true")
                 ELSE
                    (code executed if both expression 1
                    and expression2 are "false")
               END IF
```

Note: In this third form, you can repeat ELSE IF blocks as your program's logic requires; for example:

```
              IF (expression1) THEN
                    (code executed if expression1 is "true")
              ELSE IF (expression2) THEN
                    (code executed if expression2
                    is "true")
              ELSE IF (expression3) THEN
                    (code executed if expression3
                    is "true")
              ELSE IF (expression4) THEN
                    (code executed if expression4
                    is "true")
                 ELSE
                    (code executed if all expressions
                    listed above are "false")
              END IF
```

Figure 15. Block IF Statement—Valid Forms

Any one of these forms can be nested to any depth within any of the others, as the following example shows:

```
      IF (A.EQ.B) THEN
         (code sequence executed if A.EQ.B is "true")
      ELSE IF (A.GT.B) THEN
            (code sequence executed if A.GT.B is "true")
         ELSE
                (code sequence executed if A.EQ.B and A.GT.B
                 are both "false")
            IF (C.LT.D) THEN
                (code sequence executed if C.LT.D is "true")
                ELSE
                (code sequence executed if C.LT.D is "false")
                   DO
                   .
                   .
                   .
90                 CONTINUE
            END IF
      END IF
```

Here's how the statement executes:

1. The first block IF statement has an IF block that includes the range of statements to the ELSE IF statement.

2. The ELSE IF THEN statement has as its ELSE IF block the range to the ELSE statement at the same level (the first ELSE statement).

   After your program executes an ELSE IF block, control is transferred to the END IF statement at the same level.

3. The first ELSE statement is the alternative condition for the ELSE IF THEN statement.

4. The second IF THEN statement is subordinate to the first ELSE statement. This IF block continues to the first END IF statement.

5. The second ELSE statement is the alternative condition for the second IF THEN statement. Its ELSE block is the range of code to the first END IF statement.

6. The CONTINUE statement provides a convenient reference point for the DO-loop code executed in the current nested ELSE block. (The nested DO loop must be completely contained within the current block; in this example, the current nested ELSE block.)

7. The first END IF statement corresponds to the nested block IF statement.

8. The second END IF statement corresponds to the first block IF statement (the one that begins the entire code sequence).

Reference documentation for the block IF statement is given in the <u>VS FORTRAN Application Programming: Language Reference</u> manual.

Using the CONTINUE Statement

In the previous example, there's a CONTINUE statement which provides a convenient ending point for procedures within the current ELSE block. You'll find the CONTINUE statement particularly useful in this way within DO loops.

However, there's one limitation you must observe. When you use the CONTINUE statement within a block IF statement sequence, you must use the CONTINUE <u>only</u> for control transfers <u>within</u> the local code block (in this example, the local ELSE block). If you branch into the block (with GO TO statements, for example), the results are unpredictable, even though you won't get an error message.

PROGRAMMING LOOPS—DO STATEMENT

In VS FORTRAN, you can use the DO statement to execute a range of statements a specific number of times. The ways you can do it are much more flexible than in previous FORTRAN implementations:

- You can nest DO statements; if you nest one DO statement within another, you must include the range of the inner DO statement entirely within the range of the outer DO statement.

  You can use the same terminal statement for both the inner and the outer DO statement ranges. (CONTINUE is handy for this.)

- If you code DO statements within a block IF, ELSE IF, or ELSE block, make sure that the range of the DO statement is completely contained within that block.

- If you code a block IF statement within a DO loop, ensure that the entire range of the block IF statement, including END IF, is within the range of the DO loop. (You can't use the END IF as the terminal statement for the loop.)
- Don't use any of the following as terminal statements:
  - An unconditional or assigned GO TO statement
  - An arithmetic IF, ELSE IF, ELSE, or END IF statement
  - Another DO statement
  - A RETURN, STOP, or END statement
  - An INCLUDE statement
- When you execute a DO statement, the DO loop becomes active. It remains active until one of the following occurs:
  - The loop executes completely.
  - The program executes a RETURN statement within its range.
  - A transfer is made out of its range.
  - Any STOP statement is executed anywhere in the program.
  - Execution is terminated for any other reason.

In VS FORTRAN, you can specify the DO variable as an integer or real variable, and you can specify the initial value, the test value, and the increment as integer or real expressions, positive or negative. These rules give your DO loop processing much more flexibility than before.

For example, if you code the following DO statement:

```
      DO 20 VAR=START,END,INC
              .
              .
              .
20    CONTINUE
```

how the loop executes depends upon the values you place in START, END, and INC.

You can specify them all as positive quantities:

```
      START=1.0
      END=11.0
      INC=2
```

The starting value (START) for VAR is 1.0, and the ending value (END) is 11.0. Each time the loop is executed, VAR is incremented by 2.0 (INC). (Note that because VAR is a real item, that the integer value in INC is converted to a real value.) After the loop has been executed six times, VAR contains the value 13.0, and DO statement processing is completed.

You can specify a decrementing INC value, with a START value higher than the END value:

```
      START=11.0
      END=1.0
      INC=-2
```

Again the loop is executed six times, after which VAR contains the value -1.0.

You can specify values that cause the DO loop not to be executed:

```
START=10.0          The START value is higher
END=1.0             than the END value and INC
INC=2               the increment is positive.
                    After execution, VAR
                    contains the value 10.0.
```

or

```
START=1.0           The START value is lower
END=10.0            than the END value and INC
INC=-1              the increment is negative.
                    After execution, VAR
                    contains the value 1.0.
```

In either case the loop is not executed at all.

Note: Be careful when you're processing DO loops using real values for the starting or ending values, or for the increment; because real numbers are an approximation of integer values, there can be times when the loop is not executed exactly as you expect. In general, numbers that cannot be represented exactly in the computer may give unexpected results.

Reference documentation for the DO statement is given in the VS FORTRAN Application Programming: Language Reference manual.

USING PROGRAM SWITCHES—ASSIGNED GO TO STATEMENT

You can make one GO TO statement transfer control to different statements, depending upon a control variable (which contains statement number to be used). You set the control variable by means of an ASSIGN statement:

```
      ASSIGN 20 TO LVAR
      GO TO LVAR
30    (next executable statement)
```

When this GO TO statement is executed, control is transferred to statement number 20.

You can optionally include a list of statement numbers in the assigned GO TO statement:

```
      ASSIGN 20 TO LVAR
      GO TO LVAR(10, 20, 50, 100)
30    (next executable statement)
```

When this GO TO statement is executed, control is transferred to statement number 20.

When your program executes either of these assigned GO TO statements, LVAR must be assigned a valid number for an executable statement.

Reference documentation for the assigned GO TO statement is given in the VS FORTRAN Application Programming: Language Reference manual.

USING CONDITIONAL TRANSFERS—COMPUTED GO TO STATEMENT

You can transfer control conditionally to one of a number of statements, depending on the value contained in a control item:

```
      INT1=2
      GO TO (10,20,30,50,100) INT1
```

When this statement is executed, the value in INT1 (2) specifies that the second statement number is to be used for the transfer, and control is transferred to statement number 20.

You can use an integer expression as the control item:

```
INT1=20
INT2=18
GO TO (10,20,30,50,100) INT1-INT2
```

When this statement is executed, the expression INT1-INT2 is evaluated, and the resulting value (2) specifies that the second statement number in the list is to be used for the transfer. Control is transferred to statement number 20.

You can also transfer control to the next executable statement, by ensuring that the value in the control item is either less than one or greater than the number of labels listed. That is, in the previous examples, if the value in INT1 or of INT1-INT2 is less than one or greater than five, control is transferred to the next executable statement.

Reference documentation for the computed GO TO statement is given in the VS FORTRAN Application Programming: Language Reference manual.

SUSPENDING PROGRAM EXECUTION—PAUSE AND STOP STATEMENTS

You can use the PAUSE and STOP statements to suspend program execution: the PAUSE statement temporarily, the STOP statement permanently.

Suspending Execution Temporarily—PAUSE Statement

You can use the PAUSE statement to halt program execution, pending operator response:

```
PAUSE 20200
```

or

```
PAUSE 'MOUNT TEMPORARY TAPE. TO RESUME ENTER 9'
```

When the program executes either of these PAUSE statements, the message is displayed at the operator console:

```
01 IFY001A PAUSE 20200
```

or

```
01 IFY001A PAUSE MOUNT TEMPORARY TAPE. TO RESUME ENTER 9
```

The format of the operator's response to the message depends upon the operating system being used.

Stopping Programs Permanently—STOP Statement

When you end execution of your program, you can communicate a message to the system operator through the STOP statement.

The message can be a numeric string of 5 digits or less:

```
STOP 21212
```

where 21212 can have any meaning you want to assign it.

The message can also be a character constant:

```
STOP 'PROGRAM BACGAM EXECUTION COMPLETED'
```

The character constant you specify must contain no more than 72 characters.

When the program executes either of these STOP statements, the message is displayed at the operator console.

You can also use the STOP statement to stop the program permanently without sending a message to the operator:

```
      STOP
```

Reference documentation for the PAUSE and STOP statements is given in the VS FORTRAN Application Programming: Language Reference manual.

ENDING YOUR PROGRAM—END STATEMENT

In VS FORTRAN, the last statement in your program must be an END statement, and (unless your program executes a RETURN or STOP statement first) it must be the last statement executed.

For this reason, you can label the END statement. This lets you ensure that it is executed (if that's what you want), no matter which branch of the program is executed last:

```
110   END
```

(For the END statement in subprograms, see "Coding Calling and Called Programs.")

Reference documentation for the END statement is given in the VS FORTRAN Application Programming: Language Reference manual.

USE PRE-WRITTEN SOURCE CODE

If your program uses frequently-used code sequences—sequences such as blocks of common data items, or often-used error routines or input/output routines—you can write the code sequences once, place them in a system catalog, and then retrieve them for your program with the INCLUDE statement.

For example, to retrieve an error routine you cataloged under the name ERRTN, you can specify:

```
      INCLUDE (ERRTN)
```

in your source program. During compilation, the ERRTN source statements are inserted at this point in the program.

For information on cataloging the source code, see "Cataloging Your Source Program."

AVOIDING CODING ERRORS

Many coding errors can cause problems in your compilation.

While you're coding your program, be careful not to make these common programming errors:

1. Misspelling FORTRAN words.

2. Omitting required punctuation, or inserting unneeded punctuation.

3. Not observing FORTRAN formatting rules.

4. Forgetting to assign values to variables and arrays before you use them.

5. Moving data into an item that's too small for it. (This causes truncation.)

   In particular, don't initialize more array elements than the array contains; you can inadvertently destroy subsequent data and instructions.

6. Specifying subscript values that are not within the bounds of an array.

7. Inadvertently changing types defined in an IMPLICIT statement by explicit type statements.

8. Making invalid data references to equivalenced items of differing types (for example, integer and real).

9. Transferring control from outside a DO loop into an intermediate point in a DO loop.

10. Using arithmetic items for intermediate calculations that are too small to give the precision you need in the result. For example: If your final result is in the order of seven decimal digits, you may need to perform the intermediate calculations in DOUBLE PRECISION.

11. Failing to inspect code movements in programs compiled with the OPTIMIZE(3) option. For example: if an IF statement controls execution of a computation within a loop, the computation may be moved outside the loop and give you results you don't expect. See "Using the Optimization Feature."

12. Attempting to process input records after a file has been used for output. See "Programming Input and Output."

13. Writing main programs or subprograms that directly or indirectly invoke themselves (see "Coding Calling and Called Programs").

14. Writing a series of subprograms without a required main program.

15. Defining dummy arguments and actual arguments that do not agree in type and/or length. For example: arrays that do not have the same dimensions, integer actual arguments with real dummy arguments or vice versa.

16. In a subprogram, assigning new values to arguments associated with variables in common.

17. Failing to initialize VSCOM when your main program is not a FORTRAN program. (See "Appendix B. Assembler Language Considerations.")

18. Referring to a statement function with other than the defined number of arguments.

19. When using carriage control characters, not overriding all applicable default values for FORTRAN units.

20. If you recompile programs originally written for the OS FORTRAN G1, H Extended, or DOS FORTRAN F compilers, you may experience VS FORTRAN internal table overflows. If this happens, you must restructure the source code to reduce the number of unique parameter lists and/or the number of branches specified in computed GO TO statements and alternate returns.

21. The VS FORTRAN compiler uses the OS FORTRAN H extended architecture for rounding infinite binary expansions. The OS FORTRAN G1 compiler also rounds, but the DOS FORTRAN F compiler truncates. If you recompile programs originally written for the DOS FORTRAN F compiler, you may experience a difference in expected execution output.

22. If your VS FORTRAN Internal tables overflow, you must restructure the source code to reduce the number of unique parameter lists and/or the number of branches specified in computed GO TO statements and alternate returns.

Additional parameter lists are created when you recompile OS FORTRAN H Extended programs that have many arrays with adjustable dimensions. (These parameter lists are created because a library routine (IFYDSPAN) is used to perform the calculations for adjustable dimensions.) Also, in LANGLVL(77) parameter lists, character parameters generate two entries each.

COMPILING YOUR PROGRAM—ADVANCED PROGRAMMING

Once your source program is written, you must have it translated into machine language—that is, compile it. What you get as output from the translation depends upon the job control options and compiler options you specify.

AUTOMATIC CROSS COMPILATION

Cross compilation of VS FORTRAN programs is automatic. That is, you can compile your source program under any system; you can then link-edit the resulting object module to execute under any of the supported systems.

OVERALL JOB CONTROL CONSIDERATIONS

The job control statements you use, when processing VS FORTRAN programs, are dependent upon the system you're running under. Your VS FORTRAN jobs can be any of the following:

- Compile-only
- Compile, link-edit
- Compile, link-edit, execute
- Link-edit only
- Link-edit, execute
- Execute only

No matter which of these you're doing, there are certain job control statements you use for any job. These are explained in the following sections.

Syntax for Job Control Statements

In the following sections, the job control statements are shown using a formal syntax notation. The following paragraphs define how to interpret this syntax.

- **UPPERCASE LETTERS, WORDS, AND NUMBERS:** must be coded in the statement exactly as shown.

  For example, the word JOB in a format is to be coded as JOB.

- **LOWERCASE LETTERS AND WORDS:** represent variables, for which programmer-supplied information is substituted.

  For example, the word "option" in a format can be coded as NODECK.

- **SYMBOLS** in the following list must be coded exactly as shown:

apostrophe	'
asterisk	*
comma	,
equal sign	=
parentheses	( )
period	.
slash	/

- **HYPHENS (-)**: join lowercase letters and words and symbols to form a single variable name.

  For example, the word "program-name" in a format could be coded as MYPROG1.

- **SQUARE BRACKETS ([ ])**: group optional related items (such as alternative choices from which one choice can be made). For example, the sequence "option [,option]" in a format could be coded as NODECK or as NODECK,XREF as needed.

- **ELLIPSES (...)**: specify that the preceding syntactical unit can, optionally, be repeated. (A syntactical unit is a single syntactical item, or a group of syntactical items enclosed in braces or brackets.) For example, the sequence "option [,option] ... " in a format could be coded as NODECK or as NODECK,XREF or as NODECK,XREF,MAP,DUMP as needed.

- **OR SIGNS(|)**: specify that only one of the units they separate can be coded.

- **BLANKS**: are used to improve the readability of the control statement definitions. They must not appear in an operand field, unless a definition explicitly states otherwise.

See "Using VM/370-CMS with VS FORTRAN" for information on compiling, link-editing, and executing programs under VM/370-CMS.

USING THE COMPILE-TIME OPTIONS

The VS FORTRAN compile-time options let you specify details about the input source program and request specific forms of compilation output.

Each option has a default value for your organization, and was set when the compiler was installed. Ask your system administrator for the default options in force.

How you specify the compiler options depends upon the system you're using:

- In VM/370-CMS, specify them as options of the FORTVS command (see "Using VM/370-CMS with VS FORTRAN").

- In OS/VS, specify them as options in the PARM subparameter of the EXEC job control statement (see "Requesting Execution—OS/VS EXEC Statement").

- In DOS/VSE, specify them as options in the PARM parameter of the EXEC job control statement (see "Requesting Execution—DOS/VSE EXEC Statement").

For abbreviations of compiler options, see VS FORTRAN Application Programming: System Services Reference Supplement.

In the following paragraphs, each of the compile-time options is briefly explained:

DECK|NODECK
Specifies whether or not the object module in card image format is to be produced.

FIPS (S|F) | NOFIPS
Specifies whether or not standard language flagging is to be performed, and, if it is, the standard language flagging level: subset or full.

Items not defined in the current American National Standard are flagged.

If you specify LANGLVL(66) and FIPS flagging at either level, the FIPS option is ignored.

FLAG (I|W|E|S)
Specifies the level of diagnostic messages to be written: I (information) or higher, W (warning) or higher, E (error) or higher, or S (severe) or higher.

FREE|FIXED
Indicates whether the input source program is in free format or in fixed format.

GOSTMT|NOGOSTMT
Specifies whether or not internal sequence numbers (for traceback purposes) are to be generated for a calling sequence to a subprogram.

LANGLVL (66|77)
Specifies the language level in which the input source program is written: the old FORTRAN language level, or the current FORTRAN language level.

LINECOUNT (number)
Specifies the maximum number of lines on each page of the printed source listing. The number may be in the range 5 to 32765.

LIST|NOLIST
Specifies whether or not the object module listing is to be written.

MAP|NOMAP
Specifies whether or not a table of source program names and statement labels is to be written.

NAME(name)
For old FORTRAN programs only, specifies the name to be given to a main program.

When NAME is omitted, the main program is named MAIN#.

OBJECT|NOOBJECT
Specifies whether or not the object module is to be produced.

OPTIMIZE (0|1|2|3) | NOOPTIMIZE
Specifies the optimizing level to be used during compilation:

OPTIMIZE (0) OR NOOPTIMIZE specifies no optimization.

OPTIMIZE (1) specifies register and branch optimization.

OPTIMIZE (2) specifies partial code-movement optimization. OPTIMIZE(2) will not relocate any code when it has been determined that relocating the code under consideration would cause unplanned or unexpected interrupts.

OPTIMIZE (3) specifies full code-movement optimization.

SC(name1,name2,...)
Specifies the names of subroutines for which only the locations of character arguments are to be passed. (An entry for the length of the character string is not made in the parameter list.) This option can be repeated; the lists will be combined. On an @PROCESS statement, multiple names can be supplied as parameters to the SC option. However, only one parameter for each SC option can be supplied on invocation of the compiler (EXECUTE options).

SOURCE|NOSOURCE
Specifies whether or not the source listing is to be produced.

TERMINAL|NOTERMINAL
Specifies whether or not error messages and compiler diagnostics are to be written on the output data set and whether or not a summary of error messages is to be printed.

XREF|NOXREF
Specifies whether or not a cross-reference listing is to be produced.

MODIFYING BATCH COMPILATION OPTIONS—@PROCESS STATEMENT

In a batch compilation for either OS/VS or DOS/VS, the options specified when the compiler is invoked remain in force for all source programs you're compiling, unless you override them with the @PROCESS statement.

To change the compiler options, place the @PROCESS statement just before the first statement in the source program. The following rules apply:

- @PROCESS must appear in columns 1 through 8 of the statement.
- The @PROCESS statement can be followed by compiler options in columns 9 through 72 of the statement. The options must be separated by commas or blanks.
- The @PROCESS statement cannot be continued to multiple lines. However, multiple process statements can be supplied for a program unit. Columns 9 through 72 of a following PROCESS statement are appended to the previous @PROCESS statement.

  All compiler options except OBJECT and DECK are permissible.
- All options _not_ to be given default values must be specified, including the overriding options in the EXEC statement. (@PROCESS overrides _all_ options specified in the EXEC statement.)

JOB CONTROL CONSIDERATIONS—OS/VS

The simplest way to execute your program under OS/VS is to use one of the cataloged procedures described in "Using and Modifying Cataloged Procedures—OS/VS."

However, the cataloged procedures may not give you the programming flexibility you need for your more complex data processing jobs, and you may need to specify your own job control statements.

The job control statements you can use for any OS/VS job are outlined below. Reference documentation for them is given in the _VS FORTRAN Application Programming: System Services Reference Supplement_.

IDENTIFYING A JOB—OS/VS JOB STATEMENT

The JOB statement begins each OS/VS job you enter into the system:

```
//jobname JOB [parameters]
```

The jobname identifies this job to the system. The jobname must conform to the standards defined in _OS/VS2 MVS JCL_.

The parameters let you request the following:

- Accounting information for this job
- Your name
- The type of system messages to be written
- Conditions for terminating job execution
- Assignment of input and output classes
- Job priority
- Main storage requirements
- Time limit for the job

REQUESTING EXECUTION—OS/VS EXEC STATEMENT

You use the EXEC job control statement to request that execution of a program or procedure is to begin.

```
//[stepname] EXEC [PROC=]procname
                [PARM=option[,option] ...
                [other-parameters]
```

The stepname identifies this job step.

The procname is the name of a cataloged procedure you want executed.

To request a FORTRAN compilation, you specify PROC=FORTVS.

The PARM parameter lets you specify any options that differ from the defaults.

(See "Using the Compile-Time Options" and "Link-Editing Your Program—Advanced Programming.")

The other parameters let you request other information:

- A job step name (when its necessary for a later job step to refer to information from this job step)
- Conditions for bypassing execution of this job step
- Accounting information for this job step
- Time limit for this step
- Main storage requirements

DEFINING FILES—OS/VS DD STATEMENT

To define files you may need, you specify the DD statement:

```
//[ddname|procstep.ddname] DD [data-set-name][other-parameters]
```

The ddname identifies the data sets defined by this DD statement to the compiler, linkage editor, loader, or to your program. The ddnames you can use for VS FORTRAN are shown in Figure 16.

The procstep identifies the procedure step.

The data-set-name is the qualified name you've given the data set that contains your data files; for example, the name of the library containing the files you use in your INCLUDE statements.

The other parameters let you request additional information:

- The location of this data set in the system configuration

- The status of this data set at the beginning and end of the job step
- Label information for this data set's volume
- Optimization of input/output channel usage
- Device type
- Space allocation (for data sets on direct access devices)
- Characteristics of the data set records

Reference documentation on these job control statements is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

REQUESTING COMPILATION—OS/VS

In one job step under OS/VS, you can request compilation for a single source program or for a series of source programs.

COMPILING A SINGLE SOURCE PROGRAM—OS/VS

For compiling a single source program under OS/VS, the sequence of job control statements is:

```
//JOB Statement
//EXEC Statement (to execute the VS FORTRAN compiler)
//DD Statements for Compilation (as required)
  (Source program to be compiled)
/*Data Delimiter Statement (only if source program is on cards)
//End-of-Job Statement
```

These job control statements are decribed in the VS FORTRAN Application Programming: System Services Reference Supplement.

BATCH COMPILATION OF MORE THAN ONE SOURCE PROGRAM—OS/VS

In OS/VS, you can compile more than one source program during the execution of one job. The sequence of job control statements you use is:

```
//JOB Statement
//EXEC Statement (to execute the VS FORTRAN compiler)
//DD Statements (as required)
@PROCESS Statement (if needed to modify compiler options)
  (First source program to be compiled)
@PROCESS Statement (if needed to modify compiler options)
  (Second source program to be compiled)
@PROCESS Statement (if needed to modify compiler options)
  (Third source program to be compiled)
/*Data Delimiter Statement (only if source program is on cards)
//End-of-Job Statement
```

These job control statements are decribed in the VS FORTRAN Application Programming: System Services Reference Supplement.

The @PROCESS statement is described in "Modifying Batch Compilation Options—@PROCESS Statement" on page 74.

COMPILATION DATA SETS—OS/VS

For compilation under OS/VS, there are two required data sets and several optional ones.

REQUIRED OS/VS COMPILE-TIME DATA SETS

You must ensure that the following ddnames are available during compilation. They may be available through the cataloged procedure you're using for compilation (check with your system administrator). If they aren't, you must specify them through a DD statement:

SYSIN—to define the source input data set

SYSPRINT—to define the printed output data set

OPTIONAL OS/VS COMPILE-TIME DATA SETS

You specify the following data sets, through a DD statement, only if you're requesting specific compilation features:

SYSLIN—if you're requesting that an object module be produced (through the OBJECT compiler option); it defines the object module data set.

SYSPUNCH—if you're requesting an object module punched on cards (through the DECK compiler option); it defines the card image data set on which the object module is punched.

SYSTERM—when the TERMINAL compile-time option is in effect, you specify SYSTERM as the data set that will contain printed compiler output (that is, error messages, source statements in error, and compiler statistics).

SYSLIB—if your source program uses the INCLUDE statement, this ddname defines the library input data set. For INCLUDE statements, the data set records must be unblocked, fixed length, 80-character records with DSORG=PO.

USING AND MODIFYING CATALOGED PROCEDURES—OS/VS

When you're operating under OS/VS, there are a number of cataloged procedures you can use to compile, link-edit (or load), and execute your VS FORTRAN programs:

1. Compile only—FORTVC, shown in Figure 16
2. Compile and Link-edit—FORTVCL, shown in Figure 17
3. Compile, Link-edit, and Execute—FORTVCLG, shown in Figure 18
4. Link-edit and Execute—FORTVLG, shown in Figure 19
5. Execute only—FORTVG, shown in Figure 20
6. Compile and Load—FORTVCG, shown in Figure 21
7. Load only—FORTVL, shown in Figure 22

As the figures show, many of the JCL parameters in these procedures are coded as symbolic parameters (the parameter name is preceded by an ampersand). The IBM-supplied default value for each symbolic parameter is defined in the PROC statement that begins each procedure.

This means that you can execute the procedures without making any changes to them, or you can modify them for any particular run. (The use of symbolic parameters is explained in the job control language publications for the system you're operating under.)

```
//FORTVC   PROC FVPGM=FORTVS,FVREGN=800K,FVPDECK=NODECK,
//              FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',
//              FVLNSPC='3200,(25,6)'
//*
//*             PARAMETER  DEFAULT-VALUE     USAGE
//*
//*               FVPGM    FORTVS            COMPILER NAME
//*               FVREGN   256K              FORT-STEP REGION
//*               FVPDECK  NODECK            COMPILER DECK OPTION
//*               FVPOLST  NOLIST            COMPILER LIST OPTION
//*               FVPOPT   0                 COMPILER OPTIMIZATION
//*               FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*               FVLNSPC  3200,(25,6)       FORT.SYSLIN SPACE
//*
//FORT   EXEC   PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//              PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM       DD &FVTERM
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
```

Figure 16. Cataloged Procedure FORTVC, OS/VS

```
//FORTVCL  PROC FVPGM=FORTVS,FVREGN=800K,FVPDECK=NODECK,
//         FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',
//         FVPNAME=MAIN,PGMLIB='&&GOSET'
//*
//*             PARAMETER  DEFAULT-VALUE     USAGE
//*
//*               FVPGM    FORTVS            COMPILER NAME
//*               FVREGN   256K              FORT-STEP REGION
//*               FVPDECK  NODECK            COMPILER DECK OPTION
//*               FVPOLST  NOLIST            COMPILER LIST OPTION
//*               FVPNAME  MAIN              COMPILER NAME OPTION
//*               FVPOPT   0                 COMPILER OPTIMIZATION
//*               FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*               PGMLIB   &&GOSET           LKED.SYSLMOD DSNAME
//*
//FORT   EXEC   PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//              PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT),NAME(&FVPNAME)'
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM       DD &FVTERM
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(3200,(25,6)),DCB=BLKSIZE=3200
//LKED   EXEC   PGM=IEWL,REGION=200K,COND=(4,LT),
//              PARM='LET,LIST,MAP,XREF'
//SYSPRINT      DD SYSOUT=A
//SYSLIB        DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1        DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD       DD DSN=&PGMLIB.(&FVPNAME),DISP=(,PASS),UNIT=SYSDA,
//              SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN        DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//              DD DDNAME=SYSIN
```

Figure 17. Cataloged Procedure FORTVCL, OS/VS

```
//FORTVCLG PROC FVPGM=FORTVS,FVREGN=800K,FVPDECK=NODECK,
//              FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',GOREGN=100K,
//              GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',
//              GOF7DD='SYSOUT=B'
//*
//*             PARAMETER  DEFAULT-VALUE     USAGE
//*
//*               FVPGM    FORTVS            COMPILER NAME
//*               FVREGN   800K              FORT-STEP REGION
//*               FVPDECK  NODECK            COMPILER DECK OPTION
//*               FVPOLST  NOLIST            COMPILER LIST OPTION
//*               FVPOPT   0                 COMPILER OPTIMIZATION
//*               FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*               GOREGN   100K              GO-STEP REGION
//*               GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*               GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*               GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*
//FORT   EXEC   PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//              PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM       DD &FVTERM
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(3200,(25,6)),DCB=BLKSIZE=3200
//LKED   EXEC   PGM=IEWL,REGION=200K,COND=(4,LT),
//              PARM='LET,LIST,MAP,XREF'
//SYSPRINT      DD SYSOUT=A
//SYSLIB        DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1        DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD       DD DSN=&&GOSET(MAIN),DISP=(,PASS),UNIT=SYSDA,
//              SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN        DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//              DD DDNAME=SYSIN
//GO     EXEC   PGM=*.LKED.SYSLMOD,REGION=&GOREGN,COND=(4,LT)
//FT05F001      DD &GOF5DD
//FT06F001      DD &GOF6DD
//FT07F001      DD &GOF7DD
```

Figure 18. Cataloged Procedure FORTVCLG, OS/VS

```
//FORTVLG PROC LKLNDD='DDNAME=SYSIN',GOPGM=MAIN,GOREGN=100K,
//             GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B'
//*
//*
//*
//*            PARAMETER  DEFAULT-VALUE      USAGE
//*              LKLNDD   DDNAME=SYSIN       LKED.SYSLIN OPERAND
//*              GOPGM    MAIN               OBJECT PROGRAM NAME
//*              GOREGN   100K               GO-STEP REGION
//*              GOF5DD   DDNAME=SYSIN       GO.FT05F001 OPERAND
//*              GOF6DD   SYSOUT=A           GO.FT06F001 OPERAND
//*              GOF7DD   SYSOUT=B           GO.FT07F001 OPERAND
//*
//*
//LKED    EXEC PGM=IEWL,REGION=200K,
//             PARM='LET,LIST,MAP,XREF'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD      DD DSN=&&GOSET(&GOPGM),DISP=(,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD &LKLNDD
//GO      EXEC PGM=*.LKED.SYSLMOD,REGION=&GOREGN,
//             COND=(4,LT,LKED)
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 19. Cataloged Procedure FORTVLG, OS/VS

```
//FORTVG PROC  GOPGM=MAIN,GOREGN=100K,
//             GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B'
//*
//*
//*            PARAMETER  DEFAULT-VALUE      USAGE
//*
//*              GOPGM    MAIN               PROGRAM NAME
//*              GOREGN   100K               GO-STEP REGION
//*              GOF5DD   DDNAME=SYSIN       GO.FT05F001 DD OPERAND
//*              GOF6DD   SYSOUT=A           GO.FT06F001 DD OPERAND
//*              GOF7DD   SYSOUT=B           GO.FT07F001 DD OPERAND
//*
//*
//GO      EXEC PGM=&GOPGM,REGION=&GOREGN
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 20. Cataloged Procedure FORTVG, OS/VS

```
//FORTVCG  PROC FVPGM=FORTVS,FVREGN=800K,FVPDECK=NODECK,
//             FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',
//             FVLNSPC='3200,(25,6)',
//             GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B',GOREGN=100K
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*
//*              FVPGM    FORTVS            COMPILER NAME
//*              FVREGN   256K              FORT-STEP REGION
//*              FVPDECK  NODECK            COMPILER DECK OPTION
//*              FVPOLST  NOLIST            COMPILER LIST OPTION
//*              FVPOPT   0                 COMPILER OPTIMIZATION
//*              FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*              FVLNSPC  3200,(25,6)       FORT.SYSLIN SPACE
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*              GOREGN   100K              GO-STEP REGION
//*
//FORT    EXEC PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//             PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT     DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM      DD &FVTERM
//SYSPUNCH     DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN       DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//GO      EXEC PGM=LOADER,REGION=&GOREGN,
//             PARM='LET,NORES,EP=MAIN'
//SYSLIN       DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSLOUT      DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 21. Cataloged Procedure FORTVCG, OS/VS

```
//FORTVL  PROC GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B',GOREGN=100K
//*
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*              GOREGN   100K              GO-STEP REGION
//*
//*
//GO      EXEC PGM=LOADER,REGION=&GOREGN,
//             PARM='LET,NORES,EP=MAIN'
//SYSLOUT      DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 22. Cataloged Procedure FORTVL, OS/VS

The following sections describe how you can temporarily modify the OS/VS cataloged procedures.

Modifying PROC Statements—OS/VS

You can modify PROC statement parameters by specifying changes in the EXEC statement that calls the procedure. When you change a PROC statement parameter, you're assigning a temporary value to a symbolic parameter; when the cataloged procedure is executed, this value is transferred to the appropriate parameter in the EXEC or DD statement.

For example, to change the region size of the compiler from 256K to 200K bytes and the card punch output in the load module from output class B to output class C, you can use the following statement:

```
//  EXEC  FORTVCLG,FVREGN=200K,
//        GOF7DD='SYSOUT=C'
```

Note that you don't code the ampersand preceding a symbolic parameter; and that you use apostrophes to enclose a value containing a special character, as in SYSOUT=C.

Before you execute the call, the appropriate statements in FORTVCLG appear as follows:

```
//FORT   EXEC  PGM=&FVPGM,REGION=&FVREGN,...
         .
         .
         .
//FT07F001  DD  &GOF7DD
```

When the cataloged procedure is called, the statements appear as though they were coded:

```
//FORT   EXEC  PGM=FORTVS,REGION=200K,...
         .
         .
         .
//FT07F001  DD  SYSOUT=C
```

Note that a symbolic parameter not changed (PGM=&FVPGM) retains its default value.

Another method you can use to change a parameter value is to assign the new value directly to the parameter itself, not the symbolic parameter associated with it. For example, you can change the region size by specifying REGION=200K in place of FVREGN=200K. (In this example, you'd be changing the region size for all job steps; to change the region size for the compile job step only, you must code the appropriate job step name, FORT, in the parameter; for example, REGION.FORT=200K.)

Modifying EXEC Statements—OS/VS

To modify EXEC statement parameters, you must change the EXEC statement that calls the procedure.

The following rules apply to EXEC statement modifications:

- Parameters are overridden in their entirety. If you want to retain some options while changing others, you must specify those options to be retained. (However, if you don't override them, default options remain in effect.)
- To specify parameters for individual job steps, use the form:

  <u>keyword.stepname=value</u>

  where:

keyword
indicates the parameter name

stepname
indicates the procedure name, for example,

```
REGION.FORT=value
```

Parameters not specifying stepname are assumed to apply to all steps in the procedure; for example, REGION=value applies to the entire cataloged procedure.

- To make changes to more than one step, you must specify all changes for an earlier step before those for later steps.

- You can combine changes to symbolic parameters and EXEC statement parameters on the same card.

You're allowed to make the following modifications:

1. Override existing parameters: For example, to modify the LKED step by raising the condition code from 4 to 8, use the statement:

```
//SOMENAME EXEC FORTVCLG,
//              COND.LKED=(8,LT)
```

2. Add new parameters: For example, to modify FORT by specifying the TIME parameter, use the statement:

```
//ANYNAME  EXEC  FORTVCLG,TIME.FORT=5
```

3. Change more than one parameter: For example, to modify FORT by changing the region from 256K to 200K bytes and the PARM option NOLIST to LIST, use the statement:

```
//SOME EXEC FORTVCLG,
//          REGION.FORT=200K,
//          PARM.FORT=LIST
```

4. Change more than one step: For example, to modify FORT by specifying TIME and to modify LKED by raising the condition code from 4 to 8, use the statement:

```
//ANY  EXEC  FORTVCLG,TIME.FORT=5,
//           COND.LKED=(8,LT)
```

Note that you can add a parameter while revising an existing one.

5. Combine changes to symbolic parameters and EXEC statement parameters: For example, to modify the symbolic parameter FVREGN, and to add the TIME parameter to the FORT EXEC statement, use the statement:

```
//ANY  EXEC  FORTVCLG,FXREGN=200K,
//           TIME.FORT=5
```

Modifying DD Statements—OS/VS

You modify the DD statement by submitting new DD statements after the EXEC statement that calls the procedure. As with modifications to EXEC statements, you can override or add parameters to DD statements in one or many steps. In addition, you can add entirely new DD statements to any step (whenever you supply a SYSIN DD statement, you're adding a new DD statement.)

The following rules apply to DD statement modifications:

- Parameters are overridden in their entirety except for the DCB parameter, in which individual subparameters can be overridden.

- Parameters are nullified by specifying a comma after the equal sign in the parameter; for example, UNIT=,.
- Parameters are overridden when mutually exclusive parameters are specified in their place; for example, SPLIT overrides SPACE.
- DD statements must indicate the related procedure step, using the form _//procstep.ddname_; for example, //FORT.SYSIN.
- To make changes in more than one step, you must specify all changes for an earlier step before those for later steps.
- To modify more than one DD statement in a job step, you must specify the applicable DD statements in the same sequence as they appear in the cataloged procedure.

You can make the following modifications:

1. _Override existing parameters_. For example, to modify SYSLMOD so that the load module is stored in a private library rather than in the system library, you can specify the statement:

   ```
   //LKED.SYSLMOD DD DSNAME=PRIV(PROG),
   //                DISP=(MOD,PASS)
   ```

   In this example, the library PRIV is assumed to be an old library and is cataloged (that is, VOLUME and UNIT parameters need not be specified). Note that in subsequent uses of the library you must submit a JOBLIB DD statement defining the private library, to make the library available to the system.

2. _Add new parameters_. For example, to store the load module in a new, uncataloged library, you must specify the VOLUME, UNIT, and SPACE parameters. For example:

   ```
   //LKED.SYSLMOD DD DSNAME=MYLIB(FIRST),
   //                DISP=(NEW,PASS),
   //                VOLUME=SER=11234,
   //                UNIT=SYSDA,
   //                SPACE=(TRK,(50,10,2))
   ```

3. _Add new DD statements_. For example, to add new data sets having data set reference numbers 10 and 15 for processing in the GO step, you can specify the statements:

   ```
   //GO.FT10F001  DD  DSNAME=DSET1,
   //                 DISP=(NEW,DELETE),
   //                 VOLUME=SER=T1132,
   //                 UNIT=TAPE
   //GO.FT15F001  DD  DSNAME=DSET2,
   //                 DISP=(,DELETE),
   //                 VOLUME=SER=DA45,
   //                 UNIT=3350,
   //                 SPACE=(TRK,(10,10))
   ```

   Note that you can explicitly define a data set as new (DISP parameter for FT10F001), or, alternatively, permit the system to assume a new data set by default (DISP in FT15F001).

JOB CONTROL CONSIDERATIONS—DOS/VSE

Under DOS/VSE, the simplest way to execute your program is to use one of the procedures outlined in "Requesting Compilation Only—DOS/VSE."

However, these procedures may not give you the programming flexibility you need for your more complex data processing jobs, and you may need to specify your own job control statements, or write your own cataloged procedures.

USING DOS/VSE JOB CONTROL STATEMENTS

The job control statements you can use for any DOS/VSE job are outlined below. Reference documentation for them is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

IDENTIFYING A JOB—DOS/VSE JOB STATEMENT

The JOB statement begins each DOS/VSE job you enter into the system:

```
// JOB jobname
```

The jobname is the name you're giving to this job.

You can also code comments in the JOB statement, including job accounting information.

SPECIFYING LINKAGE-EDITOR OPTIONS—DOS/VSE OPTION STATEMENT

Under DOS/VSE, you use the OPTION statement to request the linkage editor, and linkage-editor options.

```
// OPTION option[,option] ...
```

In the compilation step, you can use the OPTION statement to specify that the object module is to be link-edited:

- Specify OPTION LINK for link-editing only.
- Specify OPTION CATAL for link-editing and cataloging in the core image library.

DEFINING FILES—DOS/VSE ASSGN STATEMENT

You use the ASSGN statement to assign a logical input/output unit to a physical device. (You need do this only if the logical unit is different from that defined during system generation.)

```
// ASSGN SYSxxx[,device-address|symbolic-unit][,other-entries]
```

SYSxxx is a valid system logical unit or programmer logical unit. The units valid for VS FORTRAN are shown in Figure 23.

(See the VS FORTRAN Application Programming: System Services Reference Supplement.)

The device-address specifies the channel and unit number for the specific device.

The symbolic-unit specifies a valid system or programmer logical unit.
You can also optionally specify other entries:

- That the logical unit is to remain unassigned, and that any program references to it are to cause immediate job cancellation
- That the logical unit is to remain unassigned and that all program references to it are to be ignored, without canceling the job
- That this assigned unit is to be used as an alternate unit when the capacity of the original device is reached

- Tape mode specifications

(See the VS FORTRAN Application Programming: System Services Reference Supplement.)

DEFINING FILES ON DIRECT ACCESS DEVICES—DOS/VSE

For files on direct access devices, you must specify a DLBL statement and an EXTENT statement, in addition to the ASSGN statement.

DLBL Statement—DOS/VSE

You use the DLBL statement to supply label information for both file creation and file retrieval.

```
// DLBL filename [data-set-id][,date][,code]
```

The filename is the filename that corresponds to the logical unit. See Figure 23 for a list of valid system filenames.

The data-set-id specifies the name associated with this data set on the volume. If you omit this field, the filename is used.

The date specifies the retention period or expiration date of the data set.

The code specifies the type of data set label to be used: sequential, direct or VSAM.

Other fields can also be specified.

For reference documentation about the DLBL statement, see the VS FORTRAN Application Programming: System Services Reference Supplement.

EXTENT Statement—DOS/VSE

You use the EXTENT statement to assign a data area on a direct access device to a file in your program.

```
// EXTENT SYSxxx,,,,[relative-track],[no-of-tracks]
```

SYSxxx is a valid system logical unit or programmer logical unit. See Figure 23 for units valid for VS FORTRAN.

For reference documentation, see the VS FORTRAN Application Programming: System Services Reference Supplement.

The relative-track specifies the position of the track (relative to zero) where the extent begins.

The no-of-tracks specifies the number of the tracks to be allocated to the extent.

You can also specify other optional entries:

- Whether the extent is split cylinder or no split cylinder
- The number of split cylinder tracks (if this is a split cylinder extent)

REQUESTING EXECUTION—DOS/VSE EXEC STATEMENT

You use the EXEC job control statement to request that execution is to begin.

```
// EXEC[[PGM=]program-name],SIZE=AUTO|number,-
                   [PARM='option'[,'option']...]
// EXEC[PROC=program-name][,SIZE=AUTO|number]
```

The program-name identifies either:

- When you specify PGM=, a phase in the core image library.

  For example, the VS FORTRAN compiler, the linkage editor, or a phase (problem program object module).

  To request a FORTRAN compilation, you specify PGM=VFORTRAN.

  You can omit the program-name if the phase to be executed has just been processed by the linkage editor.

  When you're requesting a VS FORTRAN compilation, you should specify the SIZE parameter as 800K.

  Use the PARM parameter to request compiler options that differ from those set for your organization. The options can be any of those described in "Using the Compile-Time Options."

- When you specify PROC=, a procedure in the procedure library.

For reference documentation about these job control statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

REQUESTING COMPILATION—DOS/VSE

Under DOS/VSE, in one job you can request compilation for a single source program or for a series of source programs.

COMPILING A SINGLE SOURCE PROGRAM—DOS/VSE

For a single source program, the sequence of job control statements you use is:

```
// JOB Statement
// OPTION Statement (as required)
// ASSGN Statements for Compilation (as required)
// DLBL Statements for Compilation (as required)
// EXTENT Statements for Compilation (as required)
// EXEC Statement (to execute the VS FORTRAN compiler)
   (Source program to be compiled)
/* Data Delimiter Statement (only if source program is on cards)
/& End-Of-Job Statement
```

BATCH COMPILATION OF MORE THAN ONE SOURCE PROGRAM—DOS/VSE

For a series of programs, the sequence of job control statements you use is:

```
// JOB Statement
// OPTION Statement (as required)
// ASSGN Statements for Compilation (as required)
// DLBL Statements for Compilation (as required)
// EXTENT Statements for Compilation (as required)
// EXEC Statement (to execute the VS FORTRAN compiler)
   (First source program to be compiled)
@PROCESS Statement (if needed to modify compiler options)
   (Second source program to be compiled)
@PROCESS Statement (if needed to modify compiler options)
   (Third source program to be compiled)
/* Data Delimiter Statement (only if source program is on cards)
/& End-Of-Job Statement
```

These job control statements are described in the VS FORTRAN Application Programming: System Services Reference Supplement.

The @PROCESS statement is described in "Modifying Batch Compilation Options—@PROCESS Statement" on page 74.

COMPILE-TIME FILES—DOS/VSE

When you're compiling under DOS/VSE, most of the the system files the compiler uses—SYSIPT, SYSLST, SYSPCH, SYSLNK, and SYSLOG—are predefined and always available; therefore, you never have to specify them.

However, if your source program uses the INCLUDE statement, you must specify SYSSLB (for the system or a private source statement library) in an ASSGN statement. The file records representing the source statements to be included must be unblocked, fixed-length, 80-character records. The file to be included must be cataloged in the G sublibrary.

WRITING AND MODIFYING CATALOGED PROCEDURES—DOS/VSE

To catalog a procedure in the procedure library, you submit a CATALP statement specifying the procedure name. Rules for naming the procedures are given in DOS/VSE System Control Statements.

The control statements to be cataloged follow the CATALP statement; they can be job control or linkage editor control statements or both. The end of the control statements to be cataloged must be indicated by an end-of-procedure delimiter, usually a /+ delimiter.

Each control statement cataloged in the procedure library should have a unique identity. This identity is required if you want to be able to modify the job stream at execution time. Therefore, when cataloging, identify each control statement in columns 73 through 79 (blanks may be embedded).

RETRIEVING CATALOGED PROCEDURES—DOS/VSE

To retrieve a cataloged procedure from the procedure library, you use the PROC parameter in the EXEC job control statement, specifying the name of the cataloged procedure.

When the job control program starts reading the job control statements in the input stream on SYSRDR and finds the EXEC statement, it knows by the operand PROC that a cataloged procedure is to be inserted. It takes the name of the procedure to be used and retrieves the procedure with that name from the procedure library. At this point, SYSRDR is temporarily assigned to the procedure library. Job control reads and processes the job control statements as usual. The statement,

```
//EXEC MYPROGM
```

causes the program MYPROGM to be loaded and given control. When execution of MYPROGM is complete, the job control program reads the next statement or statements from the procedure library and then finds the end-of-procedure indicator (/+). The end-of-procedure indicator returns the SYSRDR assignment to its permanent device, where the job control program finds the /& statement and performs end-of-job processing as usual.

TEMPORARILY MODIFYING CATALOGED PROCEDURES—DOS/VSE

You can request temporary modification of statements in a cataloged procedure by supplying the corresponding modifier statements in the input stream.

Normally not all statements need to be modified; therefore, you must establish an exact correspondence between the statement to be modified and the modifier statement by giving them the same symbolic name. This symbolic name may have from one to seven characters, and must be specified in columns 73 through 79 of both statements.

A single character in column 80 of the modifier statement specifies which function is to be performed.

A	indicates that the statement is to be inserted _after_ the statement in the cataloged procedure that has the same name.
B	indicates that the statement is to be inserted _before_ the statement in the cataloged procedure that has the same name.
D	indicates that the statement in the cataloged procedure that has the same name is to be _deleted_.

Any other character or a blank in column 80 of the modifier statement indicates that the statement is to replace (override) the statement in the cataloged procedure that has the same name.

In addition to naming the statements and indicating the function to be performed, you must inform the job control program that it has to carry out a procedure modification. This is done as follows:

1. By specifying an additional parameter (OV for overriding) in the EXEC statement that calls the procedure, and

2. By using the statement // OVEND to indicate the end of the modifier statements.

Placement of the // OVEND statement is as follows:

- Directly behind the last modifier statement or,

- If the last modifier statement overwrites a // EXEC statement and is followed by data input, between the /* and /& delimiters.

COMPILER OUTPUT

The output the compiler gives you depends upon whether you've accepted the default compiler options in force for your organization, or whether you've modified the defaults using explicit compiler options.

COMPILER OUTPUT WITH DEFAULT OPTIONS

If you accept the compiler default options, you'll usually get the following output, printed in the following order (depending on the default options in force for your organization):

- The date of the compilation—plus information about the compiler and this compilation

- A listing of your source program

- Diagnostic messages telling you of errors in the source program

- Informative messages telling you the status of the compilation

(You'll also get a machine language object module.)

Note that these defaults may be modified for your organization.

OUTPUT WITH EXPLICIT COMPILER OPTIONS

In addition to the output listed above, you can request each compilation to produce the following additional output:

- A listing of the object module (LIST option), in pseudo-assembler language (that is, the assembler instructions that would have been generated for the object module, if the compiler translated into assembler before producing the machine code)
- A copy of the object module (DECK option), in card image format
- A table of names and statement labels (MAP option) defined in the source program
- A cross-reference listing (XREF option) of variables and labels used in the source program
- Messages flagging statements that do not conform to the language standard level you've chosen

Depending on the options you've chosen, the output you'll get is shown in Figure 23 (Options that produce printed output are shown in the order in which they are printed in the output listing.)

For information on how to use the SOURCE, FLAG, MAP, XREF, and FIPS options, see "Fixing Compile-Time Errors—Advanced Programming."

For information on how to use the DECK and OBJECT options, see "Link-Editing Your Program—Advanced Programming."

For information on how to use the LIST option, see "Fixing Execution-Time Errors—Advanced Programming."

CATALOGING YOUR SOURCE PROGRAM

If you wish, you can catalog your source program so that it's available for future updating or correction.

How you do it depends upon the system you're using.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS2-TSO considerations, see "Using OS/VS2-TSO with VS FORTRAN."

CATALOGING YOUR SOURCE—OS/VS

You can create partitioned data sets for use in your SYSLIB data set.

You can then catalog your source program, and source statement sequences you'll use in INCLUDE statements, as members in that library.

The library in which you catalog the source programs or statement sequences is SYSLIB.

Then when you compile a program using the INCLUDE statement, you must specify SYSLIB in a DD statement.

(See the VS FORTRAN Application Programming: System Services Reference Supplement for additional detail.)

Compiler Option	Produces the Following Output
---	Compilation Headings: Compiler Name and Release Level, Source Program Name, Compilation Date, Listing Page Number
SOURCE	A listing of the source program
XREF	Cross reference information, showing each name and its type and usage; as well as where each name and statement is defined and used in the program
LIST	A listing of the object module—containing the relative location of each generated constant or statement, the name of the source item used in the instruction, plus section headings for: constants and data addresses, common areas, equivalenced variables, address constants, external references, parameter lists, the save area, and generated instructions
MAP	A map of the source program, showing: the program name and size, name usage and type, COMMON block information, and statement label usage and location
FLAG	A listing of error messages at the FLAG level you've chosen: I requests a listing of all messages produced W requests a listing of warning, error, severe error, and abnormal termination messages (return code 4 or higher) E requests a listing of error, severe error, and abnormal termination messages (return code 8 or higher) S requests a listing of severe error and abnormal termination messages (return code 12 or higher) For an explanation of these message codes, see "Diagnostic Message Listing—FLAG Option"
OBJECT	The object module in machine language form, produced for linkage editor input
TERMINAL	Statistics and messages directed to the SYSTERM data set; also produces an indexed summary of statistics and messages for each compilation at the end of all compilations
DECK	The object module in machine language form to be produced as an output data set for punching or for cataloging
---	Compilation statistics: source program name, number of statements compiled, generated object module size (in bytes), the number and severity of error messages produced

Figure 23. Compiler Output Using Explicit Options

CATALOGING YOUR SOURCE—DOS/VSE

You can catalog source programs and source statement sequences you'll use in INCLUDE statements as books in the source statement library, using the CATALS function.

Then, when you compile a program using the INCLUDE statement, you must specify SYSSLB in an ASSGN statement.

(See the VS FORTRAN Application Programming: System Services Reference Supplement for additional detail.)

CATALOGING YOUR OBJECT MODULE—DOS/VSE

You request an object module data set by specifying the DECK compiler option.

You can use the object data set as input to the linkage editor or loader in a later job step, or you can catalog it for later reference.

The data set is a copy of the object module, in card image format, which consists of dictionaries, text, and an end-of-module indicator. (See "Object Module as Link-Edit Data Set—DECK Option" for additional detail.)

Once you've created the object module data set, you can catalog it in a system or private library for future reference.

How you do it depends upon the system you're using.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS2-TSO considerations, see "Using OS/VS2-TSO with VS FORTRAN."

CATALOGING YOUR OBJECT MODULE—OS/VS

You can create partitioned data sets for use in your SYSLIB data set. You can then catalog your object module as a member in that library.

The library in which you catalog your object module is SYSLIB. .

Then, when you link-edit and execute, you must specify SYSLIB in a DD statement.

(See the VS FORTRAN Application Programming: System Services Reference Supplement for additional detail.)

CATALOGING YOUR OBJECT MODULE—DOS/VSE

You can catalog your object module in the relocatable library, using the CATALR function.

Then, when you link-edit and execute, you must specify SYSRLB in an ASSGN statement. (See the VS FORTRAN Application Programming: System Services Reference Supplement for additional detail.)

USING VM/370-CMS WITH VS FORTRAN

You can use the facilities of VM/370-CMS, taking advantage of quick terminal turnaround time, to develop VS FORTRAN programs.

You can compile your programs under VM/370-CMS and link-edit them to run under MVS, OS/VS1, or DOS/VSE; or you can compile, link-edit, and execute them under VM/370-CMS. However, the DOS/VSE version of the VS FORTRAN compiler and library is not supported under CMS/DOS.

THE CP AND CMS COMMANDS

There are CMS and CP commands that help you create and edit your source programs, link-edit your object modules, and execute your load modules. The VM/370-CMS commands you'll use most frequently are shown in Figure 25. Reference documentation for these commands is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

USING YOUR TERMINAL WITH CMS

You must log on your terminal, using the procedures your organization has set up.

You can then use all the CP and CMS commands to develop, test, and run your VS FORTRAN programs.

When you finish a terminal session, you log off in the usual way.

See the IBM Virtual Machine Facility/370: Terminal User's Guide for documentation on terminal usage.

CREATING YOUR SOURCE PROGRAM—CMS EDIT COMMAND

To create a source program file, you use the EDIT command. (Use the EDIT command whenever you want to create a new file and also whenever you want to edit an existing one.)

The EDIT subcommands (such as FILE, INPUT, GETFILE, etc.) help you enter and edit the lines of source code.

To create a source program file, you specify the file type of your source program file as FORTRAN.

For example, to create a source program file named MYPROG, you specify:

```
  edit myprog fortran
```

This creates an empty file for you, with the filename MYPROG, and the filetype FORTRAN. (If MYPROG already exists, the EDIT command retrieves it for you and makes it available for editing.)

You can now enter your source program into the file, line by line, according to the rules for fixed or free form source programs.

Fixed format FORTRAN files contain 80-character records; you use the first 72 characters for FORTRAN statements and continuation lines.

CMS Command	Usage
ACCESS	Activates a virtual disk for your use.
EDIT	Puts you in EDIT mode to create and edit source program and data files, and lets you use the following EDIT subcommands: **FILE** Places a file on your disk; takes you out of EDIT mode. **FMODE** Changes a file's filemode. **FNAME** Changes a file's filename. **GETFILE** Includes an existing file in the file you're creating or editing. **INPUT** Enters INPUT mode and accepts lines as part of the file you're creating or editing. **QUIT** Lets you stop editor processing without making any permanent changes to the file.
EXEC	Executes a file that consists of one or more CMS commands.
FILEDEF	Defines a file and its input/output devices.
FORTVS	Invokes the VS FORTRAN compiler.
GLOBAL	Specifies text libraries to be searched to resolve external references in a program being loaded.
INCLUDE	Specifies additional TEXT files for use during program execution.
LISTFILE	Displays a list of your files.
LOAD	Places a TEXT file in storage and establishes the linkages for execution.
PRINT	Prints a file on the off-line printer.
PUNCH	Punches a card file on the off-line card punch.
RENAME	Changes the filetype, filename, and/or filemode of a file.
RUN	Causes compilation, link-editing, and execution of a source program file.
START	Begins execution of a previously loaded and link-edited file.
TYPE	Types all or part of a file.

Figure 24. VM/370-CMS Commands Often Used with VS FORTRAN

IBM EXTENSION

If you're using free form input, you can enter your source program into the file, line by line, according to the rules for free form source programs.

The maximum line length you can enter is 81 characters; however, your source statements (excluding statement numbers and statement break characters) can be up to 1320 characters long.

You must ensure that sequence numbers do not appear in your free form source (columns 73-80). The use of the filetype FORTRAN (which is necessary for both fixed and free form source) may automatically generate sequence numbers.

END OF IBM EXTENSION

(Reference documentation for creating source programs is given in the VS FORTRAN Application Programming: Language Reference manual.)

USING THE FORTRAN INCLUDE STATEMENT—CMS

If your source programs use the INCLUDE statement, you must specify a FILEDEF command for SYSLIB, to make the library containing the INCLUDE source code available. For INCLUDE statements, the data set records must be unblocked, fixed-length, 80-character records.

1. Create one or more members with a filetype of COPY.

```
edit member1 copy a
INPUT
     common/com1/a1,a2,a3,a4
     common/com2/b1,b2,b3,b4

file
```

2. Create a FORTRAN source program.

```
edit myprog fortran a
input
     include (member1)
     z = a1 * b1
          .
          .
          .
     end

file
```

3. Create a CMS macro library.

```
mac gen test01 member1
```

COMPILING YOUR PROGRAM—USING CMS

If you have used an INCLUDE statement in your source program, you need to define SYSLIB for use by the compiler:

```
filedef syslib disk test01 maclib a(perm
```

If you want to compile MYPROG with the defaults your organization uses, you specify:

```
fortvs myprog
```

If, however, you want to compile MYPROG using nondefault compiler options, specify, for example:

```
fortvs myprog (free flag(e) deck map)
```

which tells the compiler that your source program is in free form, and which gives you a compilation with only E level messages or higher flagged (FLAG(E)), a copy of the object deck (DECK), and a map of names and labels (MAP); all the options you don't specify are the default options.

Reference documentation for the FORTVS command is given in the _VS FORTRAN Application Programming: System Services Reference Supplement_.

COMPILER OUTPUT—CMS

Depending on your organization's compile-time defaults and/or the options you select in your FORTVS command, you may get a LISTING file and/or a TEXT file as output, placed in your disk storage for easy reference.

The LISTING File—CMS

The LISTING file contains the compiler output listing; see "Fixing Compile-Time Errors—Advanced Programming" in Part 2 for an explanation of what the compiler output listing contains and how to use it. It has the filename of your source program, and the filetype LISTING. For example, the file for MYPROG is MYPROG LISTING.

You can display the LISTING file, using the TYPE command:

```
type myprog listing
```

and the listing is displayed at your terminal.

You can get a printed copy of the LISTING file, using the PRINT command:

```
print myprog listing
```

and the listing is printed on the system printer.

The TEXT File—CMS

The TEXT file contains the object code the compiler created from your source program. The TEXT file contents are explained in "Link-Editing Your Program—Advanced Programming" in Part 2.

It is placed in your storage with the filename of your source program and a filetype of TEXT. For example, the file for MYPROG is MYPROG TEXT.

This file remains in your disk storage until you erase it.

You can link-edit the TEXT file under any of the systems that VS FORTRAN supports to get a load module (or phase).

MAKING LIBRARIES AVAILABLE—CMS GLOBAL COMMAND

Before you link-edit and execute your VS FORTRAN programs, you must make the VS FORTRAN libraries accessible to your disk storage. (Ask your system administrator for the name(s) of the library files you can use.)

For example, if the names of the libraries available are FORTLIB and CMSLIB, you specify:

```
global txtlib fortlib cmslib
```

TXTLIB tells CMS that both FORTLIB and CMSLIB are text libraries available to your programs.

It's a good idea to include a GLOBAL command in your profile EXEC, so that the VS FORTRAN execution-time library modules are always automatically available to you.

LOADING AND EXECUTING YOUR PROGRAM UNDER CMS

To load and execute your program under CMS, use the LOAD and INCLUDE commands to create a load module from one or more object modules (plus any needed VS FORTRAN library modules).

The input object modules must be TEXT files.

USING THE LOAD AND INCLUDE COMMANDS—CMS

You use the LOAD command to create and execute a load module. Input you use consists of your object module, VS FORTRAN library routines, and any other secondary input (such as TEXT files of called subprograms).

For example, if you want to load and execute the TEXT files for MYPROG, and its subprogram SUBPROG, you specify:

```
load myprog (start)
```

Because you've made the TXTLIB available (through the GLOBAL command), CMS link-edits the TEXT files for MYPROG and SUBPROG into a load module and then execution begins (because you specified START in the load command).

If you don't specify the START option in the LOAD command, execution doesn't begin until you issue a START command:

```
start myprog
```

If you don't make the TEXT library available, you must define secondary input, using the INCLUDE command:

```
load myprog
include subprog
start myprog
```

The INCLUDE command tells CMS that SUBPROG is needed as secondary input during execution of MYPROG.

Creating a Nonrelocatable Load Module—GENMOD Command

You can also create a nonrelocatable load module and save it for future reference, using the LOAD, INCLUDE, and GENMOD commands:

```
load myprog
include subprog
genmod myproga
```

This creates a a nonrelocatable load module named MYPROGA on your CMS disk.

You can now invoke MYPROGA, by entering:

```
myproga
```

at the terminal, and your program is executed.

DEFINING SEQUENTIAL AND DIRECT DATA FILES—CMS

When you execute your program, you must make any files it uses available for processing, through a FILEDEF command. The FILEDEF command uses a file identifier for each file.

Before you can process a direct file, you must preformat it. "OS/VS Considerations—Direct Files" tells how to do this.

Specifying a File Identifier—CMS

You must identify every file you use in the following form:

```
filename filetype [filemode]
```

where the filename is the name that identifies the file to the system; the filetype defines the kind of file this is. The filetypes you'll most often use are:

DATA	for your data files
FORTRAN	for all your FORTRAN source programs

The filemode is optional, and can be any valid CMS filemode; you need to specify it only when you want to store the file on disk storage other than your own.

USING THE FILEDEF COMMAND—CMS

The form of the FILEDEF command you use varies, depending on the type of file you're processing: sequential or direct disk, tape, or unit record.

Defining Sequential and Direct Disk Files—CMS

To define sequential and direct disk files, you specify the FILEDEF command as follows:

```
filedef FTxxFyyy DISK filename filetype [filemode] [options]
```

You specify the FTxxFyyy field to agree with the FORTRAN reference numbers in the source program.

For the xx field, see Figure 29 in Part 2 for reference numbers you should specify.

For the yyy field, specify 001 if you're not using multiple files. if you're using multiple files, you can specify 001 through 999.

The options are any FILEDEF options valid for disk files.

See the VS FORTRAN Application Programming: System Services Reference Supplement for additional information.

Defining Tape Files—CMS

To define tape files, you specify the FILEDEF command as follows:

```
filedef FTxxFyyy TAPn [options]
```

You specify the FTxxFyyy field to agree with the FORTRAN reference numbers in the source program:

For the xx field, see Figure 29 in Part 2 for reference numbers you should specify.

For the yyy field, specify 001 if you're not using multiple files. if you're using multiple files, you can specify 001 through 999.

For the n field, you specify any valid tape unit (1 through 4).

The options are any FILEDEF options valid for tape files.

See the VS FORTRAN Application Programming: System Services Reference Supplement for additional information.

Defining Terminal Files—CMS

To define terminal files, you specify the FILEDEF command as follows:

```
filedef FTxxF001 TERMINAL [options]
```

You specify the FTxxF001 field to agree with the FORTRAN reference numbers in the source program.

For the xx field, see Figure 29 in Part 2 for reference numbers you should specify.

You always specify 001 for the yyy field.

The options are any FILEDEF options valid for terminal files.

For input terminal files, your program should always notify you when to enter data; if it doesn't, you may inadvertently cause long system waits.

See the VS FORTRAN Application Programming: System Services Reference Supplement for additional information.

Defining Unit Record Files—CMS

To define unit record files, you specify the FILEDEF command as follows:

For Card Reader Files:

```
filedef FTxxF001 READER [options]
```

For Card Punch Files:

```
filedef FTxxF001 PUNCH [options]
```

For Printer Files:

```
filedef FTxxF001 PRINTER [options]
```

You specify the FTxxF001 field to agree with the FORTRAN reference numbers in the source program:

For the xx field, see Figure 29 in Part 2 for reference numbers you should specify.

You always specify 001 for the yyy field.

The options are any FILEDEF options valid for the type of unit record file you're processing.

See the VS FORTRAN Application Programming: System Services Reference Supplement for additional information.

DEFINING VSAM SEQUENTIAL AND DIRECT FILES—CMS

VS FORTRAN allows you to process VSAM sequential files (using ESDS data sets) and VSAM direct files (using RRDS data sets).

To define and use such VSAM files under CMS, you must first define the file to CMS, then you define a catalog entry for the file. When you want to access the file (to write or read it), you must use job control commands.

Defining a VSAM File to CMS

To define a VSAM file to CMS, you must specify the following commands in the following order:

The DLBL command to define the VSAM cluster.

The EDIT command, to create a file containing the DEFINE CLUSTER command you'll use to create the catalog entry for your file.

The AMSERV command, to execute the DEFINE CLUSTER command, in the file you've created; this creates the VSAM catalog entry.

Using the DLBL Command—CMS

You must execute the DLBL command to name the VSAM cluster and to define the VSAM files you'll use for program input/output. For example, to identify the FILE name used in your DEFINE CLUSTER command, you specify:

```
  dlbl myfile data DSN mastcat (vsam
```

which identifies MYFILE to the system as a VSAM file named MYFILE contained in the catalog MASTCAT.

Creating a VSAM DEFINE Command—CMS EDIT Command

To create a catalog entry, you must issue the EDIT command, giving the name of the entry; the filetype is AMSERV:

```
  edit catfile amserv
```

This puts you in edit mode, and you can then enter the Access Method Services DEFINE CLUSTER command as data into the file; for example:

```
DEFINE CLUSTER (NAME(MASTCAT)
        FILE(MYFILE)
        VOLUME(123456)
        NONINDEXED
        RECORDS(20)
```

which defines a catalog entry in MASTCAT, for a VSAM sequential file (NONINDEXED) named MYFILE, on volume 123456, with 20 tracks of space allocated.

Making the Catalog Entry—AMSERV Command

Once you've entered the DEFINE CLUSTER command in an AMSERV file, you can request Access Method Services to process the command for you, using the AMSERV command:

```
  amserv catfile
```

This command sends the DEFINE CLUSTER command to Access Method Services for processing, and entry into MASTCAT.

See the <u>VS FORTRAN Application Programming: System Services Reference Supplement</u> for additional information.

Creating and Processing VSAM Files—CMS

When you execute a FORTRAN program to create or process a VSAM file, you define the file in a DLBL command: For example, to process MYFILE in a FORTRAN program called MYPROG, you specify:

```
  dlbl myfile data DSN mastcat (vsam
  start myprog text
```

When MYPROG is executed, the DLBL statement makes MYFILE available to the program.

USING OS/VS2-TSO WITH VS FORTRAN

You can use the facilities of OS/VS2-TSO, taking advantage of quick terminal turnaround time, to develop VS FORTRAN programs.

You can compile your programs under OS/VS2-TSO and link-edit them to run under MVS, or you can compile, link-edit, and execute them under OS/VS2-TSO.

USING THE TSO COMMANDS

The TSO commands help you create and edit your source programs, link-edit your object modules, and execute your load modules. The TSO commands you'll use most frequently are shown in Figure 25. Reference documentation for these commands is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

USING YOUR TERMINAL WITH TSO

You must log on your terminal, using the procedures your organization has set up.

You can then use all of the TSO commands to develop, test, and run your VS FORTRAN programs.

When you finish a terminal session, you log off in the usual way.

See the OS/VS2 TSO Terminal User's Guide for documentation on terminal usage.

CREATING YOUR SOURCE PROGRAM—TSO EDIT COMMAND

To create a source program file, you use the EDIT command. Use the EDIT command whenever you want to create a new file and also whenever you want to edit an existing one.

(You can also use the Structured Programming Facility (SPF) editor to create source program files. You can use SPF to allocate sequential files or partitioned data sets, although this isn't necessary, since the ALLOCATE and ATTRIB commands are also available.)

The EDIT subcommands (such as COPY, INPUT, INSERT, etc.) help you enter and edit the lines of source code.

To create a source program file, you can specify the descriptive qualifier of your source program file as FORT. Alternatively, you can specify the descriptive qualifier as DATA.

For example, to create a source program file named MYPROG, you specify:

```
edit myprog.fort

  or

edit myprog.data
```

This creates an empty data set for you, with the name MYPROG, and the descriptive qualifier FORTRAN or DATA. (If MYPROG.FORT already exists, the EDIT command retrieves it for you and makes it available for editing.)

TSO Command	Usage	
ALLOCATE	Allocates data sets needed for compilation, link-editing, or execution.	
ATTRIB	Builds a list of data sets you intend to allocate dynamically.	
CALL	Invokes compiler, linkage-editor, or load module for execution.	
DELETE	Deletes one or more data set entries or one or more members of a partitioned data set.	
EDIT	Puts you in EDIT mode to create and edit source program and data files, and lets you use the following EDIT subcommands:	
	COPY	Copies records within the data set you are editing.
	END	Ends edit mode with or without saving the data set.
	HELP	Gives information about EDIT subcommands.
	INPUT	Enters INPUT mode and accepts lines as part of the data set you're creating or editing.
	INSERT	Inserts one or more lines of data into the data set that you're creating or editing.
	LIST	Displays one or more lines of your data set at the terminal.
	MOVE	Move one or more records in your data set to another position in the same data set.
	SAVE	Ends EDIT mode and places the current data set on disk storage.
FREE	Deallocates files allocated for a job.	
HELP	Gives information about commands other than EDIT subcommands.	
LINK	Converts one or more object modules into a load module.	
LOADGO	Loads one or more object modules into storage and executes them.	
STATUS	Checks execution status of a submitted batch job.	
SUBMIT	Submits a JCL file to MVS to run as a batch (background) job (requires SUBMIT logon capabilities).	
TEST	Tests an object program for proper execution and locates programming errors.	

Figure 25. OS/VS2-TSO Commands Often Used with VS FORTRAN

You can now enter your source program into the file, line by line, according to the rules for fixed or free form source programs.

Fixed format FORTRAN files contain 80-character records; you use the first 72 characters for FORTRAN statements and continuation lines.

IBM EXTENSION

If you're using free form input, you can enter your source program into the file, line by line, according to the rules for free format source programs.

The maximum line length you can enter is 80 characters; however, your source statements (excluding statement numbers and statement break characters) can be up to 1320 characters long.

This is particularly handy when you're using a terminal—you don't have to pay attention to card image restrictions.

END OF IBM EXTENSION

(Reference documentation for fixed-form and free-form source program input is given in the VS FORTRAN Application Programming: Language Reference manual.)

COMPILING YOUR PROGRAM—TSO ALLOCATE AND CALL COMMANDS

To compile your program, use the ALLOCATE and CALL commands.

Allocating Compilation Data Sets—TSO ALLOCATE Command

First, you allocate the data sets you'll need for compilation as shown in Figure 26.

```
allocate dataset(myprog.fort) file(sysin) old
READY
allocate dataset(myprog.list) file(sysprint) new block(120) space(60,10)
READY
allocate dataset(myprog.obj) file(syslin) new block(80) space(100,10)
READY
allocate dataset(*) file(systerm)
READY
```

Figure 26. Allocating TSO Compilation Data Sets

For any compilation, you must allocate the SYSIN AND SYSPRINT data sets.

Allocate the SYSLIN data set only if you wish to produce an object module (you've specified the OBJECT compiler option).

Allocate the SYSTERM data set only if you wish to receive diagnostics at the terminal (you've specified the TERMINAL option).

You can enter these ALLOCATE commands in any order. However, you must enter all of them before you invoke the FORTRAN compiler.

Requesting Compilation—CALL Command

Once you've allocated the data sets you'll need, you can issue a CALL command, requesting compilation.

You can request compilation, using the default compiler options:

```
call 'sys1.linklib(vsfort)'
```

or you can request one or more compiler options explicitly:

```
call 'sys1.linklib(vsfort)' 'free,term,source,map,list,object'
```

which tells the compiler that:

- FREE—your source program is in free form.
- TERM—diagnostic messages are to be directed to your terminal.
- SOURCE—the source program is to be printed in the output listing.
- MAP—a storage map is to be printed in the output listing.
- LIST—the object program is to be printed in the output listing.
- OBJECT—an object module is to be produced.

Reference documentation for the CALL command is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

COMPILER OUTPUT—TSO

Depending on your organization's compile-time defaults and/or the options you select in your CALL command, you may get a LIST file and/or an OBJ file as output, placed in your disk storage for easy reference, under the name(s) you specified in the ALLOCATE command.

The LIST File—TSO

The LIST file contains the compiler output listing; see "Fixing Compile-Time Errors—Advanced Programming" in Part 2 for an explanation of what the compiler output listing contains and how to use it.

It has the name of your source program, and the qualifier LIST. For example, the qualified name for MYPROG is MYPROG.LIST.

The OBJ File—TSO

The OBJ file contains the object code the compiler created from your source program. The OBJ file contents are explained in "Link-Editing Your Program—Advanced Programming" in Part 2.

It is placed in your storage with the name of your source program and the qualifier OBJ. For example, the qualified name for MYPROG is MYPROG.OBJ.

This file remains in your disk storage until you erase it, using the DELETE command.

You can link-edit the OBJ file under any of the systems that VS FORTRAN supports to get a load module.

LINK-EDITING AND EXECUTING YOUR PROGRAM UNDER TSO

To link-edit and execute your program under TSO, use the LINK command to create a load module from one or more object modules (plus any needed VS FORTRAN library modules), and then use the CALL command to execute the load module.

The input object module must be OBJ files, for example:

userid.name.OBJ

LINK-EDITING YOUR PROGRAM—TSO LINK COMMAND

You use the LINK command to create and execute a load module. Input you use consists of your object module, VS FORTRAN library routines, and any other secondary input (such as OBJ files of called subprograms).

For example, if you want to load and execute the OBJ files for MYPROG, and its subprogram SUBPROG, you specify:

```
link (myprog,subprog) load(myprog) lib('vfortlib')
```

When the commands are executed, the OBJ files for MYPROG and SUBPROG are link-edited together into a load module, and then execution begins.

You must request the linkage editor to search the library to resolve external references. In the last example, you are therefore requesting a search of SYS1.FORTLIB.

Linkage Editor Listings—TSO LINK Command

You can also use the LINK command to specify linkage editor options. In the last example, you can request the listings to be printed, either on the system printer or at your terminal:

On the System Printer:

```
link (myprog,subprog) lib('vfortlib') load(myprog) print
```

The qualified name of the file to be sent to the system printer is MYPROG.LINKLIST. To print the file, you must use a PRINT command, or the SPF HARDCOPY command.

At Your Terminal:

```
link (myprog,subprog) lib('vfortlib') load(myprog) print(*)
```

When you specify PRINT(*), the linkage editor listings are displayed at your terminal.

EXECUTING YOUR PROGRAM—TSO

To execute your program, use the CALL command.

USING THE CALL COMMAND—TSO LOAD MODULE EXECUTION

For example, to execute MYPROG.LOAD, you specify the ALLOCATE commands needed to allocate the input and output data sets it uses, as well as any work data sets.

Then you issue the CALL command, as follows:

```
allocate dataset(myprog.indata)   file(F05FT001)  (as needed)
READY
allocate dataset(myprog.outdata)  file(F06FT001)  (as needed)
READY
allocate dataset(myprog.workfil)  file(F10FT001)  (as needed)
READY
call myprog
```

and program TEMPNAME from file MYPROG.LOAD is executed.

Once program execution is complete, and if you no longer need them, you should issue the DELETE command to free the disk space used by the data sets you've named in the ALLOCATE commands and in the CALL command:

```
delete (myprog.indata myprog.outdata myprog.workfil)
READY
```

If you do need them, don't issue the DELETE command; you can then reuse the data sets as necessary.

Using the Loader—TSO LOADGO Command

Using the LOADGO command, you can invoke the loader program to link-edit and execute your program all in one step. This is efficient when you have several object modules you want combined into one load module for a quick test. When you use the LOADGO command, the load module created is automatically deleted when program execution ends.

First, you must allocate any needed data sets, as outlined under "Allocating Compilation Data Sets—TSO ALLOCATE Command."

Then you issue the LOADGO command. In this example, you're linking and executing the MYPROG object module:

```
loadgo (myprog) lib('vfortlib')
```

The VFORTLIB operand makes the SYS1.FORTLIB data set available to the loader program. The loader program can then resolve any external references in MYPROG, and load the needed object modules.

You can also use the LOADGO command to execute a link-edited load module, for example, one named MYPROG:

```
loadgo myprog tempname
```

You can also use the LOADGO command to specify loader options. In the last example, you can request a load module map and the listings to be printed, either on the system printer or at your terminal:

On the Printer File:

```
loadgo (myprog) map print
```

The qualified name of the file sent to the system printer is MYPROG.LOADLIST.

At Your Terminal:

```
loadgo (myprog) map print(*)
```

When you specify PRINT(*), the loader listings are displayed at your terminal.

FIXING EXECUTION ERRORS UNDER TSO

When you're developing programs using TSO, you can make use of all the FORTRAN debugging aids described in "Fixing Execution-Time Errors—Advanced Programming."

You can't use the TEST command to debug your source programs.

However, you can use the TSO TEST command, together with its associated subcommands, to debug your object program.

The easiest way to use TEST is to establish the point in your program where an abnormal termination occurred.

For best results, you should be familiar with the basic assembler language and addressing conventions.

Reference documentation on using the TEST command is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

REQUESTING COMMAND PROCEDURE PROCESSING UNDER TSO

You can create a command procedure (CLIST) for a number of different processing options under TSO. This is useful, because you can preallocate all the options you need once, when you create the command procedure. Then, every time you execute the command procedure, there's no need to respecify the options.

You can create command procedures to process your jobs either in the foreground or in the background.

COMMAND PROCEDURES FOR FOREGROUND PROCESSING

To create a command procedure to link-edit and run a FORTRAN program in the foreground, do the following:

1. Edit a file named LINK.MYPROG.CLIST, which contains:

   ```
   LINK MYPROG LIB('VFORTLIB') LOAD(MYPROG(MYNAME) TEST
        LIST LET XREF PRINT(MYPROG)
   ```

   These options will allow you to use the TEST command (TEST), list the link-edit control statements (LIST), allow the procedure to complete execution even if there are errors (LET), produce a cross-reference table (XREF), and produce a linkage-editor listing on the MYPROG.LIST data set (PRINT).

2. Use the SAVE subcommand to save the command procedure.

3. To execute the link and run command procedure, enter:

   ```
   ex link.myprog
   ```

   and press ENTER.

   This tells TSO to create a load module, MYNAME, in file userid.MYPROG.LOAD

To run a prevously link-edited program, do the following:

1. Edit a file name RUN.MYPROG.CLIST which contains:

   ```
   ALLOC FILE(FTxxFyyy)    (work and data files
         .                  as needed)
         .
         .
   CALL MYPROG(MYNAME)
   ```

2. Use the SAVE subcommand to save the command procedure.

3. Use the END subcommand to exit from the EDIT command.

4. To execute the run command procedure, enter:

   ```
   ex run.myprog
   ```

   and press ENTER.

   This tells TSO to execute MYNAME in file userid.MYPROG.LOAD in the foreground.

COMMAND PROCEDURES FOR BACKGROUND EXECUTION

If your source programs are small, foreground execution can be quite convenient. However, if your programs will take some time to compile and/or to execute, you may prefer to batch process them in the background. This frees your terminal for other work while the batch job is running.

To create a command procedure for background compilation, do the following:

1. Edit a file named COMPILE.MYPROG.CNTL, placing in it the following job control statements:

   ```
   //userid JOB (acct. information),'yourname', other information
   //COMPILE EXEC VSFORTC
   //VSFORT.SYSIN DD DSN=userid.MYPROG.FORT,DISP=SHR
   ```

2. Use the SAVE subcommand to save the command procedure.

3. To execute the run command procedure, enter:

   ```
   submit compile.myprog
   ```

   and press ENTER.

   This tells TSO to compile MYPROG in file userid.MYPROG.FORT in the background.

To create a command procedure for background compilation, link-editing, and execution, do the following:

1. Edit a file named RUN.MYPROG.CNTL, placing in it the following job control statements:

   ```
   //userid JOB (acct. information),'yourname', other information
   //RUN EXEC FORTCLG
   //GO.FTnnFxxx DD DSN=userid.MYPROG.FORT,
                    DISP=SHR
   /*
   //GO.FTxxFyyy DD DSN=userid.MYPROG.DATA,
                    DISP=SHR    (other files as needed)
   ```

2. Use the SAVE subcommand to save the command procedure.

3. Use the END subcommand to exit from the EDIT command.

4. To execute the compile-link-execute command procedure, enter:

   ```
   submit run.myprog
   ```

   and press ENTER.

   This tells TSO to compile, link-edit, and execute MYPROG in file userid.MYPROG.FORT in the background.

TSO FILE NAMING CONVENTIONS

When you're creating or processing files under TSO, you must use the TSO naming conventions, as follows:

```
[identification.]name.qualifier
```

[identification.]
is the identification you supply in your LOGON command, or that you assign with the PROFILE command.

If you omit the identification, your LOGON identification is assumed.

name.
is the name you give the file.

qualifier
identifies the contents of the file. Qualifiers useful for FORTRAN are:

Qualifier	Has File Contents
CLIST	TSO commands and subcommands
DATA	Data files you're creating or processing
FORT	FORTRAN source programs—in either fixed or free form
LIST	FORTRAN compiler output listings
LOAD	FORTRAN load modules
LOADLIST	Loader output listings
OBJ	FORTRAN object modules

If the file you're referring to is a sequential or direct file, you can use the name and the qualifier alone to identify the data set. For example:

```
myfile1.data
```

If the file you're referring to is a member of a partitioned data set, you can add the member-name in parentheses after the qualifier. For example, to refer to member-name COMMON1 in data set INCLIB1, you specify:

```
inclib1.data(common1)
```

FILE IDENTIFICATION—TSO ALLOCATE COMMAND

Before compiling, link-editing, or executing your program, you must allocate the data sets you'll need, using the ALLOCATE command. For example, you could allocate the following files when processing a source program named MYPROG:

For the Source Program as Compiler Input:

```
allocate dataset(myprog.fort) file(sysin) old
```

This ALLOCATE command tells TSO that the file named MYPROG.FORT is an existing file (OLD), available on the SYSIN data set.

For Compiler Output Listings:

```
allocate dataset(myprog.list) file(sysprint) new   -
                    block(120)     space(60,10)
```

This ALLOCATE command tells TSO that the file named MYPROG.LIST is a new file (NEW), to be produced on the SYSPRINT data set. The line length is 120 characters; the primary space allocation is 60 lines.

To print the listing, use the PRINT command.

For an Object Deck:

```
allocate dataset(myprog.obj) file(syspunch) new   -
                    block(80) space (120,20)
```

This ALLOCATE command tells TSO that the file named MYPROG.OBJ is a new file (NEW), to be produced on the SYSPUNCH data set. The record length is 80 characters.

For the Object Module:

```
allocate dataset(myprog.obj) file(syslin)  -
                           new block(80) space(100,10)
```

This ALLOCATE command tells TSO that the file named MYPROG.OBJ is a new file (NEW), to be produced on the SYSLIN data set. The record size (and block size) must be 80 characters. The space you can specify as any size you need.

For Terminal Input/Output:

```
allocate dataset(*) file(systerm)
```

This ALLOCATE command tells TSO that the file identified by the asterisk (*) is available on the SYSTERM data set. You can then use the terminal to receive error mesage output.

(The listing output is on the SYSPRINT data set.)

For Program Data Sets:

```
allocate dataset(identifier.name.data) file(FnnFTxxx)
```

This ALLOCATE command tells TSO that the file identified by the qualified name is available on the FnnFTxxx data set. Valid values for nn and xxx are documented in Figure 29.

Reference documentation for the ALLOCATE command is given in the VS FORTRAN Application Programming: System Services Reference Supplement. For valid values for FnnFTxxx, see "Defining Files—OS/VS DD Statement" in Part 2.

Before you can load a direct file, you must preformat it. "OS/VS Considerations—Direct Files" tells how to do this.

DEFINING VSAM SEQUENTIAL AND DIRECT FILES—TSO

VS FORTRAN allows you to process VSAM sequential files (ESDS data sets) and VSAM direct files (RRDS data sets).

To define and use VSAM files under TSO, you can enter and execute the Access Method Services commands (such as DEFINE CLUSTER) directly from your terminal. See your system administrator for further information about catalogs and volumes.

When you want to access the file (to write or read it), you must use the ALLOCATE command, if you're referencing the VSAM file using FnnFTyyy. If you aren't, you don't need an ALLOCATE command, because you are accessing the file directly.

SYSTEM CONSIDERATIONS UNDER OS/VS2-TSO

When you're developing programs using the TSO facilities, your FORTRAN programs must not require system actions for which you are not authorized (for example, protected data set access or volume mounting).

In addition, if your FORTRAN programs make use of assembler subroutines, there are a few system considerations you must take into account, when coding the assemler routines, as follows:

1. Your assembler subprograms must execute as nonauthorized problem programs.

2. Your assembler subprograms must use standard MVS system service interfaces.

3. Your assembler subprogram must not use TSO-specific storage subpools.

4. The address spaces your assembler subprograms use must not be sensitive to MVS control block structures.

5. Your assembler subprogram must use only the data set characteristics available through the ALLOCATE command.

FIXING COMPILE-TIME ERRORS—ADVANCED PROGRAMMING

VS FORTRAN gives you a good deal of assistance in finding and correcting errors it detects during compilation. The compiler output listing gives you much useful information to aid you in debugging.

In addition, if you're creating programs that must conform to 1978 Standard FORTRAN rules, you can use the Standard Language Flagger to tell you when you've used program elements that don't conform.

USING THE COMPILER OUTPUT LISTING

The compiler output listing is designed to help you pinpoint any errors you've made in your source program. The listing is described in the following sections, together with hints on how to use it.

COMPILATION IDENTIFICATION

This portion of the listing helps you identify each run you make, even separate runs on the same day.

The heading on each page of the output listing gives the name of the compiler and its release level, the name of the source program, and the date and time of the run in the format:

```
      DATE: month day, year    TIME: hour.minute.second
```

That is, for example:

```
      DATE: MAY 1, 1981     TIME: 14.31.01
```

The TIME given is the time the job was completed. The TIME is shown on a 24-hour clock; that is, 14.31.01 is the equivalent of 2.31.01 PM.

The first page of the listing also shows the compiler options (default and explicit) in effect for this compilation.

Note: Your organization may show the date as:

```
  year month day
```

instead of the format shown above.

SOURCE PROGRAM LISTING—SOURCE OPTION

Use the source listing for desk checking to make sure that your source statements conform to VS FORTRAN syntax. The internal sequence numbers (which the compiler provides for you) help you identify the statements causing diagnostic messages.

The statements printed in the source program listing are identical to the FORTRAN statements you submitted in the source program, except for the addition of internal sequence numbers (ISN) as a prefix. Figure 27 shows an example of the source program listing.

Note: When this program is executed, the diagnostic messages shown in Figure 28 are produced.

```
REQUESTED OPTIONS (EXECUTE):  MAP
REQUESTED OPTIONS (PROCESS):  LIST XREF MAP
OPTIONS IN EFFECT: LIST MAP XREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRACE
                   OPTIMIZE(0) LANGLVL(77) NOFIPS FLAG(I) NAME(MAIN   ) LINECOUNT(60)
                        *....*...1.......2.......3.......4.......5.......6.......7.*......8

                        C SAMPLE PROGRAM TO DEMONSTRATE THE NEW VS FORTRAN
 ISN        1                 REAL*8 PI
 ISN        2                 PARAMETER (PI = .314159265D1)
 ISN        3                 COMPLEX*8     C8V, C8A, C8B
 ISN        4                 COMPLEX*32    C32
 ISN        5                 LOGICAL*1     L1
 ISN        6                 REAL*8        R8A, R8V, R8VNU
 ISN        7                 CHARACTER*14  CHAR14

 ISN        8                 EQUIVALENCE  (R4A(4), R8A(2), C32(1,1))
 ISN        9                 EQUIVALENCE  (I2(3), L1)

 ISN       10                 DIMENSION     R8A(7), C32(4,5), R4A(11), I2(3)

 ISN       11                 COMMON /COM1/    R4A, C8A
 ISN       12                 COMMON /COM2/    L1,  C8B

 ISN       13          111    FORMAT('1OUTPUT FOR ', A14, 14F10.7, E15.7, 2(F20.16))

 ISN       14                 DATA A2/3.14159/
 ISN       15                 DATA CHAR14/'SHARE PROGRAM '/

 ISN       16                 ASSIGN 111 TO J

 ISN       17                 A1 = (A2 + R8A(2))*3

 ISN       18                 IF (A1 .EQ. 0) GOTO 200

 ISN       19                 IF (A2 .EQ. 0) GOTO 200

 ISN       20                 DO 100 I = 1,7
 ISN       21                    IF (L1)
                             1      R8V =  R8V + R8A(I) + (.0007, .0021) + FLOAT(I)
 ISN       23          100    CONTINUE

 ISN       24                 CALL CXSUB(*300,R8V,A1,PI)

 ISN       25                 R8A = 1.0002 + R8A(1)

 ISN       26          200    WRITE(6,J) CHAR14, R8A, A1, C32

 ISN       27                 DATA C8V/(2.540005, 2.781828)/

 ISN       28          300    PAUSE 'THE END'
 ISN       29                 END
```

Figure 27. Source Program Listing Example—SOURCE Option

DIAGNOSTIC MESSAGE LISTING—FLAG OPTION

If the level of the message is greater than or equal to that you've specified in the FLAG option and there are syntax errors in your VS FORTRAN source program, the compiler detects them and gives you a message. The messages are self explanatory, making it easy to correct your syntax errors before recompiling your program.

Examples of VS FORTRAN messages are shown in Figure 28.

```
*** VS FORTRAN ERROR MESSAGES ***

IFX1027I   RPLC   12(S)    27    NON-SUBSCRIPTED ARRAY NAME APPEARS AS LEFT-OR-EQUAL
                                 SIGN VARIABLE. SPECIFY A SUBSCRIPTED ARRAY NAME
                                 OR A VARIABLE NAME.

IFX2323I   COMN    4(W)          VARIABLE "R8A" IN COMMON "COM1" IS INEFFICIENTLY
                                 ALIGNED. VARIABLES SHOULD BE ALIGNED ON
                                 APPROPRIATE BYTE BOUNDARIES.

IFX2323I   COMN    4(W)          VARIABLE "C32" IN COMMON "COM1" IS INEFFICIENTLY
                                 ALIGNED. VARIABLES SHOULD BE ALIGNED ON
                                 APPROPRIATE BYTE BOUNDARIES.

IFX2332I   COMN   12(S)          THE VARIABLE "L1" WILL CAUSE COMMON "COM2" TO EXTEND
                                 TO THE LEFT BECAUSE OF ITS POSITION IN EQUIVALENCE
                                 STATEMENT AT ISN "9". CHECK VARIABLE PLACEMENT.

IFX2323I   COMN    4(W)          VARIABLE "C8B" IN COMMON "COM2" IS INEFFICIENTLY
                                 ALIGNED. VARIABLES SHOULD BE ALIGNED ON
                                 APPROPRIATE BYTE BOUNDARIES.
```

Figure 28. Examples of Compile-Time Messages—FLAG Option

Messages you'll get are all in the following format:

    IFXnnnnI mmmm level [isn] 'message text'

where each one of the areas has the following meaning:

IFX The message prefix identifying all VS FORTRAN compiler messages

nnnnI The unique number identifying this message

mmmm The identifier for the compiler module issuing the message

level The severity level of the error diagnosed. Compiler diagnostic messages are assigned severity levels as follows:

0(I) Indicates an informational message; messages at the I level merely give you information about the source program and how it was compiled.

The severity code is 0 (zero).

4(W) Is a warning message; it usually tells you that a minor error, which does not violate the VS FORTRAN syntax, was detected.

The severity code is 4.

If no messages produced exceed this level, you can link-edit and execute the compiled object module.

8(E) Is an error message; usually, a VS FORTRAN syntax error was detected. The compiler makes a corrective assumption, and completes the compilation.

The severity code is 8.

It's possible that the program will execute correctly.

12(S) Is a serious error message; an error was detected which violates VS FORTRAN syntax, and for which the compiler could make no corrective assumption.

The severity code is 12.

You shouldn't attempt execution, except possibly for debugging purposes.

16(U) Is an abnormal termination message; an error was detected which stopped the compilation before it could be completed.

The severity code is 16.

[isn] gives the internal sequence number of the statement in which the error occurred, if the internal sequence number can be determined.

'message-text' explains the source program error that was detected.

OS/VS Cataloged Procedures and Compiler Message Codes

Unless you increase the permissible condition code in the COND parameter of the compilation EXEC statement, severity levels higher than level 4 prevent link-edit processing.

DOS/VSE Message Code Considerations

Severity levels higher than level 4 prevent link-edit processing.

USING THE MAP AND XREF OPTIONS

The MAP and XREF compiler options, described in the following sections, are a big aid in debugging your programs—both for fixing compile-time errors and also for fixing execution-time errors. You can use them to cross check for the following common source program problems:

- Are all variables defined as you expected?
- Are variables misspelled?

  If you've declared all variables, then the following are suspect:

  1. Unreferenced variables
  2. Variables referenced in only one place

- Are all referenced variables set before they're used? (Except for variables in COMMON, parameters, initialized variables, etc.)
- Are one or more variables unexpectedly equivalenced with some other variable?
- Are there unreferenced labels? (If there are, you may at some point have miskeyed a label.)
- Have you accidentally redefined one of the standard library functions? (For example, through a statement function definition.)
- Are the types and lengths of arguments correct across subroutine calls? (You'll need both listings for this.)

- For a variable passed to the main entry of a subroutine, have you inadvertently altered it at a subordinate entry point?

These questions and more about your source program syntax and logic can be answered through the information supplied you by the Source Program Map and the Cross Reference Dictionary.

SOURCE PROGRAM MAP—MAP OPTION

If you've specified the MAP option, the storage map shows you how you've used each item you've defined in your program; this can help you figure out obscure syntax or logic errors that aren't immediately apparent from the source listing.

A storage map shows the use made of each variable, statement function, subprogram, or intrinsic function within a program. An example of a storage map is given in Figure 29.

```
STORAGE MAP
TAG: SET(S), REF'D(F), USED AS ARG(A), COMN(C), EQUV(E), STMT.FUNCT.(T), SUBPROG(X), NAMED CONSTANT(K), INITIAL VALUE(I)
PROGRAM NAME: MAIN  .   SIZE OF PROGRAM:     350 HEX BYTES.
 NAME   MODE   TAG  ADDR.          NAME    MODE   TAG   ADDR.        NAME    MODE   TAG   ADDR.        NAME    MODE   TAG   ADDR.
 A1       R*4  SFA   000140         A2        R*4  FI     000144      CHAR14  CHAR   FI    000148      CXSUB           FX    000000
 C32     C*32  FCE   00000C         C8A       C*8  C      UNREFD      C8B       C*8  C     UNREFD      C8V       C*8   I     UNREFD
 FLOAT    R*4  X     UNREFD         I         I*4  FA     000138      I2        I*4  CE    UNREFD      J         I*4   F     00013C
 L1       L*1  FCE   000000         PI        R*8  FK     000054      R4A       R*4  CE    UNREFD      R8A       R*8   FCE   000004
 R8V      R*8  SFA   000130         R8VNU     R*8         UNREFD      VSCOM#         FX    000000      VSERH#          FX    000000

COMMON INFORMATION
  NAME:   COM1.   SIZE:    28C HEX BYTES.      (E) - EQUIVALENCED
    NAME       MODE     DISPL.        NAME       MODE     DISPL.       NAME       MODE     DISPL.       NAME       MODE     DISPL.
    R8A(E)      R*8     000004        C32(E)     C*32     00000C       R4A(E)      R*4     000000       C8A         C*8     00002C

  NAME:   COM2.   SIZE:      9 HEX BYTES.      (E) - EQUIVALENCED
    NAME .     MODE     DISPL.        NAME       MODE     DISPL.       NAME       MODE     DISPL.       NAME       MODE     DISPL.
    I2(E)       I*4     FFFFF8        L1(E)       L*1     000000       C8B         C*8     000001

LABEL INFORMATION.          (NN NNN NNNNN IS A GENERATED LABEL)
    LABEL      DEFINED  ADDR.        LABEL      DEFINED  ADDR.        LABEL      DEFINED  ADDR.        LABEL      DEFINED  ADDR.
          100       23  000292              111       13  000030              200       26  0002CE              300       28  000312
01 000 00001         1  0001CC    01 000 00019        19  0001FE    01 000 00020        20  00020C    01 000 00022        22  000228
01 000 00025        25  0002C4    26 001 00020        20  000214    02 002 00020        23  0002A8
```

Figure 29. Example of a Storage Map—MAP Option

The following paragraphs describe each area of a storage map, such as that shown in Figure 29.

The first line of a storage map gives the name and size of the source program; the size is given in hexadecimal format.

NAME Column

The first column is headed NAME. It shows the name of each variable, statement function, subprogram, or implicit function in the program.

MODE Column

The second column is headed MODE—it gives the type and (except for character items) length of each name, in the format:

type*length

where the <u>type</u> can be:

C	for complex
CHAR	for character (length not displayed)
I	for integer
L	for logical
R	for real or double precision

TAG Column

The third column is headed TAG. It displays use codes for each name and variable. The use codes are:

A	for a variable used as an actual argument in a parameter list
C	for a variable in a COMMON block
E	for a variable in an equivalenced block
F	for a variable whose value was referred to during some operation
I	for a variable specified with an initial value
K	for a constant
S	for a variable whose value was set during some operation
T	for a statement function
X	for an external function

Address Column

The fourth column is headed ADDR—it gives the relative address assigned to a name. (External functions, statement functions, and COMMON block references have a relative address of 00000.)

For unreferenced variables, this column contains the letters UNREFD instead of a relative address.

COMMON Block Maps—MAP Option

If your source program contains COMMON statements, you'll also get a storage map for each COMMON block.

The map for a COMMON block contains much the same kind of information as for the main program. The DISPL column shows the displacement from the beginning of the COMMON block.

Any equivalenced COMMON variable is listed with its name followed by (E); its displacement (offset) from the beginning of the block is also given.

Statement Label Map—MAP Option

The MAP option also gives you a statement label map, a table of statement numbers used in the program. The label map shows the following forms of statement numbers:

- Source statement labels—as entered
- Compiler-generated statement labels, in the form:

  aa bbb ccccc

  where

  aa
  is the identification code that indicates which compiler module caused the label to be generated. The codes are shown in Figure 30.

  bbb
  is the sequence number within internal sequence number (ISN) ccccc, starting with 000.

  ccccc
  is the ISN of the statement at which the label is generated.

- FORMAT statement labels—as entered

It also gives you the internal sequence number (ISN) for the statement in which the label is defined and the address assigned to the label.

Code	Module
01	IFX1CNTL
02	IFX1DODO IFX1ENDO
03	IFX1LOGL
04	IFX1IFTH
05	IFX1ELSF
06	IFX1ELSE
07	IFX1PRNS
08	IFX1IMPD
0B	IFX1RELS
0C	IFX1IOST
0D	IFX1IOMN
21	IFX1FORM

Figure 30. Module Identification Codes

SOURCE PROGRAM CROSS-REFERENCE DICTIONARY—XREF OPTION

If you've specified the XREF option, the cross-reference dictionary shows you where in your program you used each item you defined. This gives you a record of what interactions between items you can expect; it also shows up conflicting usages of items.

During the later debugging stages, when you're debugging execution errors, it can also be of great assistance in detecting logic errors.

Figure 31 shows an example of a cross reference listing.

A cross-reference dictionary shows the names and statement labels in the source program, together with the internal statement numbers in which they appear.

Data Item Dictionary—XREF Option

The first line of the data item cross-reference dictionary defines the codes used in the TAG column.

From left to right, the subsequent columns give you the following information:

NAME COLUMN: Names are listed in alphabetic order.

MODE COLUMN: Each name is followed in the second column, the column headed MODE, by its type, in the same format as for the MAP option.

TAG COLUMN: The type for each name is followed in the third column, the column headed TAG, by its status, which can be:

A	an array
C	an item in COMMON
D	a dummy argument
E	an equivalenced item
F	a statement function
I	an intrinsic function
K	a named constant
T	an item defined in an explicit type statement
X	an external function

DECLARED COLUMN: The DECLARED column gives the internal sequence number where the data item is defined.

REFERENCES COLUMN: The REFERENCES column gives the internal sequence number of each statement in the source program where the data item is referred to.

If there are no references within the program, this column contains UNREFERENCED.

Statement Label Dictionary—XREF Option

In the statement label dictionary, the following columns are defined:

LABEL COLUMN: Statement labels, including compiler-generated labels, are listed in ascending order.

TAG COLUMN: The LABEL column is followed in the second column, the column headed TAG, by its status, which can be:

A	used as an argument
B	an object of a branch
F	label for a FORMAT statement

```
CROSS REFERENCE DICTIONARY

TAG: ARRAY(A), COMMON(C), EQUIV(E), DUMMY ARG(D), NAMED CONSTANT(K), STMT.FUNCT(F), EXT.FUNCT(X), INTR.FUNCT(I), EXPL.TYPE(T)

 NAME    MODE   TAG    DECLARED REFERENCES

A1        R*4                    17    18    24    26
A2        R*4                    14    17    19
CHAR14   CHAR   T            7   15    26
CXSUB           X                24
C32      C*32   AET          4    8    10    26
C8A       C*8   CT           3   11
C8B       C*8   CT           3   12
C8V       C*8   T            3   27
FLOAT     R*4   I                22
I         I*4                    20    22    22
I2        I*4   AE                9    10
J         I*4                    16    26
L1        L*1   CET          5    9    12    21
PI        R*8   KT           1    2    24
R4A       R*4   ACE               8    10    11
R84       R*8   AET          6    8    10    17    22    25    26
R8V       R*8   T            6   22    22    24
R8VNU     R*8   T            6   UNREFERENCED
VSCOM#          X                 1
VSERH#          X                 1

TAG: FORMAT(F), NON-EXECUTABLE(N), USED AS ARGUMENT(A), OBJECT OF BRANCH(B), USED IN ASSIGN STATEMENT(S)

    LABEL      TAG  DEFINED  REFERENCES

          100          23      20
          111  NFS     13      16
          200  B       26      18    19
          300  B       28      24
01 000 00001            1      UNREFERENCED
01 000 00019           19      UNREFERENCED
01 000 00020           20      UNREFERENCED
01 000 00022           22      UNREFERENCED
01 000 00025           25      UNREFERENCED
02 001 00020   B       20      20
02 002 00020   B       23      20
```

Figure 31. Example of a Cross-Reference Dictionary—XREF Option

N label for a nonexecutable statement

S label used in an ASSIGN statement

If a label is used in more than one way, all tags that apply are printed.

DEFINED COLUMN: This column displays the internal sequence number (ISN) of the statement in which the label is defined.

REFERENCES COLUMN: This column displays the internal sequence number (ISN) of all statements in which there are references to the label.

If there are no program references to the label, the word UNREFERENCED is printed.

END OF COMPILATION MESSAGE

The last entry of the compiler output listing is the informative message:

```
********************END OF COMPILATION n ***********************
```

Where n is the number identifying this program's position in a batch compilation.

USING THE STANDARD LANGUAGE FLAGGER—FIPS OPTION

Through the FIPS option, you can help ensure that your program conforms to the current FORTRAN standard—American National Standard Programming Language FORTRAN, ANSI X3.9-1978.

You can specify standard language flagging either at the full language level or at the subset language level:

FIPS=F	requests the compiler to issue a message for any language element not included in full American National Standard FORTRAN.
FIPS=S	requests the compiler to issue a message for any language element not included in subset American National Standard FORTRAN.
NOFIPS	requests no flagging for nonstandard language elements.

The messages tell you which language items are not included in the current American National Standard. They're all in the same format as the other diagnostic messages, and they're all at the 0(I) (information) level.

For the formats of diagnostic messages, see "Diagnostic Message Listing—FLAG Option."

LINK-EDITING YOUR PROGRAM—ADVANCED PROGRAMMING

You must link-edit any object module before you can execute your program, combining this object module with others to construct an executable load module.

For VM/370-CMS considerations on loading and executing your programs, see "Using VM/370-CMS with VS FORTRAN."

For TSO considerations on loading and executing your program, see "Using OS/VS2-TSO with VS FORTRAN."

AUTOMATIC CROSS-SYSTEM SUPPORT

In VS FORTRAN, you can compile your source program under any supported operating system. You can then link-edit the resulting object module under the same system, or under any other supported system.

For example, you could request compilation under VM/370-CMS and then link-edit the resulting object module for execution under DOS/VSE.

You don't have to request anything special during compilation to do this; VS FORTRAN uses the execution-time library for all system interfaces, so the operating system under which you link-edit determines the system under which you execute.

LINKAGE EDITOR INPUT

Your input to the linkage editor can be the object module in machine-language format (which you request through the OBJECT compile-time option), or as a machine-language input data set (which you request through the DECK compile-time option).

You request the DECK option when you want to catalog the object module and save it for future link-edit runs.

Request the OBJECT option when you want to combine the link-edit task with the compilation task. You can then catalog and/or execute the load module produced.

OBJECT MODULE AS LINK-EDIT DATA SET—DECK OPTION

You use the DECK compile-time option to request an object module punched into a card deck. The deck produced is in 80-character fixed format.

The deck is a copy of the object module—which consists of dictionaries, text, and an end-of-module indicator. (Object modules are described in greater detail in the appropriate linkage editor and loader publications, as listed in the Preface.)

The object deck consists of four types of records, identified by the characters ESD, RLD, TXT, or END in columns 2 through 4. Column 1 of each card contains a 12-2-9 punch. Columns 73 through 80 contain the first four characters of the program name followed by a four-digit sequence number. The remainder of the card contains program information.

ESD Record

ESD records describe the entries of the External Symbol Dictionary, which contains one entry for each external symbol defined or referred to within a module. For example, if program MAIN calls subprogram SUBA, the symbol SUBA will appear as an entry in the External Symbol Dictionaries of both the program MAIN and the subprogram SUBA.

The linkage editor matches the entries in the dictionaries of other included subprograms and, when necessary, to the automatic call library.

ESD records are divided into four types, identified by the digits 0, 1, 2, or 5 in column 25 of the first entry, and column 57 of a third entry (there can be 1, 2, or 3 external-symbol entries in a record).

The contents of each type of ESD record are:

ESD Type	Contents
0	Name of the program or subprogram and indicates the beginning of the module.
1	Entry point name appearing in an ENTRY statement of a subprogram.
2	Name of a subprogram referred to by the source module through CALL statements, EXTERNAL statements, and explicit and implicit function references. (Some VS FORTRAN intrinsic functions are so complex that a function subprogram is called in place of in-line coding. Such calls are defined as implicit function references).
5	Information about a COMMON block.

TXT Record

TXT records contain the constants and variables your source program uses, any constants and variables generated by the compiler, coded information for FORMAT statements, and the machine instructions generated by the compiler from the source module.

RLD Record

RLD records describe entries in the Relocation Dictionary, which contains one entry for each address that the linkage editor or loader must resolve before the module can be executed.

The Relocation Dictionary contains information that enables absolute storage addresses to be established when the module is loaded into main storage for execution. These addresses cannot be determined earlier because the absolute starting address of a module cannot be known until the module is loaded.

The linkage editor or loader consolidates RLD entries in the input modules into a single relocation dictionary when it creates a load module.

RLD records contain the storage addresses of subprograms called through ESD type 2 records.

END Record

The END record indicates:

- The end of the object module to the linkage editor
- The relative location of the main entry point
- The length (in bytes) of the object module.

The structural order of a typical VS FORTRAN object module is shown in Figure 32.

Record Type	Usage
ESD (Type 0)	Names object module
ESD (Type 1)	Names entry points (from ENTRY statements)
TXT	For FORMAT statements
TXT	For compiler-generated constants
ESD (Type 5)	For COMMON areas
ESD (Type 2)	For external references in CALL and EXTERNAL statements, and statements using subprograms
RLD	For external references in CALL and EXTERNAL statements, and statements using subprograms
TXT	For source program constants
ESD (Type 2)	For compiler-generated external references
RLD	For compiler-generated external references
TXT	For object module instructions
TXT	For the branch list
RLD	For the branch list
END	End of object module

Figure 32. Object Module Structure

PRODUCING A LOAD MODULE—OBJECT OPTION

Specify the OBJECT option when you want either to link-edit and execute your program immediately, or to link-edit the program and catalog the load module for execution at some later time.

LINK-EDITING FOR IMMEDIATE EXECUTION

The simplest way to link-edit for immediate execution is to use a link-edit-execute cataloged procedure. See "Executing Your Program—Advanced Programming" for details.

You can also execute the program immediately after link-editing by including the execution step immediately after the link-edit step. See "Executing Your Program—Advanced Programming" for details on the execution step.

LINK-EDITING YOUR PROGRAM—OS/VS

The following sections show how to catalog your object module or load module under OS/VS, and how to use the OS/VS linkage editor or loader.

CATALOGING YOUR LOAD MODULE—OS/VS

You can catalog the data set containing the load module by defining it in a SYSLMOD DD statement during link-edit processing.

See the VS FORTRAN Application Development: System Services Reference Supplement for details.

EXECUTING A LINK-EDIT—OS/VS

Under OS/VS, there are two different programs you can use to perform the link-edit: the linkage editor or the loader. Which one you use depends upon the output you want produced.

THE LINKAGE EDITOR: Use the linkage editor when you want to reduce storage requirements through overlays, or to use additional libraries as input, or to define the structural segments of the program.

THE LOADER: Use the loader when your input is a small object module that doesn't require overlay, that doesn't require additional linkage editor control statements, and that you'll be executing immediately.

VS FORTRAN supplies you with cataloged procedures that let you link-edit or load your programs easily. See "Using and Modifying Cataloged Procedures—OS/VS" for details.

Using the Linkage Editor—OS/VS

When you use the linkage editor, rather than the loader, you have many processing options and optional data sets you can use, depending on the link-edit processing you want done.

LINKAGE EDITOR PROCESSING OPTIONS—OS/VS: Through the PARM option of the EXEC statement, you can request additional optional output and processing capabilities:

> **MAP**—specifies that a map of the load module is to be produced on SYSPRINT, giving the length and location of the main program and all subprograms.
>
> **XREF**—specifies that a cross reference listing of the load module is to be produced on SYSPRINT, for the main program and all subprograms.
>
> **LET**—specifies that the linkage editor is to allow load module execution, even when abnormal conditions have been detected that could cause execution to fail.
>
> **NCAL**—specifies that the linkage editor is not to attempt to resolve external references.
>
> If your program attempts to call external routines, you'll get an abnormal termination.
>
> **LIST**—specifies that the linkage editor control statements are to be listed in the SYSPRINT data set.

OVLY—specifies that the load module is to be in overlay format. That is, that segments of the program will share the same storage at different times during processing. (For more details, see "Coding Calling and Called Programs.")

SIZE—specifies the amount of virtual storage to be used for this link-edit job.

REQUIRED LINKAGE EDITOR DATA SETS—OS/VS: For any link-edit job, you must make certain that at least the following data sets are available:

SYSLIN—used for compiler output and linkage editor input.

SYSLMOD—used for linkage editor output.

SYSPRINT—makes the system print data set available, used for writing listings and messages. This data set can be a direct access, magnetic tape, or printer data set.

SYSUT1—direct access work data set needed by the link-edit process.

OPTIONAL LINKAGE EDITOR DATA SETS—OS/VS: In addition, depending on what you want the linkage editor to do for you, you can optionally specify the following data sets:

SYSLIB—direct access data set that makes the automatic call library (SYS1.FORTLIB) available. (Required for execution.)

SYSTERM—used for writing error messages and the compiler statistics listing. This data set can be on a direct access, magnetic tape, or printer device.

For reference information on these linkage editor data sets, see the VS FORTRAN Application Programming: System Services Reference Supplement.

USING LINKAGE EDITOR CONTROL STATEMENTS—OS/VS: Under OS/VS, you can use the INCLUDE and LIBRARY linkage editor control statements.

INCLUDE Statement: Use the INCLUDE statement to specify additional object modules you want included in the output load module.

LIBRARY Statement: Use the LIBRARY statement to specify additional libraries to be searched for object modules to be included in the load module.

For reference informaton on these statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

Using the Loader—OS/VS

You choose the loader when you want to combine link-editing into one job step with load module execution. The loader combines your object module with other modules into one load module, and then places the load module into main storage and executes it.

The loader options you can use, and the loader data sets are described in the following paragraphs.

LOADER OPTIONS—OS/VS: When you execute the loader, you can specify the following options through the PARM parameter of the EXEC statement:

MAP|NOMAP—specifies whether a map of the load module is to be produced on SYSPRINT, giving the length and location of the main program and all subprograms.

LET|NOLET—specifies whether the linkage editor is to allow load module execution, even when abnormal conditions that could cause execution to fail have been detected.

CALL|NCAL—specifies whether or not the loader is to call attempt to resolve external references. If you specify NCAL and your program attempts to call external routines, you'll get an abnormal termination.

SIZE—lets you specify the amount of storage to be allocated for loader processing.

EP—lets you specify the name of the entry point of the program being loaded.

PRINT|NOPRINT—specifies whether or not loader messages are to be listed in the data set defined by the SYSLOUT DD statement.

RES|NORES—specifies whether or not the link pack area is to be searched to resolve external references.

SIZE—specifies the amount of storage to be allocated for loader processing; this size includes the size of your load module.

LOADER DATA SETS—OS/VS: For any loader job, you must make certain that at least the SYSLIN data set (used for compiler output) and the SYSPRINT data set (used for printed output) are available.

In addition, depending on what you want the loader to do for you, you can optionally specify the following data sets:

SYSLIB—direct access data set that makes the automatic call library (SYS1.FORTLIB) available. This is the library used for FORTRAN library subroutines.

SYSLOUT—makes the system output data set available, used for writing listings. This data set can be a direct access, magnetic tape, or printer data set.

FTnnFnnn—data sets for user-defined files—can be unit record, magnetic tape, or direct access data sets.

LINK-EDITING YOUR PROGRAM—DOS/VSE

The following sections show how to catalog your object module or load module under DOS/VSE, and how to use the DOS/VSE linkage editor.

CATALOGING YOUR LOAD MODULE—DOS/VSE

You can catalog the load module (phase) in the core image library; this can be either the system core image library or your own private core image library.

When you catalog the load module in the core image library, your link-edit step must include the following statement:

```
// OPTION CATAL
```

which invokes the linkage editor and places the phase output into a core image library.

Place the statement before the first link-edit control card, and before the PHASE statement for the program you want cataloged.

See the VS FORTRAN Application Development: System Services Reference Supplement for details.

EXECUTING A LINK-EDIT—DOS/VSE

Under DOS/VSE, the only control statement you need to link-edit your program is the EXEC LINKEDT statement. DOS/VSE has the autolink feature which, unless you suppress it, always resolves all object module references to external-names, after all the input modules have been read from SYSLNK, SYSIPT, and/or the relocatable library. Ordinarily, you shouldn't suppress it. However, if you use service routines or extended error handling routines, you will need INCLUDE statements, as shown in Figure 37 on page 143.

In addition to the EXEC LINKEDT statement and the autolink feature, there are other linkage editor statements you can use to control linkage editor functions:

ACTION specifies linkage editor options, as follows:

- **CANCEL** requests immediate cancellation if errors occur.
- **CLEAR** initializes the temporary portion of the core image library to binary zeros.
- **BG/F1-F6** specifies how input is to be processed:
  - **BG** background execution
  - **F1-F6** Execution in one of the foreground partitions, F1 through F6
- **MAP** specifies that a storage map and messages are to be printed.
- **NOMAP** suppresses the MAP option; messages are to be printed.
- **NOAUTO** suppresses the autolink feature for this run.

INCLUDE specifies that a module from the relocatable library or from SYSIPT (if the compile-time DECK option was specified) is to be included; a number of modules may be specified.

PHASE specifies a name for the phase (load module) to be produced; the starting address and a relocation factor can also be specified.

You can also specify control statements to control overlay requirements; for details, see "Coding Calling and Called Programs."

Logical Units Used for Link-Editing—DOS/VSE

The following logical units are used during link-editing:

SYSLNK	used for linkage editor input
SYSRES	used for input in form of relocatable object modules
SYSRLB	used for linkage editor input from the system relocatable library; this is the library used for FORTRAN library subroutines.
SYSLST	used for input from relocatable object modules in a private library
SYSCLB	used for output placed in a private core image library
SYS001	used as a linkage-editor work file

LINKAGE EDITOR OUTPUT

Output from the linkage editor is in the form of load modules (or phases) in executable form. The exact form of the output depends upon the options in effect when you requested the link-edit, as described in the previous sections.

EXECUTING YOUR PROGRAM—ADVANCED PROGRAMMING

When you execute the load module (or phase), you can either execute it directly as output from the link-edit (or loader) step, or specify that it be called from a library of load modules.

When you execute a load module, you may need many different files, as outlined in the following sections.

For VM/370-CMS considerations on program execution, see "Using VM/370-CMS with VS FORTRAN."

EXECUTING YOUR PROGRAM—OS/VS

The following sections describe the data sets you may need, and outline the job control language you must use to execute your programs under OS/VS.

USING LOAD MODULE DATA SETS—OS/VS

If you're using cataloged procedures, or if you're using the device assignments as shipped by IBM, you must use the DD names shown in Figure 33.

FORTRAN Ref. Number	ddname	Function	Device Type	Usage
5	SYSIN	Input data set to load module	Card Reader, Magnetic Tape, Direct Access	Load Module Input Data
5	FT05F001	Input data set to load module	Card Reader, Magnetic Tape, Direct Access	Load Module Input Data
6	FT06F001	Printed output data	Printer, Magnetic Tape, Direct Access	Load Module Output Data
7	FT07F001	Punched output data	Card Punch, Magnetic Tape, Direct Access	Load Module Output Data
0-4 8-99	FTnnFnnn	Sequential Data Set	Unit Record, Magnetic Tape, Direct Access	Program Data
0-4 8-99	FTnnFnnn	Direct Access Data Set	Direct Access	Program Data
0-4 8-99	FTnnFnnn	Partitioned Data Set Member Using Sequential Access	Direct Access	Load Module Input Data

Figure 33. Load Module Data Sets—OS/VS

Using Cataloged Load Modules—OS/VS

You can execute cataloged load modules using either a STEPLIB DD or a JOBLIB DD statement.

USING JOBLIB DD :If you specify a JOBLIB DD statement for the load module, the JOBLIB library is available through all job steps of the job.

To ensure that the library remains available, you must specify the JOBLIB DD statement immediately after the JOB statement.

USING STEPLIB DD :If you specify a STEPLIB DD statement for the load module, the STEPLIB library is available for only this one step of the job.

You can place the STEPLIB DD statement anywhere among the DD statements for this job step.

EXECUTING THE LOAD MODULE—OS/VS

How you execute the load module depends on the kind of job you're running: execute only, link-edit and execute, or compile link-edit and execute.

VS FORTRAN supplies you with cataloged procedures that let you compile, link-edit or load, and/or execute easily. See "Using and Modifying Cataloged Procedures—OS/VS" for details.

Execute Only—OS/VS

The job control statements you use are:

```
//JOB Statement
//EXEC Statement        (load module)
//DD Statements         (as required for execution)
  (Input Data to be processed)
/*End-of-Data Statement (if input data is on cards)
//End-of-Job Statement
```

Link-Edit and Execute—OS/VS

The job control statements you use are:

```
//JOB Statement
//EXEC Statement        (linkage editor)
//DD Statements         (as required for linkage editing)
  (Link-edit is performed)
//EXEC Statement        (load module)
//DD Statements         (as required for execution)
  (Input data to be processed)
/*Statement             (if input data is on cards)
//End-of-Job Statement
```

Compile, Link-Edit, and Execute—OS/VS

The job control statements you use are:

```
//JOB Statement
//EXEC Statement        (VS FORTRAN Compiler)
//DD Statements         (as required for compilation)
  (Source program to be compiled)
/*End-of-Data Statement (if source program is on cards)
//EXEC Statement        (linkage editor)
//DD Statements         (as required for link-editing)
  (Link-edit is performed)
//EXEC Statement        (load module)
//DD Statements         (as required for load module execution)
  (Input data to be processed)
```

/*End-of-Data Statement (if input data is on cards)
//End-of-Job Statement

Reference documentation for these job control statements is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

EXECUTING YOUR PROGRAM—DOS/VSE

The following sections describe the logical units you may need, and outline the job control statements you must use to execute your programs under DOS/VSE.

LOAD MODULE LOGICAL UNITS—DOS/VSE

If you're using cataloged procedures, or if you're using the logical units as shipped by IBM, you must specify them as shown in Figure 34.

FORTRAN Ref. Number	Logical Unit	DOS File Name	Function (Primary)	Device Type
0	SYS000	IJSYS00	Program data set	Unit record Magnetic tape Direct access
1	SYS001	IJSYS01	Program data set	Unit record Magnetic tape Direct access
2	SYS002	IJSYS02	Program data set	Unit record Magnetic tape Direct access
3	SYS003	IJSYS03	Program data set	Unit record Magnetic tape Direct access
4	SYS004	IJSYS04	Program data set	Magnetic tape Direct access
5	SYSIPT or SYSIN	IJSYSIP	Input data set to load module	Unit record Magnetic tape Direct access
6	SYSLST	IJYSYLS	Punched output data	Unit record Magnetic tape Direct access
7	SYSPCH	IJSYSPC	Printed output data	Printer Magnetic tape Direct access
8 thru 99	SYS005 thru SYS096	IJSYS05 thru IJSYS96	Program data set	Unit record Magnetic tape Direct access

Note: Units 9 through 99 may be added by reassembling the unit assignment table module (IFYUATBL). See "Using the VSFORTL Macro" in VS FORTRAN Installation and Customization.

Figure 34. Load Module Logical Units—DOS/VSE

EXECUTING THE LOAD MODULE—DOS/VSE

How you execute the load module (or phase) depends on the kind of job you're running: execute only, link-edit and execute, or compile link-edit and execute.

Execute Only—DOS/VSE

The job control statements you use are:

```
// JOB Statement
// ASSGN Statements      (as required for execution)
// EXTENT Statements     (as required for execution)
// DLBL/TLBL Statements  (as required for execution)
// EXEC Statement        (load module (or phase))
   (Input data to be processed)
/* End-of-Data Statement (if input data is on cards)
/& End-of-Job Statement
```

Link-Edit and Execute—DOS/VSE

The job control statements you use are:

```
// JOB Statement
// OPTION LINK           (sets link option)
    or
// OPTION CATAL          (sets link option and catalogs phase)
    INCLUDE Statements   (as required for linkage editing)
// ASSGN Statements      (as required for linkage editing)
// DLBL/TLBL Statements  (as required for linkage editing)
// EXEC Statement        (linkage editor)
   (Linkage editor execution)
// ASSGN Statements      (as required for execution)
// DLBL/TLBL Statements  (as required for execution)
// EXTENT Statements     (as required for execution)
// EXEC Statement        (load module)
    (Input data to be processed)
/* End-of-Data Statement (if input data is on cards)
/& End-of-Job Statement
```

Note: Unless you specify OPTION CATAL, the phase is deleted from the core image library after execution is completed.

Compile, Link-Edit, and Execute—DOS/VSE

The job control statements you use are:

```
// JOB Statement
// OPTION LINK           (sets link option)
   or
// OPTION CATAL          (sets link option and catalogs phase)
// EXEC Statement        (VS FORTRAN Compiler)
   (VS FORTRAN source program)
/* Statement             (if source program is on cards)
// ASSGN Statements      (as required for linkage editing)
// DLBL/TLBL Statements  (as required for linkage editing)
// EXEC Statement        (linkage editor)
     (Linkage Editor execution)
// ASSGN Statements      (as required for execution)
// DLBL/TLBL Statements  (as required for execution)
// EXTENT Statements     (as required for execution)
// EXEC Statement        (load module)
     (Input Data to be processed)
/* End-of-Data Statement (if input data is on cards)
/& End-of-Job Statement
```

Note: Unless you specify OPTION CATAL, the phase is deleted from the core image library after execution is completed.

Reference documentation for these job control statements is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

LOAD MODULE EXECUTION-TIME OUTPUT

The output that execution of your load module gives you depends upon whether or not there are errors in your program.

EXECUTION WITHOUT ERROR

If your program executes without error, and gives the results you expect, your task of program development is completed.

EXECUTION WITH ERRORS

When your program has errors in it, your execution-time output may be incorrect, or nonexistent.

You may or may not get error messages as well. Any VS FORTRAN execution-time error messages you get come from the VS FORTRAN Library. These messages are in a format similar to the compiler message format (see "Library Diagnostic Messages").

If you get output from the program itself, it may be exactly what you expected, or (if there are logic errors in the program) it may be output you didn't expect at all.

When this happens, you must proceed to the next step in program development, described in "Fixing Execution-Time Errors—Advanced Programming."

FIXING EXECUTION-TIME ERRORS—ADVANCED PROGRAMMING

You can begin to fix execution-time errors by scanning the source program for the kinds of errors described in "Fixing Execution-Time Errors—Simplified Programming" in Part 1.

However, some execution-time errors are difficult to find and a simple scan of the source program isn't always very helpful. VS FORTRAN has a number of features to help you find errors. The major ones are described in the following sections.

EXECUTION-TIME MESSAGES

Execution-time messages are issued by the execution-time library. There are three types of messages issued:

Library Diagnostic Messages—which give information about errors occurring when the library subroutines are executed—for example, input/output or mathematical subroutine errors

Program Interrupt Messages—which give information about errors that occur when system rules are violated

Operator Messages—which communicate with the operator when program execution makes it necessary (for example, when a PAUSE statement is executed)

LIBRARY DIAGNOSTIC MESSAGES

The library diagnostic messages have a format similar to that of the compiler messages. The library messages give you information on execution of input/output routines, mathematical subroutines, and the utility routines.

The messages all have the prefix IFY; they are in the following format:

```
IFYnnnnI origin 'message text'
```

where each of the areas has the following meaning:

IFY	is the message prefix identifying all VS FORTRAN library messages
nnnnI	is the unique number identifying this message
origin	is the abbreviated name for the library module that originated the message.
'message-text'	explains the execution-time error that was detected.

The action the program takes after a message is issued depends upon your extended error handling routines. For further information, see "Using Extended Error Handling."

The execution-time messages are documented in the VS FORTRAN Application Programming: Library Reference manual.

Using the Optional Traceback Map

Whenever you get a library diagnostic message, you can also, optionally, get a traceback map.

Your organization may have set this as the default you get whenever a library message is generated.

If this is not the default for your organization, you can request a traceback map, using the CALL ERRTRA routine. See "Using the Optional Traceback Map" for a further discussion.

The traceback map gives you guidance in determining where the error occurred.

The traceback map lists the names of called routines, internal sequence numbers within routines, and contents of registers as follows:

ROUTINE
lists the names of all routines entered in the current calling sequence.

Names are shown with the latest routine called at the top and the earliest routine called at the bottom of the listing except when the earliest name shown is VSCOM.

CALLED FROM ISN
lists the FORTRAN program's internal sequence number (ISN) that called the routine, except when calls were made to VSCOM.

Internal statement numbers are available to the traceback routine only if you specified the GOSTMT compiler option.

REG. 14
lists the absolute storage location of the instruction that calls ROUTINE.

REG. 15
lists the absolute location of the entry point to ROUTINE.

REG. 0
lists the results of function subprogram operations, when applicable.

REG. 1
lists the address of any argument list passed to ROUTINE.

ENTRY POINT=address
shows the entry point of the earliest routine entered.

The control program executes its own routine to recover from the error, and displays the following message:

STANDARD FIXUP TAKEN, EXECUTION CONTINUING

If your organization uses its own error recovery routine, the word USER replaces STANDARD in this message.

After the error recovery, execution continues.

The summary of errors printed at the end of the listing can help you determine how many times an error was encountered. If your source program contains many input/output statements, locating an error can become a formidable task. By pinpointing the exact FORTRAN statement involved, the traceback map makes it much easier for you to locate execution errors.

If you specify the GOSTMT compiler option, the traceback map lists the internal sequence number (ISN) calling each routine. Using the ISN, you can locate the source statement and module called.

LIST Compiler Option and Traceback Maps

If you specify the LIST compiler option, you can use the traceback map to locate the last assembler language instruction executed. See Figure 33 for an example of the object program listing. The steps to take are:

1. For the topmost routine listed under the heading REG.14, subtract the 6 low-order hexadecimal digits in the number shown under ENTRY POINT. This produces the relative location of the instruction in the listing.

2. Find this location in the object code listing.

3. Using this location as a beginning point, scan upward in the column that identifies statement numbers to locate the nearest number occurring before the instruction; this is the statement number of the FORTRAN statement involved in the error.

4. Investigate the FORTRAN statement in the source module listing.

5. If the source statement is correctly specified, investigate the corresponding job control statement for accuracy.

PROGRAM INTERRUPT MESSAGES

During program execution, you'll get messages whenever the program is interrupted because of the following exceptions: operation, fixed-point division, decimal division, floating-point division, exponent overflow, or exponent underflow.

Program interrupt messages are written in the output data set. Such an interrupt message gives you guidance in determining the cause of the error; it indicates what system rule was violated.

Exception codes themselves appear in the eighth digit of the PSW and indicate the reason for the interruption. Their meanings are as follows:

Code	**Meaning**
1	is an operation exception, that is, the operation is not one that is defined to the operating system.
4	is a protection exception, that is, an illegal reference is made to an area of storage protected by a key.
5	is an addressing exception, that is, a reference is made to a storage location outside the range of storage available to the job.
6	is a specification exception, for example, a unit of information does not begin on its proper boundary.
7	is a data exception, that is, the arithmetic sign or the digits in a number are incorrect for the operation being performed.
9	is a fixed-point-divide exception that is, an attempt is made to divide by zero.
C	is an exponent-overflow exception, that is, a floating-point arithmetic operation produces a positive number mathematically too large to be contained in a register (the mathematically largest number that can be contained is 16^{63} or approximately 7.2×10^{75}). Exponent-overflow generates the additional message:

REGISTER CONTAINED number

where number is the floating-point number in hexadecimal format. (When extended-precision is in use, the message prints out the contents of two registers.) A standard corrective action is taken and execution continues.

D indicates an exponent-underflow exception, that is, a floating-point arithmetic operation generates a number with a negative exponent mathematically too small to be contained in a register (mathematically smaller than 16^{-65} or approximately 5.4×10^{-79}). Exponent-underflow also generates the message:

REGISTER CONTAINED number

where number is the number generated.

(When extended-precision is in use, the message prints out the contents of two registers.) A standard corrective action is taken and execution continues.

F is a floating-point-divide exception, that is, an attempt is being made to divide by zero in a floating-point operation. Floating-point divide also generates the message:

REGISTER CONTAINED number

(When extended-precision is in use, the message prints out the contents of two registers.) A standard corrective action is taken and execution continues.

The standard corrective action for each type of interrupt is described in the "Program Interrupt Messages" section of the VS FORTRAN Application Programming: Library Reference manual.

Notes:

1. Operation, protection, addressing, and data exceptions (codes 1, 4, 5, and 7) ordinarily cause abnormal termination without any corresponding message.

2. Protection and addressing exceptions (codes 4 and 5) generate a message only if a specification exception (code 6) or an operation exception (code 1) has also been detected.

3. A data exception (code 7) generates a message only if a specification exception has also been detected. When a message is generated for codes 4, 5, or 7, the job will terminate.

   The completion code in the dump indicates that job termination is due to a specification or operation exception; however, the error message indicates the true exception that caused the termination.

OPERATOR MESSAGES

Operator messages are generated when your program executes a PAUSE or STOP n statement. Operator messages are written on the system device specified for operator communication, usually the console. The message can guide you in determining how far your FORTRAN program has executed.

Figure 35 shows the form that the operator message may take.

```
yy n | 'message'
```

Figure 35. Operator Message Format

The meaning of the lowercase characters in the figure is as follows:

Character	Meaning
yy	message identification number assigned by the system.
n	string of 1 through 5 decimal digits you specified in the PAUSE or STOP statement. For the STOP statement, this number is placed in register 15.
'message'	character constant you specified in the PAUSE or STOP statement.
0	printed when a PAUSE statement containing no characters is executed. (Nothing is printed for a similar STOP statement.)

A PAUSE message causes program execution to halt, pending operator response. To resume program execution, the operator issues the command:

```
REPLY yy, 'z'
```

where yy is the message identification number and z is any letter or number.

A STOP message causes program termination.

IBM EXTENSION

USING DEBUG PACKETS

A debug packet helps you locate obscure sources of error in your source program. It consists of the DEBUG statement, a set of FORTRAN statements, and an END DEBUG statement. The DEBUG statement and debug packets, when used, must be the first statements in your program.

In debugging packets you can use the following statements:

DEBUG
specifies the debugging options you want performed, which can be:

- Check the validity of array subscripts.
- Trace the order of execution of all or part of the program.
- Display array or variable values each time they change during program execution.

AT
specifies the statement number before which the debugging packet is to be executed.

The AT statement begins each new debugging packet in the program and ends the previous one.

TRACE ON|TRACE OFF
begin or end program execution tracing.

DISPLAY
writes a list of variable or array values that you specify.

END DEBUG
ends the last debugging packet specified.

In addition to these specific debugging statements (valid only in a debugging packet) you can also use most FORTRAN procedural statements to gather information about what's happening during execution.

When you specify a debugging packet, you'll get information about program execution in a form that's easy to understand and easy to use. Figure 36 shows how you can use VS FORTRAN debugging statements.

If you use a debugging packet in your source program and compile it using OPTIMIZE(1), OPTIMIZE(2), or OPTIMIZE(3), the compiler changes the optimization parameter to NOOPTIMIZE.

END OF IBM EXTENSION

USING EXTENDED ERROR HANDLING

Extended error handling can be either by default or you can control it, using predefined CALL statements.

EXTENDED ERROR HANDLING BY DEFAULT

Your organization has a default value for the following execution-time conditions:

- The number of times an error can occur before the program is terminated.

- The maximum number of times an execution-time message is printed.

- Whether or not a traceback map is to be printed with the message.

- Whether or not your organiation's error-exit routine is to be called.

These are the extended error handling facilities that may be available to you.

When extended error handling by default is in effect for your organization, the following actions take place when an error occurs.

The FORTRAN error monitor (ERRMON) receives control.

The error monitor prints the necessary diagnostic and informative messages:

- A short message, along with an error identification number.

  The data in error (or some other associated information) is printed as part of the message text.

  For a complete listing of execution-time messages, see the VS FORTRAN Application Programming: Library Reference manual.

- A summary error count, printed when the job is completed.

The error count, telling you how many times each error occurred.

- A traceback map, tracing the subroutine flow back to the main program, after each error occurrence.

Then the error monitor takes one of the following actions:

- Terminates the job.
- Returns control to the calling routine, which takes a standard corrective action and then continues execution.
- Calls a user-written closed subroutine to correct the data in error and then returns to the routine that detected the error, which then continues execution.

Program Code With A Debugging Packet:

```
A       DEBUG SUBCHK(ARRAY1), TRACE, INIT(ARRAY1)
B       AT 10
C       TRACE ON
D        (procedural code for debugging)
E       AT 40
F       TRACE OFF
G       DISPLAY I, J, K, L, M, N, ARRAY1
H       END DEBUG
           .
           .
           .
  10    DO ...     (program tracing begins here; procedural debugging
           .        code executed)
           .
           .
  30    CONTINUE
  40    WRITE ... (program tracing ends here; values of I, J, K, L,
                   M, N, and ARRAY1 are displayed)
```

How Each Debugging Statement is Used:

A The DEBUG statement begins the first debugging packet and specifies the following:

- SUBCHK(ARRAY1) requests validity checking for the values of ARRAY1 subscripts
- TRACE specifies that tracing will be allowed within the debugging packet
- INIT(ARRAY1) specifies that ARRAY1 will be displayed when values within it change

B AT 10 begins the first debugging packet.

C TRACE ON turns on program tracing at statement number 10.

D (Procedural debugging code contains valid FORTRAN statements to aid in debugging; for example, to initialize variables.)

E AT 40 ends the first debugging packet and begins the second.

F TRACE OFF turns off program tracing at statement number 40.

G The DISPLAY statement writes the values of I, J, K, L, M, N, and ARRAY1.

H END DEBUG ends the second (and last) debugging packet.

Figure 36. Using Batch Symbolic Debugging Statements

The actions of the error monitor are controlled by settings in the option table. IBM provides a standard set of option table entries; your system administrator may have provided additional entries for your organization.

By altering the option table at execution-time, through ERRSET, you can specify the user exit to be taken.

If no corrective action, either standard or user-written, is to be taken, make sure the table entry specifies that only one error is to be allowed before an abnormal termination.

To make changes to the option table dynamically at load module execution-time, you can use the predefined CALL subroutines, summarized in the next section.

CONTROLLING EXTENDED ERROR HANDLING—CALL STATEMENTS

For each error condition detected, you have both dynamic and default control over:

- The number of times the error is allowed to occur before program termination
- The maximum number of times each message may be printed
- Whether or not the traceback map is to be printed with the message
- Whether or not a user-written error-exit routine is to be called

The action that takes place is governed by information stored in the option table, which is present in main storage. (A permanent copy of the option table is maintained in the FORTRAN library.)

Reference documentation for using the option table is given in the VS FORTRAN Application Programming: Language Reference manual.

The predefined CALL routines let you request extended error handling, so that you get greater control over load module errors:

- **CALL ERRMON**—causes execution of the error control monitor.
- **CALL ERRTRA**—causes execution of the traceback routines. (See "Using the Optional Traceback Map.")
- **CALL ERRSAV**—copies an option table entry into an area accessible to your program.
- **CALL ERRSTR**—stores an entry into the option table from your program.
- **CALL ERRSET**—changes up to 5 values associated with an entry in the option table.

When you're using these routines, specify them as follows:

1. Issue CALL ERRMON to make the error handling facilities available.
2. Issue CALL ERRSAV to make an entry accessible to your program.
3. Issue CALL ERRSET to change the options in the entry (for example, to change the number of errors allowed before termination).
4. Issue CALL ERRSTR to store the changed entry back into the option table.

When you're planning to use these routines, be sure to consult your system administrator for options and values you can specify.

When you're setting option table entries, don't allow more than 255 occurrences of any error; infinite program looping can result.

The changes you make through these CALL routines are in effect only during your own program's execution.

These routines are described in "Using The Execution-Time Library."

For reference documentation about these predefined CALL routines, see the VS FORTRAN Application Programming: Language Reference manual.

Extended Error Handling—DOS/VSE Considerations

Under DOS/VSE, when you're using any of the extended error handling routines, or any of the service routines, you must identify the routine by its library module name through a linkage editor INCLUDE statement. Names you can use are shown in Figure 37.

VS FORTRAN Source Name	VS FORTRAN Library Name
ERRMON	IFYVMOPT
ERRSAV	IFYVMOPT
ERRSET	IFYVMOPT
ERRSTR	IFYVMOPT
ERRTRA	IFYVMOPT
DVCHK	IFYDVCHK
DUMP/PDUMP	IFYVDUMP
CDUMP/CPDUMP	IFYVDUMP
EXIT	IFYVEXIT
OPSYS	IFYOPSYS
OVERFL	IFYVOVER

Figure 37. Library Names for DOS/VSE Error Handling and Service Routines

OBJECT MODULE LISTING—LIST OPTION

The object module listing is useful when you can't discover any syntax errors in your VS FORTRAN source statements, and yet they aren't doing what you expected. The object module listing shows you (in pseudo-assembler format) the machine code the compiler generated from your source statements. A careful examination of this can often give you an idea of what's wrong with your source.

The object module listing is especially useful when you're compiling and executing using one of the OPTIMIZE compiler options. Further details are given in "Using the Optimization Feature."

You request an object module listing by specifying the LIST option.

The object module listing is in pseudo-assembler language format showing each assembler language instruction and data item, as shown in Figure 38.

Each line of the listing is formatted (from left to right) as follows:

- A 6-digit number shows the relative address of the instruction or data item, in hexadecimal format.
- The next area gives the storage representation of the instruction or initialized data item, in hexadecimal format.
- The next area (not always present) gives names and statement labels, which may be either those appearing in the source program or those generated by the compiler (compiler-generated labels are in the form nn nnn nnnnnnn).
- The next area gives the pseudo-assembler language format for each statement.
- The last area gives the source program items referred to by the instruction, such as entry points of subprograms, variable names, or other statement labels.

The object module listing contains sections in the following order:

1. Entry code
2. Format statements
3. Temporary storage for fix/float
4. Constants
5. Variables in COMMON areas
6. Other variables
7. Address constants for ASSIGNED FORMAT statements, COMMON areas, and external references
8. NAMELIST dictionaries
9. Program code
10. Prolog and epilog code
11. Address constants for prologs, the save area, epilogs, and parameter lists
12. Temporary storage areas and generated constants
13. Address constants for block labels

At the end of the listing, you'll find compiler statistics:

- The name of the source program
- The number of statements compiled
- The size of the generated object module (in bytes)
- The number of error messages, listed by severity, produced during this compilation

```
ENTRY CODE
     000000  47F0 F012            MAIN          BC     15,18(0,15)
     000004  06                                 DC     XL1'06'
     000005  D4C1C9D5404040                     DC     CL7'MAIN   '
     00000C  F8F14BF1F1F2                       DC     CL6'81.112'
     000012  90EC D00C                          STM    14,12,12(13)
     000016  9823 F028                          LM     2,3,40(15)
     00001A  5030 D008                          ST     3,8(13)
     00001E  50D0 3004                          ST     13,4(0,3)
     000022  07F2                               BCR    15,2

FORMAT STATEMENTS
     000030  021A0CF1D6E4E3D7     111           DC     CL8'   1OUTP'
     000038  E4E340C6D6D94030                   DC     CL8'UT FOR  '
     000040  000E060E0A0A070C                   DC     CL8'        '
     000048  0F0704020A14101C                   DC     CL8'        '
     000050  22                                 DC     CL1' '
```

Figure 38 (Part 1 of 4). Object Module Listing Example—LIST Compiler Option

```
TEMPORARY FOR FIX/FLOAT
     0000C0  0000000000000000                DC    XL8'0000000000000000'
     0000C8  4E00000000000000                DC    XL8'4E00000000000000'

CONSTANTS
     0000D0  4F08000000000000                DC    XL8'4F08000000000000'
     0000D8  4E00000080000000                DC    XL8'4E00000080000000'
     0000E0  413243F6A791A9E1   PI           DC    XL8'413243F6A791A9E1'
     0000E8  4130000000000000                DC    XL8'4130000000000000'
     0000F0  00000000                        DC    XL4'00000000'
     0000F0  00000000                        DC    XL4'00000000'
     0000F4  00000000                        DC    XL4'00000000'
     0000F8  00000001                        DC    XL4'00000001'
     0000FC  0000000E                        DC    XL4'0000000E'
     000100  00000004                        DC    XL4'00000004'
     000104  00000007                        DC    XL4'00000007'
     000108  00000008                        DC    XL4'00000008'
     00010C  00000020                        DC    XL4'00000020'
     000110  00000080                        DC    XL4'00000080'
     000114  000000A0                        DC    XL4'000000A0'
     000118  00000000                        DC    XL4'00000000'
     00011C  3E2DE00D                        DC    XL4'3E2DE00D'         (REAL)
     000120  3E89A027                        DC    XL4'3E89A027'         (IMAG)
     000124  00000019                        DC    XL4'00000019'
     000128  00000006                        DC    XL4'00000006'

VARIABLES IN 'COM1  ' COMMON.
     000000  NO INITIAL DATA    R4A          DS    11F                   (ARRAY)
     000004  NO INITIAL DATA    R8A          DS    7D                    (ARRAY)
     00000C  NO INITIAL DATA    C32          DS    80D                   (ARRAY)
     00002C  NO INITIAL DATA    C8A          DS    F                     (REAL)
     000030  NO INITIAL DATA                 DS    F                     (IMAG)

VARIABLES IN 'COM2  ' COMMON.
     FFFFF8  NO INITIAL DATA    I2           DS    3F                    (ARRAY)
     000000  NO INITIAL DATA    L1           DS    X
     000001  NO INITIAL DATA    C8B          DS    F                     (REAL)
     000005  NO INITIAL DATA                 DS    F                     (IMAG)

VARIABLES
     000130  NO INITIAL DATA    R8V          DS    D
     000138  NO INITIAL DATA    I            DS    F
     00013C  NO INITIAL DATA    J            DS    F
     000140  NO INITIAL DATA    A1           DS    F
     000144  413243F4           A2           DC    XL4'413243F4'
     000148  E2C8C1D9C540D7D9   CHAR14       DC    CL8'SHARE PR'
     000150  D6C7D9C1D440                    DC    CL6'OGRAM '

ADCONS FOR ASSIGNED FORMAT STATEMENTS
     000158  00000030                        DC    AL4'00000030'

ADCONS FOR COMMONS
     00015C  00000000                        DC    AL4'00000000'   COM1
     000160  FFFFFFFC                        DC    AL4'FFFFFFFC'   COM1
     000164  FFFFFF6C                        DC    AL4'FFFFFF6C'   COM1
     000168  00000000                        DC    AL4'00000000'   COM2
     00016C  FFFFFFF4                        DC    AL4'FFFFFFF4'   COM2

ADCONS FOR EXTERNAL REFERENCES
     000170  00000000                        DC    AL4'00000000'   CXSUB  (SUBR)
     000174  00000000                        DC    AL4'00000000'   VSCOM# (SUBR)
     000178  00000000                        DC    AL4'00000000'   VSERH# (SUBR)
```

Figure 38 (Part 2 of 4). Object Module Listing Example—LIST Compiler Option

```
PROGRAM CODE
      0001CC   5800 D104         01 000 00001  L      0,260(0,13)          111
      0001D0   5000 D0E8                       ST     0,232(0,13)            J
      0001D4   5870 D10C                       L      7,268(0,13)
      0001D8   6800 7010                       LD     0,16(0,7)            R8A
      0001DC   6000 D12C                       STD    0,300(0,13)        .S01
      0001E0   6800 D07C                       LD     0,124(0,13)     4F080000
                                                                      00000000
      0001E4   7800 D0F0                       LE     0,240(0,13)           A2
      0001E8   6A00 D12C                       AD     0,300(0,13)        .S01
      0001EC   6C00 D094                       MD     0,148(0,13)     4130...0
      0001F0   7000 D0EC                       STE    0,236(0,13)           A1
      0001F4   7900 D0C4                       CE     0,196(0,13)            0
      0001F8   5850 D170                       L      5,368(0,13)          200
      0001FC   0785                            BCR    8,5
      0001FE   7800 D0F0         01 000 00019  LE     0,240(0,13)           A2
      000202   7900 D0C4                       CE     0,196(0,13)            0
      000206   5850 D170                       L      5,368(0,13)          200
      00020A   0785                            BCR    8,5
      00020C   5800 D0A4         01 000 00020  L      0,164(0,13)            1
      000210   5000 D0E4                       ST     0,228(0,13)            I
      000214   5870 D114         02 001 00020  L      7,276(0,13)
      000218   1B00                            SR     0,0
      00021A   4300 7000                       IC     0,0(0,7)              L1
      00021E   1B55                            SR     5,5
      000220   5860 D164                       L      6,356(0,13)          100
      000224   8705 6000                       BXLE   0,5,0(6)
      000228   5860 D0E4         01 000 00022  L      6,228(0,13)            I
      00022C   8960 0003                       SLL    6,3
      000230   6800 D0DC                       LD     0,220(0,13)          R8V
      000234   5870 D10C                       L      7,268(0,13)
      000238   6A06 7000                       AD     0,0(6,7)             R8A
      00023C   6000 D12C                       STD    0,200(0,13)         .S01
      000240   2B00                            SDR    0,0
      000242   6000 D134                       STD    0,308(0,13)         .S03
         .
         .
         .
      0002D2   45E0 F004                       BAL    14,4(0,15)
      0002D6   20000000                        DC     XL4'20000000'
      0002DA   00000128                        DC     XL4'00000128'
      0002DE   0000013C                        DC     XL4'0000013C'
      0002E2   45E0 F008                       BAL    14,8(0,15)
      0002E6   0A00D0F4                        DC     AL4'0A00D0F4'     CHAR14
      0002EA   000000FC                        DC     AL4'000000FC'         14
      0002EE   45E0 F00C                       BAL    14,12(0,15)
      0002F2   00000004                        DC     AL4'00000004'        R8A
      0002F6   06000007                        DC     XL4'06000007'
      0002FA   45E0 F008                       BAL    14,8(0,15)
      0002FE   0700D0EC                        DC     AL4'0700D0EC'         A1
      000302   45E0 F00C                       BAL    14,12(0,15)
      000306   0000000C                        DC     AL4'0000000C'        C32
      00030A   0E000014                        DC     XL4'0E000014'
      00030E   45E0 F010                       BAL    14,16(0,15)
      000312   58F0 D120                   300 L      15,288(0,13)      VSCOM#
      000316   45E0 F038                       BAL    14,56(0,15)
      00031A   07                              DC     XL1'07'
      00031B   E3C8C540C5D5C4                  DC     XL7'E3C8C540C5D5C4'
      000322   58F0 D120                       L      15,288(0,13)       VSCOM#
      000326   45E0 F034                       BAL    14,52(0,15)
      00032A   05                              DC     XL1'05'
      00032B   40404040F0                      DC     XL5'40404040F0'
```

Figure 38 (Part 3 of 4). Object Module Listing Example—LIST Compiler Option

```
EPILOGUE CODE
     000330  58F0 D120                                    L      15,288(0,13)      VSCOM#
     000334  45E0 F034                                    BAL    14,52(0,15)
     000328  05                                           DC     CL1' '
     000339  40404040F0                                   DC     CL5'    0'

PROLOGUE CODE
     000340  58F0 3120                                    L      15,288(0,3)       VSCOM#
     000344  45E0 F040                                    BAL    14,64(0,15)
     000348  18D3                                         LR     13,3
     00034A  58F0 D150                                    L      15,336(0,13)
     00034E  07FF                                         BCR    15,15

ADCON FOR PROLOGUE
     000028  00000340                                     DC     XL4'00000340'

ADCON FOR SAVE AREA
     00002C  00000054                                     DC     XL4'00000054'

ADCON FOR EPILOGUE
     000054  00000330                                     DC     XL4'00000330'

ADCONS FOR BRANCH TABLES
     0000A0  000002C4                                     DC     XL4'000002C4'
     0000A4  00000312                                     DC     XL4'00000312'

ADCONS FOR PARAMETER LISTS
     0000AC  00000130                                     DC     AL4'00000130'       R8V
     0000B0  00000140                                     DC     AL4'00000140'       A1
     0000B4  800000E0                                     DC     AL4'800000E0'       413243F6
                                                                                   A791A9E1
     0000B8  80000124                                     DC     AL4'80000124'       25

TEMPORARIES AND GENERATED CONSTANTS
     00017C  00000000                                     DC     XL4'00000000'
     000180  00000000                                     DC     XL4'00000000'
     000184  00000000                                     DC     XL4'00000000'
     000188  00000000                                     DC     XL4'00000000'
     00018C  00000000                                     DC     XL4'00000000'
     000190  00000000                                     DC     XL4'00000000'
     000194  00000000                                     DC     XL4'00000000'
     000198  00000000                                     DC     XL4'00000000'
     00019C  00000000                                     DC     XL4'00000000'
     0001A0  00000000                                     DC     XL4'00000000'

ADCONS FOR B BLOCK LABELS
     0001A4  000001CC                                     DC     XL4'000001CC'
     0001A8  000001FE                                     DC     XL4'000001FE'
     0001AC  0000020C                                     DC     XL4'0000020C'
     0001B0  00000214                                     DC     XL4'00000214'
     0001B4  00000228                                     DC     XL4'00000228'
     0001B8  00000292                                     DC     XL4'00000292'
     0001BC  000002A8                                     DC     XL4'000002A8'
     0001C0  000002C4                                     DC     XL4'000002C4'
     0001C4  000002CE                                     DC     XL4'000002CE'
     0001C8  00000312                                     DC     XL4'00000312'
```

Figure 38 (Part 4 of 4). Object Module Listing Example—LIST Compiler Option

REQUESTING DUMPS

You can request dynamic dumps of specific areas of storage during program execution, using the FORTRAN dump subprograms.

REQUESTING DYNAMIC DUMPS—CALL STATEMENT

Four VS FORTRAN predefined CALL routines let you request a dynamic dump of selected areas of storage during program execution:

- CALL PDUMP—dumps the requested areas and allows processing to continue
- CALL DUMP—dumps the requested areas and terminates processing
- CALL CPDUMP—dumps the requested character storage areas and allows processing to continue
- CALL CDUMP—dumps the requested character storage areas and terminates processing

When you use the DUMP and PDUMP subprograms, you specify as parameters:

- The variables delimiting the area to be dumped
- A code specifying the format in which the items are to be dumped

For example, if you wanted to dump one item and continue processing, you could specify:

```
CALL PDUMP (A,A,5)
```

which would dump the variable A in real format (the code 5 specifies real format).

You can also dump an entire range of items in storage:

```
CALL DUMP (A,M,0)
```

which would dump every item in storage, beginning with variable A and continuing through variable M, in hexadecimal format (the code 0 specifies hexadecimal format). Processing would then be terminated.

When you're using CDUMP and CPDUMP, the output is always in character format. Therefore, you specify only the delimiting variables in the CALL statement. For example, to dump a range of character variables from character variable C1 to character variable C19, specify:

```
CALL CDUMP (C1,C19)    (and execution terminates)
```

or

```
CALL CPDUMP (C1,C19)   (and execution continues)
```

For reference documentation about these routines, see the VS FORTRAN Application Programming: Language Reference manual.

Dynamic Dumps—DOS/VSE Considerations

Under DOS/VSE, when you're requesting a dynamic dump, you must also specify its library name with a FORTRAN or a linkage editor INCLUDE statement. Figure 37 gives the names to specify.

REQUESTING AN ABNORMAL TERMINATION DUMP

How you request an abnormal termination dump depends on the system you're using.

Information on interpreting dumps is found in the appropriate debugging guide, as listed in the Preface.

Requesting a Dump—OS/VS

Under OS/VS1, program interrupts causing abnormal termination produce a dump, called an indicative dump, which displays the completion code and the contents of registers and system control fields.

To display the contents of main storage as well, you must request an abnormal termination (ABEND) dump by including a SYSUDUMP DD statement in the appropriate job step. The following example shows how the statement may be specified for IBM-supplied cataloged procedures:

```
  //GO.SYSUDUMP  DD  SYSOUT=A
```

To specify a dump under MVS, you should include a SYSUDUMP DD statement.

Requesting a Dump—DOS/VSE

Under DOS/VSE, to request a dump you can specify:

```
// OPTION PARTDUMP
```

This provides a dump of the partition storage, the registers, and the areas of the supervisor control blocks that relate to this partition.

PART 3—FORTRAN SPECIAL FEATURES

This part gives you guidance information on using the following VS FORTRAN special features:

- "Programming Input and Output"
- "Coding Calling and Called Programs"
- "Using the Optimization Feature"
- "Using The Execution-Time Library"
- "Using VM/370-CMS with VS FORTRAN"
- "Using OS/VS2-TSO with VS FORTRAN"

PROGRAMMING INPUT AND OUTPUT

This chapter describes how you use VS FORTRAN to create and process the following types of files:

- Sequential files on unit record, magnetic tape, and direct access devices
- Direct files on direct access devices
- VSAM files on direct access devices
- Internal files for data conversions to and from the CHARACTER type

IBM EXTENSION

- Sequential files using list-directed input/output statements
- Asynchronous files for high-speed sequential input/output

END OF IBM EXTENSION

USING VS FORTRAN INPUT/OUTPUT STATEMENTS

For each of these forms of input/output you can use the VS FORTRAN input/output statements:

FORMAT statement—specifies the structure of FORTRAN records.

OPEN statement—connects a file to a FORTAN program.

WRITE statement—transmits a record to an external or internal unit.

READ statement—retrieves a record from an external or internal unit.

ENDFILE statement—writes an end-of-file record on an external sequential file.

BACKSPACE statement—backspaces a sequential file one record.

REWIND statement—positions the file so that the next READ or WRITE statement processes the first record in the file.

CLOSE statement—disconnects a file from a FORTRAN program.

INQUIRE statement—requests information about a file.

IBM EXTENSION

WAIT statement—completes an asynchronous input/output transmission.

END OF IBM EXTENSION

USING COMMON OPTIONS FOR INPUT/OUTPUT

For the OPEN, WRITE, READ, ENDFILE, BACKSPACE, and REWIND statements, there are common options you can specify:

The UNIT Number
which specifies the I/O unit number of the file to be processed, which you code as an integer or as a variable expression.

When you specify the UNIT number as a variable expression, you can use one input/output statement to process more than one file; that is, between one execution of the statement and the next, you can change the value of the unit number and thus change the external unit that the statement refers to.

The I/O Status
which, after the input/operation is completed, gives you the result:

Zero, if no error was detected

Positive, if an error was detected

Negative, at sequential end-of-file

If this is a VSAM file, the VSAM return and reason codes

An Error Routine
which you can use to specify special processing after an error occurs during execution of the I/O statement.

This lets you code a special routine that's executed when an error occurs. The routine can obtain information about the last record processed.

For example, the routine could close any other open files, and display information useful in debugging, such as accumulated totals or current values in selected data items.

Other options are available with specific input/output statements and with specific file processing techniques.

CONNECTING TO A FILE—OPEN STATEMENT

The OPEN statement connects your program to an external file. It's required for direct files and VSAM files; for other files, it's optional. When you specify it, it must be the first input/output statement executed for the file.

The OPEN statement lets you specify special processing options for the file.

In addition to the common processing options previously listed, you can also optionally specify:

The File Status
which lets you specify the file status of this file, as follows:

NEW	for a file that you're creating for the first time
OLD	for a file that already exists

SCRATCH	for a temporary file to be be used during this job and then erased at the end of the job
UNKNOWN	for a file whose status is not currently known to the program; it may or may not currently exist

The External Filename
lets you specify the name of the file you're connecting.

Depending upon the file's status, this option may or may not be required. If the STATUS is:

NEW	the filename is optional
OLD	the filename is optional
SCRATCH	the filename must not be specified
UNKNOWN	the filename is optional

For CMS, MVS, and VS1, the filename specified in the OPEN statement is the DD name.

Example (CMS):

```
FILEDEF MYFILE1 DISK SAMPLE DATA A
      .
      .
      .
    OPEN (UNIT=15, FILE='MYFILE1',...)
```

Example (MVS,VS1):

```
//MYFILE1 DD DSN=SAMPLE,...
      .
      .
      .
    OPEN (UNIT=15, FILE='MYFILE1',...)
```

For DOS/VSE, the filename specified is contained in the DLBL statement.

Example (DOS):

```
//DLBL MYFILE1
      .
      .
      .
    OPEN (UNIT=15, FILE='MYFILE1',...)
```

The Access Method
which lets you specify the access method for this file: SEQUENTIAL or DIRECT.

SEQUENTIAL files are always assumed; therefore for these types of files, specifying the access option is not required. However, specifying it is useful for documentation.

For DIRECT files, you must specify the access method. A direct file must always exist before it is opened. The OPEN itself neither allocates the file nor initializes it with skeleton records if the file is new.

How Blanks are Treated
lets you specify how blanks in numeric fields of an input record are to be treated:

NULL	blanks are ignored
ZERO	any blanks that aren't leading blanks are treated as zeros

The Formatting
lets you specify whether this file is connected for FORMATTED or for UNFORMATTED input/output.

The Record Length
lets you specify the logical record length of a direct file.

If you specify FORMATTED I/O, this length is the number of characters (bytes) in the record.

If you specify UNFORMATTED I/O, the length is the number of bytes in the record.

For example, if the record contains two character items, each 10 characters long, and 2 double precision items, the length of the record is 36 bytes:

10 bytes each (20 bytes total) for the two character items

8 bytes each (16 bytes total) for the two double precision items

Reference documentation for the OPEN statement is given in _VS FORTRAN Application Programming: Language Reference_.

CREATING FILE RECORDS—WRITE STATEMENT

You can use the WRITE statement for two different purposes:

1. To transfer data items from internal storage to a record in an external file
2. For internal files, to transfer a number of data items (each of which may have a different data type) into one character item

The form of WRITE statement you specify depends upon the access you're using. See the descriptions, later in this chapter, of each type of access.

With the WRITE statement, you can specify any of the common processing options previously described.

Reference documentation for the WRITE statement is given in the _VS FORTRAN Application Programming: Language Reference_ manual.

RETRIEVING FILE RECORDS—READ STATEMENT

You use the READ statement in two different ways:

1. To transfer a record from an external file to data items in internal storage
2. For internal files, to transfer one character item into a number of data items, each of which may have a different data type

The form of READ statement you specify depends upon the access you're using. See the descriptions, later in this chapter, of each type of access.

With the READ statement, you can specify any of the common processing options previously described.

Reference documentation for the READ statement is given in the _VS FORTRAN Application Programming: Language Reference_ manual.

OBTAINING FILE INFORMATION—INQUIRE STATEMENT

You use the INQUIRE statement to gather information about an external sequential or direct file, or about a particular external unit. Your program can then take alternative actions, depending upon the information provided.

The INQUIRE statement is never required in a program, and you can execute it whether or not the file or unit is currently connected with your program.

You can ask for information about either a file or a unit number; and you can specify the I/O Status and Error Routine options, previously described.

In addition, you can request the following information:

- Whether or not the file or unit exists.
- Whether or not the file or unit is connected.
- Get the unit number of the file or unit.
- Whether or not the file has a name.
- The access to this file—sequential or direct.
- If the file can be connected for sequential I/O.
- If the file can be connected for direct I/O.
- Whether the file is connected for formatted or for unformatted I/O.
- If you can connect the file for formatted I/O.
- If you can connect the file for unformatted I/O.
- The record length, if this is a direct access file.
- The number of the next record in the file, if this is a direct access file.
- Whether input blanks are treated as zeros or as nulls.

Reference documentation for the INQUIRE statement is given in the VS FORTRAN Application Programming: Language Reference manual.

DISCONNECTING A FILE—CLOSE STATEMENT

You use the CLOSE statement to break the connection between an external file and a FORTRAN unit.

The CLOSE statement is never required, but it lets you specify special processing when the connection is broken off.

In addition to the common processing options previously described, you can also specify whether or not the file still exists (internally to the FORTRAN program) after the CLOSE statement is executed.

If you don't explicitly specify that the file is to be deleted, the file still exists after you've closed it. In this case, you can subsequently open the file for updating or retrieval.

If you specify that the file is to be deleted, you can subsequently open the file only for file creation.

Note: The CLOSE statement does not override what you specify in the job control statements for the file.

Reference documentation for the CLOSE statement is given in the VS FORTRAN Application Programming: Language Reference manual.

USING UNFORMATTED AND FORMATTED I/O

For sequential and direct files, you can specify either of two forms of READ and WRITE statements: unformatted or formatted.

Unformatted I/O—lets you use a list of FORTRAN data items to control the transfer of data; the length of the FORTRAN items in storage controls the amount of data transferred.

Each unformatted READ or WRITE statement processes a record at a time, transferring the data items without conversion. (This means the transfer is quicker than when the program must convert each item as it is processed.)

Formatted I/O—lets you control input/output by specifying the format of the FORTRAN records and the form of data fields within the records.

This form of I/O also lets you convert items from one type to another during the data transfer. (The conversions cause the data transfer to be slower than with unformatted input/output.)

In this form of input/output, you specify a FORMAT statement to be used in conjunction with the READ and WRITE statements. The FORMAT statement specifies the format of the FORTRAN records—the receiving field in a WRITE statement and the sending field in a READ statement.

FORMATTING FORTRAN RECORDS—FORMAT STATEMENT

When you're using formatted I/O, the FORMAT statement lets you specify the format of the FORTRAN records in READ or WRITE statements. You can place FORMAT statements anywhere between the first and last statements in your program unit; you must specify a statement label.

You can use the FORMAT statement with both external and internal files.

When you use it for external files, you must ensure that the size of the FORMAT record doesn't exceed the size of the input/output medium; for example, if you're sending the record to a printer, it must not be longer than the printer line length. See "Appendix A. Device Information" for details on external devices.

Each field in the FORTRAN record is described with a FORMAT code specifying the data type for the field. The order in which you specify the codes is their order in the record. Some of the codes available with VS FORTRAN are shown in Figure 39.

FORMAT Code	Meaning
	Data Field Codes
Aw	Character data field (optional length specification)
aAw	Character data field (optional repeat count)
paEw.dEe	Real data field (optional exponent (Ee))
aIw.m	Integer data field (with minimum number of digits to be displayed (.m))
	IBM EXTENSION
aGw.dEe	Integer, real, or logical data field (optional exponent (Ee))
	END OF IBM EXTENSION
	Edit Codes:
:	End of format control, but only if I/O list is completely processed
/	End of record
BN	Nonleading blanks in a numeric field are ignored on input
BZ	Blanks treated as zeros on input
S	Specifies display of an optional plus sign
SP	Specifies plus sign must be produced on output
SS	Specifies plus sign is not to be produced on output
TLr	Data transfer starts r characters to left
TRr	Data transfer starts r characters to right

The lower case letters have the following meanings:

a	an optional repeat count
d	number of decimal places to be carried
e	number of digits in the exponent field
m	minimum number of digits to be displayed
p	the number of digits for the scale factor
r	a character displacement in a record
w	the total number of characters in a field

Figure 39. Some Codes Used with the FORMAT Statement

For example, if you want to define the format of an output record, you could specify your FORMAT statement as follows:

```
200   FORMAT (SP,2A10,I6.4,2E14.5E2)
```

which specifies that the output line is to be formatted as follows:

SP	specifies that if the value of any of the numeric fields is positive a plus sign is to be displayed. (If the value is negative, a minus sign is always displayed.)
2A10	specifies that the first and second items are character items of length 10
I6.4	specifies that the third item is an integer item of total length 6, and that when the line is produced the display is: (blank)(+ or -)(4 digits)
2E14.5E2	specifies that the fourth and fifth items are real items of total length 14; the display for each is shown in Figure 40. In this example, the minimum total width you can specify for this field is 12: one character for a leading blank, seven characters for the numeric field (including the leading sign and the decimal point), and four characters for the exponent (including the E and the sign).

In this example, the length of the record you've defined is 54 characters (bytes).

```
(3 blanks)(sign)(decimal point)(5 digits)(E)(sign)(2 digits)
          |______________________________||_________________|
                   Numeric Field                Exponent

    (The sign is displayed as a + or a -)
```

Figure 40. Display for FORMAT E14.5E2

A formatted WRITE statement uses the statement label for this FORMAT statement and writes a record in this format.

When you execute a formatted READ statement, your program expects the external data to be in this format.

Reference documentation for the FORMAT statement is given in the VS FORTRAN Application Programming: Language Reference manual.

Group FORMAT Specifications

VS FORTRAN lets you specify group specifications nested within the overall FORMAT specification, by specifying the group within parentheses. The group can contain a combination of format codes and groups, each separated by commas, slashes, or colons.

For example, you could specify an input record as follows:

```
100   FORMAT (BZ,A4,(A8,(I4,E8.4,(E4,E8.2))))
```

which specifies that the input receiving fields are structured as follows:

BZ specifies that blanks in the input are treated as zeros.

The first field in the record:

```
        A4
```

is a character field of length 4.

It is followed by a group field

```
        (A8,(I4,E8.4,(E4,E8.2),),)—
```

consisting of a character field eight characters long

```
        A8
```

followed by a group field

```
        (I4,E8.4,(E4,E8.2),)
```

This nested group field contains yet another nested group field:

```
        (E4,E8.2)
```

In the VS FORTRAN FORMAT statement, the limit of two for nesting group fields has been removed.

Using Specifications Repeatedly—FORMAT Control

Your FORMAT statements need not contain a format specification for each field in the READ or WRITE I/O list. If the end of the FORMAT specification list is reached before the last item in the I/O list is processed, control is returned to the rightmost left parenthesis in the format list; if there aren't any embedded parentheses, then control is returned to the first item in the format list:

```
10    FORMAT (A4,2(I2,I4),3(I4,I4),E4.2)

          (Control returns to 3(I4,I4)
```

You can take advantage of this VS FORTRAN feature to reduce your coding effort, but be careful to ensure that the items repeated are the items you want repeated.

Using One FORMAT Statement with Variable Formats

You can specify variable FORMAT statements, by placing a format specification into an array during execution. You could read the specification in from external storage, or you could initialize the array using a DATA statement or an explicit specification statement. You can then use the array as the format specification in READ or WRITE statements.

Using this feature, you can refer to a different array element each time you execute the READ or WRITE statement, and thus change the format.

This is an efficient way to use one READ or WRITE statement to process a file that contains records in many different formats.

This feature (together with a variable unit number) lets you use a single READ or WRITE statement to process more than one file.

USING SEQUENTIAL FILES

Sequential files are those in which the records are arranged and processed in a serial order. The records are arranged in the file in the same order they were created. They can be retrieved only in that same order.

You can store sequential files on unit record, magnetic tape, or direct access devices.

If you concatenate sequential files with differing record lengths, make certain that the first file is the one with the longer record length. If you don't, the record lengths of the files are not compatible.

SOURCE PROGRAM CONSIDERATIONS

The FORTRAN statements you can use with sequential file processing are the OPEN, WRITE, READ, ENDFILE, BACKSPACE, REWIND, and CLOSE statements.

Using the OPEN Statement—Sequential Files

It's never necessary to specify an OPEN statement with sequential files, unless the file is a VSAM sequential file. However, the OPEN statement lets you take advantage of the special processing it makes available. You can, for example, specify the status of the file—NEW, OLD, SCRATCH, or UNKNOWN—as well as specifying special processing to be performed if the OPEN statement fails.

If your OPEN statement doesn't specify the formatting, FORMATTED is assumed.

If you don't specify an OPEN statement, the first READ statement for the file establishes the file connection.

See "Using Common Options For Input/Output" and "Connecting to a File—OPEN Statement" for descriptions of the options you can use.

Using the WRITE Statement—Sequential Files

You can use either an unformatted or formatted WRITE statement with a sequential file; for example:

Unformatted:

```
WRITE (UNIT=10,ERR=300,IOSTAT=INT)A,E,I,O,U
  or
WRITE (10,ERR=300,IOSTAT=INT) A,E,I,O,U
```

Formatted:

```
WRITE (UNIT=10,FMT=40,ERR=300,IOSTAT=INT) A,E,I,O,U
  or
WRITE (10,40,ERR=300,IOSTAT=INT)A,E,I,O,U
```

where, in this example:

10	is the unit number of the external file.
40	is the statement label of the FORMAT statement (used only with the formatted WRITE statement).
300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.

INT	is the name of an integer variable or an array element into which is placed a positive or zero value indicating failure or success of the WRITE operation.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists to be included in the output record. Within main storage, these items need not be contiguous.

See "Using Common Options For Input/Output" for a description of the options you can use.

Using the READ Statement—Sequential Files

You can use either an unformatted or a formatted READ statement with a sequential file; for example:

Unformatted:

```
      READ (UNIT=11,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U

        or

      READ (11,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U
```

Formatted:

```
      READ (11,FMT=40,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U

        or

      READ (11,40,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U
```

where, in this example:

11	is the unit number of the external file.
40	is the statement label of the FORMAT statement (used only with the formatted READ statement)
300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs
INT	is the name of an integer variable or array element into which is placed a positive or zero value, indicating failure or success of the READ operation
200	is the statement label of the FORTRAN statement to which control is transferred when end-of-file is reached.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists into which the input record is transferred. Within main storage, these items need not be contiguous.

See "Using Common Options For Input/Output" for a description of the options you can use.

For an unformatted READ statement:

If an external record contains more data than the items in the list, the excess external data is skipped.

If an external record contains less data than the items in the list, an error occurs and processing continues.

Using the ENDFILE Statement—Sequential Files

You can use the ENDFILE statement to write an end-of-file record on an external file. The file must be connected when you issue the statement.

You can use the ENDFILE statement when you need to write an end-of-file record for an output file. For example, the following ENDFILE statement:

```
ENDFILE (UNIT=10,IOSTAT=INT,ERR=300)

        or

ENDFILE (10,IOSTAT=INT,ERR=300)
```

performs the following actions:

- Writes an end-of-file record on unit number 10.
- Returns a positive or zero value in INT to indicate failure or success.
- Transfers control to statement label 300 if an error occurs.

Using the REWIND Statement—Sequential Files

You use the REWIND statement to reposition a sequentially accessed file at its beginning point. The file must be connected when you execute the statement.

For example, the following REWIND statement:

```
REWIND (UNIT=11,IOSTAT=INT,ERR=300)

        or

REWIND (11,IOSTAT=INT,ERR=300)
```

performs the following actions:

- Positions the file on unit number 11 to its beginning point.
- Returns a positive or zero value in INT to indicate failure or success.
- Transfers control to statement label 300 if an error occurs.

Using the BACKSPACE Statement—Sequential Files

You use the BACKSPACE statement to reposition a sequentially accessed file to the beginning of the record last processed. The file must be connected when you execute the statement.

Before your program issues a BACKSPACE statement, it must issue a READ, WRITE, or REWIND statement, or the BACKSPACE statement is ignored.

You can use the BACKSPACE statement to check the accuracy of records your program writes on a magnetic tape file or sequential direct access file:

```
WRITE ...     (writes the record to the file)
BACKSPACE ... (positions the file at the beginning
              of the record just written)
READ ...      (retrieves the record for checking)
```

You can use the BACKSPACE statement to replace a record in a magnetic tape file or a sequential direct access file:

```
READ ...        (retrieves the record to be replaced)
BACKSPACE ... (positions the file at the beginning
                of the record just retrieved)
WRITE ...       (writes the new record)
```

After execution of this WRITE, no records exist in the file following this record. Any records that did exist are lost.

You can use the I/O common processing options with the BACKSPACE statement. For example, the following BACKSPACE statement:

```
BACKSPACE (UNIT=10,IOSTAT=INT,ERR=300)

      or

BACKSPACE (10,IOSTAT=INT,ERR=300)
```

performs the following actions:

- Positions the file on unit number 10 at the beginning of the record last read or written.
- Returns a positive or zero value in INT to indicate failure or success.
- Transfers control to statement label 300 if an error occurs.

Using the CLOSE Statement—Sequential Files

You use the CLOSE statement to terminate the connection between the external file and the unit.

For sequential files, the CLOSE statement is optional; however, you can use it to specify specific processing actions when you disconnect from the external file.

See the previous description of the CLOSE statement, in "Using VS FORTRAN Input/Output Statements," for a description of the options you can specify.

IBM EXTENSION

USING ASYNCHRONOUS INPUT/OUTPUT

Asynchronous input/output statements let you transfer unformatted data quickly between external sequential files and arrays in your FORTRAN program, and, while the data transfer is taking place, continue other processing within your FORTRAN program.

Because the processing overlaps, you must have a method to ensure that your program doesn't make references to the data until the data transfer is complete.

The asynchronous input/output statements have special features to achieve this:

- The WAIT statement—to ensure that data transmission is complete before your program begins processing the data
- A unique identifier to identify a particular READ, WRITE, or WAIT statement—and to connect it with other related asynchronous statements

Using the Asynchronous WRITE Statement

To create an asynchronous input/output file, you use a special form of the WRITE statement:

To Transfer an Entire Array:

```
WRITE (10,ID=6) ARAY1
```

To Transfer Part of an Array:

```
WRITE (10,ID=6) ARAY1(2,2)...ARAY1(5,6)

WRITE (10,ID=6) ARAY1(2,2)...
```

where, in this example:

10	is the unit number for the asynchronous file.
ID=6	is a unique identifier for this WRITE statement, used in the WAIT statement.
ARAY1	is an array whose contents are to be transferred.

In the first WRITE statement, the contents of the entire array are transferred.

In the second WRITE statement, the contents of ARAY1(2,2) through ARAY1(5,6) are transferred.

In the third WRITE statement, the contents of ARAY1(2,2) through the end of ARAY1 are transferred.

Using the Asynchronous READ Statement

To retrieve an asynchronous input/output file, you use a special form of the READ statement:

To Transfer an Entire Array:

```
READ (10,ID=6) ARAY1
```

To Transfer Part of an Array:

```
READ (10,ID=6) ARAY1(2,2)...ARAY1(5,6)

READ (10,ID=6) ARAY1(2,2)...
```

where, in this example:

10	is the unit number for the asynchronous file.
ID=6	is a unique identifier for this READ statement, used in the WAIT statement.
ARAY1	is an array whose contents are to be transferred.

In the first READ statement, the contents of the entire array are transferred.

In the second READ statement, the contents of ARAY1(2,2) through ARAY1(5,6) are transferred.

In the third READ statement, the contents of ARAY1(2,2) through the end of ARAY1 are transferred.

Using the Asynchronous WAIT Statement

After you've executed an asynchronous WRITE or READ statement, you must ensure that the I/O operation is complete before you make any further program references to the array being processed. The WAIT statement tells the program to suspend operations until the data transfer is complete; that is, it synchronizes the WRITE or READ statement with the rest of the program.

For example, you can use the following WAIT statement with the previously described READ or WRITE statements:

```
      WAIT (10,ID=6) ARAY1

      or

      WAIT (10,ID=6) ARAY1(2,2)...ARAY1(5,6)
```

where, in this example:

10	is the unit number for the asynchronous file.
ID=6	is a unique identifier for this WAIT statement; it ties this WAIT statement to the READ or WRITE statement with the same identifier (the operation the program is marking time for).
ARAY1	is an array. In the first WAIT statement, the data transfer for the entire array is being synchronized. In the second WAIT statement, the data transfer for array elements ARAY1(2,2) through ARAY1(5,6) is being synchronized.

END OF IBM EXTENSION

USING LIST-DIRECTED INPUT/OUTPUT

List-directed input/output statements—READ and WRITE—simplify your data entry for sequential files. They let you use formatted input/output—that is, input/output statements that perform data conversions as the data is transferred between internal and external storage—without the restrictions of a FORMAT statement. You can enter the data to be transferred without regard for column, line, or card boundaries.

In addition, there's no need for you to define the file through job control statements or CMS FILEDEF commands, because list-directed input/output uses the standard system data sets or logical units.

This makes list-directed READ and WRITE statements particularly useful for terminal input and output, and for developing program test data.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS2-TSO considerations, see "Using OS/VS2-TSO with VS FORTRAN."

Input Data—List-Directed I/O

You enter list-directed input data as a series of FORTRAN constants and separators:

- Each constant can be any valid FORTRAN constant. (Enter character constants within apostrophes.)

  Each constant you specify must agree with its corresponding item in the data list.

  You can sign numeric constants, but the sign must immediately precede the first digit of the data (no intervening blanks).

- You can specify a repetition factor for any constant or null item. For example:

```
3*2.6
```

specifies that the real constant 2.6 is to appear three times in the data input stream.

- Each separator can be:
  - one or more blanks
  - a comma
  - a line advance (for terminal input)
  - an end of card (on card devices)
  - a slash (/) (See the following paragraph)

  A combination of more than one separator, except for the comma, represents one separator.

  A slash (/) separator indicates that no more data is to be transferred during this READ operation:

  - Any items following the slash are not retrieved during this READ operation.
  - If all the items in the list have been filled, the slash is not needed.
  - If there are fewer items in the record than in the data list, and you haven't ended the list with a slash separator, an error is detected.
  - If there are more items in the record than in the data list, the excess items are ignored.
- A null item is represented by two successive commas.

List-Directed READ Statement

The list-directed READ statement retrieves a record from an input file, with conversions to internal forms of data:

```
READ (FMT=*,UNIT=5,IOSTAT=INT1) A,E,I,O,U
```

This READ statement specifies:

FMT=*
: specifies that this is a list-directed READ statement.

UNIT=5
: specifies that 5 is the unit from which the data is to be retrieved.

IOSTAT=INT1
: defines INT1 as the FORTRAN integer variable into which information about the last operation is placed; by testing INT1 you can specify special programming actions for special conditions.

  INT1 contains a positive value when an error occurs.

  INT1 contains a negative value at end-of-file.

  INT1 contains a zero value when neither condition occurs.

A,E,I,O,U
: are the data items in which you want the data placed.

  When the READ statement is executed, the data placed in A, E, O, and U is converted to REAL data of length 4; the data placed in I is converted to INTEGER data of length 4.

List-Directed WRITE Statement

The list-directed WRITE statement writes a record on an output file, with conversions from internal forms of data:

```
WRITE (UNIT=6,IOSTAT=INT1,FMT=*) A,E,I,O,U
```

This WRITE statement specifies:

UNIT=6
: specifies that 6 is the unit on which the data is to be written.

IOSTAT=INT1
: defines INT1 as the FORTRAN integer variable into which information about the last operation is placed; by testing INT1 you can specify special programming actions for special conditions.

  INT1 contains a positive value when an error occurs.

  INT1 contains a negative value at end-of-file.

  INT1 contains a zero value when neither condition occurs.

FMT=*
: specifies that this is a list-directed WRITE statement

A,E,I,O,U
: are the source data items for the data transfer.

When the WRITE statement is executed, the data is converted from the internal formats to external EBCDIC format.

USING INTERNAL FILES

Internal files let you move data from one internal storage area to another while converting it from one format to another. This gives you a convenient and standard method of making such conversions.

If your external file is coded in EBCDIC character data, you can execute an unformatted READ statement to bring it into a character item (VAR1) in storage:

For Sequential Files:

```
READ (UNIT=11,ERR=300,IOSTAT=INT,END=200) VAR1

     or
```

For Direct Files:

```
READ (UNIT=11,REC=KEY,ERR=300,IOSTAT=INT,END=200) VAR1
```

You can now execute a formatted internal READ statement to convert individual items in the record from EBCDIC to their internal formats.

For internal READ and WRITE statements, you specify the UNIT identifier as an internal data item—a character variable or substring, or as a character array or array element.

For reference documentation about the internal READ and WRITE statements, see the VS FORTRAN Application Programming: Language Reference manual.

Using the READ Statement—Internal Files

A READ statement referring to an internal unit converts the data from character format to the internal format(s) of the receiving item(s):

```
READ (UNIT=VAR1,FMT=40,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U

  or

READ (VAR1,40,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U
```

where, in this example:

VAR1	is a character variable in this program unit. (It could also be a character array or array element, or a character substring.)
40	is the statement label of the FORMAT statement.
300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.
INT	is the name of an integer variable or array element into which is placed a positive or zero value, indicating failure or success of the READ operation.
200	is the statement label of the FORTRAN statement to which control is transferred when the end of the storage area is reached.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists into which the input record is transferred. Within main storage, these items need not be contiguous.

Using the WRITE Statement—Internal Files

A WRITE statement referring to an internal unit converts the data transferred from internal format to character format:

```
WRITE (UNIT=VAR1,FMT=40,ERR=300,IOSTAT=INT) A,E,I,O,U

  or

WRITE (VAR1,40,ERR=300,IOSTAT=INT)A,E,I,O,U
```

where, in this example:

VAR1	is a character variable in this program unit. (It could also be a character array or array element, or character substring.)
40	is the statement label of the FORMAT statement.
300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.
INT	is the name of an integer variable or array element into which is placed a positive or zero value indicating failure or success of the WRITE operation.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists to be included in VAR1.

Within main storage, these items need not be contiguous.

SYSTEM CONSIDERATIONS—SEQUENTIAL FILES

Each sequential file you use must be defined to the system, through job control statements. The statements you use vary, depending upon the system you're executing under—OS/VS, DOS/VSE, or VM/370-CMS.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS2-TSO considerations, see "Using OS/VS2-TSO with VS FORTRAN."

For reference documentation about job control statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

OS/VS Considerations—Sequential Files

To define each file to the system, you specify a DD statement; see "Defining Files—OS/VS DD Statement" in Part 2 for details.

The data sets you can specify and the ddnames you can use for them are shown in Figure 29 in Part 2.

DOS/VSE Considerations—Sequential Files

To define each file to the system, you optionally specify an ASSGN statement; if it is a file on a direct access device, you must also specify a DLBL and an EXTENT statement. See "Defining Files—DOS/VSE ASSGN Statement" in Part 2 for details.

The logical units you can specify and the names you can use for them are shown in Figure 30 in Part 2.

USING DIRECT FILES

Direct files are those in which all the records are arranged in the file according to the relative addresses of their keys. Each record is the same size, and each occupies a predefined position in the file, depending upon its relative record number.

In a direct file, the first record has relative record number 1, the tenth record has relative record number 10, the fiftieth record has relative record number 50. You can think of the file as a series of slots, each of which may or may not actually contain a record. That is, record 50 may hold an actual record, and be identified as record number 50, even though records 24, 38, and 42 are vacant slots.

You can process the records by supplying the relative record number of the record you want with each READ or WRITE statement. In order to do this, you'd ordinarily develop an algorithm that lets you identify each record uniquely and that converts to a relative record number for record creation and retrieval.

For a discussion of randomizing techniques useful in developing such algorithms, see the Introduction to IBM Direct Access Storage Devices and Organization Methods.

You can store direct files only upon direct access devices.

SOURCE PROGRAM CONSIDERATIONS

The FORTRAN statements you can use to process direct files are the OPEN, WRITE, READ, and CLOSE statements.

Using the OPEN Statement—Direct Files

You must specify an OPEN statement with a direct file; you must specify the following options:

ACCESS—to specify the file as direct

RECL— to specify record length

The record length you specify when you create the file is the record length you must specify when you retrieve records from the file.

If you don't specify the formatting, UNFORMATTED is assumed.

See "Using Common Options For Input/Output" and "Connecting to a File—OPEN Statement" for descriptions of other options you can use.

Using the WRITE Statement—Direct Files

You can use either an unformatted or formatted WRITE statement with a direct file; for example:

Unformatted:

```
     WRITE (UNIT=15,REC=KEY,ERR=300,IOSTAT=INT)A,E,I,O,U

       or

     WRITE (15,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U
```

Formatted:

```
     WRITE (UNIT=15,FMT=40,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U

       or

     WRITE (15,40,REC=KEY,ERR=300,IOSTAT=INT)A,E,I,O,U
```

where, in this example:

15	is the unit number of the external file. It must identify a direct file.
40	is the statement label of the FORMAT statement (used only with the formatted WRITE statement).
KEY	Is an integer variable into which you place the relative record number for the record you're writing.

You can specify the variable as an integer item of length 4.

IBM EXTENSION

You can also specify the variable as an integer item of length 2.

END OF IBM EXTENSION

300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.

INT	is the name of an integer variable or array element into which is placed a positive or a zero value indicating failure or success of the WRITE operation.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists to be included in the output record. Within main storage, these items need not be contiguous.

Using the READ Statement—Direct Files

You can use either an unformatted or a formatted READ statement with a direct file, for example:

Unformatted:

```
READ (UNIT=11,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U

  or

READ (11,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U
```

Formatted:

```
READ (UNIT=11,FMT=40,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U

  or

READ (11,40,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U
```

where, in this example:

11	is the unit number of the external file. It must identify a direct file.
40	is the statement label of the FORMAT statement (used only with the formatted READ statement).
KEY	is an integer variable into which you place the relative record number of the record you want to retrieve. You can specify the variable as an integer item of length 4.

IBM EXTENSION

You can also specify the variable as an integer item of length 2.

END OF IBM EXTENSION

300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.
INT	is the name of an integer variable or array element into which is placed a positive or a zero value, indicating failure or success of the READ operation.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists into which the input record is transferred. Within main storage, these items need not be contiguous.

For an unformatted READ statement:

> If an external record contains more data than the items in the list, the excess external data is skipped.
>
> If an external record contains less data than the items in the list, an error occurs and processing continues.

Using the CLOSE Statement—Direct Files

You use the CLOSE statement to terminate the connection between the external file and the unit. The CLOSE statement is never required for direct files; however, you can use it to specify special processing to occur when you disconnect from the external file.

See the previous description of the CLOSE statement, in "Using VS FORTRAN Input/Output Statements," for a description of the options you can specify.

SYSTEM CONSIDERATIONS—DIRECT FILES

You must define each direct file to the system, through job control statements.

The job control statements differ, depending upon the system you're operating under—OS/VS, DOS/VSE, or VM/370-CMS.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS2-TSO considerations, see "Using OS/VS2-TSO with VS FORTRAN."

For reference documentation about job control statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

OS/VS Considerations—Direct Files

Before you can write records into the file, it must be initialized with empty records. Your organization should have a utility program to do this; check with your system administrator.

To define each file to the system, you specify a DD statement; in the DCB parameter, for a direct file, you must specify:

RECFM=F which specifies a fixed record size.

BLKSIZE=rl where rl is the record length.

> For example, if the record length is 80, you must specify BLKSIZE=80.

The OPEN statement provides the default block size for the file, unless you override it through the BLKSIZE parameter.

For other DD statement options you can specify, see "Defining Files—OS/VS DD Statement" in Part 2.

The data sets you can specify and the ddnames you can use for them are shown in Figure 29 in Part 2.

DOS/VSE Considerations—Direct Files

Before you can write records into the file, you must preformat it using the CLEAR disk utility. For documentation, see the DOS/VSE System Utilities manual.

To define each file to the system, you optionally specify an ASSGN statement.

You must specify a DLBL statement, using:

BLKSIZE to specify the block size.

The BLKSIZE you specify must be the same size as the record length you specify in the FORTRAN OPEN statement.

You must also specify an EXTENT statement, which defines each area the file occupies on the direct access device. See "Defining Files—DOS/VSE ASSGN Statement" in Part 2 for details.

The logical units you can specify and the names you can use for them are shown in Figure 30 in Part 2.

USING VSAM FILES

VS FORTRAN lets you use VSAM to process three kinds of files:

- VSAM sequential files, using Entry Sequenced Data Sets (ESDS), which can be processed only sequentially
- VSAM direct files, using Relative Record Data Sets (RRDS), which can be processed either sequentially or directly
- Under DOS/VSE—you can also process VSAM-managed sequential files (using the VSAM Space Management for SAM feature); such files can be processed only sequentially

For documentation explaining VSAM Entry Sequenced Data Sets (ESDS), Relative Record Data Sets (RRDS), and DOS/VSE VSAM-Managed SAM files, see the VSAM documentation for the system you're running under. (VSAM publications titles are given in the "Related Publications" section at the beginning of this manual.)

Generally speaking, VSAM files are best used as permanent files, that is, as files that are processed again and again by one or more application programs. You shouldn't try to use VSAM files as "scratch" files, because VSAM files are more difficult to allocate and erase than other files. (For this reason, FORTRAN doesn't let you use the NEW or SCRATCH options of the OPEN statement, or the DELETE option of the CLOSE statement.)

The following general programming considerations apply to VSAM files:

- Use VSAM sequential files (using ESDS) for applications in which you create a complete file, one in which you'll never update any records but to which you may add records in the future.
- Use VSAM direct files (using RRDS) for work files, or for files in which records must be created and later updated in place.
- Under DOS/VSE, use VSAM-Managed SAM files to reduce the amount of manual control needed to organize and maintain your non-VSAM sequential files.

The next section, on Source Language, gives programming considerations for each type of VSAM file.

SOURCE LANGUAGE CONSIDERATIONS—VSAM FILES

While a VSAM sequential file (ESDS) is similar to other sequential files and a VSAM direct file (RRDS) is similar to other direct files, their organizations are actually different from other sequential and direct files, and the same source language can give different results. You must take these differences into account to get the results you expect.

When you're processing VSAM files, you can use all the VS FORTRAN input/output statements.

However, the ENDFILE statement has no meaning for a VSAM file and is treated as documentation. If your program contains an ENDFILE statement and processes a VSAM file, you'll get a warning message to inform you of this.

Figure 41 summarizes the FORTRAN input/output statements you can use with each form of access.

File Type	VSAM Sequential (ESDS)	VSAM Direct (RRDS)			VSAM-Managed Sequential
Access	Sequential	Sequential		Direct	Sequential
VALID FORTRAN STATEMENTS		Empty File:	Nonempty File:		
	OPEN (sequential)	OPEN (sequential)	OPEN (sequential)	OPEN (direct)	
	WRITE	WRITE		WRITE (update or replace)	WRITE
	READ		READ	READ	READ
	BACKSPACE	BACKSPACE (has effect of CLOSE)	BACKSPACE		BACKSPACE
	REWIND	REWIND (has effect of CLOSE)	REWIND		REWIND
					ENDFILE
	CLOSE	CLOSE	CLOSE	CLOSE	CLOSE

Figure 41. FORTRAN Statements Valid with VSAM Files

In some instances, the VSAM input/output statements have a different effect than they have for other file processing techniques. The differences are documented in the following sections.

Processing VSAM Sequential Files

VSAM sequential files use VSAM Entry Sequenced Data Sets (ESDS); processing of such files can only be sequential.

When you're processing VSAM sequential files, there are special considerations for the OPEN, CLOSE, READ, WRITE, BACKSPACE, and REWIND statements, as described in the following paragraphs.

OPEN AND CLOSE STATEMENTS—VSAM SEQUENTIAL FILES: When your program processes a VSAM sequential file, you must specify the OPEN statement. For VSAM sequential files, specify:

```
ACCESS='SEQUENTIAL'
```

For VSAM files, the STATUS specifier of an OPEN statement may not be NEW or SCRATCH, and the STATUS specifier of a CLOSE statement may not be DELETE.

USING THE READ STATEMENT—VSAM SEQUENTIAL FILES: The READ statement for a VSAM sequential file has the same effect it has for other sequential files; records are retrieved in the order they are placed in the file. Therefore, you must use the sequential forms of the READ statement.

USING THE WRITE STATEMENT—VSAM SEQUENTIAL FILES: For VSAM sequential files, the WRITE statement places the records into the file in the order that the program writes them. If a VSAM sequential file is nonempty when your program opens it, a WRITE statement always adds a record at the end of the existing records in the file; thus you can extend the file without first reading all the existing records in the file.

Once you've written a record into a VSAM sequential file, you can only retrieve it; you cannot update it. Thus, when processing a VSAM sequential file, you can't update records in place. That is, if you code the following statements:

```
READ ...
BACKSPACE ...
WRITE ...
```

the WRITE statement does not update the record you have just retrieved. Instead, it places the updated record at the end of the file. (If you want to update records, you should define the VSAM file as direct. See the following section on "Processing VSAM Direct Files".)

USING THE BACKSPACE STATEMENT—VSAM SEQUENTIAL FILES: For VSAM sequential files, you can use the BACKSPACE statement to make the last record processed the current record:

- For a READ statement followed by a BACKSPACE statement, the current record is the record you've just retrieved. You can then retrieve the same record again.
- For a WRITE statement followed by a BACKSPACE statement, the current record is the record you've just written, that is, the last record in the file. You can then retrieve the record at this position.

USING THE REWIND STATEMENT—VSAM SEQUENTIAL FILES: The REWIND statement for VSAM sequentially accessed files has the same effect it has for other sequential files: the first record in the file becomes the current record.

For VSAM sequential files, this means that you can rewind the file and then process records for retrieval only. If you attempt to update the records, you'll simply add records at the end of the file.

After a BACKSPACE or REWIND statement is executed, you cannot update the current record. If you attempt it, you'll simply add another record at the end of the file.

Processing VSAM Direct Files

VSAM direct files use VSAM Relative Record Data Sets (RRDS). You can process VSAM direct files using either direct or sequential access.

Using direct access, you supply the relative record number of the record you want to process. You should use direct access when there are gaps in the relative record sequence for the file, or when you want to update records in place.

Using sequential access, you access each record in turn, one after another, and you have no control over the relative record number. For this reason, if you use sequential access to load the file, there should be no gaps in the relative record number sequence.

When you're processing VSAM direct files, there are special considerations for the OPEN and CLOSE statements, and for sequential and direct access, as described in the following paragraphs.

USING OPEN AND CLOSE STATEMENTS—VSAM DIRECT FILES: When your program processes a VSAM direct file, you must specify the OPEN statement. The options you can use are:

- ACCESS='SEQUENTIAL' for sequential access
- ACCESS='DIRECT' for direct access

For VSAM files, the STATUS specifier of an OPEN statement may not be NEW or SCRATCH, and the STATUS specifier of a CLOSE statement may not be DELETE.

USING SEQUENTIAL ACCESS—VSAM DIRECT FILES: You can use sequential access to load (place records into) an empty VSAM direct file using the WRITE statement, or to retrieve records from a VSAM direct file using the READ statement. The records are processed sequentially, one after the other, exactly as a sequential file is processed, and the relative record numbers of the records are ignored. In other words, when you're loading the file, there should not be any gaps in the relative record number sequence, because space for any missing records will not be reserved in the file.

For a direct file opened in the sequential access mode, you can use the WRITE statement only to load (place records into) a file that is empty when the file is opened. During loading, if you specify a BACKSPACE or REWIND statement, you cannot specify any more WRITE statements.

If the sequentially accessed VSAM direct file already contains one or more records when it is opened and you issue a WRITE statement, your program is terminated. In other words, for a VSAM direct file opened in the sequential access mode, once the file is loaded, you can't add or update records with FORTRAN programs. (For updating and adding records, you must use direct access.)

The READ statement for a sequentially accessed VSAM direct file, retrieves the records in the order they are placed in the file. The VS FORTRAN program gives you no way of determining the relative record number of any particular record you retrieve.

(If you need to use the relative record number, you must use direct access.)

Except during file loading, the REWIND statement for a sequentially accessed VSAM direct file has the same effect it has for VSAM sequential files: the first record in the file becomes the current record, which is then available for retrieval. During file loading, the REWIND statement has the same effect as a CLOSE statement followed by an OPEN statement; the first record in the file is then available for retrieval.

Except during file loading, the BACKSPACE statement for a sequentially accessed VSAM file has the same effect it has for VSAM sequential files; the last record processed becomes the current record, which is then available for retrieval. During file loading, the BACKSPACE statement has the same effect as a CLOSE statement, followed by an OPEN statement, followed by file positioning to the last record written; the last record in the file is then available for retrieval.

USING DIRECT ACCESS—VSAM DIRECT FILES: You can use direct access to place records into a VSAM direct file using the WRITE statement, or to retrieve records from a VSAM direct file using the READ statement.

For VSAM direct files, if the relative record numbers for the file are not strictly sequential—that is, if there are gaps in the key sequence, for example:

```
  1, 2, 3, 10, 12, 15, 16, 17, 20
```

—you must load (create records in) the file, using direct access WRITE statements to provide the relative record number for each record you write.

Otherwise (if the relative record numbers for the file _are_ strictly sequential—no gaps), you should sort the records according to the ascending order of their record numbers and then load them into the file using sequential access. This is because sequential access is faster than direct access.

For a VSAM direct file opened in the direct access mode, a WRITE statement uses the relative record number you supply to place a new record into the file, or to update an existing record.

The method you follow, either for record insertion or record update, is as follows:

1. In the OPEN statement, specify ACCESS='DIRECT' for the file.

2. Set the REC variable to the relative record number of the record to be inserted or updated.

3. Then code the WRITE statement, using the preset REC variable.

4. Repeat steps 2 and 3 until you've processed all the records you need to process.

When you are loading (initially placing records into) a file, you must not use duplicate record numbers during processing. In other words, you are not allowed to update records while you are loading the file.

To retrieve records from a directly accessed VSAM direct file, use the direct access forms of the READ statement. You cannot open the same file in the same programming unit for both sequential and direct access processing.

Don't execute the BACKSPACE or REWIND statements with a directly accessed VSAM direct file; if you do, your program is terminated.

Processing VSAM-Managed Sequential Files

You cannot use an OPEN statement with a VSAM-managed sequential file.

For VSAM-managed sequential files, you can use all other input/output statements valid for other non-VSAM sequential files.

See the section "Using Sequential Files" on page 161.

Obtaining the VSAM Return Code—IOSTAT Option

VS FORTRAN uses the VSAM program to process all VSAM requests.

If you specify the IOSTAT option for VSAM input/output statements, and an error occurs while VSAM is processing it, you'll get the VSAM return code for the operation attempted in the IOSTAT data item.

(If the error occurrs while FORTRAN is processing it, you'll get an IOSTAT code of +1.)

The VSAM return code is formatted in the IOSTAT data item as follows:

1. The VSAM return code is placed in the first two bytes.

2. The VSAM reason code is placed in the second two bytes.

IBM EXTENSION

To inspect the codes, you can equivalence the IOSTAT variable with two integer items, each of length 2. After a VSAM input/output operation, you can then write out the two integer items, which contain the pair of VSAM codes.

END OF IBM EXTENSION

The VSAM documentation for the system you're operating under gives the meaning of these return and reason codes. See the list of "Related Publications" at the beginning of this manual for VSAM publications titles.

DEFINING A VSAM FILE

To define and use a VSAM file, you must first define a catalog entry for the file, using Access Method Services commands. When you execute the commands, you create a VSAM catalog entry for the file.

You use Access Method Services to create a catalog entry for your file. The form of the entry depends upon the kind of file you'll be creating: a VSAM sequential file (ESDS), VSAM direct file (RRDS), or a VSAM-managed sequential file.

For VSAM sequential and direct files, the following examples assume that the data space your file is using has already been defined as VSAM space by the system administrator.

For reference documentation about the DEFINE commands, see the VS FORTRAN Application Programming: System Services Reference Supplement.

Defining a VSAM Sequential File

To define a VSAM sequential file (ESDS), you can specify:

```
DEFINE CLUSTER                                      -
       (NAME(MYFILE)                                -
       FILE(MYFILE1)                                -
       VOLUMES(666666)                              -
       NONINDEXED                                   -
       RECORDS(180)                                 -
       RECORDSIZE(80 200)                           -
       CATALOG(USERCAT))
```

which defines a file named MYFILE1 as a VSAM file.

NONINDEXED specifies that this is a VSAM sequential file (ESDS).

VOLUMES(666666) specifies that the file is contained on volume 666666.

RECORDS(180) specifies that there can be a maximum of 180 records in the space.

RECORDSIZE(80 200) specifies that the average length of the records in the file is 80 bytes, and the maximum length of any record is 200 bytes.

CATALOG(USERCAT) specifies the catalog in which this file is entered.

To actually create the entry in the catalog, you must then use job control statements to execute the DEFINE CLUSTER command.

Defining a VSAM Direct File

To define a VSAM direct file (RRDS), you can specify:

```
DEFINE CLUSTER                                  -
       (NAME(MYFILE2)                           -
       FILE(MYFILE3)                            -
       VOLUMES(666666)                          -
       NUMBERED                                 -
       RECORDS(200)                             -
       RECORDSIZE(80 80)                        -
       CATALOG(USERCAT))
```

which defines a file named MYFILE3 as a VSAM file.

NUMBERED specifies that the file is a VSAM direct file (RRDS).

VOLUMES(666666) specifies that the file is contained on volume 666666.

RECORDS(200) specifies that there can be a maximum of 200 records allowed in the space.

RECORDSIZE(80 80) specifies that all the records in the file are 80 bytes long.

CATALOG(USERCAT) specifies the catalog in which this file is entered.

To actually create the entry in the catalog, you must then use job control statements to execute the DEFINE CLUSTER command.

Defining a DOS/VSE VSAM-Managed Sequential File

To define a DOS/VSE VSAM-managed sequential file (using the DOS/VSE VSAM Space Management for SAM feature), you can specify:

```
DEFINE CLUSTER                                      -
       (NAME(MYFILE3)                               -
        NONINDEXED                                  -
        RECORDFORMAT(VB120)                         -
        RECORDSIZE(500)                             -
        RECORDS(200)                                -
        VOLUMES(666666))
```

which defines a sequential file named MYFILE3, suballocated in VSAM space.

RECORDFORMAT(VB120) specifies that the file has variable blocked format with average logical records 120 bytes long.

RECORDSIZE(500) specifies that there are four logical records in each block; 120 bytes for each logical record, plus 4 bytes for each record descriptor, plus 4 bytes for the block descriptor.

RECORDS(200) specifies that there can be a maximum of 200 records in the space.

VOLUMES(666666) specifies that the file is contained on volume 666666.

To actually create the entry in the catalog, you must then use job control statements to execute the DEFINE CLUSTER command.

PROCESSING DEFINE COMMANDS

Once you've created your DEFINE command, you must execute it, using Access Method Services, to create an entry in a VSAM catalog. The job control statements you use depend upon the system you're operating under: VM/370-CMS, OS/VS, or DOS/VSE.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS2-TSO considerations, see "Using OS/VS2-TSO with VS FORTRAN."

Processing DEFINE Commands—OS/VS

Under OS/VS, you specify the following job control statements to catalog your VSAM DEFINE CLUSTER commands:

```
//VSAMJOB   JOB
//STEP      EXEC  PGM=IDCAMS
//SYSPRINT  DD    SYSOUT=A
//MYFILE1   DD    VOL=SER=MYVOL,UNIT=SYSDA,DISP=OLD
//SYSIN     DD    *

 (The DEFINE CLUSTER command as data)

/*
//
```

If the DEFINE command FILE name is MYFILE1, then this is the DD name you specify in the DD control statement.

Processing DEFINE Commands—DOS/VSE

Under DOS/VSE, you specify the following job control statements to execute your DEFINE commands, both DEFINE CLUSTER and DEFINE NONVSAM:

```
// JOB DEFINE
// DLBL MYFILE2
// EXTENT  (as required)
// EXEC IDCAMS,SIZE=AUTO

 (The DEFINE command as data)

/*
/&
```

If the DEFINE command FILE name is MYFILE2, then this is the file name you specify in the DLBL control statement.

CREATING AND PROCESSING VSAM FILES

Once you've created a catalog entry for the file, you can actually load the file (place records in it for the first time). You can load and process the file using VS FORTRAN statements.

When you execute the FORTRAN program, you must define the file using job control statements for the system you're using.

For reference documentation about job control statements, see the VS FORTRAN Application Programming: System Services Reference Supplement.

Creating and Processing VSAM Files—OS/VS

Under OS/VS, when you execute a FORTRAN program to create or process a VSAM file, you define the file in a DD statement.

For example, to process MYFILE1 in a FORTRAN load module called MYPROG, you specify:

```
//VSAM1    JOB
//         EXEC PGM=MYPROG
//MYFILE1 DD DSN=MYFILE,DISP=SHR
//
```

When MYPROG is executed, the DD statement makes MYFILE1 (and the information in its catalog entry) available to the program. In the FORTRAN OPEN statement, MYFILE1 is the name you use for the FILE option.

Creating and Processing VSAM Files—DOS/VSE

Under DOS/VSE, when you execute a FORTRAN program to create or retrieve records in a VSAM file, you define the file in DLBL and EXTENT statements.

For example, to process MYFILE1 in a FORTRAN load module (phase) called MYPROG, you specify:

```
// JOB VSAM1
// DLBL MYFILE1
// EXTENT SYS015,VSAMVOL
// EXEC MYPROG,SIZE=xxK
//
```

When MYPROG is executed, the DLBL and EXTENT statements make MYFILE1 (and the information in its catalog entry) available to the program.

In the EXTENT statement, you need specify only the logical unit (SYS015) and the volume ID (VSAMVOL), which is the volume containing the VSAM catalog.

In the FORTRAN OPEN statement, the unit you specify must be equivalent to that specified in the EXTENT statement. Your system administrator can tell you the units valid for your organization.

For VSAM files, you must specify the SIZE parameter in the EXEC statement. Do not specify a size larger than the size of the partition the program will run in.

SYSTEM CONSIDERATIONS—INPUT/OUTPUT

For every file your program uses, you may need labels. Record formats the system uses are also of importance. Both are described in the following sections.

USING INPUT/OUTPUT LABELS

Files stored on magnetic tape devices can be labeled or unlabeled. Files stored on direct access devices must have labels. The labels are identifiers that help the system keep track of the files current at any one time.

The tape and direct access files you create can have volume labels, standard file labels, or user standard labels. The format and use of magnetic tape and direct access labels are different; therefore, the following sections discuss each separately.

Magnetic Tape Labels

Magnetic tape labels can be volume labels, standard file labels, or user standard labels.

VOLUME LABELS—TAPE FILES: Whenever you specify standard or user labels in your program, the system also creates volume labels. The first four characters identify the volume label.

Volume labels precede standard labels on the tape.

STANDARD FILE LABELS—TAPE FILES: When you specify standard labels, the system creates standard file labels. The first three characters specify if this is a header, end-of-volume, or end-of-file label.

Standard file labels follow volume labels and precede user standard labels (if any) on the tape.

USER STANDARD LABELS—TAPE FILES: The system uses the first four characters to identify the label as a user header or trailer label; you supply the contents of the remaining 76 characters.

Your user standard labels follow the standard file labels on the tape.

When you specify user standard labels, you should also create an assembler routine to process them.

PROCESSING MAGNETIC TAPE LABELS: Whenever you process a labeled tape file, the system processes the labels:

When you're creating the file, the system creates labels for it.

When you're retrieving the file, the system checks the labels.

The system processes volume and standard header labels when you open the file.

The system processes trailer labels when you close the file, or when the physical end of a reel is reached.

The system processes end-of-volume labels when an end-of-reel indicator is reached.

The system processes end-of-file trailer labels when you close the file.

OS/VS Tape Label Considerations

You specify magnetic tape labels through the LABEL parameter of the DD statement; through this parameter, you can specify the position of the file on the tape, the type of label, if the data set is password protected, and the type of file processing allowed.

Reference documentation for the DD statement is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

For additional detail on magnetic tape label processing, see the OS/VS Tape Labels manual.

DOS/VSE Tape Label Considerations

You specify magnetic tape labels through the TLBL statement; through this statement, you can specify the position of the file on the tape, the generation and version number for this file, and the expiration date.

Using the LBLTYP statement, when you're processing labeled tape files, you must also tell the linkage editor to reserve storage for label processing.

Reference documentation for the TLBL and LBLTYP statements is given in the VS FORTRAN Application Programming: System Services Reference Supplement.

For additional detail on magnetic tape label processing, see the DOS/VSE Tape Labels manual.

Direct Access Device Labels

The system uses direct access device labels to identify and protect files stored on such devices. Files on direct access devices must be labeled.

For each volume on a direct access device, there's always a volume label. For each file on that volume, there's always a standard file label. For each file, you can also specify and process user standard labels.

Volume Labels

Volume labels are preceded by a 4-byte key field. The key field and the first four bytes of the data field each contain the volume identification.

STANDARD FILE LABELS: Standard file labels can be in one of the following three formats:

- Sequential file format
- Direct file format

- DOS/VSE format for files using more than three extents on one volume

USER STANDARD LABELS: User standard labels have the same format as user standard labels for magnetic tape files.

If you specify user standard labels, you should provide an assembler language subroutine to process them.

Processing Direct Access Device Labels

The system processes the volume and standard file labels when you open and close the file.

OS/VS Direct Access Label Considerations

You specify direct access labels through the LABEL parameter of the DD statement; through this parameter, you can specify the position of the file on the volume, the type of label, if the data set is password protected, and the type of file processing allowed.

Reference documentation for the DD statement is given in the _VS FORTRAN Application Programming: System Services Reference Supplement_.

For additional detail on direct access label processing, see the _OS/VS Data Management Services Guide_.

DOS/VSE Direct Access Label Considerations

You specify direct access labels through the DLBL statement; through this statement, you can specify the identification of the file on the volume, the type of data set label to be used (sequential or direct), and the expiration date.

Reference documentation for the DLBL statement is given in the _VS FORTRAN Application Programming: System Services Reference Supplement_.

For additional detail on direct access label processing, see _DOS/VSE DASD Labels_.

DEFINING FORTRAN RECORDS—SYSTEM CONSIDERATIONS

Your FORTRAN programs must define the characteristics of the data records it will process: their formats, their record length, their blocking, and the type of device upon which they reside.

For CMS considerations about record formats, see "Using VM/370-CMS with VS FORTRAN."

Record Formats—OS/VS

Under VS FORTRAN, you can specify the format of the data records as:

Fixed-Length Records
All the records in the file are the same size and each is wholly contained within one block. Blocks can contain more than one record, and there is usually a fixed number of records in each block.

Variable-Length Records
The records can be either fixed or variable in length. Each record must be wholly contained within one block. Blocks can contain more than one record.

Each record contains a record-descriptor field, and each block contains a block-descriptor field. These fields are used by the system; they are not available to FORTRAN programs.

Spanned Records
The records can be either fixed or variable in length and each record can be larger than a block. If a record is larger than the remaining space in a block, a segment of the record is written to fill the block. The remainder of the record is stored in the next block (or blocks, if required). Only complete records are made available to FORTRAN programs.

Each segment in a block, even if it is the entire record, includes a segment-descriptor field, and each block includes a block-descriptor field. These fields are used by the system; they are not available to FORTRAN programs.

Undefined-Length Records
The records may be fixed or variable in length. There is only one record per block. There are no record-length, block-descriptor, or segment-descriptor fields.

SEQUENTIAL EBCDIC DATA SETS: You can define FORTRAN records in an EBCDIC data set as formatted or unformatted, that is, they may or may not be defined in a FORMAT statement. List-directed I/O statements are considered formatted.

You can specify formatted records as fixed length, variable length, or undefined length.

You can specify unformatted records only as variable length.

If you're processing records using asynchronous input/output, the records must not be blocked.

Unformatted Records: Unformatted records are those not described by a FORMAT statement. The size of each record is determined by the input/output list of READ and WRITE statements.

Always specify unformatted records as variable and spanned.

In addition, they may be blocked or unblocked.

Use blocked records wherever possible; blocked records reduce processing time substantially.

SEQUENTIAL ASCII DATA SETS: ASCII data sets may have sequential organization only. For system considerations, see the documentation for the system you're using.

FORTRAN records in an ASCII data set must be formatted and unspanned and may be fixed length, undefined length, or variable length records.

DIRECT-ACCESS DATA SETS: FORTRAN records may be formatted or unformatted, but must be fixed in length and unblocked only.

The OPEN statement specifies the record length and buffer length for a direct-access file. This provides the default value for the block size.

Defining Records—OS/VS

Under OS/VS, you define data record characteristics through the DCB parameter of the DD statement.

Through the DCB parameter, you can specify:

- Record format—fixed length, variable length, or undefined

- Record length—either the exact length (fixed or undefined), or the length of the longest record (variable)
- Blocking information—such as the block size
- Buffer information—the number of buffers to be assigned
- Whether the data set is encoded in the EBCDIC or the ASCII character set
- Special information for tape files
- Special information for direct access files
- Information to be used from another data set

For reference documentation on the DCB parameter, see the VS FORTRAN Application Programming: System Services Reference Supplement.

Record Formats—DOS/VSE

The DOS/VSE system keeps control information about your data files in an internal DTF table. There is a DTF table for each logical unit, including the system logical units, that your program uses—except for SYSLOG.

The system builds the DTF tables dynamically as each logical unit is opened for the first time, using information it obtains from your program. Guidance information on the DTF tables is included in DOS/VSE Data Management Concepts and DOS/VSE Macro User's Guide; reference information on the DTF tables is included in DOS/VSE Macro Reference.

DOS/VSE FORTRAN produces and accepts records that have a particular format depending on logical unit class, device type, and the type of FORTRAN input/output operations applying to that record.

The maximum length of a formatted record depends on the logical unit or device, as shown in Figure 42.

DOS/VSE Logical Unit Name	Device Permitted	Type Of Operation	Maximum Record Length
SYSIPT SYSIN	Card reader, tape unit, or disk storage unit	Input	80 bytes
SYSPCH	Punch card device, tape unit, or disk storage unit	Output	80 bytes
SYSLST	Printer, tape, or disk storage unit	Output	145 bytes for tape and printer 121 bytes for disk.
SYSLOG	Console typwriter or printer	Output	256 bytes for console typewriter. Printer—the number of print positions plus one for carriage control.

Figure 42. DOS/VSE Logical Units and Devices Allowed

By device, the maximum size permitted for each record follows:

Device	**Maximum Bytes**
Card Reader	80
Card Punch	80
Printer	Number of print position, plus one byte for carriage control
Tape	260
Disk (sequential access)	Device Dependent (see Appendix A.)
Direct access	As specified in an OPEN statement.

For unformatted input/output, a single WRITE or READ may cause the transfer of more bytes than are contained in a single record. The system organizes such data into two or more records.

The system provides eight bytes of control information at the beginning of each record block. This information indicates the size of the record and whether it is part of a record that is continued in one or more other blocks. There is never more than one record for each block.

When this control information is needed it is provided and maintained by the system. There's no need to consider it in writing your input/output instructions.

The first four bytes of control information for unformatted records constitute a block descriptor word; the next four bytes constitute a segment-descriptor word.

CODING CALLING AND CALLED PROGRAMS

You may need to write programs which require a specific operation to be performed again and again, with different data for each repetition; for example, a complex mathematical operation.

You can simplify the writing of such programs if you write the statements in a separate subprogram that performs the repetitive operation once; you can then simply refer to the subprogram throughout the program, as if the operation itself were inserted at the points you need to use it.

For example, you can write a general routine to take the cube root of any number; you can then link-edit that routine, as an external reference, with any program in which cube root calculations are required.

The program that requires the services of the generalized routine is the calling program.

The generalized routine itself (for example, the cube root routine) is the called program.

Calling and called programs make up a hierarchy of programs. The first program unit to call others is the main program, which is required in every program unit. The main program invokes subprograms; however, subprograms cannot invoke the main program.

Subprograms can invoke other subprograms to any depth. However, a subprogram cannot invoke itself, and it cannot invoke any subprogram in the hierarchy that invoked it. For example:

A	invokes	B and E
B	invokes	C and F
C	invokes	D

Program D cannot invoke program A, B, or C; it can, however, invoke program E or F.

Kinds of Called Programs

Called programs in VS FORTRAN can be of three kinds:

- **Subroutine Subprograms**—which are invoked in the calling program through the CALL statement. For example:

  ```
  CALL CROOT (INTR)
  ```

  This CALL statement makes the value in INTR available to subprogram CROOT and transfers control to the first executable statement in subprogram CROOT. When CROOT has completed execution, control is transferred back to the main program.

  The point at which execution resumes depends upon the processing within the subprogram.

  Depending upon the logic of the program, CROOT may or may not return a value in INTR.

- **Function Subprograms**—which are invoked in the calling program through function references. For example:

  ```
  ANS = LNGTH * CROOT1(INTR)
  ```

When this statement is executed, the main program makes the value in INTR available to function subprogram CROOT1 and transfers control to the first executable statement in it.

As soon as CROOT1 finishes processing, control is returned to the main program together with a value that is multiplied by LNGTH to give ANS.

- **FORTRAN-Supplied Intrinsic Functions**—which are a special form of function subprograms with fixed names, available in the execution-time library. The intrinsic functions perform often-used mathematical and character functions, such as obtaining logarithms of numbers, exponentiation, trigonometric and hyperbolic evaluations, and character manipulations.

  For example, to obtain the square root of INTR, you can use the SQRT function:

  ```
        ANS = LNGTH * SQRT(INTR)
  ```

  which executes the SQRT intrinsic function using the value of INTR; the function then returns a value to the calling program, which then uses the returned value in executing the assignment statement, multiplying LNGTH by the value returned by SQRT and placing the result in ANS.

  The names ARSIN, ARCOS, DARSIN, and DARCOS are intrinsic function names when using the option LANGLVL(66). The corresponding names for LANGLVL(77) are ASIN, ACOS, DASIN, and DACOS. The extended precision names for LANGLVL(66) and LANGLVL(77) are QARSIN and QARCOS.

  If you use the names ARSIN, ARCOS, DARSIN, or DARCOS under LANGLVL(77), the corresponding functions are considered to be external; that is, it is assumed you are supplying these functions in your library or as one of your subprograms. However, if you fail to supply your own and are using the MVS or CMS system, the corresponding FORTRAN functions supplied for the LANGLVL(66) names will be obtained from the library and used.

  However, if your are using DOS, this resolution is not made and you must supply your own function. In this case, no indication is given to the non-DOS user that the resolution was made to the FORTRAN library routine instead of to your own routine.

SHARING DATA BETWEEN PROGRAMS

Calling and called programs can share data between them, as we have seen in the previous examples.

In FORTRAN, there are two ways to share data: by passing arguments (data names identifying data items) between the programs, or by using common data areas (areas that can be shared by more than one program).

PASSING ARGUMENTS BETWEEN PROGRAMS

You can pass data values between a calling program and a subprogram through the use of paired lists of actual and dummy arguments. The paired lists must contain the same number of items, and be in the same order; in addition, items paired with each other must be of the same type and length. You can use such paired lists in both SUBROUTINE and FUNCTION subprograms.

Passing Arguments to a FUNCTION Subprogram

You can use actual and dummy arguments when you're invoking a function subprogram. If your main program contains the following function reference:

```
      G = B * ZCALC(INUM,X,Y)
```

the actual arguments in function ZCALC are INUM, X, and Y; they contain the actual values you want to make available to the function subprogram.

In the ZCALC function subprogram, you define the dummy arguments:

```
      FUNCTION ZCALC(M,X,ZZ)
```

The dummy arguments of function subprogram ZCALC are M, X, and ZZ.

When the calling program executes the statement containing the function reference, the values in the actual arguments are made available to the dummy arguments:

The Value of:	Is Made Available in:
INUM	M
X	X
Y	ZZ

again, according to their positions in the argument lists.

M, X, and ZZ can then be used in operations within the function subprogram.

When control returns to the calling program, a value is returned to the calling program; then the assignment statement is executed, using the value returned.

Passing Arguments to a SUBROUTINE Subprogram

You can use actual and dummy arguments to pass data between a calling program and a SUBROUTINE subprogram. For example, if the calling program contains the statement:

```
      CALL MAXNUM(PI,FOURV,XYZ,BIGM,HH)
```

PI, FOURV, XYZ, BIGM, and HH are actual arguments; they contain values you want to make available to the SUBROUTINE subprogram.

The MAXNUM subprogram, in order to make the values available, must contain a matching list of dummy arguments:

```
      SUBROUTINE MAXNUM(A,B,C,D,E)
```

The dummy arguments of SUBROUTINE subprogram MAXNUM are A, B, C, D, and E.

When the CALL statement is executed, the addresses of the actual arguments are used as the addresses of the matching dummy arguments:

The Address of:	Becomes the Address of:
PI	A
FOURV	B
XYZ	C
BIGM	D
HH	E

When MAXNUM is executed, the newly assigned values of A, B, C, D and E can be used in operations.

When control returns to the calling program, the current values in A, B, C, D, and E are also the current values of PI, FOURV, XYZ, BIGM, and HH in the calling program.

General Rules for Arguments

You must define dummy arguments to correspond in number, order, and type with the actual arguments. For example, if you define an actual argument as an integer constant of length 4, you must define the corresponding dummy argument as an integer of length 4.

Actual arguments are passed by name; if you alter the value of an argument in the subroutine or function subprogram, you're altering the value in the calling program as well.

If you define an actual argument as an array, then the size of your paired dummy array must not exceed the size of the actual array.

If you define a dummy argument as an array, you must define the corresponding actual argument as an array or an array element.

If you define the actual argument as an array element, your paired dummy array must not be larger than the part of the actual array which follows and includes the actual array element you specify.

Assigning Argument Values

If your subprogram assigns a value to a dummy argument, you must ensure that its paired actual argument is a variable, an array element, or an array. Never specify a constant or expression as an actual argument, unless you are certain that the corresponding dummy argument is not assigned a value in the subprogram.

Your subprograms should not assign new values to dummy arguments that are associated with other dummy arguments in the subprogram, or with variables in COMMON. You may get unexpected results, but the compiler cannot give you a warning message.

For example, if you define the subprogram DERIV as:

```
SUBROUTINE DERIV (X,Y,Z)
COMMON W
```

and if you include the following elements in the calling program:

```
COMMON B
  .
  .
  .
CALL DERIV (A, B, A)
```

the DERIV subprogram should not assign new values to X, Y, Z, and W:

X and Z because they are both associated with the same argument, A.

Y because it is associated with argument B, which is in COMMON.

W because it also is associated with B.

For reference documentation about dummy arguments and actual arguments, see the VS FORTRAN Application Programming: Language Reference manual.

SHARING DATA STORAGE—COMMON STATEMENT

You can use the COMMON statement to share data storage areas between two or more program units, and to specify the names of variables and arrays occupying the shared area.

There are two reasons why you might want to share data storage:

1. To conserve storage, by using only one storage allocation for variables and arrays used by several program units

2. To implicitly transfer arguments between program units

Arguments passed in a common area are subject to the same rules as arguments passed in a SUBROUTINE subprogram argument list (see "General Rules for Arguments").

For reference documentation about the COMMON statement, see the VS FORTRAN Application Programming: Language Reference manual.

Data Item Order—COMMON Statement

Entries in a common area share storage locations; therefore, the order in which you specify them is significant when you use the common area to transmit arguments. For example, you specify the following COMMON statements in the calling program:

```
COMMON A, B, C, R(100)
REAL A, B, C
INTEGER R
```

and you specify the following COMMON statement in your subprogram:

```
COMMON X, Y, Z, S(100)
REAL X, Y, Z
INTEGER S
```

Figure 43 shows the order in which these variables and arrays are placed in the common area, and how they share common storage. Each column of variables starts at the beginning of the common area. Variables on the same line share the same storage location.

```
Calling Program   Shares   Called Program   Displacement Bytes From
Item              With     Item             Beginning of COMMON

-----                       -----           0
  A                           X
-----                       -----           4
  B                           Y
-----                       -----           8
  C                           Z
-----                       -----           12
R(1)                        S(1)
-----                       -----           16
R(2)                        S(2)
-----                       -----           20
  .                           .
  .                           .
  .                           .
-----                       -----           408
R(100)                      S(100)
-----                       -----           412
```

Figure 43. Example of Shared Data Areas—COMMON Statement

The calling program can assign values to A, B, C, and R, and those values are available to X, Y, Z, and S of the subprogram.

Conversely, the subprogram can assign values to X, Y, Z, and S, and those values are available to A, B, C, and R of the calling program.

Using the COMMON statement in this way, you can share the values in a number of data items without transmitting them in the argument list of a CALL statement.

Type and Length Considerations—COMMON Statement

In order to pass arguments using the COMMON statement, you must define the items that are to share common storage with the same type and length.

For example, if you define a common area in a main program and in three subprograms, as follows:

Main Program:	COMMON A,B,C (A and B are 8 storage locations, C is 4 storage locations)
Subprogram 1:	COMMON D,E,F (D and E are 8 storage locations, F is 4 storage locations)
Subprogram 2:	COMMON Q,R,S,T,U (4 storage locations each)
Subprogram 3:	COMMON V,W,X,Y,Z (4 storage locations each)

How these variables are arranged within common storage is shown in Figure 44.

```
Main                                                               Displacement
Program        Subprogram 1      Subprogram 2      Subprogram 3    (Bytes)

-----          -----             -----             -----           0
                                   Q      <--->      V
  A    <--->     D               -----             -----           4
                                   R      <--->      W
-----          -----             -----             -----           8
                                   S      <--->      X
  B    <--->     E               -----             -----           12
                                   T      <--->      Y
-----          -----             -----             -----           16
  C    <--->     F      <--->      U      <--->      Z
-----          -----             -----             -----           20
```

Figure 44. Transmitting Values Between Common Areas

The main program can transmit values for A, B, and C to subprogram 1, provided that

- A is of the same type as D.
- B is of the same type as E.
- C is of the same type as F.

However, the main program and subprogram 1 cannot, by assigning values to the variables A and B, or D and E, respectively, transmit values to the variables Q, R, S, and T in subprogram 2, or V, W, X, and Y in subprogram 3, because the lengths of their common variables differ.

In the same way, subprogram 2 and subprogram 3 cannot transmit values to variables A and B, or to D and E.

Values can be transmitted between variables C, F, U, and Z if each is the same data type as the others.

Also, if each is the same data type, values can be transmitted between A and D, between B and E, and between Q and V, R and W, S and X, and T and Y.

However, any assignment of values to A or D destroys any values assigned to Q, R, V, and W (and vice versa); and any assignment to B or E destroys the values of S, T, X, and Y (and vice versa).

Efficient Arrangement of Variables—COMMON Statement

Your programs lose some object-time efficiency unless you ensure that all of the common variables have proper boundary alignment. (However, it isn't necessary for you to align COMPLEX, INTEGER, LOGICAL, or REAL variables; your programs will still execute correctly.)

You can ensure proper alignment either by arranging the non-CHARACTER type variables in a fixed descending order according to length, or by defining the block so that dummy variables force proper alignment.

FIXED ORDER OF VARIABLES—COMMON STATEMENT: If you use the fixed order, non-CHARACTER type variables must appear in the following order:

Length	Type
IBM EXTENSION	
32	COMPLEX
16	COMPLEX or REAL
8	REAL
END OF IBM EXTENSION	
8	COMPLEX or DOUBLE PRECISION
4	REAL, INTEGER, or LOGICAL
IBM EXTENSION	
2	INTEGER
1	LOGICAL
END OF IBM EXTENSION	

USING DUMMY VARIABLES—COMMON STATEMENT: If you don't use the fixed order, you can ensure proper alignment by constructing the block so that the displacement of each variable can be evenly divided by the reference length associated with the variable. (Displacement is the number of storage locations, or bytes, from the beginning of the block to the first storage location of the variable.) The reference length in bytes for each type of variable is as follows:

Type Specification	Length Specification	Reference Length (Bytes)
LOGICAL	4	4
INTEGER	4	4
REAL	4	4
DOUBLE PRECISION	8	8
COMPLEX	8	8

IBM EXTENSION

LOGICAL	1	1
INTEGER	2	2
REAL	8	8
REAL	16	8
COMPLEX	16	8
COMPLEX	32	8

END OF IBM EXTENSION

The first variable in every common block is positioned as though its length specification were 8. Therefore, you can assign a variable of any length as the the first in a COMMON block.

To obtain the proper alignment for the other variables in the same block, you may find it necessary to add a dummy variable to the block.

For example, your program uses the variables A, K, and CMPLX (defined as REAL*4, INTEGER*4, and COMPLEX*8, respectively) in a COMMON block defined as:

COMMON A, K, CMPLX

The displacement of these variables within the block is:

```
Variable      Displacement (Bytes)
              in COMMON

-----         0

  A

-----         4

  K

-----         8


CMPLX


-----         16
```

The displacements of K and CMPLX are evenly divisible by their reference numbers.

IBM EXTENSION

However, if you define K as an integer of length 2, then CMPLX is no longer properly aligned (its displacement of 6 is not evenly divisible by its reference length of 8). In this case, you can ensure proper alignment by inserting a dummy variable (DV) of length 2 either between A and K or between K and CMPLX.

```
Variable        Displacement (Bytes)
                in COMMON

-----           0
  A
-----           2
  DV
-----           4

  K

-----           8


CMPLX


-----           16
```

END OF IBM EXTENSION

EQUIVALENCE Considerations—COMMON Statement

When you use the EQUIVALENCE statement together with the COMMON statement, there are additional complications resulting from storage allocations. The following examples illustrate programming considerations you must take into account.

Your program contains the following items:

```
REAL  R4A, R4B, R4M(3,5), R4N(7)
DOUBLE PRECISION  R8A, R8B, R8M(2)
```

IBM EXTENSION

```
LOGICAL*1 L1A
```

END OF IBM EXTENSION

```
LOGICAL L4A
```

which are defined in COMMON as follows:

```
COMMON R4A, R8M, L1A , R8A, L4A, R4M
```

and which results in the following inefficient displacements:

Name	Displacement	Boundary
R4A	0	Doubleword
R8M	4	Word (should be doubleword)

IBM EXTENSION

L1A	20	Word

END OF IBM EXTENSION

R8A	21	Byte (should be doubleword)
L4A	29	Byte (should be word)
R4M	33	Byte (should be word)

Now add an EQUIVALENCE statement to this inefficient COMMON statement:

1. First Example (valid but inefficient):

```
EQUIVALENCE (R4M(1,1), R4B)
EQUIVALENCE (R4B, R8B)
```

This results in the following additional inefficiencies:

Name	Displacement	Boundary
R4B	33	Byte (same as R4M(1,1))
R8B	33	Byte (same as R4M(1,1) and R4B)

which means that now both R4B and R8B are now also inefficiently aligned.

2. Second Example (illegal):

```
EQUIVALENCE (R8A, R4N(7))
```

This is illegal, because the seventh element of R4N has the same displacement as R8A, or 21.

This means that the first element of R4N is located 24 bytes (4*6) before this, at displacement -3. It is illegal to extend a COMMON area to the left in this way.

3. Third Example (valid but inefficient):

```
EQUIVALENCE (R8A, R4N(2))
EQUIVALENCE (R4M, R4N(5))
```

This has the following results:

Name	Displacement	Boundary
R4N(2)	21	Byte
R4N(3)	25	Byte
R4N(4)	29	Byte
R4N(5)	33	Byte
R4M	33	Byte (same position as R4N(5)

This is valid because the EQUIVALENCE statement places R4M at displacement 33, the same displacement as that specified in the COMMON statement. However, it is inefficient because both R4N and R4M begin at a byte boundary.

4. Fourth Example (illegal):

```
EQUIVALENCE (R8A, R4N(2))
EQUIVALENCE (R4M, R4N(4))
```

This has the following illegal results:

Name	Displacement	Boundary
R4N(2)	21	Byte
R4N(3)	25	Byte
R4N(4)	29	Byte
R4N(5)	33	Byte
R4M	29	Byte (same position as R4N(4)

This is illegal, because the EQUIVALENCE statement (which places R4M at displacement 29) contradicts the COMMON statement (which places R4M at displacement 33). The COMMON statement controls the displacement of R4M, not the EQUIVALENCE statement.

BLANK AND NAMED COMMON

- There are two forms of common storage you can specify: blank common and named common.

Blank Common

In the preceding example, the common storage area (common block) is a blank common area, since you gave no name to the storage area. The variables you specified in the COMMON statements were assigned locations relative to the beginning of this blank common area.

When you specify items in blank common, you can't use a BLOCK DATA program to initialize them.

Named Common

You can name common storage areas (or blocks of storage)—known as named common. Blocks given the same name occupy the same space.

You can place variables and arrays in separate common areas

This lets you specify that a calling program is to share one common block with one subprogram, and to share another common block with another subprogram. It also gives you better program documentation.

Using Blank Common and Named Common

There are different FORTRAN rules for blank and named common areas, which may cause you to choose one type over the other, depending on what you want your program to do.

The differences are:

- You can define only one blank common block in an executable program, although you can specify more than one COMMON statement defining items in blank common; you cannot assign blank common a name. You can define many named common blocks, each with its own name.

- You can define blank common as having different lengths in different program units. You must define a given named common block with the same length in every program unit that uses it.

- You can't assign initial values to variables and array elements in blank common.

- In named common, you can assign initial values to variables and array elements, through a BLOCK DATA subprogram that contains DATA statements or explicit specification statements.

In a COMMON statement, you specify a common block name by enclosing it in slashes. The following example defines a named common block, PAYROL, that contains the variables FICA, MANHRS, SICKDA:

```
      COMMON/PAYROL/FICA,MANHRS,SICKDA
```

You can define blank common in a COMMON statement by omitting a name, and defining the blank common area first:

```
      COMMON A,C,G/PAYROL/FICA,MANHRS,SICKDA
```

and the variables A, C, and G are placed in blank common.

You can also specify blank common items after named common items, by placing two consecutive slashes before the list of blank common variables:

For example, in the following statement:

```
      COMMON A, C, G /PAYROL/FICA,MANHRS,SICKDA// JJ, VMN, LP7
```

you've defined the variables A, C, G, JJ, VMN, and LP7 in blank common, and in that order. You've defined PAYROL as named common, containing FICA, MANHRS, and SICKDA, in that order.

If you specify more than one COMMON statement in a program, the definitions are cumulative through the program. For example, if you specify the following two COMMON statements:

```
      COMMON A, B, C /R/ D, E /S/ F
      COMMON G, H /S/ I, J /R/P // W
```

they have the same effect as if you specified the single statement:

```
      COMMON A,B,C,G,H,W /R/ D, E, P /S/ F, I, J
```

Using Blank and Named Common—Example

The differences in usage between blank and named common are illustrated in the following example.

Your programs specify that A, B, C, K, X, and Y each occupy four locations of storage, H and G each occupy eight locations, and D, E, and F each occupy two locations.

```
  Calling Program                  Subprogram

  COMMON H, A /R/ X, D, E // B     SUBROUTINE MAPMY(...)
       .                           COMMON G, Y, C /R/ K, E, F
       .                              .
       .                              .
```

In the calling program, the statement:

```
      COMMON H, A /R/ X, D, E // B
```

reserves the following storage:

16 locations in blank common (eight locations for H, and four each for A and B)

Eight locations in named common R (four for X, and two each for D and E).

The following statement in the subprogram MAPMY:

```
      COMMON G,Y,C/R/K,E,F
```

causes the following common storage sharing:

In blank common, the variables G, Y, and C share the same storage as H, A, and B, respectively.

In named common, the variables K, E, and F share the same storage as X, D, and E respectively.

The common storage is laid out as shown in Figure 45.

```
                   Calling         Called     Displacement (Bytes)
                   Program         Program    in COMMON

Blank Common       -----           -----      0  Blank Common
                     H      <-->     G

                   -----           -----      8
                     A      <-->     Y
                   -----           -----      12
                     B      <-->     C
                   -----           -----      16

Named Common R     -----           -----      0 Named Common R
                     X      <-->     K
                   -----           -----      4
                    D/E     <-->    E/F
                   -----           -----      8
```

Figure 45. Blank and Named Common Storage Sharing

CODING MAIN PROGRAMS—CALLING PROGRAMS

When you're writing a series of program units that communicate with each other, the first program in the calling chain must be a main program.

The main program can invoke subprogram units, which in turn can invoke other subprogram units; however, none of the subprogram units can invoke the main program, and no program can invoke itself.

NAMING YOUR MAIN PROGRAM—PROGRAM STATEMENT

To give your main program a name, use the PROGRAM statement as the first statement in the program, as shown below.

```
       PROGRAM CUBES
       DO 10 I=12,26
       J = I**3
          .
          .
          .
 10   CONTINUE
      END
```

The PROGRAM statement names your main program CUBES. (The name is appropriate, since the program cubes the numbers from 12 to 26 and then manipulates those values, perhaps printing both I and J each time the program goes through the loop.)

If you don't use the PROGRAM statement, your main program is named MAIN.

For reference documentation about the PROGRAM statement, see the VS FORTRAN Application Programming: Language Reference manual.

INVOKING FORTRAN-SUPPLIED FUNCTIONS

There are a number of FORTRAN-supplied functions (intrinsic functions) you'll find useful, including mathematic functions to derive trigonometric values, logarithms, exponential values, maximum and minimum values, sign conversions, absolute values, error functions, and functions to manipulate character operands.

Invoke a FORTRAN-supplied function by referring to the function name within an arithmetic statement; the function name is replaced by the value returned from the invoked function, after the invoked function has completed its calculations.

For instance, you can invoke the FORTRAN-supplied function subprogram that returns the square root of a number, with statements in your program such as:

```
      Y = SQRT(X+Z) * 3.1
```

If the sum of X and Z is 9.0, then the square root of X+Y is 3.0, and the value assigned to Y would be 3.0 times 3.1, or 9.3.

In VS FORTRAN, you can use the generic function name, and the compiler will select the function your program actually should use,

```
      DOUBLE PRECISION  X, Z, Y
      Y = SQRT(X+Z) * 3.1
```

In this case, you've specified SQRT, the generic function name; however, during compilation the compiler selects the DSQRT function, which gives a double precision result.

When you are using the current language level and specify an intrinsic function name in an explicit type statement, the intrinsic function is not removed from its status as an intrinsic function; this is true whether you specify the predefined function type or whether you respecify it as another type. When the intrinsic function is executed, the mode used is the mode predefined to the compiler.

However, when you are using the old language level and specify an intrinsic function name in an explicit but conflicting type statement, you remove it from its intrinsic status and it becomes the name of a user-supplied external function.

Using FORTRAN-Supplied Functions as Arguments—INTRINSIC Statement

If you want to pass FORTRAN-supplied function names as arguments, you must specify an INTRINSIC statement in the invoking program:

```
INTRINSIC CSIN,CCOS,TAN,SINH,COSH
```

This lets you use the FORTRAN-supplied CSIN, CCOS, TAN, SINH, and COSH mathematical subroutines as arguments in your own functions or subroutine programs.

In the INTRINSIC statement, you can specify a given function name only once. That is, in the preceding example, it would be invalid to repeat CSIN at the end of the list of function names.

When you specify an INTRINSIC statement, it must precede all the statement function definitions and executable statements in the program unit.

For reference documentation about FORTRAN-supplied functions, see the VS FORTRAN Application Programming: Library Reference manual.

For reference documentation about the INTRINSIC statement, see the VS FORTRAN Application Programming: Language Reference manual.

Comparing Character Operands—FORTRAN-Supplied Functions

When your programs compare character operands using the following intrinsic functions:

LLT—logically less than

LGT—logically greater than

LLE—logically less than or equal to

LGE—logically greater than or equal to

the character operands are compared using the ASCII collating sequence.

For example, if C1 contains "3AB" and C2 contains "XYZ", the following comparison:

```
L = LGT(C1,C2)
```

will evaluate as "false."

(However, if the same operands are compared, using a relational operator:

```
L = C1.GT.C2
```

the EBCDIC collating sequence is used for the comparison, and the statement evaluates as "true.")

The EBCDIC and ASCII collating sequences are listed in the VS FORTRAN Application Programming: Language Reference manual.

INVOKING FUNCTION AND SUBROUTINE SUBPROGRAMS

FUNCTION and SUBROUTINE subprograms are useful for performing often-repeated routines that might be shared by many programs—such as error recovery routines for input/output statements, or often-repeated mathematical calculations.

Both types of subprograms are discussed in the following sections.

Invoking FUNCTION Subprograms

Invoke a function subprogram of your own in the same way you invoke an IBM-supplied function—through a program reference. The function name is replaced by the value returned from the invoked function, after the invoked function has completed its calculations.

For instance, you can invoke a function subprogram you've named BCALC, with the following statement in your invoking program:

```
R = BCALC(A+B) * 3.1
```

Your program calculates the sum of A and B and passes that value to the function subprogram BCALC; when BCALC completes executing, it returns the value to the invoking program, which then multiplies it by 3.1 to give the value of R.

FUNCTION SUBPROGRAMS AND THE EXTERNAL STATEMENT: You use the EXTERNAL statement with your FUNCTION subprograms in two different ways:

- If you name the program with an IBM-supplied function name, you must list the name in an EXTERNAL statement.

  For example, if you write your own square root routine, and you name it SQRT, you must specify it in an EXTERNAL statement:

  ```
  EXTERNAL SQRT
  ```

which tells the compiler that you want any SQRT references in your program to invoke your own SQRT routine rather than the IBM-supplied SQRT routine.

- If you want to pass a function subprogram as an argument, you must specify the name of the subprogram in an EXTERNAL statement.

When you specify an EXTERNAL statement, it must precede all the statement function definitions and executable statements in your program. The names you specify in an EXTERNAL statement can be names of external procedures, dummy procedures, or BLOCK DATA subprograms.

You can't use the same name in both an EXTERNAL statement and an INTRINSIC statement.

For reference documentation about the INTRINSIC and EXTERNAL statements, see the _VS FORTRAN Application Programming: Language Reference_ manual.

Invoking SUBROUTINE Subprograms—CALL Statement

To execute a SUBROUTINE subprogram at a certain point in a program, issue a CALL statement at that point in the invoking program. The CALL statement can optionally pass actual arguments to replace the dummy arguments in the called SUBROUTINE subprogram:

```
CALL OUT                          (no actual arguments passed)
CALL SUB1(X+Y*5,ABDF(IND),SINE)   (three actual arguments passed)
```

When it's executed, the CALL statement transfers control to the SUBROUTINE subprogram, and associates the dummy variables in the SUBROUTINE subprogram with the actual arguments that appear in the CALL statement, as shown in Figure 46.

```
Calling Program              SUBROUTINE Subprogram

DIMENSION X(90),Y(90)
   .                               SUBROUTINE COPY(A,B,N)
   .                               DIMENSION A(N),B(N)
   .                               DO 10 I = 1,N
CALL COPY (X,Y,90)              10 B(I) = A(I)
   .                               RETURN
   .                               END
   .
```

Figure 46. CALL Statement Execution

When the CALL COPY statement is executed:

The addresses of the actual arguments, array X and array Y, become the addresses of the dummy arguments, array A and array B, in the subprogram.

The variable N in the subprogram is associated with the value 90.

Thus a call to subprogram COPY, in this instance, results in the 90 elements of array X being copied into the 90 elements of array Y.

CODING SUBPROGRAMS—CALLED PROGRAMS

There are four types of subprograms that you can use in VS FORTRAN: FORTRAN-supplied functions, FUNCTION subprograms, SUBROUTINE subprograms, and BLOCK DATA subprograms.

FORTRAN-supplied functions let you use predefined calculations for commonly used mathematical and character manipulations.

FUNCTION and SUBROUTINE subprograms are useful when your program must perform the same set of computations with different data at various points in the program.

In a calling program, you invoke a FUNCTION subprogram by a statement reference to the function name. After the function has been executed, the function name has a value and can be used in a calculation just as a variable is used.

In a calling program, you invoke a SUBROUTINE subprogram by a CALL statement that specifies the subprogram name.

Your calling program can also invoke a SUBROUTINE subprogram by a CALL statement that specifies an ENTRY name within the subprogram.

You code a BLOCK DATA subprogram when you want to initialize variables in named common blocks. These subprograms are neither called nor invoked by a reference to the subprogram name; they initialize the variables at compile time.

CODING FUNCTION SUBPROGRAMS

The first statement in a FUNCTION subprogram (excluding debugging statements) is a FUNCTION statement, identifying the program. For example:

```
      FUNCTION TRIG (DELTA, THETA, ABSVAL)
```

This statement identifies the subprogram named TRIG as a FUNCTION subprogram, with dummy arguments DELTA, THETA, and ABSVAL.

The data type of the function is real of length 4—derived from the predefined naming conventions. (The data type of the function determines the data type of the value it returns to the invoking program.)

You can also explicitly specify the data type of the function:

```
      DOUBLE PRECISION FUNCTION TRIG (DELTA,THETA,ABSVAL)
```

which specifies that the data type of TRIG is real of length 8.

IBM EXTENSION

If you want TRIG to be a real function of length 8, you can alternatively specify:

```
      REAL FUNCTION TRIG*8(DELTA,THETA,ABSVAL)
```

END OF IBM EXTENSION

You can also specify a FUNCTION subprogram as being of CHARACTER type:

```
      CHARACTER*10 FUNCTION TEXT1 (WORD1,WORD2)
```

which defines TEXT1 as a CHARACTER function which returns a CHARACTER value of length 10, using dummy arguments WORD1 and WORD2.

In a function subprogram, you can use any FORTRAN statements, except PROGRAM (which would define it as a main program), SUBROUTINE (which would define it as a SUBROUTINE subprogram), or BLOCK DATA (which would define it as a BLOCK DATA subprogram).

You must code the last statement in a FUNCTION subprogram as an END statement.

You can also specify any number of RETURN statements.

Both the END and RETURN statements return control to the statement making the function reference in the calling program.

For reference documentation about the FUNCTION statement, see the VS FORTRAN Application Programming: Language Reference manual.

CODING SUBROUTINE SUBPROGRAMS

The first statement in a SUBROUTINE subprogram (excluding debugging statements) is a SUBROUTINE statement, identifying the program:

```
      SUBROUTINE TRIG (DELTA, THETA, ABSVAL)
```

This statement identifies the subprogram named TRIG as a SUBROUTINE subprogram, with dummy arguments DELTA, THETA, and ABSVAL.

In a SUBROUTINE subprogram, you can use any FORTRAN statements, except PROGRAM (which would define it as a main program), FUNCTION (which would define it as a FUNCTION subprogram), or BLOCK DATA (which would define it as a BLOCK DATA subprogram).

You must code the last statement in a SUBROUTINE subprogram as an END statement. You can also specify any number of RETURN statements. Both of these statements return control to the statement following the CALL statement in the calling program, except when you specify an alternate return.

For reference documentation about the SUBROUTINE statement, see the VS FORTRAN Application Programming: Language Reference manual.

SPECIFYING ALTERNATIVE ENTRY POINTS—ENTRY STATEMENT

When you're developing either FUNCTION or SUBROUTINE subprograms, you can specify alternative entry points within the program, using the ENTRY statement.

For example, subprogram TRIG could have an alternative entry point, depending on the data type of the values you wanted returned.

Alternative Entry Points in FUNCTION Subprograms

If FUNCTION TRIG had an alternative entry point, the sequence of statements would look something like this:

```
      FUNCTION TRIG (DELTA,THETA,ABSVAL)
      DOUBLE PRECISION BETA,ZETA,ABVAL1,TRIG8
          .
          .
          .
      RETURN
      ENTRY TRIG8 (BETA,ZETA,ABVAL1)
          .
          .
          .
      END
```

This FUNCTION subprogram can be executed in either of two ways:

1. When the calling program uses TRIG in a function reference, the FUNCTION subprogram is entered at the first executable statement, and the value returned is a real value of length 4.

   The RETURN statement returns control to the calling program.

2. When the calling program uses TRIG8 in a function reference, the FUNCTION subprogram is entered at the first executable statement following TRIG8, which is defined as a double precision function; therefore, the value returned is a real value of length 8.

   The END statement returns control to the calling program.

Alternative Entry Points in SUBROUTINE Subprograms

If SUBROUTINE TRIG had an alternative entry point, the sequence of statements would look something like this:

```
SUBROUTINE TRIG (DELTA,THETA,ABSVAL)
DOUBLE PRECISION BETA,ZETA,ABVAL1
      .
      .
      .
RETURN
ENTRY TRIG8 (BETA,ZETA,ABVAL1)
      .
      .
      .
END
```

This SUBROUTINE subprogram can be executed in either of two ways:

1. When the calling program uses TRIG in a CALL statement, the SUBROUTINE subprogram is entered at the first executable statement, and the subprogram uses the arguments DELTA, THETA, and ABSVAL, which are real items of length 4.

   The RETURN statement returns control to the calling program.

2. When the calling program uses TRIG8 in a CALL statement, the SUBROUTINE subprogram is entered at the first executable statement following TRIG8, and the subprogram uses the arguments BETA, ZETA, and ABVAL1, which are real items of length 8.

   The END statement returns control to the calling program.

SPECIFYING ALTERNATIVE RETURN POINTS—RETURN STATEMENT

When you're developing SUBROUTINE subprograms, you can specify alternative return points within the calling program, using the RETURN statement.

The RETURN statement, with no operands, can serve as one alternative return point, as the previous examples illustrate.

You can also code the RETURN statement with an integer variable operand; this allows you to specify variable return points. That is, you can return control to any labeled statement in the calling program.

For example, SUBROUTINE subprogram TRIG could have a variable return point, depending on the data values it develops:

```
Calling Program                        Called Program

      .                                SUBROUTINE TRIG (X,Y,Z,*,*)
      .                                   .
      .                                   .
10 CALL TRIG (A,B,C,*30,*40)              .
20 Y = A + B                           100 IF (M) 200,300,400
      .                                200 RETURN
30 Y = A + C                           300 RETURN 1
      .                                400 RETURN 2
40 Y= B - C                            END
```

When statement 10 of the calling program is executed, control is transferred to the first executable statement in TRIG. In TRIG, when statement 100 is executed, the value of M in the arithmetic IF statement determines which RETURN statement is executed:

> If M is less than zero, control is transferred to statement 200, and the RETURN statement returns control to statement 20 in the calling program. (This is the statement following the CALL statement.)
>
> If M is equal to zero, control is transferred to statement 300, and the RETURN 1 statement returns control to the first statement number in the calling program's actual argument list (*30). (Thus, control is returned to statement 30 in the calling program.)
>
> If M is greater than zero, control is transferred to statement 400, and the RETURN 2 statement returns control to the second statement number in the calling program's actual argument list (*40). (Thus, control is returned to statement 40 in the calling program.)

Execution then continues in the calling program.

RETAINING SUBPROGRAM VALUES—SAVE STATEMENT

The current FORTRAN standard states that, when a RETURN or an END statement in a subprogram is executed, all variables become undefined except for those in blank common, those in the argument list, and those specified in a SAVE statement. Thus, you use the SAVE statement to retain such undefined values.

For program portability, you can use the SAVE statement to ensure, if the program is recompiled on some other FORTRAN compiler, that values in specific named common blocks, variables, or arrays are saved when a RETURN or an END statement is executed.

In VS FORTRAN, these values are still available after a RETURN or an END statement is executed; however, the compiler accepts the SAVE statement and treats it as documentation.

For reference documentation about the SAVE statement, see the VS FORTRAN Application Programming: Language Reference manual.

INITIALIZING NAMED COMMON—BLOCK DATA SUBPROGRAMS

BLOCK DATA subprograms let you initialize data items in named common. (You can't initialize data items in blank common.)

The first statement you specify in a BLOCK DATA subprogram must be the BLOCK DATA statement. For example:

```
BLOCK DATA

or

BLOCK DATA COMDAT
```

where COMDAT is the name (optional) you've given the BLOCK DATA subprogram. If you specify a name, it must not be the same as the name of any other program, of an alternate entry point, of a common block, or of any data item within this BLOCK DATA subprogram.

The only statements you can specify in a BLOCK DATA subprogram are:

- BLOCK DATA (first statement in program)
- IMPLICIT (if used, must immediately follow BLOCK DATA statement)
- PARAMETER
- SAVE
- DIMENSION
- COMMON (must specify named common areas, each defined once)
- EQUIVALENCE
- DATA (must follow data item definitions in named COMMON statements, and must specify only data items in named common)
- Type statements
- END (must be last statement in subprogram)

The presence of a BLOCK DATA subprogram initializes named common data values in main programs or subprograms that refer to the named common blocks. Therefore, your programs must not contain CALL statements or function references to BLOCK DATA subprograms.

The following example shows how a BLOCK DATA subprogram might be coded:

```
      BLOCK DATA
      COMMON /ELJ/JC,A,B/DAL/Z,Y
      REAL B(4)/1.0,0.9,2*1.3/,Z*8(3)/3*5.42311849D0/
      INTEGER*2 JC(2)/74,77/
      END
```

this program initializes items in two named common areas, ELJ and DAL:

- The REAL type statement initializes item B in ELJ and item Z in DAL.
- The INTEGER type statement initializes item JC in ELJ.
- Because they're not included in either type statement, item A in ELJ and item Y in DAL are not initialized.

For reference documentation about BLOCK DATA subprograms, see the VS FORTRAN Application Programming: Language Reference manual.

SYSTEM CONSIDERATIONS

In order to use the subprograms you write, you must catalog them in a library—so they're available to calling programs. How you do this depends on the system you're using.

For VM/370-CMS considerations, see "Using VM/370-CMS with VS FORTRAN."

For OS/VS and DOS/VSE considerations, see "Cataloging Your Object Module—DOS/VSE" in Part 2.

OVERLAYING PROGRAMS IN STORAGE

When you use the overlay features of the linkage editor, you can reduce the main storage requirements of your program by breaking the program up into two or more segments that don't need to be in main storage at the same time. These segments can then be assigned the same storage addresses and can be loaded at different times during execution of the program.

You must specify linkage editor control statements to indicate the relationship of segments within the overlay structure.

Keep in mind, that although overlays reduce storage, they also can drastically increase program execution time. In other words, you probably shouldn't use overlays unless they're absolutely necessary.

The SAVE statement has no effect on overlaid programs. That is, when a program is overlaid by another, variable values in the overlaid program become undetermined.

For reference documentation about overlays, see the VS FORTRAN Application Programming: System Services Reference Supplement.

Specifying OS/VS Overlays

In OS/VS, overlay is initiated at execution time when a subprogram not already in main storage is referred to. The reference to the subprogram may be either a FUNCTION name or a CALL statement to a SUBROUTINE subprogram name. When the subprogram reference is found, the overlay segment containing the required subprogram is loaded—as well as any segments in its path not currently in main storage.

When a segment is loaded, it overlays any segment in storage with the same relative origin. It also overlays any segments that are lower (farther from the root segment) in the path of the overlaid segment.

Whenever a segment is loaded it contains a fresh copy of the program units that it comprises; any data values that may have been established or altered during previous processing are returned to their initial values each time the segment is loaded.

For this reason, you should place subprograms whose data values must be retained for longer than a single load phase into the root segment.

The linkage-editor control statements you use to process an overlay load module in OS are:

- OVERLAY linkage-editor control statement—which indicates the beginning of an overlay segment and gives the symbolic name of the relative origin.

  OVERLAY control statements are followed by object decks, INSERT control statements, or INCLUDE control statements.

- INSERT linkage-editor control statement—which positions previously compiled routines, when the object decks are not available, within the overlay structure.

  The INSERT control statement gives the names of one or more control sections (CSECTs) that are to be inserted.

  To place the control section in the root segment, position the INSERT control statement before the first OVERLAY control statement.

- INCLUDE linkage-editor control statement—which includes control sections from libraries, if the control sections reside in partitioned data sets or sequential data sets.

  When you use an INCLUDE control statement in an overlay program, you should position it in the input stream at the point where the control section to be included is required.

  The control sections added by an INCLUDE control statement can be manipulated through use of the INSERT control statement.

- ENTRY linkage-editor control statement—which specifies the first instruction of the program to be executed, giving the name of an instruction in the root segment. Usually that name will be either MAIN# or the name you've given in the PROGRAM statement (if specified).

These control statements appear in the input stream after the //SYSLIN DD statement (or after the //LKED.SYSLIN DD statement if you use a cataloged procedure).

Specifying DOS/VSE Overlays

The linkage-editor control statements you need to create an overlay phase in DOS are:

- PHASE linkage-editor control statement—which lets you divide your program into a number of phases.

- INCLUDE linkage-editor control statement—which lets you specify that a module from the relocatable library is to be included in the present phase.

OVERLAY PROCEDURE: To keep from running into difficulty when you're using multiple phases, you should develop your overlay programs using the following procedures:

1. Select a program name for the root phase of the program, the first four characters of which are unique in the core image library.

2. Determine the phase structure of the program and assign phase names, the first four characters of which are the same as the root phase name.

3. Write the root phase to serve as a monitor, loading each of the other phases as needed.

4. Determine the subroutine requirements for each phase.

5. If any modules used are not in the root phase but in the relocatable library, specify NOAUTO in the PHASE statement for each phase that refers to a module appearing in a later phase; when the PHASE statement is executed, the relocatable library is not searched and the later phases are not loaded.

   You must explicitly specify each subsequent phase in an INCLUDE control statement for the root phase.

USING THE CALL OPSYS STATEMENT: Within the FORTRAN program, when you want to have a phase loaded, issue a CALL to the library routine OPSYS. Your program must do this before you invoke the SUBROUTINE or FUNCTION that you want to execute.

For example, to load and execute subprograms PHASE4 and PHASE6, you specify:

```
      CALL OPSYS('LOAD','PHASE4  ')
           .
           .
           .
      CALL PHASE4(A,B)
      CALL OPSYS('LOAD','PHASE6  ')
           .
           .
           .
      CALL PHASE6(D)
```

These CALL OPSYS statements result in the phases named PHASE4 and PHASE6 being loaded. (The phase name in the CALL OPSYS statement is always eight alphameric characters, with the name left-adjusted within the field and padded on the right with blanks.)

The CALL PHASE4 and CALL PHASE6 statements cause the subprograms PHASE4 and PHASE6 to be executed.

For reference documentation about the CALL OPSYS statement, see the VS FORTRAN Application Programming: Language Reference manual.

Note: You must explicitly identify the OPSYS service routine by its library module name (IFYOPSYS) and supply a linkage editor INCLUDE statement using this name to get the OPSYS module included into the root phase.

USING THE OPTIMIZATION FEATURE

When you use the OPTIMIZE(1/2/3) options, you can get faster program execution. However, when you use these options, you should be aware of programming practices that can help or hinder optimization.

Some of the suggestions are obvious, some are not. However, it's easy to forget even the obvious ones when you're developing or revising programs over a period of time.

The following paragraphs suggest ways you can make your programs use the OPTIMIZE features to best advantage.

SELECT THE HIGHER OPTIMIZATION LEVELS

In general, you should select the highest level of optimization. The higher the optimization level you select, the longer the compilation time; however, both object program storage and execution time are reduced.

Very few iterations through most subroutines can cause optimization savings to exceed optimized compilation cost.

You should use NOOPTIMIZE or OPTIMIZE(1) only for testing purposes. For example, they're useful if you want to check the syntax of the program without executing it, or to debug a subroutine that doesn't iterate correctly.

Whenever you can, however, you should choose either OPTIMIZE(2) or OPTIMIZE(3) for most programs.

WRITE PROGRAMS OF EFFICIENT SIZE

For efficient optimization, programs can be either too large or too small.

Keep programs smaller than 8,192 bytes in size. Programs larger than this cause the compiler to remove an address from optimization and reserve it as a program address register.

Don't write subroutines that perform a small amount of work. The work done by a small subroutine can be lost in the cost of calling it. Instead, you should code such subroutines directly into the main program. This lets the compiler optimize the operations in the subroutine together with those of the main program.

USE UNFORMATTED I/O

Unformatted I/O takes less processing time and uses less storage than formatted I/O. Unformatted I/O also exactly maintains the precision of the data items you're processing.

With formatted I/O, each data element is converted between internal and external format. This takes time and storage. In addition, rounding errors can accumulate during conversion.

USE ARRAY OPERANDS FOR I/O TRANSFERS

In a READ or a WRITE statement, each operand is interpreted individually by the data transfer routines. (This is true for both unformatted and formatted I/O.)

By using an array operand without subscripts, you can read or write an entire array with a single operand.

You can also transfer a subset of an array by specifying an implied DO loop. For example:

```
      (A(I),I=11,99)
```

which transfer elements 11 through 99 of the array A.

However, when you use implied DO loops in this way, keep the subscript expressions simple. Don't interleave elements from several arrays in one implied DO loop; instead, write each array subscript in a separate implied loop. For example:

```
      ((A(I),B(I),C(I)I=11,99)
```

requires one call to the I/O subroutines for each array element, but

```
      (A(I),I=11,99),(B(I),I=11,99),(C(I),I=11,99)
```

requires only three such calls for the entire array.

USE LOGICAL VARIABLES OF LENGTH 4

LOGICAL variables of length 4 can be accessed directly without clearing a register.

IBM EXTENSION

Every reference to a LOGICAL*1 variable causes a register to be cleared before the variable is accessed. In some situations, the compiler allocates a register throughout an entire loop for this purpose; if not, it must at least generate an extra instruction.

END OF IBM EXTENSION

USE INTEGER VARIABLES OF LENGTH 4

INTEGER variables of length 4 are optimized by strength reduction; they're also generated into branch-on-index instructions. You should always use INTEGER variables of length 4 for DO loop indexes.

IBM EXTENSION

INTEGER*2 variables are not optimized by strength reduction, and they aren't generated into branch-on-index instructions. Therefore, they're less efficient than INTEGER*4 variables.

END OF IBM EXTENSION

ELIMINATE EQUIVALENCE STATEMENTS

Equivalenced variables often cannot be optimized. Optimization can't be performed for a variable equivalenced to another variable of a different type (for example, logical to integer), or for a scalar variable equivalenced to an array.

INITIALIZE LARGE ARRAYS DURING EXECUTION

If you initialize large arrays using a DO loop, you get faster execution and you use less storage than if you initialize using a DATA statement.

For example, the following statements:

```
DOUBLE PRECISION A(5000)
DATA A(5000)/5000*0.0/
```

generate 40,000 bytes of object module information—more than 500 TXT cards. The 40,000 zeros must be placed in the object module by the compiler, placed in the load module by the linkage editor, and fetched into storage when you execute the program.

USE COMMON BLOCKS EFFICIENTLY

Each reference to a variable in COMMON requires that the address of the COMMON block be in a register. The following recommendations are based on this fact.

1. Don't use multiple COMMON blocks. The following example shows why:

Three Registers Required:	**One Register Required:**
COMMON /X/ A	COMMON /Q/ A,B,C
COMMON/Y/ B	A=B+C
COMMON /Z/ C	
A=B+C	

   As the example shows, you should group concurrently referenced variables into the same COMMON block.

2. Place scalar variables before arrays in a given COMMON block. The following example shows why:

Two Registers Required:	**One Register Required:**
COMMON /Z/ X(5000),Y	COMMON /Z/ Y, X(5000)
X(1)=Y	X(1)=Y

   In the same way, you should place small arrays before large ones. All the scalar variables and the first few arrays can then be addressed through one address constant. The subsequent larger arrays will probably each need a separate address constant.

3. If a scalar variable in a COMMON block is frequently referred to, assign it from the COMMON block into a local variable. References to the local variable will not then require that the COMMON block address be in a register.

   If you do this, always remember that changing the local variable does not change the COMMON variable.

PASS SUBROUTINE ARGUMENTS IN COMMON BLOCKS

If you pass subroutine arguments in a COMMON block rather than as parameters, you'll avoid the overhead of moving each argument between the parameter list and local storage.

You must evaluate the effect of placing parameters into common for both the calling and the called routine.

DON'T USE VARIABLY DIMENSIONED ARRAYS

Subscripting of variably dimensioned arrays requires additional indexing computations. In addition, if you use a variably dimensioned array as a subroutine parameter, there are additional calculations that must be performed on each entrance into the subroutine.

You can lessen the amount of extra processing, however, in the following ways:

1. If the location and size of the array do not change during repeated calls to the subroutine, you can insert an initialization call to the subroutine to define the array; subsequent execution calls need not then refer to the array, as the following example shows:

```
Dimensions Calculated              Dimensions Calculated
Once:                              At Each Entrance:


Main Program                       Main Program

      CALL INIT(A,I,J)
      DO 1 N=1,10                        DO 1 N=1,10
1     CALL EXEC                    1     CALL EXEC(A,I,J)


Subprogram                         Subprogram

      SUBROUTINE INIT(A,I,J)       SUBROUTINE EXEC(A,I,J)
      REAL*8 A(I,J)                REAL *8 A(I,J)
      RETURN
      ENTRY EXEC
```

2. Make the variable dimensions of an array the high-order dimensions, if the indexing can be varied in the low-order dimensions. This reduces the number of computations needed for indexing the array, as the following example shows:

```
Computation not              Computation (I*N)
Required:                    Required:

SUBROUTINE EXEC(Z,N)         SUBROUTINE EXEC(Z,N)
REAL *8 Z(9,N)               REAL *8 Z(N,9)
Z(I,5)=A                     Z(5,I)=A
```

WRITE CRITICAL LOOPS INLINE

If your program has a short heavily-referenced DO loop, it's probably worth the effort to remove the loop and expand the code inline in the program. Each loop iteration will execute faster.

ENSURE RECOGNITION OF DUPLICATE COMPUTATIONS

If components of a computation are duplicates, make sure you code the duplicate elements in one of the following ways:

- At the left end of the computation
- Within parentheses

The compiler must follow the left-to-right FORTRAN rules, and this order of computations follows those rules.

The following examples illustrate this concept:

```
Duplicates Recognized:   No Duplicates Recognized:

A=B*(X*Y*Z)              A=B*X*Y*Z
C=X*Y*Z*D                C=X*Y*Z*D

E=F+(X+Y)                E=F+X+Y
G=X+Y+H                  G=X+Y+H
```

In the pair of examples at the left, the compiler can recognize X*Y*Z and X+Y as duplicates because they're either coded in parentheses or coded at the left end of the computation. In the pair of examples at the right, these rules are not followed, and therefore the compiler cannot recognize these duplicates.

ENSURE RECOGNITION OF CONSTANT COMPUTATIONS

In a loop, when several components of a computation are constant, ensure that they can be recognized by following one of these coding rules:

1. Move all the constant computations to the left end of the computation.

2. Group constant computations within parentheses.

The compiler follows the left-to-right FORTRAN rules, and this order of computations allows the compiler to recognize the constant portions of the computations.

If C, D, and E are constant and V, W, and X are variable, the following examples show the difference in evaluation:

Constant Computations Recognized	Constant Computations Not Recognized
V*W*X*(C*D*E)	V*W*X*C*D*E
C+D+E+V+W+X	V+W+X+C+D+E

ENSURE RECOGNITION OF CONSTANT OPERANDS

The compiler can recognize only local variables as having a constant value. (It must always assume that operands in common or in a parameter list can change, and therefore cannot optimize them.)

Therefore, for such items you should define constant operands as local variables.

ELIMINATE SCALING COMPUTATIONS

If your program performs calculations representing physical values of some kind, you can save computation time by using factoring, as the following simple example shows:

```
Not Using Factoring          Using Factoring

  SUM=0.0                      SUM=0.0
  DO 1 I=1,9                   DO 1 I=1,9
1 SUM=SUM+FAC*ARR(I)         1 SUM=SUM+ARR(I)
                               SUM=SUM*FAC
```

In many programs, you can use factoring much more extensively than this simple example shows.

DEFINE ARRAYS WITH IDENTICAL DIMENSIONS

If all your arrays have the same shape, then the compiler can use a subscript calculated for one array to subscript the others.

In some cases, therefore, you should consider expanding some smaller arrays to match the dimensions of the other arrays with which they're involved. The compiler can then maintain only one index for all the arrays defined as having the same dimensions.

DEFINE ARRAYS WITH IDENTICAL ELEMENT SPECIFICATIONS

If you define arrays as having the same dimensions <u>and</u> the same element specifications, the compiler can compute a subscript for one array and then use it without change for the others.

In some cases, therefore, you should consider expanding smaller arrays to match the elements in the others. You should always do this for arrays with integer or logical operands.

USE CRITICAL VARIABLES CAREFULLY

Certain variables cannot be optimized in certain circumstances:

- Control variables for direct access input/output data sets cannot be optimized at all.
- Variables in input/output statements and in argument lists cannot be optimized by register optimization in the loops that contain the statements.
- Variables in COMMON blocks cannot be optimized across subroutine calls.

You shouldn't use DO loop indexes for any of these purposes.

AVOID UNNEEDED FIXED/FLOAT CONVERSIONS

Avoid forcing the compiler to convert numbers between the integer and the floating-point internal representations; each such conversion requires several instructions, including some double-precision floating-point arithmetic.

The following example shows one method of avoiding such unnecessary conversions:

```
  One Conversion Needed:    Multiple Conversions Needed:

        X=1.0
        DO 1 I=1,9                  DO 1 I=1,9
        A(I)=A(I)*X           1     A(I)=A(I)*I
  1     X=X+1.0
```

When you can't avoid using mixed-mode arithmetic, then code the fixed-point and floating-point arithmetic as much as possible in separate computations.

MINIMIZE CONVERSIONS BETWEEN SINGLE AND DOUBLE PRECISION

Two, or even three, instructions are required to convert data between single and double precision.

USE SCALAR VARIABLES AS ACCUMULATORS

When you're accumulating intermediate summations, keep the result in a scalar variable rather than in an array. Array accumulators require load and store instructions; scalar variable accumulators can be maintained in a register.

USE EFFICIENT ARITHMETIC CONSTRUCTIONS

Wherever possible, change subtractions into additions and divisions into multiplications.

In subtraction operations, if only the negative is required, change the subtraction operations into additions, as follows:

```
  Efficient:                   Inefficient:

        Z=-2.0
        DO 1 I=1,9                   DO 1 I=1,9
  1     A(I)=A(I)+Z*B(I)       1     A(I)=A(I)-2.0*B(I)
```

In division operations, do the following:

- For constants, use one of the following constructions:

  ```
  X*(1.0/2.0)
  0.5*X
  ```

  rather than the construction X/2.0.

- For a variable used as a denominator in several places, use the same technique.

USE IF STATEMENTS EFFICIENTLY

In general, use the logical IF statement rather than the arithmetic IF statement.

If you must use an arithmetic IF statement, try to make the next succeeding statement one of the branch destinations.

For multiple branches, either use the computed GO TO statement, or, if the branch can be initialized so that it remains invariant, use an assigned GO TO statement.

In logical IF statements, if your tests involve a series of AND and/or OR operators, try to do the following:

- Put the simplest conditions tested in the leftmost positions.
- Also, put the tests most likely to be decisive in the leftmost positions.
- Put the more complex conditions (such as tests involving function references) in the rightmost positions.

If the first part of the expression causes the logical condition to test as true, then the rest of the expression need not be evaluated, saving execution time.

USE THE OBJECT PROGRAM LISTING

Use the object program listing, which you obtain through the LIST compiler option, to find out what machine instructions the compiler has generated for your program. You can often tell whether the program has been well or poorly optimized.

The essential work for most FORTRAN programs is to compute floating-point numbers (rather than subscripts or DO loop indexes). Take a quick look at the inner loops for such programs; if they contain essentially no fixed-point instructions, the program is efficiently optimized.

Similarly, you can tell from a FORTRAN source program which additions and multiplications and other operations are necessary and which ought to disappear under optimization. You can examine the object program to discover whether there's a reasonable correlation between the generated program and your expectation.

Using the object code listing in this way, is the best way you can study the efficiency of source program optimization.

Neither of these examinations requires detailed knowledge of assembler language.

SOURCE CONSIDERATIONS WITH OPTIMIZE(3)

When you're using the OPTIMIZE(3) compiler option, there are additional coding considerations you should be aware of.

COMMON EXPRESSION ELIMINATION—OPTIMIZE(3)

OPTIMIZE(3) evaluates expressions and eliminates those that are common to more than one statement. That is, if an expression occurs more than once and the path of execution always executes the first expression and then the second, with no change in the expression value, the first value is saved and used instead of the second expression. OPTIMIZE(3) does this even for intermediate expressions within expressions. For example, if your program contains the following statements:

```
 10   A=C+D
        .
        .
        .
 20   F=C+D+E
```

the common expression C+D is saved from its first evaluation at 10, and is used at 20 in determining the value of F.

COMPUTATIONAL REORDERING—OPTIMIZE(3)

OPTIMIZE(3) may move an expression outside of a loop when the operands of the expression are not defined as part of the loop. This can cause execution differences from nonoptimized code.

For example, when an IF statement controls the execution of a computation within a loop, and the computation is moved outside the loop, program execution results may change:

```
      DO 11 I=1,10
      DO 12 J=1,10
  9   IF (B(I).LT.0) GO TO 11
 12   C(J)=SQRT(B(I))
 11   CONTINUE
```

OPTIMIZE(3) moves the library function call to precede statement 9, which causes the square root computation to be made before the test for zero.

To avoid this unwanted code movement, use the OPTIMIZE(2) option.

You can also get unexpected results when you use CALL OVRFL or CALL DVCHK, because the computations causing overflow, underflow, or divide check conditions could be moved out of the loop in which the test occurs.

INSTRUCTION ELIMINATION—OPTIMIZE(3)

If your program defines nonsubscripted variables, and their values are not used between two definitions within one block, or have not been used before the exit from the block, the compiler may eliminate any intermediate storing of the variables.

This can cause a change in execution logic if, for example, you initialize the variables just before issuing a READ statement:

```
      I=7
      J=8
      K=9
      READ (8,*,END=1) I,J,K
      WRITE (6,*) I,J,K
  1   END
```

OPTIMIZE(3) considers the READ statement as a redefinition of I, J, and K, and the END statement delimits the block; therefore, the instructions usually generated to store I, J, and K are eliminated by the compiler. During execution, if the READ statement terminates normally before reading into all three variables, those not read into are not initialized. (If you omit the END and/or the ERR parameters, the store instructions are not eliminated.)

USING THE EXECUTION-TIME LIBRARY

The VS FORTRAN execution-time library contains intrinsic function routines, and subprogram routines for error handling and service functions. There are several kinds of routines available:

Mathematical and Character Functions—mathematical and character routines that are either inline functions or subroutine functions.

Service Routines—these subprogram routines test for mathematical exceptions or terminate execution.

Error Routines—these subprograms handle errors automatically; you can also control error handling directly.

Each category is explained more fully in the following sections.

MATHEMATICAL AND CHARACTER FUNCTIONS

These routines provide intrinsic functions you can use in mathematical and character operations. There are several categories of mathematical and character functions:

- Logarithmic and exponential routines
- Trigonometric routines
- Hyperbolic function routines
- Miscellaneous mathematical routines
- Character manipulation routines

When your VS FORTRAN program requests an intrinsic function, the routine is either:

- Inserted inline into the program, or
- Included in the load module as a called subroutine during link-editing.

You can use the generic name for a function; VS FORTRAN will then select the particular function you need, depending upon the precision of the data you're using.

You can, alternatively, use the name of the specific alternative entry point you want to use. A prefix to the generic name specifies an alternative entry point.

See the VS FORTRAN Application Programming: Library Reference manual for a description of the mathematical and character functions, and for names of generic functions and their alternative entry points.

SERVICE ROUTINES

These routines give you control over certain mathematical exceptions and over program termination when unusual conditions occur. Using the service routines, you can perform the following functions:

CALL OVRFLW—Test for exponent overflow or underflow and return a value indicating the condition that exists.

CALL DVCHK—Test for a divide-check exception and return a value indicating the condition that exists.

CALL EXIT—Terminate load module execution and return control to the operating system.

CALL DUMP/PDUMP—Dynamically dump a specified area of storage and either terminate (CALL DUMP) or continue execution (CALL PDUMP). You can specify the format of the output.

CALL CDUMP/CPDUMP—Dynamically dump a specified area of storage, in character format. and either terminate (CALL CDUMP) or continue execution (CALL CPDUMP). The output is always in character format.

These routines are described in the VS FORTRAN Application Programming: Library Reference manual.

Reference documentation for these routines is given in the VS FORTRAN Application Programming: Language Reference manual.

ERROR-HANDLING ROUTINES

The library supplies two kinds of error-handling support: automatic error handling during compilation or execution, and error handling under your control.

Automatic Error Handling

During compilation, if the compiler detects an S-level error, it inserts a call for a library function instead of generating the code for the statement; during execution, if and when this statement in the program is reached, an error message (including the internal statement number) is written and the program is terminated.

During program execution, the library issues messages when the following program interrupts occur:

- Operation
- Fixed-point divide
- Decimal divide
- Floating-point divide
- Exponent overflow
- Exponent underflow

Error Handling Under Your Control

In addition, the execution-time error-handling routines give you dynamic control over:

- The number of times an error can occur before your program is terminated
- The maximum number of times a message is printed
- If a traceback map is to be printed with the message
- If your organization's error routine is to be called

You specify dynamic error control during your program's execution through:

- CALL ERRMON—causes execution of the error monitor.

- CALL ERRSAV—copies an option table entry into an area accessible to your program.
- CALL ERRSET—changes up to five values associated with an entry in the option table (for example, the number of errors permitted or number of messages to be printed).
- CALL ERRSTR—stores an entry into the option table from your program.
- CALL ERRTRA—causes execution of the error trace routines.

For a description of how you use these routines, see "Fixing Execution-Time Errors—Advanced Programming" in Part 2.

Reference documentation for these routines is given in the *VS FORTRAN Application Programming: Language Reference* manual.

APPENDIXES

This section contains appendixes documenting the following auxiliary VS FORTRAN material:

- Appendix A—Device Information
- Appendix B—Assembler Language Considerations

APPENDIX A. DEVICE INFORMATION

This appendix gives information regarding specific input/output devices that can be used with a VS FORTRAN program.

MINIMUM AND MAXIMUM BLOCK SIZE VALUES

The minimum and maximum block sizes that can be specified for specific devices are shown in Figure 47.

Device Type	Fixed and Undefined Records Block Size (Bytes)		Variable Records Block Size (Bytes)	
	Minimum	Maximum	Minimum	Maximum
Card Reader	1	80	9	80
Card Punch	1	81	9	89
Printer Line Length (1403, 3800, etc.)				
120 characters	1	121	9	129
132 characters	1	133	9	141
144 characters	1	145	9	153
150 characters	1	151	9	159
Magnetic Tape	18	32760	18	32760
Direct Access				
2314	1	7294	9	7294
3330	1	13030	9	13030
3340	1	8368	9	8368
3350	1	19069	9	19069

Notes:

1. For DOS/VSE Fixed Block Architecture devices, see the manuals describing the devices you're using.
2. For direct access devices with the track overflow feature, the maximum is 32760 for each device.

Figure 47. VS FORTRAN Devices—Minimum and Maximum Block Sizes

DIRECT ACCESS DEVICE CAPACITIES

The capacities of specific direct access devices are shown in Figure 48.

Device Type	Track Capacity	Tracks per Cylinder	Number of Cylinders	Total Capacity
2314(2319)	7294	20	100	19,176,000
3330	13030	19	404	101,751,270
3330-11	13030	19	808	203,502,340
3340	8535	12	348	34,944,768
3350	19254	30	815	317,498,850

Notes:

1. For DOS/VSE Fixed Block Architecture devices, see the manuals describing the devices you're using.
2. For the 3375 and 3380 devices, device information will be provided after the devices are available.

Figure 48. Direct Access Device Capacities

APPENDIX B. ASSEMBLER LANGUAGE CONSIDERATIONS

You can use assembler language subprograms with your FORTRAN main programs. In your FORTRAN programs, you can invoke the assembler subprogram in either of two ways: through CALL statements or through function references in arithmetic expressions.

This appendix describes the linkage conventions you must use in such assembler language subprograms to communicate with the FORTRAN program.

For documentation about assembler language programs, see:

> OS/VS, DOS/VS, and VM/370 Assembler Language, GC33-4010
>
> OS/VS, VM/370 Assembler Programmer's Guide, GC33-4021
>
> Guide to the DOS/VSE Assembler, GC33-4024

SUBPROGRAM REFERENCES IN FORTRAN

For each subprogram reference, the compiler generates:

- A contiguous argument list containing the addresses of the arguments; this makes the arguments accessible to the subprogram.

  If the calling program was compiled with LANGLVL(77), and if any arguments in the call to the assembler language subprogram are of character type, there will be two items in the argument list for each character argument. The first item is the address of the start of the character argument, and the second is the address of the length of the character argument contained in the full word. If the passed length is needed in the assembler subprogram, then, when the arguments are fetched from the parameter list, the length argument must be received; otherwise, the length argument must be skipped over. The generation of the length address may be suppressed for LANGLVL(77) by using the SC compile option. See the section "SC(name1,name2,...)" under "Using the Compile-Time Options" on page 72.

- A save area in which the subprogram can place information about the calling program.

- A calling sequence to pass control to the subprogram, using standard linkage conventions.

THE ARGUMENT LIST

The argument list contains addresses of variables, arrays, and subprogram names used as arguments.

Each entry in the argument list is four bytes long and is aligned on a fullword boundary. The last three bytes of each entry contain the address of an argument. The first byte of every entry except the last contains zeros; in the first byte of the last entry, the sign bit is set to binary 1.

The calling program places the address of the argument list in general register 1.

THE SAVE AREA

The calling program contains a save area in which the subprogram places information: the entry point for this program, the address to which the subprogram returns, general register contents, and addresses of save areas used by programs other than this subprogram.

The calling program reserves 18 words of storage for this area.

The calling program places the address of the save area in general register 13.

THE CALLING SEQUENCE

The FORTRAN compiler generates a calling sequence to transfer control to the subprogram, placing the following addresses in the following registers:

- Register 13—the address of the save area.
- Register 1—the address of the argument list. (If there is no argument list, 0 (zero) is placed in general register 1.)
- Register 15—the entry address.
- Register 14—the return address.

The program then branches to the address in general register 15.

You can also use register 15 as a condition code register, and as a RETURN and STOP code register. The values you should use for these codes are:

16	for a terminal error detected during subprogram execution. (A FORTRAN message is also generated.)
0	when a RETURN or STOP statement is executed in the subprogram.
4*i	when a RETURN i statement is executed in the subprogram.
n	when a STOP n statement is executed in the subprogram.

LINKAGE IN ASSEMBLER SUBPROGRAMS

You can use two types of assembler subprograms:

Called Subprograms—that is, assembler language subprograms that don't call another subprogram.

Called and Calling Subprograms—that is, assembler language subprograms that do call another subprogram.

The rules for coding such subprograms are somewhat different, so they are documented in separate sections following.

CALLED ASSEMBLER SUBPROGRAMS

For assembler subprograms that don't call other subprograms, you must include the following linkage instructions:

1. An instruction naming the entry point for this subprogram.
2. Instructions to save any general registers this subprogram uses in the save area reserved by the calling program. (You don't need to save the contents of linkage registers 0 and 1.)

3. Before returning control to the calling program, instructions to restore the saved registers.

4. An instruction setting the first byte of the fourth word in the save area to ones, to indicate return of control to the calling program.

5. An instruction returning control to the calling program.

In addition to these instructions, if arguments are passed, the assembler subprogram must transfer the arguments from the calling program and return the arguments to the calling program, using the address passed in general register 1.

CALLED AND CALLING ASSEMBLER SUBPROGRAMS

A called and calling assembler language subprogram must contain the same linkage instructions as a called subprogram; it must also simulate the FORTRAN linkage conventions for calling subprograms. Therefore, it must also include:

- A save area and instructions to place entries into its save area
- A calling sequence and parameter list for the subprogram it is calling
- An instruction indicating an external reference to the subprogram it is callin.
- Additional instructions in the return routine to retrieve entries from the save area

MAIN PROGRAMS

If the main program is not a FORTRAN main program, you must establish certain FORTRAN linkages after you've established the save area and before you call the FORTRAN subroutine..

The assembler instructions you must code to establish the linkages are:

```
LR X,13          (where X is any register 2 through 12
                 to SAVE register 13)
L 13,4(13)
L 15,=V(VSCOM#)
BAL 14,64(15)
LR 13,X          (where X is the same as above)
```

The linkages you establish cause initialization of return coding and interrupt exceptions, as well as opening of the error message data set.

If you don't do this and the FORTRAN subprogram terminates in error or with a STOP statement, any open FORTRAN data sets are not closed, and the results of the program termination are unpredictable.

USING FORTRAN DATA IN ASSEMBLER SUBPROGRAMS

Your assembler language subprograms can use data defined in FORTRAN subprograms, either data contained in common areas or in argument lists.

USING COMMON DATA IN ASSEMBLER SUBPROGRAMS

Assembler language subprograms can access data in both blank and named common areas.

Using Blank Common Data in Assembler Programs

To refer to the blank common area, the assembler language program must also define a blank common area, using the COM assembler instruction. Only one blank common area is generated, and the data it contains is available both to the FORTRAN program containing the blank COMMON statement and to the assembler language program containing the COM statement.

In the assembler language program, you can specify the following linkage:

```
      COM
name  DS 0F
      .
      .
      .
      L 11,=A(name)
      USING name,11
```

Using Named Common Data in Assembler Programs

To refer to named common areas, your assembler program can use a V-type address constant:

```
name  DC V(name of common area)
```

RETRIEVING ARGUMENTS IN AN ASSEMBLER PROGRAM

The argument list contains addresses of variables, arrays, and program statement labels used as arguments.

Each entry in the argument list is four bytes long and is aligned on a fullword boundary. The last three bytes of each entry contain the address of an argument. The first byte of every entry except the last contains zeros; in the first byte of the last entry, the sign bit is set to binary 1.

The calling program places the address of the argument list in general register 1.

Retrieving Variables from the Argument List

The argument list contains the address of a variable. The assembler program can retrieve the variable using the following instructions:

```
      L    Q,x(1)
      MVC  LOC(y),z(Q)
```

where:

Q	is any general register except 0.
LOC	is the location that will contain the variable.
x	is the displacement of the variable from the start of the argument list.
y	is the length of the variable itself.
z	is either 0 or the displacement to correct the value for an array element.

For example, if a REAL*8 variable is the second item in the argument list, you could code the following assembler instructions to retrieve it:

```
      L     5,4(1)
      MVC   LOC(8),0(5)
```

Retrieving Arrays and Array Elements from the Argument List

The address of the first element of an array is placed in the argument list. If you must retrieve any other elements in the array, you must specify the displacement for that element from the beginning of the array:

```
        L     Q,x(1)
        L     R,disp
        L     S,0(Q,R)
        ST    S,LOC
```

where:

Q and R	are any general registers except 0.
disp	is the displacement of the element within the array.
S	is any general register.
LOC	is the location that will contain the array element.

Returning to Alternative Return Points

To simulate the FORTRAN subprogram RETURN **stn** statement, however, the assembler program can place the value:

```
        4*stn
```

in general register 15, where stn is the actual argument position to which return is to be made.

For example, to return to the second statement in the actual argument list, the assembler language program must contain:

```
        LA 15,8
```

INTERNAL REPRESENTATION OF FORTRAN DATA

If you're using FORTRAN data in your assembler language programs, you should be aware of the formats FORTRAN uses within the computer.

The following examples show how FORTRAN data items appear in internal storage.

CHARACTER Items in Internal Storage

CHARACTER items are treated internally as one EBCDIC character for each character in the item.

LOGICAL Items in Internal Storage

LOGICAL items are treated internally as items either 1 byte or 4 bytes in length. Their value can be "true" or "false".

Their internal representation in hexadecimal notation is:

IBM EXTENSION

```
 ----
| 01 |                  "true"
 ----

 ----
| 00 |                  "false"
 ----
1 byte
```

END OF IBM EXTENSION

```
 ---- ---- ---- ----
| 00 | 00 | 00 | 01 |        "true"
 ---- ---- ---- ----

 ---- ---- ---- ----
| 00 | 00 | 00 | 00 |        "false"
 ---- ---- ---- ----
<------4 bytes------>
```

INTEGER Items in Internal Storage

INTEGER items are treated internally as fixed-point signed operands, either 2 bytes or 4 bytes in length.

Their internal representation is:

IBM EXTENSION

INTEGER *2

```
 -------------------------
|S|        |              |
 -------------------------
<-------2 bytes--------->
```

END OF IBM EXTENSION

INTEGER *4

```
 ----------------------------------------------------
|S|         |             |             |             |
 ----------------------------------------------------
<--------------------4 bytes---------------------->
```

S = the sign bit

REAL Items in Internal Storage

The compiler converts REAL items into 4-byte, 8-byte, or 16-byte floating-point numbers.

Their internal representation is:

REAL *4

```
 ----------------------
|S| C  | F  |    |     |
 ----------------------
<-------4 bytes------->
```

DOUBLE PRECISION (REAL *8)

```
 ----------------------------------------
|S| C  | F  |    |    |    |    |    |    |
 ----------------------------------------
<----------------8 bytes----------------->
```

For REAL *4 and DOUBLE PRECISION items, the codes shown are:

```
S = sign bit (bit 0)
C = characteristic (or exponent), in bit positions 1 through 7
F = fraction, which occupies bit positions. as follows:
    REAL *4            positions 8 through 31
    DOUBLE PRECISION   positions 8 through 63
```

IBM EXTENSION

REAL *16 (Extended Precision)

```
┌-------------//---------------------//-------┐
|S| C   | F   | // |    |S| C   | F   | // |       |
└-------------//---------------------//-------┘
0       8                64      72
<-----------------16 bytes------------------>
```

For Extended Precision Items, the codes are:

```
S = sign bit (sign for the item in bit 0; a + sign in bit 64)
C = characteristic (or exponent), in bit positions 1 through 7
    and 65 through 71 (the value in bit positions 63 through 71
    is 14 less than that in bit positions 1 through 7)
F = fraction, in bit positions 8 through 63, and 72 through 127
```

END OF IBM EXTENSION

COMPLEX Items in Internal Storage

The compiler converts COMPLEX items into a pair of REAL numbers. The first number in the pair represents the real part; the second number in the pair represents the imaginary part.

The internal representations of COMPLEX numbers are:

COMPLEX *8

```
┌----------------------┐
|S| C   | F   |     |      |  (real)
|----------------------|
|S| C   | F   |     |      |  (imag.)
└----------------------┘
<-------4 bytes------->
```

For COMPLEX *8 items, the codes shown are:

```
S = sign bit (bit 0)
C = characteristic (or exponent), in bit positions 1 through 7
F = fraction, which occupies bit positions 8 through 31
```

IBM EXTENSION

COMPLEX *16

```
┌-------------------------------------------┐
|S| C   | F   |    |    |    |    |    |    |      |  (real)
|-------------------------------------------|
|S| C   | F   |    |    |    |    |    |    |      |  (imag.)
└-------------------------------------------┘
<-----------------8 bytes----------------->
```

COMPLEX *32

```
┌-------------//---------------------//-------┐
|S| C   | F   | // |    |S| C   | F   | // |       |  (real)
|-------------------//-----------------------|
|S| C   | F   | // |    |S| C   | F   | // |       |  (imag.)
└-------------//---------------------//-------┘
0       8                64      72
<-----------------16 bytes------------------>
```

For COMPLEX *16 items, the codes shown are:

S = sign bit (bit 0)
C = characteristic (or exponent), in bit positions 1 through 7
F = fraction, which occupies bit positions 8 through 63

For COMPLEX *32 Items, the codes are:

S = sign bit (sign for the item in bit 0; a + sign in bit 64)
C = characteristic (or exponent), in bit positions 1 through 7 and 65 through 71 (the value in bit positions 63 through 71 is 14 less than that in bit positions 1 through 7)
F = fraction, in bit positions 8 through 63, and 72 through 127

END OF IBM EXTENSION

In VS FORTRAN, operands of logical type are not permitted with relational operators. FORTRAN H and FORTRAN H Extended permit this nonstandard usage. Under LANGLVL(66), this nonstandard usage is accepted and a warning message is issued.

In VS FORTRAN, operands of logical type are not permitted with arithmetic operators. FORTRAN H and FORTRAN H Extended permit this nonstandard usage.

The Extended Language features permitted with use of the XL option from FORTRAN H and FORTRAN H Extended are not supported in VS FORTRAN.

In VS FORTRAN, the DEBUG statement and the debug packets precede the program source statements. The new END DEBUG statement delimits the debug-related source from the program source. For FORTRAN G1, the DEBUG statement and the debug packets are placed at the end of the source program.

In VS FORTRAN, evaluation of arithmetic expressions involving constants is performed at compile time (including those containing mixed-mode constants).

In VS FORTRAN, the number of arguments is checked in statement function references. The mode of arguments is checked for statement function references under LANGLVL(77) option only.

In VS FORTRAN, the form of the compiler option to name a program is NAME(nam) under LANGLVL(66).

Arguments are received only by location (or name) in LANGLVL(77). The default in LANGLVL(66) and for FORTRAN H and FORTRAN H Extended is receipt by value with the facility to allow receipt by name by the use of slashes around the dummy argument in the SUBROUTINE, FUNCTION, or ENTRY statements.

The appearance of an intrinsic function name in a conflicting type statement has no effect in LANGLVL(77), but is considered user-supplied under LANGLVL(66) and FORTRAN H and FORTRAN H Extended.

Direct access files must be preformatted when using LANGLVL(77). This is done by the DEFINE FILE statement under LANGLVL(66), FORTRAN G1, FORTRAN H, and FORTRAN H Extended. (The reason for this is that there is no provision in the OPEN statement for the number of records to be contained in the file as there is in the DEFINE FILE statement.)

The use of a scale factor with an integer format specifier is considered a severe error by VS FORTRAN, but is accepted by FORTRAN G1, FORTRAN H, and FORTRAN H Extended.

For an implied DO in an input/output list with a variable incrementation value that is assigned a value of zero at execution time:

- VS FORTRAN and FORTRAN G1 do not diagnose this invalid case and the program loops.
- FORTRAN H Extended makes its diagnoses at execution time.
- In a standard DO-loop, FORTRAN G1, FORTRAN H Extended, and VS FORTRAN under LANGLVL(66) all loop. VS FORTRAN under LANGLVL(77) bypasses the loop, because it does not meet the standard requirements for execution.

In VS FORTRAN, when a variable has been initialized with a DATA statement, that variable cannot appear in an explicit type statement and a severe level diagnostic is issued. FORTRAN H and FORTRAN H Extended allow typing following the data initialization. This is nonstandard usage. FORTRAN G1 issues a level 8 error diagnostic.

The record designator for direct-access I/O is required to be an integer expression for both LANGLVL(66) and LANGLVL(77). If it is not, VS FORTRAN diagnoses with a level 12 error message. FORTRAN H and FORTRAN H Extended permit this designator to be of real type. FORTRAN G1 diagnoses with a level 8 error message.

In VS FORTRAN, the use of literal and Hollerith constants to initialize integer, real, or logical type variables is permitted under option LANGLVL(66), as it is in FORTRAN G1, and FORTRAN H Extended. However, in VS FORTRAN under LANGLVL(77), a character constant (that is, a quoted constant), may be used to initialize character data type only and, in fact, is the only way to initialize character type variables and arrays. Hexadecimal constants may be used in the DATA statement to initialize integer, real, or logical type variables or arrays under both LANGLVL(66) and LANGLVL(77).

In VS FORTRAN, for LANGLVL(77), character arguments are passed to a subprogram with both a pointer to the character string and a pointer to the length of the character string. This is required because the receiving program may have declared the dummy character arguments to have inherited length (that is, the length of the dummy argument is the length of the actual argument). The parameter list is therefore longer than for LANGLVL(66), because every character argument generates two items in the parameter list. For LANGLVL(66):

- Literal constants passed as arguments generate only one item in the parameter list.
- Hollerith constants may be passed as subroutine or function arguments.

In LANGLVL(77), a level 8 message is received if Hollerith constants are passed as arguments.

In both languages, only one item is generated in the parameter list for Hollerith arguments. (See also the section "SC(name1,name2,...)" under "Using the Compile-Time Options" on page 72.)

In VS FORTRAN, all calculations for adjustably dimensioned arrays are performed by use of a library routine called at all entry points that specify such arrays. This method was required for LANGLVL(77), because it permits redefinition of the adjustable dimension parameters within the subprogram but requires that the array properties do not change from those existing at the entry point.

APPENDIX D. INTERNAL LIMITS IN VS FORTRAN

NESTED DO LOOPS

Nested DO loops and nested implied DO loops are limited to 25 each.

EXPRESSION EVALUATION

The maximum depth of the push-down stack for expression evaluation is 150. This means that, for any given expression, the maximum number of operator tokens that can be considered before any intermediate text can be put out is 150. For example, if an expression starts with 150 left parentheses before any right parentheses, this expression will exceed the push-down stack limit.

NESTED STATEMENT FUNCTION REFERENCES

The total number of statement function arguments in any nested reference is limited to 50.

The total number of nested statement function references is limited to 50.

The total number of arguments in any statement function definition is limited to 20.

NESTED INCLUDE STATEMENTS

The maximum number of nested INCLUDE statements is 16.

NESTED BLOCK IF STATEMENTS

Block If statements may be nested to a depth of 25. That is, the number of IF... THEN, ELSE, and ELSEIF... THEN statements occurring before the occurrence of an ENDIF statement must be no greater than 25.

CHARACTER CONSTANTS

Character symbolic constants that are defined using the PARAMETER statement are limited to a maximum length of 255. Character constants used in PAUSE or STOP statements are limited to 72 characters. Character data types, defined with an IMPLICIT statement or an explicit type statement, are limited to a maximum of 500 characters.

REFERENCED VARIABLES

The maximum number of referenced variables in a program unit is 660.

PARENTHESES GROUPS

The maximum number of parentheses groups in a format is 50.

STATEMENT LABELS

Allowance has been made for up to 2000 user source labels and compiler-generated labels. However, if optimization level 2 or 3 causes table overflow, the problem may be alleviated by removing all unreferenced user labels.

GLOSSARY

This glossary includes definitions developed by the American National Standards Institute (ANSI), and the International Organization for Standarization (ISO). This material is reproduced from the American National Dictionary for Information Processing, copyright 1977 by the Computer and Business Equipment Manufacturers Association, copies of which may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

An asterisk (*) to the right of an item number indicates an ANSI definition in an entry that also includes other definitions.

The symbol "(ISO)" at the beginning of a definition indicates that it has been discussed and agreed upon at meetings of the International Organization for Standardization Technical Committee 97/Subcommittee 1 (Data Processing Vocabulary), and has also been approved by ANSI and included in the American National Dictionary for Information Processing.

alphabetic character. A character of the set A, B, C, ... Z. See also "letter."

IBM EXTENSION

In VS FORTRAN, the currency symbol ($) is considered an alphabetic character.

END OF IBM EXTENSION

alphameric. Pertaining to a character set that contains letters, digits, and other characters, such as punctuation marks.

alphameric character set. A character set that contains both letters and digits and also contains control characters, special characters, and the space character.

argument. A parameter passed between a calling program and a SUBROUTINE subprogram, a FUNCTION subprogram, or a statement function.

arithmetic constant. A constant of type integer, real, double-precision, or complex.

arithmetic expression. One or more arithmetic operators and/or arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, array element, or function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs VS FORTRAN to perform an arithmetic operation. The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

array. An ordered set of data items identified by a single name.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit type statement.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name. The name of an ordered set of data items that make up an array.

assignment statement. A statement that assigns a value to a variable or array element. It is made up of a variable or array element, followed by an equal sign (=), followed by an expression. The variable, array element, or expression can be of type character, logical, or arithmetic. When the assignment statement is executed, the expression to the right of the equal sign replaces the value of the variable or array element to the left.

basic real constant. A string of decimal digits containing a decimal point, and expressing a real value.

blank common. An unnamed common block.

character constant. A string of one or more alphameric characters enclosed in apostrophes. The delimiting apsotrophes are not part of the constant.

character expression. An expression in the form of a single character constant, variable, array element, substring, function reference, or another expression enclosed in parentheses. A character expression is always of type character.

character type. A data type that can consist of any alphameric characters; in storage, one byte is used for each character.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first real constant of the pair is the real part of the complex number; the second is the imaginary part.

complex type. An approximation of the value of a complex number, consisting of an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

connected file. A file that has been connected to a unit and defined by a FILEDEF command or by job control statements.

constant. An unvarying quantity. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and hexadecimal data.

control statement. Any of the statements used to alter the normal sequential execution of FORTRAN statements, or to terminate the execution of a FORTRAN program. FORTRAN control statements are any of the forms of the GO TO, IF, and DO statements, or the PAUSE, CONTINUE, and STOP statements.

data. (1) * (ISO) A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. (2) In FORTRAN, data includes constants, variables, arrays, and character substrings.

data item. A constant, variable, array element, or character substring.

data set. The major unit of data storage and retrieval consisting of data collected in one of several prescribed arrangements and described by control information to which the system has access.

data set reference number. A constant or variable in an input or output statement that identifies a data set to be processed.

data type. The properties and internal representation that characterize data and functions. The basic types are integer, real, complex, logical, double precision, and character.

*** digit.** (ISO) A graphic character that represents an integer. For example, one of the characters 0 to 9.

DO loop. A range of statements executed repetitively by a DO statement.

double precision. The standard name for real data of storage length 8.

DO variable. A variable, specified in a DO statement, that is initialized or incremented prior to each execution of the statement or statements within a DO range. It is used to control the number of times the statements within the range are executed.

dummy argument. A variable within a subprogram or statement function definition with which actual arguments from the calling program or function reference are positionally associated. Dummy arguments are defined in a SUBROUTINE or FUNCTION statement, or in a statement function definition.

executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

executable statement. A statement that causes an action to be taken by the program; for example, to calculate, to test conditions, or to alter the flow of control.

existing file. A file that has been defined by a FILEDEF command or by job control statements.

expression. A notation that represents a value: a constant or a reference appearing alone, or combinations of constants and/or references with operators. An expression can be arithmetic, character, logical, or relational.

external file. A set of related external records treated as a unit; for example, in stock control, an external file would consist of a set of invoices.

external function. A function defined outside the program unit that refers to it.

external procedure. A SUBROUTINE OR FUNCTION subprogram written in FORTRAN.

file. A set of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

file definition statement. A statement that describes the characteristics of a file to a user program. For example, the OS/VS DD statement or DOS/VSE ASSGN statement for batch processing, or the FILEDEF command for CMS processing.

file reference. A reference within a program to a file. It is specified by a unit identifier.

formatted record. (1) A record, described in a FORMAT statement, that is transmitted, when necessary with data conversion, between internal storage and internal storage or to an external record. (2) A record transmitted with list-directed READ or WRITE statements and an EXTERNAL statement.

FORTRAN-supplied procedure. See "intrinsic function".

function reference. A source program reference to an intrinsic function, to an external function, or to a statement function.

function subprogram. A subprogram invoked through a function reference, and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

IBM EXTENSION

hexadecimal constant. A constant that is made up of the character Z followed by two or more hexadecimal digits.

END OF IBM EXTENSION

hierarchy of operations. The relative priority order used to evaluate expressions containing arithmetic, logical, or character operations.

implied DO. An indexing specification, similar to a DO statement, causing repetition over a range of data elements. (The word DO is omitted, hence the term "implied.")

integer constant. A string of decimal digits containing no decimal point and expressing a whole number.

integer expression. An arithmetic expression whose values are of integer type.

integer type. An arithmetic data type, capable of expressing the value of an integer. It can have a positive, negative, or zero value; it must not include a decimal point.

internal file. A set of related internal records treated as a unit.

intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical or character operations.

*** I/O.** Pertaining to either input or output, or both.

I/O list. A list of variables in an input or output statement specifying which data is to be read or which data is to be written. An output list may also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

labeled common. See "named common".

length specification. A source language specification of the number bytes to be occupied by a variable or an array element.

letter. A symbol representing a unit of the English alphabet.

list-directed. An input/output specification that uses a data list instead of a FORMAT specification.

logical constant. A constant that can have one of two values: "true" or "false."

logical expression. A combination of logical primaries and logical operators. A logical operator can have one of two values: true or false.

logical operator. Any of the set of operators .NOT. (negation), .AND. (connection: both), or .OR. (inclusion: either or both), .EQV. (equal), .NEQV.(not equal).

logical primary. A primary that can have the value "true" or "false." See also "primary".

logical type. A data type that can have the value "true" or "false" for VS FORTRAN. See also "data type".

looping. Repetitive execution of the same statement or statements. Usually controlled by a DO statement.

main program. A program unit, required for execution, that can call other program units but cannot be called by them.

name. A string of from one through six alphameric characters, the first of which must be alphabetic. Used to identify a constant, a variable, an array, a function, a subroutine, or a common block.

named common. A separate common block consisting of variables, arrays, and array declarators, and given a name.

nested DO. A DO statement whose range of statements is entirely contained within the range of another DO statement.

nonexecutable statement. A statement that describes the characteristics of the program unit, of data, of editing information, or of statement functions, but does not cause an action to be taken by the program.

nonexisting file. A file that has not been defined by a FILEDEF command or by job control statements.

*** numeric character.** (ISO) Synonym for digit.

numeric constant. A constant that expresses an integer, real, or complex number.

preconnected file. A unit or file that was defined at installation time. However, a preconnected file does not exist for a program if the file is not defined by a FILEDEF command or by job control statements.

predefined specification. The implied type and length specification of a data item, based on the initial character of its name in the absence of any specification to the contrary. The initial characters I-N type data items as integer; the initial characters A-H, O-Z, and $ type data items as real. No other data types are predefined. For VS FORTRAN, the length for both types is 4 bytes.

primary. An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

procedure subprogram. A function or subroutine subprogram.

program unit. A sequence of statements constituting a main program or subprogram.

real constant. A string of decimal digits that expresses a real number. A real constant must contain either a decimal point or a decimal exponent and may contain both.

real type. An arithmetic data type, capable of approximating the value of a real number. It can have a positive, negative, or zero value.

record. A collection of related items of data treated as a unit.

relational expression. An expression that consists of an arithmetic expression, followed by a relational operator, followed by another arithmetic expression or a character expression followed by a relational operator, followed by another character expression. The result is a value that is true or false.

relational operator. Any of the set of operators that can express a comparison between arithmetic expressions, and that can be either true or false:

.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

scale factor. A specification in a FORMAT statement that changes the location of the decimal point in a real number (and, if there is no exponent, the magnitude of the number).

specification statement. One of the set of statements that provides the compiler with information about the data used in the source program. In addition, the statement supplies the information required to allocate data storage.

specification subprogram. A subprogram headed by a BLOCK DATA statement and used to initialize variables in named common blocks.

statement. The basic unit of a FORTRAN program, that specifies an action to be performed, or the nature and characteristics of the data to be processed, or information about the program itself. Statements fall into two broad classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated to an arithmetic, logical, or character expression. In the remainder of the program the name can be used as a substitute for the expression.

statement function definition. A statement that defines a statement function. Its form is a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic, logical, or character expression.

statement function reference. A reference in an arithmetic, logical, or character expression to the name of a previously defined statement function.

statement label. See "statement number".

statement number. A number of from one through five decimal digits that is used to identify a statement. Statement numbers can be used to transfer control, to define the range of a DO statement, or to refer to a FORMAT statement.

subprogram. A program unit that is invoked by another program unit in the same program. In FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

subroutine subprogram. A subprogram whose first statement is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

* **subscript.** (1) (ISO) A symbol that is associated with the name of a set to identify a particular subset or element.

(2) A subscript quantity or set of subscript quantities, enclosed in parentheses and used with an array name to identify a particular array element.

subscript quantity. A component of a subscript: an integer constant, an integer variable, or an expression evaluated as an integer constant.

IBM EXTENSION

In VS FORTRAN, a subscript quantity may also be a real constant, variable, or expression.

END OF IBM EXTENSION

type declaration. The explicit specification of the type of a constant, variable, array, or function by use of an explicit type specification statement.

unformatted record. A record that is transmitted unchanged between internal storage and an external record.

unit. A means of referring to a file in order to use input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unit identifier. The number that specifies an external unit.

1. An integer expression whose value must be zero or positive. For VS FORTRAN, this integer value of length 4 must correspond to a DD name, a FILEDEF name, or an ASSGN name.
2. An asterisk (*) that corresponds on input to FT05F001 and on output to FT06F001.
3. The name of a character array, character array element, or character substring for an internal file.

variable. (1) * A quantity that can assume any of a given set of values.

(2) A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program execution.

INDEX

Special Characters

. (period), job control syntax 71
... (ellipses), job control syntax 72
+ as addition symbol 15, 55
| (OR sign), job control syntax 72
$ as first character in names 9
* (asterisk)
  job control syntax 71
  multiplication symbol 15, 55
  two as exponentiation symbol 15, 55
- (hyphen), job control syntax 72
- as subtraction symbol 15, 55
/ (slash)
  division symbol 15, 55
  format code 158
  job control syntax 71
/+ (end-of-procedure delimiter, DOS/VSE 88
// (concatenation operator), in character expressions 56
, (comma), job control syntax 71
[ ] (square brackets)
  job control syntax 72
: (colon)
  format code 158
  in array declarators 48
  in substring notation 49
() (parentheses)
  arithmetic expressions 15
  in array declarators 48
  in substring notation 49
  job control syntax 71
@PROCESS statement 73, 74
' (apostrophe or single quote)
  delimits character constants 44
  job control syntax 71
  within character constant 45
= (equal sign)
  assignment statement 14, 58
  job control syntax 71

A

A, format code 158
abnormal termination
  exceptions causing 138
  not initializing, common error 68
about this book iii-vi
ACCESS command, VM/370-CMS 94
access method option, OPEN statement 154
Access Method Services, catalogs DEFINE commands 181
ACTION, DOS/VSE linkage editor control statement 128
actual argument
  common coding errors 69
  description 191
  rules for use 192
  statement function references 61
addition, evaluation order 15, 55
address column, in storage map 117
algebraic equation, similar to assignment statement 58
ALLOCATE command for compilation, OS/VS2-TSO 104
ALLOCATE command, OS/VS2-TSO 103, 110-111
alphabetic character, definition 240
alphameric character set, definition 240
alphameric, definition 240
American National Standard FORTRAN iv
  flagging for 120-121
AMSERV CMS command, processes DEFINE commands 100
apostrophe
  delimits character constants 44
  job control syntax 71
  within character constant 45
appendixes
  A, device information 226-227
  B, assembler language considerations 228-235
  overall description 225
ARCOS 190
argument
  array, and assembler subprograms 232
  assembler programs and 231-232
  assigning values to 192
  COMMON statement and 193
  cross reference dictionary lists 119
  definition 240
  FUNCTION subprograms and 191
  general rules 192
  intrinsic functions as 202
  passing between programs 190-192
  SUBROUTINE subprograms and 191-192
  variable, and assembler subprograms 231
arithmetic efficiency, for optimization 218
arithmetic errors, common 69
arithmetic expression
  assignment statement processes 14-15
  definition 240
  description 53-55
  evaluation of 53
  evaluation rules 14-15
  in assignment statement 59
  operator precedence in 15
arithmetic expression, definition 240
arithmetic IF statement
  arithmetic operators in 53
  simplified FORTRAN programming 16
arithmetic operator, definition 240
arithmetic operators and their meanings 55
arithmetic results, ensuring needed precision 55
array declarator, definition 240
array element, definition 240
array element, internal file unit 168
array name, definition 240
arrays
  as actual arguments 192
  assembler subprograms and 232
  character, substrings of elements 49

cross reference dictionary lists 119
defining 12-13
definition 240
description 46-49
DO statement processes 17-18
efficient common arrangement 195-197
elements, in assignment statement 58
EQUIVALENCE statement and 51
explicit lower bounds 48
expressions and 53
implicit and explicit lower bounds, illustration of 48
implicit lower bounds 47
implicit sharing through EQUIVALENCE 52
initializing 50-51
initializing an entire 51
initializing character elements 50
initializing efficiently 214
initializing elements 50
initializing, common error 68
input/output operations and 213
internal file unit 168
multidimensional 13, 47
multidimensional, processing 17
negative lower bounds, illustration of 49
one-dimensional 12, 47
one-dimensional, processing 17
optimizing identically dimensioned 217
optimizing with identical elements 217
physical layout, illustration 13
programming considerations 49
signed subscripts and 48-49
storage sharing between, illustration 52
stored in column-major order 47
subscript references invalid, common error 69
variably dimensioned, recommendation against 215
ARSIN 190
ASCII character comparisons, in intrinsic functions 202
ASCII collating sequence, used in character intrinsic functions 57
ASCII encoded files, record formats 186
assembler language considerations
appendix B 228-235
common data in 230-231
FORTRAN data 230-235
FORTRAN subprogram references 228-229
linkage considerations 229-230
LIST option listing and 143
main programs 230
retrieving arguments 231-232
subprograms 230-235
ASSGN control statement, DOS/VSE
description 85-86
sequential files 170
when required for DOS/VSE compilation 88
ASSIGN statement.cross reference dictionary lists 119
ASSIGN statement, sets GO TO variable 66
assigned GO TO, invalid as DO loop terminal statement 65
assignment statement
arithmetic 59
character 59
definition 240
description 58-60
evaluates arithmetic expressions 14-15
evaluation rules 14
initializes data items 14
initializing principal diagonal using 51
intrinsic functions in 15-16
logical 60
simplified FORTRAN programming 13-16
statement functions in 60-61
substring references valid in 49
asterisk (*), job control syntax 71
asynchronous input/output statements 164-166
AT statement, description 139-140
ATTRIB command, OS/VS2-TSO 103
audience iii
autolink feature, DOS/VSE 128
automatic cross compilation 71
automatic cross system support, link-editing 122
avoiding coding errors 68-70

B

background command procedures, TSO 108
BACKSPACE statement
checking accuracy of records with 163
invalid for directly accessed VSAM direct files 178
replace a record 163
sequential files 163-164
sequentially accessed VSAM direct files 177, 178
VSAM sequential file considerations 176
basic real constant with real exponent, defining 43
basic real constant, definition 43, 240
BG, DOS/VSE linkage editor control option 128
blank character ( ), job control syntax 72
blank common
definition 240
description 199-201
example of use 200-201
illustration of use 201
initializing invalid 199
must be unnamed 199
only one allowed 199
rules for use 199-200
blank common, example of use 200
blanks, OPEN statement and 154
BLKSIZE parameter, for direct files 174
BLKSIZE, for direct files 173
BLOCK DATA statement
BLOCK DATA subprogram and 209
description of use 208
initializes named common 209
BLOCK DATA subprograms
coding 208-209
coding example 209
statements valid 209
block IF statement
description 61-64
execution rules 63-64

invalid as DO loop terminal statement 65
nested DO statements and 64
nesting 63
valid forms 63
BN, format code 158
BZ, format code 158

C

CALL CDUMP library routine 223
CALL command for compilation, OS/VS2-TSO 104-105
CALL command, OS/VS2-TSO 103, 106
CALL CPDUMP library routine 223
CALL DUMP library routine 223
CALL DVCHK library routine 222
CALL ERRMON error-handling routine 223
CALL ERRSAV error-handling routine 224
CALL ERRSET error-handling routine 224
CALL ERRSTR error-handling routine 224
CALL ERRTRA error-handling routine 224
CALL EXIT library routine 223
CALL loader option, OS/VS 127
CALL OVRFLW library routine 222
CALL PDUMP library routine 223
CALL statement
CDUMP 223
CPDUMP 223
DUMP 223
DVCHK 222
ERRMON 223
ERRSAV 224
ERRSET 224
ERRSTR 224
ERRTRA 224
EXIT 223
extended error handling 143
extended error handling and 143
general description 189
OPSYS, DOS/VSE 211
OVRFLW 222
PDUMP 223
calling and called programs
assembler considerations 228-235
called program, definition 189
calling program, definition 189
coding 201-209
common coding errors 69
detailed description 189-212
invocation example 189
main program, definition 189
shared data areas, example 193
subprogram definition 189
system considerations 209-212
CANCEL, DOS/VSE linkage editor control option 128
capacities, direct access devices 227
card deck, for source input 20
card punch files, VM/370-CMS 99
card reader files, VM/370-CMS 99
cataloged procedure
compile-only, OS/VS 20
DOS/VSE, writing and using 88-89
modifying , OS/VS 77-84
modifying, DOS/VSE 88-89
OS/VS compilation data sets and 77
retrieving, DOS/VSE 88
unique identifier, DOS/VSE 88
using for program output 134
using, OS/VS 77-84
cataloging
DOS/VSE record size 88
load module 127
object module 92
subprograms 209
VSAM DEFINE commands 181-182
CATALP, and DOS/VSE cataloged procedures 88
CATALR DOS/VSE function, catalogs object module 92
CATALS DOS/VSE function, catalogs source program 91
changing program execution sequence 16-18
character array element, internal file unit 168
character constant
definition 240
maximum size 46
PAUSE statement and 67
STOP statement and 67
character data type
constant 42
constant, defining by value 44-45
defining 10
definition 241
description 39-41
EQUIVALENCE statement restriction 52
general description 9
internal representation 232
substrings of items 49
valid lengths 39
variables, hexadecimal constants initialize 45
character expression
character operators in 53
definition 240
description 55-56
in assignment statement 59
character library functions 222
character substring, internal file unit 168
CLEAR, DOS/VSE linkage editor control option 128
CLOSE statement
description 156-157
direct files 173
sequential files 164
VSAM direct file considerations 177
VSAM sequential file considerations 176
closed loop errors 29
CMS commands, using 93
CMS considerations
See VM/370-CMS considerations
codes
compiler messages 22
compiler messages, error severity 22
coding errors to avoid 19
coding form, FORTRAN 7, 38
coding your program
coding errors to avoid 68-70
complete FORTRAN programming 38-70
OPTIMIZE(3) considerations 219-221
sharing data 190-201
simplified FORTRAN programming 7-19
colon (:)
in array declarators 48
in substring notation 49
comma (,), job control syntax 71
command procedures, TSO 108
common block, definition 241

COMMON blocks, storage maps and 117
common errors
  source program 19, 68-70
common expression elimination, OPTIMIZE(3) 220
common input/output statement options 152
common items, in cross reference dictionary 119
COMMON statement
  argument usage 193
  assembler programs and 230
  blank common 199-201
  BLOCK DATA subprogram and 209
  data item order in 193-194
  defines arrays 47
  description of use 193-201
  dummy variables for alignment 195
  efficient data arrangement 195-197
  example of use 200-201
  fixed order variable alignment 195
  length considerations 194-195
  named common 199-201
  passing subroutine arguments using 215
  rules for use 199-200
  shared data areas, example 193
  storage maps and 117
  transmitting values using 194
  type considerations 194-195
  using efficiently 215
compilation
  automatic cross compilation 71
  batch, DOS/VSE 87
  batch, OS/VS 76
  complete FORTRAN programming 71, 92
  DOS/VSE compile, link-edit, and execute procedure 28
  DOS/VSE files for 88
  entering source program 20
  fixing errors 22-23
  link-edit and execute, DOS/VSE 133
  link-edit and execute, OS/VS 131
  modification of defaults 89
  OS/VS compile, link-edit, and execute procedure 27
  OS/VS2-TSO 104-105
  output, illustration of 91
  requesting 20, 21
  requesting under DOS/VSE 87-88
  requesting under OS/VS 76
  requesting, DOS/VSE 21
  requesting, OS/VS 20-21
  simplified FORTRAN programming 20-21
  statistics in object listing 144
  VM/370-CMS 95
compile time options
  DECK 72
  defaults for 72
  EXEC statement specifies in DOS/VSE 87
  EXEC statement specifies in OS/VS 72
  FIPS 72
  FIXED 73
  FLAG 73
  FORTVS command specifies in VM/370 CMS 72
  FREE 73
  GOSTMT 73
  LANGLVL(66|77) 73
  LINECOUNT 73
  LIST 73
  MAP 73
  NAME 73
  NODECK 72
  NOFIPS 72
  NOGOSTMT 73
  NOLIST 73
  NOMAP 73
  NOOBJECT 73
  NOOPTIMIZE 73
  NOSOURCE 74
  NOTERMINAL 74
  NOXREF 74
  OBJECT 73
  OPTIMIZE(0|1|2|3) 73
  OPTION statement specifies in DOS/VSE 72
  SC 73
  SOURCE 74
  TERMINAL 74
  XREF 74
compile-only, when to request 20
compiler messages
  See diagnostic messages
compiler output
  cross reference listing, XREF option 115-120
  default options and 89
  description 89-92
  end of compilation message 120
  explicit options and 90
  illustration of 91
  LIST file, OS/VS2-TSO 105
  listing 112-121
  LISTING file, VM/370-CMS 96
  listing, header 112
  message listing, FLAG option 113-115
  OBJ file, OS/VS2-TSO 105
  simplified FORTRAN programming 21
  source program 90-91
  source program listing 112
  source program, DOS/VSE 91
  source program, OS/VS 90
  standard language flagging 120-121
  storage map listing, MAP option 115-120
  TEXT file, VM/370-CMS 96
  under OS/VS2-TSO 105
  under VM/370-CMS 96
complete FORTRAN programming
  coding errors to avoid 68-70
  compilation 71, 92
  description 33-150
  executing your program 130-134
  fixing compile time errors 112-121
  fixing execution-time errors 135-150
  general description 33
  link-editing your program 122-129
  OPTIMIZE(3) considerations 219-221
  program coding 38-70
completion code in dumps 138
complex constant, definition 44, 241
complex data type
  constant 42
  constant, defining by value 44
  defining 10
  definition 241
  description 39
  general description 9
  internal representation 234-235
  reference length in common 196
  valid in arithmetic expressions 55
  valid lengths 39
computational reordering, OPTIMIZE(3) 220

computations, how compiler recognizes duplicate 216, 217
concatenation operator (//), in character expressions 56
conditional control transfers, GO TO statement 66-67
connected file, definition 241
constant
  arithmetic, description 42
  assignment statement 58
  character, description 42
  character, STOP statement and 67
  complex, description 42
  cross reference dictionary lists 119
  defining by name 11-12, 46
  defining by value 11, 42-45
  definition 241
  description 42-46
  expressions and 53
  hexadecimal 42
  Hollerith 42
  integer, description 42
  literal, old FORTRAN 42, 45
  logical, description 42
  real, description 42
  restrictions as actual arguments 192
  subscripts 47
constant computations, how compiler recognizes 217
constant operands, recognition of 217
CONTINUE statement, description 64
control statement
  complete FORTRAN programming 61-68
  definition 241
  simplified FORTRAN programming 16-18
control variable for GO TO, set by ASSIGN statement 66
COPY, EDIT subcommand, OS/VS2-TSO 103
CP commands, using 93
critical variables, limitations on optimizing 218
cross reference dictionary
  compiler output 90
  description 118-120
  using 115-120
  XREF option requests 74
cross system support, link-editing 122
current FORTRAN, definition iv
current FORTRAN, description 1
current language documented in this book 2
current language, definition iv
current standard, flagging for 120

D

DARCOS 190
DARSIN 190
data
  assignment statement assigns values to 58
  classifications of 42-49
  constants 11-12, 42-46
  defining 8-13, 39-41
  definition 7, 38, 241
  dictionary, compiler output 119
  efficient arrangement, common areas 195-197
  EQUIVALENCE statement, reuses storage for 51-53
  explicit type statement and 10, 41
  IMPLICIT statement and 10, 40-41
  incorrect input, causes errors 30
  initializing 14, 50-51
  items, description 13
  length considerations, common areas 194-195
  order in COMMON areas 193-194
  retrieval of, with READ 8
  shared areas, example 193
  sharing between programs 190-201
  sharing storage for 193-201
  truncation, common coding error 68
  type considerations, common areas 194-195
  types and lengths 39
  types, general description 9
  typing groups of 40-41
  using efficiently 50-53
  variables 42
  WRITE statement displays 18
data files
  DOS/VSE test files 27
  OS/VS test files 26
  VM/370-CMS 98-101
data item, definition 241
data set reference number, definition 241
data set, definition 241
DATA statement
  BLOCK DATA subprogram and 209
  initializes arrays 50-51
  initializes data 50-51
data type
  COMMON statement considerations 194-195
  constants and 42-46
  cross reference dictionary and 119
  definition 241
  descriptions 39-41
  explicit type statement and 41
  general description 9
  IMPLICIT statement and 40
  initializing required 14
  predetermined definitions 9, 40
  specifying for groups of items 40-41
  specifying for single items 41
  valid lengths 39
DATA, VM/370-CMS filetype 98
date of compilation, compiler default 89
date, in output listing header 112
DCB parameter, defines OS/VS records 186
DD control statement, OS/VS
  defining direct files using 173
  description 75-76
  direct access labels and 185
  modifying in cataloged procedures 83-84
  sequential files 170
  tape labels and 184
  VSAM file processing 182
debug packets
  detailed description 139-140
  general description 30
DEBUG statement, description 139-140
debugging
  debug packets for 139-140
  dumps, requesting 149-150
  extended error handling and 140-143
  GOSTMT option and 136
  LIST option and 137

symbolic debugging example 141
decimal divide interrupt message 223
DECK compiler option 72
DECK option output, detailed description 122-124
DECK option, compiler output 90
DECLARED column in cross reference dictionary 119
defaults
  data types 9
  extended error handling 140
defaults, modification of 89
DEFINE command, VSAM
  creates catalog entry 179-181
  creating under VM/370-CMS 100
  execution of 181-182
  VSAM direct files 180
  VSAM sequential files 180
  VSAM-managed sequential files 181
DEFINED column in cross-reference dictionary 120
defining records 185-188
DELETE command, OS/VS2-TSO 103
designing your program
  complete FORTRAN programming 34-37
  resources available and 34
  simplified FORTRAN programming 6
  top-down development and 34-37
  typical structure and 34
device information
  appendix A 226-227
  maximum block size values 226
  minimum block size values 226
  when corrections required 22
devices valid for DOS/VSE execution 132
devices valid for OS/VS execution 130
diagnostic messages
  compile time, example 114
  compiler default 89
  compiler module identifier in 114
  compiler output and 21
  detailed description 113-115
  DOS/VSE considerations 115
  example 22
  execution time 29
  execution-time 135-139
  format of 22-23
  GOSTMT option and 136
  identifier for 22
  IFX compiler message prefix 114
  IFY, execution error message prefix 135
  language standard flagging 90
  library 135
  library message number 135
  library message, origin 135
  library, program action after 135
  LIST option and 137
  message number identifies 114
  operator 138-139
  OS/VS considerations 115
  prefix meanings 29
  program interrupt 137-138
  self explanatory 113
  severity level in 114
  traceback map and 135-136
  using 23
digit, definition 241
DIMENSION statement
  BLOCK DATA subprogram and 209
  defines arrays 13, 47
  severity level in 22
  simplified programming examples 12
direct access device capacities 227
direct access files, control statements for 86
direct access labels
  description of use 184-185
  processing 185
direct access, VSAM direct files 178
direct file processing
  CMS FILEDEF command and 98
  complete FORTRAN programming 170-174
  direct access devices valid 170
  formatted, description 157-160
  internal files and 168
  record formats 186
  source program considerations 171-173
  unformatted, description 157
  valid VSAM source statements, summary 175
  VM/370-CMS 98-99
  VSAM considerations 174-183
  VSAM direct access for 178
  VSAM sequential access for 177-178
  VSAM source language 177-179
diskette, as source input device 20
displacement, definition 195
DISPLAY statement, description 140
displaying program results 18
division, evaluation order 15, 55
DLBL command, VM/370-CMS 100
DLBL control statement, DOS/VSE
  description 86
  direct access labels 185
  direct files 174
  sequential files 170
  VSAM file processing 182
DO list, implied, in DATA statement 51
DO loop, definition 241
DO statement
  batch compilation 87
  block IF statement and 64
  branches into loops, warning 64
  CONTINUE statement useful with 64
  control transfers into, common coding error 69
  description 64-66
  DO loop, active upon execution 65
  flexibility of processing with 65
  incrementing loop controls 65
  invalid as DO loop terminal statement 65
  multidimensional arrays 17-18
  nested 17-18, 64
  one-dimensional arrays 17
  preventing loop execution 65
  simplified FORTRAN programming 17-18
  when execution completed 65
  writing loops inline 216
DO variable, definition 241
DOS/VSE considerations
  abnormal termination dumps 150
  automatic cross compilation 71
  cataloged procedures, writing and using 88-89
  cataloging load modules 127
  cataloging source program 91
  common input/output error 69
  compilation files 88
  compile, link-edit, and execute job 133
  compiler options and 72
  DEFINE command 182
  defining records 187-188

direct access labels 185
direct files 174
execute-only job 133
job control statement descriptions 85-87
link-edit and execute job 133
link-edit, executing 128
linkage editor, using 127-128
logical units needed for execution 132
maximum record length 188
message codes 115
object module, cataloging 92
overlay procedure 211
overlays 211-212
phase execution 133
program execution 27-28
publications vi
requesting compilation 21, 87-88
sequential files 170
tape labels 184
VSAM file creation 182-183
VSAM file processing 182-183
VSAM-managed sequential files 181

double precision data type
- conversions of 218
- defining 10
- double precision, definition 241

double precision data type, reference length in common 196
dummy argument
- common coding errors 69
- cross reference dictionary lists 119
- definition 241
- description 191
- restrictions on assigning values 192
- rules for use 192
- statement function definitions and 60

dummy variables, alignment using 195
DUMP, requests dynamic dump 149
dumps
- completion code in 138
- requesting 149-150

duplicate computations, how compiler recognizes 216
dynamic dumps, requesting 149

E

E, error code 22, 114
E, format code 158
EBCDIC collating sequence, used in relational expressions 57
EBCDIC data sets, record formats 186
EBCDIC data, and internal files 168
EBCDIC items, hexadecimal constants initialize 45
EDIT command, OS/VS2-TSO 103
EDIT command, OS/VS2-TSO, source program creation 102-104
EDIT command, VM/370-CMS 94
EDIT command, VM/370-CMS, source program creation 93-95
elimination of instructions, OPTIMIZE(3) 220
ellipses (...), job control syntax 72
ELSE IF statement, description 61-64
ELSE IF, invalid as DO loop terminal statement 65
ELSE statement, description 61-64
ELSE, invalid as DO loop terminal statement 65
END DEBUG statement, description 140
END IF statement, description 61-64
END IF, invalid as DO loop terminal statement 65
end of compilation message 120
END record, in object module 124
END statement
- BLOCK DATA subprogram and 209
- description 68
- executable 19, 68
- invalid as DO loop terminal statement 65
- simplified FORTRAN programming 19

end-of-file labels, when processed 184
end-of-procedure delimiter, DOS/VSE 88
end-of-volume labels, when processed 184
END, EDIT subcommand, OS/VS2-TSO 103
ENDFILE statement
- sequential files 163
- VSAM files treat as documentation 175

ending a program 19, 68
ENTRY control statement, OS/VS 211
entry point, traceback map lists 136
entry sequenced data set (ESDS), source language considerations 175-179
ENTRY statement
- description 41
- in subprograms 206-207

EP loader option, OS/VS 127
equal sign (=)
- assignment statement 14, 58
- job control syntax 71

EQUIVALENCE statement
- arrays and, illustration 52
- BLOCK DATA subprogram and 209
- COMMON statement and 197-198
- description 51-53
- errors using 198
- execution time efficiency, illustration 54
- execution-time efficiency and 52-53
- invalid references, common error 69
- optimization and 214
- restriction on CHARACTER items 52

equivalenced items, in cross reference dictionary 119
ERRMON, requests extended error control 142
error code 22
error handling library functions 223-224
error messages
- See diagnostic messages

error occurrences, warning on number of 143
error routine option, input/output 153
error summary, in traceback map 136
errors
- fixing compilation 22-23, 112-121
- fixing execution 29-30, 135-150

errors to avoid 19, 68-70
ERRSAV, alters entry in option table 142
ERRSET, changes entry in option table 142
ERRSTR, stores entry in option table 142
ERRTRA, requests traceback 142
ERRTRA, requests traceback maps 136
ESD record types 123

ESD record, in object module 123
ESDS, under OS/VS2-TSO 111
ESDS, under VM/370-CMS 99-101
ESDS, VSAM sequential files 174
exception codes, program interrupt messages 137
EXEC command, VM/370-CMS 94
EXEC control statement, DOS/VSE
  cataloged procedures and 88
  cataloged procedures use PROC parameter 88
  description 86-87
  DOS/VSE compilation size 87
  LINKEDT, for link-editing 128
  OV (override) parameter 89
  SIZE parameter required for VSAM files 183
EXEC control statement, OS/VS
  description 75
  linkage editor options, OS/VS 125
  loader data sets 127
  loader processing options 126
  modifying in cataloged procedures 82-83
executable program, definition 241
executable statement, definition 7, 38, 241
EXECUTE option 73
executing a program, OS/VS2-TSO 106-107
execution errors, fixing
  complete FORTRAN programming 135-150
  simplified FORTRAN programming 29-30
execution time efficiency and EQUIVALENCE 52-53
execution time efficiency, illustration of EQUIVALENCE 54
execution time error messages
  detailed description 135-139
  interrupt 223
  library 135
  operator 138-139
  program interrupt 137-138
  traceback map with 135-136
execution-time library
  automatic error handling 223
  description 222-224
  DOS/VSE SYSRLB used for 128
  user controlled error handling 223-224
execution, compiler
  See compilation
execution, program
  complete FORTRAN programming 130-134
  DOS/VSE compile, link-edit, and execute procedure 28
  DOS/VSE considerations 27-28
  error messages 29
  OS/VS compile, link-edit, and execute procedure 27
  OS/VS considerations 26-27
  OS/VS2-TSO 106-107
  output 28
  simplified FORTRAN programming 26-28
  test data files and 27
  under OS/VS2-TSO 105
  under VM/370-CMS 97
existing file, definition 241
explicit type statement
  BLOCK DATA subprogram and 209
  cross reference dictionary and 119
  defines arrays 47
  simplified FORTRAN programming 10
  type changes using, common coding error 69
exponent overflow interrupt message 223
exponent underflow interrupt message 223
exponentiation, evaluation order 15, 53, 55
expression
  common, OPTIMIZE(3) eliminates 220
  defining and using 53-58
  definition 7, 38, 241
  integer or real, in DO statement 65
  integer, in computed GO TO 66
  PARAMETER statement and 46
  restrictions as actual arguments 192
  scaling elimination 217
extended error handling
  complete FORTRAN programming 140-143
  control through CALL statements 143
extensions, how documented 3
EXTENT control statement, DOS/VSE
  description 86
  parameters required for VSAM 182
  required for direct files 174
  sequential files 170
  VSAM file processing 182
external file, definition 241
external function, definition 241
external procedure, definition 241
EXTERNAL statement, description 203-204

F

factoring expressions 217
file definition statement, definition 242
file identifier, VM/370-CMS 98
file labels, direct access 184
file labels, tape files 183
file reference, definition 242
file status option, OPEN statement 153
file, definition 241
FILE, EDIT subcommand, VM/370-CMS 94
file, INQUIRE statement and 156
FILEDEF command for data files 98-99
FILEDEF command, VM/370-CMS 94, 98-99
filemode, VM/370-CMS 98
filename option, OPEN statement 154
filename, VM/370-CMS 98
filetype, VM/370-CMS 98
finding execution errors 29-30
FIPS compiler option 72
FIPS option, output for 120-121
FIXED compiler option 73
fixed form input 38
fixed length records, description 185
fixed order variable alignment 195
fixed-point divide interrupt message 223
fixed-point items, conversions of 218
fixing compilation errors 23
fixing compile-time errors
  complete FORTRAN programming 112-121
  simplified FORTRAN programming 22-23
  using compiler output listing 112-121
fixing execution-time errors
  complete FORTRAN programming 135-150
  simplified FORTRAN programming 29-30
FLAG compiler option 73, 113

FLAG compiler option, detailed description 115
floating-point divide interrupt message 223
floating-point items, conversions of 218
FMODE, EDIT subcommand, VM/370-CMS 94
FNAME, EDIT subcommand, VM/370-CMS 94
foreground command procedures, TSO 108
FORMAT statement
  codes, examples 158
  cross reference dictionary lists 119
  description 157-160
  display example 159
  group specifications 159-160
  nested specifications 159-160
  repeated specifications 160
  variable specifications 160
formatted I/O
  external 157-160
  internal READ statement 169
  internal WRITE statement 169
formatted record, definition 242
formatting information, in READ statement 8
formatting rules, common errors 68
FORTRAN
  See VS FORTRAN
FORTRAN publications
  current FORTRAN source programs 7
  current language documented in this book 2
  old FORTRAN source programs 7
  related systems publications v-vi
  usage of 2-3
  usage, illustration 3
  VS FORTRAN Application Programming iv
  VS FORTRAN Installation and Customization iv-v
FORTRAN 77, definition iv
FORTRAN-supplied functions
  See intrinsic functions
FORTRAN, VM/370-CMS filetype 98
FORTVC cataloged procedure, OS/VS 78
FORTVCG cataloged procedure, OS/VS 81
FORTVCL cataloged procedure, OS/VS 78
FORTVCLG cataloged procedure, OS/VS 79
FORTVG cataloged procedure, OS/VS 80
FORTVL cataloged procedure, OS/VS 81
FORTVLG cataloged procedure, OS/VS 80
FORTVS command, VM/370-CMS 94, 95-96
FREE command, OS/VS2-TSO 103
FREE compiler option 73
FREE compiler option, OS/VS2-TSO considerations 105
FREE compiler option, VM/370-CMS considerations 96
free form input 38, 95
FTnnFnnn, data sets for OS/VS execution 130
FTnnFnnn, optional OS/VS loader data set 127
full FIPS flagging 121
function reference
  definition 242
  evaluation order 15, 55
  explicit type statement and 202
  general description 189, 190
FUNCTION statement, in subprogram 205
FUNCTION subprograms
  arguments in 191-192
  CALL OPSYS loads, DOS/VSE 211
  coding 205-206
  definition 242
  ENTRY statement in 206-207
  general description 189
  invoking 203-204
  paired arguments in 190
  RETURN statement in 207-208
  SAVE statement as documentation 208
  top-down development and 37
  using 203-204
F1-F6, DOS/VSE linkage editor control options 128

G

G, format code 158
general description, VS FORTRAN 1-3
general logic structure of programs 6
GENMOD command, VM/370-CMS 97
GETFILE, EDIT subcommand, VM/370-CMS 94
GLOBAL command, VM/370-CMS 94, 96-97
glossary
  ANSI definitions 240
  definitions of terms 240-244
  ISO definitions 240
GO TO statement
  assigned, description 66
  computed, description 66-67
  control transfer to next executable statement 67
  warning on branches into loops 64
  when invalid as DO loop terminal statement 65
GOSTMT compiler option 73
  description 73
  traceback map and 136
group format specifications 159-160

H

HELP command, OS/VS2-TSO 103
HELP, EDIT subcommand, OS/VS2-TSO 103
hexadecimal constant
  definition 242
  description 42, 45
hierarchy of operations, definition 242
Hollerith constant, description 42, 45
hyphen (-), job control syntax 72

I

I/O list, definition 242
I/O status option, input/output 153
I/O, definition 242
I, format code 158
I, informational code 22, 114
identity matrix, initializing 51
IF block 62
IF statement
  block, repeated ELSE IF statements in 63
  block, valid forms 63
  common errors using OPTIMIZE(3) 69

logical, optimization of 219
logical, simplified FORTRAN programming 16-17
optimization and 219
OPTIMIZE(3) and 220
simplified programming 16-17
IF-level 62
IFX, compiler message prefix 22, 114
IFY, execution error message prefix 29, 135
imaginary part of a complex constant, defining 44
IMPLICIT statement
BLOCK DATA subprogram and 209
data initialization and 50
description 40-41
simplified FORTRAN programming 10
type changes using, common coding error 69
wraparound scan of 40
implied DO list, in DATA statement 51
implied DO, definition 242
INCLUDE command, VM/370-CMS 94, 97
INCLUDE statement, FORTRAN
advantages in top-down development 37
can be nested 36
CMS record size 95
OS/VS record size 77
SYSLIB required for CMS 95
SYSLIB required under OS/VS 77
SYSSLB required under DOS/VSE 88
top-down development and 36-37
INCLUDE, DOS/VSE linkage editor control 211
INCLUDE, DOS/VSE linkage editor control statement 128
INCLUDE, OS/VS linkage editor control statement 126, 211
incorrect input, and errors 30
incorrect logic, and errors 30
industry standards iii-iv
information messages, compiler default 89
informational code 22
initialization errors, common 69
initializing named common 199
input/output
common errors 69
common statement options 152
data sets for OS/VS 130
detailed description 152-188
DOS/VSE logical units 132
formatted 157-160
formatted, description 157-160
optimization and 213
statement list 152
unformatted, description 157
INPUT, EDIT subcommand, OS/VS2-TSO 103
INPUT, EDIT subcommand, VM/370-CMS 94
INQUIRE statement, description 156
INSERT control statement, OS/VS 210
INSERT, EDIT subcommand, OS/VS2-TSO 103
instruction elimination, OPTIMIZE(3) 220
integer constant with real exponent, defining 43
integer data type
constant 42
constant, defining by value 43
default names for 9
defining 10
definition 242
description 39-41
division gives integer results 55
general description 9
internal representation 233
optimization efficiency and 214
predetermined definition 9
reference length in common 196
valid in arithmetic expressions 55
valid lengths 39
variable as 13
variable in DO statement 65
variable, array subscripts and 47
integer expression in computed GO TO 66
integer expression, definition 242
internal file
complete FORTRAN programming 168-170
definition 242
internal sequence number (ISN)
compile-time messages optionally contain 115
compiler messages 23
source program listing 21
source program listing prints 112
traceback map uses 136
intrinsic functions
ASCII character comparisons in 202
cross reference dictionary lists 119
definition 242
explicit type statement and 202
general description 190
invoking 201-203
simplified FORTRAN programming 15-16
storage map lists 116
TSO usage of 106
INTRINSIC statement
description of use 202
IOSTAT Option, VSAM return code placed in 179
ISO FORTRAN iv
ISO, identifies ISO glossary definitions 240

J

job control considerations
DOS/VSE 84-87
OS/VS 74-76
job control statement syntax 71-72
JOB control statement, DOS/VSE
description 85
JOB control statement, OS/VS
description 74-75
JOBLIB DD, using 131
jobname
DOS/VSE compile-only example 21
OS/VS compile-only procedure 20

L

LABEL column in cross reference dictionary 119
LABEL parameter, DD statement 185
LABEL parameter, tape files 184
labels, description of use 183-185
LANGLVL(66|77) compiler option 73
language extensions, how documented 3
language levels, manuals valid 38

LBLTYP control statement, tape labels and 184
length specification, definition 242
LET linkage editor option, OS/VS 125
LET loader option, OS/VS 126
letter, definition 242
level codes
  description of 114
  E (error) 114
  I (information) 114
  S (serious error) 114
  U (abnormal termination) 115
  W (warning) 114
  0 (information) 114
  12 (serious error) 114
  16 (abnormal termination) 115
  4 (warning) 114
  8 (error) 114
library
  See execution-time library
library messages 135
  See also diagnostic messages
library module, identified in messages 135
LIBRARY, OS/VS linkage editor control statement 126
LINECOUNT compiler option 73
LINK command, OS/VS2-TSO 103, 106
link-editing
  complete FORTRAN programming 122-129
  cross system support, automatic 122
  DECK option and 122
  DOS/VSE compile, link-edit, and execute procedure 28
  DOS/VSE logical units 128
  execution 128
  execution with load module, OS/VS 131
  execution with phase, DOS/VSE 133
  for immediate execution 124
  for later execution 127
  input for 122
  optional OS/VS linkage editor data sets 126
  OS/VS compile, link-edit, and execute procedure 27
  OS/VS linkage editor control statements 126
  output 129
  required OS/VS linkage editor data sets 126
  simplified FORTRAN programming 24-25
  TSO listings 106
  under DOS/VSE 24-25, 128
  under OS/VS 24, 125-127
  under OS/VS2-TSO 105-107
  under VM/370-CMS 97
linkage editor program, OS/VS 125-127
LIST compiler option 73
  description of listing 143-148
  example of output 148
  format of listing 144-148
  traceback map and 137
LIST compiler option, OS/VS2-TSO considerations 105
list directed input/output
  description 166-168
  input data 166
  READ statement 167
  WRITE statement 168
LIST file, OS/VS2-TSO 105
LIST linkage editor option, OS/VS 125
LIST option, compiler output 90
list-directed, definition 242
LIST, EDIT subcommand, OS/VS2-TSO 103
LISTFILE command, VM/370-CMS 94
LISTING file, VM/370-CMS 96
literal constant, description 42
literal constant, old FORTRAN 45
LOAD command, VM/370-CMS 94, 97
load module
  cataloging, DOS/VSE 127
  DOS/VSE execution 133
  DOS/VSE logical units 132
  execution of 130, 134
  execution output 134
  OS/VS execution 131-132
  OS/VS program data sets 130
  producing 124
  using cataloged 131
loader program under TSO 107
loader program, OS/VS 126-127
LOADGO command, OS/VS2-TSO 103
logic errors, MAP option helps find 116
logic structure of programs, general 6
logic, incorrect, and errors 30
logical constant, definition 242
logical data type
  constant 42
  constant, defining by value 44
  data items used in logical expressions 57
  definition 242
  description 39
  internal representation 232
  optimization efficiency and 214
  reference length in common 196
  simplified FORTRAN programming 16-17
  valid lengths 39
logical expression
  definition 242
  description 57-58
  in assignment statement 60
  logical IF statement and 16-17
  logical operators in 53
logical operator, definition 242
logical operators and their meanings 57
logical primary, definition 242
logical units for DOS/VSE linkage editing 128
logoff, OS/VS2-TSO 102
logoff, VM/370-CMS 93
logon, OS/VS2-TSO 102
logon, VM/370-CMS 93
looping, definition 242
loops, programming
  block IF statement and 64
  branches into, warning 64
  complete FORTRAN programming 64-66
  CONTINUE statement useful with 64
  description 64-66
  errors in 29
  for multidimensional arrays 17-18
  for one-dimensional arrays 17
  nested 17-18, 64
  OPTIMIZE(3) and 220
  simplified FORTRAN programming 17-18
  writing inline 216
lower bound, in substring notation 49
lower bounds, in arrays 48
lower case items, job control syntax 71

M

magnetic tape labels
  See tape labels
main program
  coding 201-204
  common coding errors 69
  definition 242
  general description 189
  invocation example 189
  naming 201
manual organization iii
manuals, valid for language levels 38
MAP compiler option
  compiler output 90
  description 73, 115-118
  example 116
MAP compiler option, OS/VS2-TSO considerations 105
MAP linkage editor option, OS/VS 125
MAP loader option, OS/VS 126
MAP option, using 115
MAP, DOS/VSE linkage editor control option 128
mathematical equivalence, when implied 52
mathematical errors 30
mathematical functions
  assignment statement uses 15-16
mathematical functions, in assignment statement 15
mathematical library functions 222
maximum block size values 226
maximum record length, DOS/VSE 188
message format, operator 139
message number, compiler 114
message prefix, compiler 114
message text
  compiler messages 23, 115
  library messages 135
  programmer-specified in PAUSE statement 139
  programmer-specified in STOP statement 139
messages
  See diagnostic messages
minimum block size values 226
misspelling words, common error 68
MODE column in cross reference dictionary 119
MODE column, in storage map 116
modification of compiler defaults 89
modifier statements, DOS/VSE cataloged procedures 88
module identifier, compiler messages 114
MOVE, EDIT subcommand, OS/VS2-TSO 103
multiplication, evaluation order 15, 55
MVS considerations
  abnormal termination dumps 150
  automatic cross compilation 71
  batch compilation 76
  cataloged load modules, using 131
  cataloged procedures 77-84
  cataloging load modules 125
  cataloging source program 90
  compilation data sets 76-77
  compile-only cataloged procedure 20
  compile, link-edit and execute job 131
  compiler options and 72
  data sets needed for execution 130
  defining records 186-187
  direct access labels 185
  direct files 173
  execution-only job 131
  job control statements 74-76
  link-edit and execute job 131
  link-edit execution 125-126
  linkage editor, using 125-126
  load module execution 131-132
  loader program under TSO 107
  loader, using 126-127
  message codes 115
  object module, cataloging 92
  overlays 210-211
  program execution 26-27
  publications v
  requesting compilation 20-21, 76
  sequential files 170
  tape labels 184
  VSAM DEFINE command 181
  VSAM file creation 182
  VSAM file processing 182

N

n, programmer-specified in PAUSE statement 139
n, programmer-specified in STOP statement 139
name
  complete FORTRAN programming 40-49
  cross reference dictionary and 119
  definition 242
  PARAMETER statement uses 46
  simplified FORTRAN programming 9-13
  table of, compiler output 90
name column in cross reference dictionary 119
NAME column, in storage map 116
NAME compiler option 73
named common
  definition 242
  description 199-201
  example of use 200-201
  illustration of use 201
  initializing 199
  length restriction 199
  naming 199
  rules for use 199-200
named common, BLOCK DATA programs initialize 208
named common, example of use 200
NCAL linkage editor option, OS/VS 125
NCAL loader option, OS/VS 127
nested DO loops 17-18
nested DO, definition 242
nested format specifications 159-160
NOAUTO, DOS/VSE linkage editor control option 128
NODECK compiler option 72
NOFIPS compiler option 72
NOGOSTMT compiler option 73
NOLET loader option, OS/VS 126
NOLIST compiler option 73
NOMAP compiler option 73
NOMAP loader option, OS/VS 126
NOMAP, DOS/VSE linkage editor control option 128

nonexecutable statement, definition 7, 38, 243
nonexisting file, definition 243
NOOBJECT compiler option 73
NOOPTIMIZE compiler option 73
NOPRINT loader option, OS/VS 127
NORES loader option, OS/VS 127
NOSOURCE compiler option 74
notation, substring 49
NOTERMINAL compiler option 74
NOXREF compiler option 74
number, library message 135
numeric character, definition 243
numeric constant, definition 243
numeric constant, PAUSE statement and 67
numeric constant, STOP statement and 67

O

OBJ file, OS/VS2-TSO 105
OBJECT compiler option
  description 73
  produces load module 124
OBJECT compiler option, OS/VS2-TSO considerations 105
object module
  card image listing, compiler output 90
  cataloging 92
  cataloging, DOS/VSE 92, 127
  cataloging, OS/VS 92
  compiler default 89
  compiler output 21
  DECK option and 122
  END record in 124
  ESD record in 123
  example of listing 148
  example of structure 124
  link-editing 122-129
  link-editor input 122
  listing, compiler output 90
  listing, optimization usage 219
  obtaining listing of 143-148
  RLD record in 123
  TXT record in 123
object of a branch, cross reference dictionary lists 119
OBJECT option, produces load module 127
object time efficiency, illustration of EQUIVALENCE 54
old FORTRAN, definition iv
old FORTRAN, description 1
old language, definition iv
OPEN statement
  description 153-155
  direct files 171
  EXTENT statement and, DOS/VSE VSAM files 183
  invalid for VSAM-managed sequential files (DOS/VSE) 179
  sequential files 161
  VSAM direct file considerations 177
  VSAM sequential file considerations 176
operands, recognition of constant 217
operation interrupt message 223
operator message format 139
operator message identification 139
operator messages 138-139
operator precedence in expressions
  arithmetic 55
  arithmetic operators 15
  logical 57
  order of 53
operator responses
  PAUSE statement 67, 139
operator, communicating with 67-68
OPSYS, using in DOS/VSE 211
OPTIMIZE compiler option
  arithmetic conversions, avoiding 218
  array initialization 214
  array operands and 213
  arrays, optimizing identically dimensioned 217
  arrays, optimizing with identical elements 217
  arrays, variably dimensioned not recommended 215
  COMMON blocks, using efficiently 215
  common expressions, OPTIMIZE(3) eliminates 220
  constant operand recognition 217
  description 73
  double precision conversions and 218
  duplicate computation recognition 216, 217
  efficient accumulator usage 218
  efficient arithmetic constructions and 218
  efficient program size 213
  EQUIVALENCE statement not recommended 214
  higher levels best 213
  IF statement and 219
  instruction elimination, OPTIMIZE(3) 220
  integer variables and 214
  logical variables and 214
  loops and OPTIMIZE(3) 220
  object listing and 219
  object module useful with 143
  OPTIMIZE(3) considerations 219-221
  passing subroutine arguments in COMMON 215
  scaling elimination 217
  single precision conversions and 218
  source program considerations 213-221
  unformatted input/output and 213
  variables, optimization limitations 218
  writing loops inline 216
OPTIMIZE(3), common errors using 69
OPTION control statement, DOS/VSE
  description 85
option table, warning on error occurrences 143
OR sign (|), job control syntax 72
organization of this manual iii
origin of library messages 135
OS/VS1 considerations
  abnormal termination dumps 150
  automatic cross compilation 71
  batch compilation 76
  cataloged load modules, using 131
  cataloged procedures 77-84
  cataloging load modules 125
  cataloging source program 90
  compilation data sets 76-77
  compile-only cataloged procedure 20
  compile, link-edit and execute job 131

compiler options and 72
data sets needed for execution 130
defining records 186-187
direct access labels 185
direct files 173
execution-only job 131
job control statements 74-76
link-edit and execute job 131
link-edit execution 125-126
linkage editor, using 125-126
load module execution 131-132
loader, using 126-127
message codes 115
object module, cataloging 92
overlays 210-211
program execution 26-27
publications v-vi
requesting compilation 20-21, 76
sequential files 170
tape labels 184
VSAM DEFINE command 181
VSAM file creation 182
VSAM file processing 182
output listing
general description 89-90
illustration of 91
simplified programming 21
using 112-120
using the object module listing 143-148
output listing header 112
output, link-editing 129
output, program
error free 29, 134
with errors 30, 134
OVEND (end of modifiers) statement, DOS/VSE 89
overlay 210-212
OVERLAY control statement, OS/VS 210
overview, VS FORTRAN 1-3
OVLY linkage editor option, OS/VS 126

P

PARAMETER statement
advantages in using 46
BLOCK DATA subprogram and 209
data initialization and 50
names constants 11-12, 46
parameter, symbolic 77-84
parentheses (), in arithmetic expressions 15
parentheses (), in array declarators 48
parentheses (), in substring notation 49
parentheses (), job control syntax 71
passing arguments between programs 190-192
PAUSE statement
description 67
operator message and 139
PDUMP, requests dynamic dump 149
period (.), job control syntax 71
PHASE control statement, DOS/VSE
linkage editor 128
overlay and 211
phase, DOS/VSE
execution of 130, 133-134
logical units needed for execution 132
precision errors, common 69
precision in arithmetic results, ensuring needed 55
preconnected file, definition 243
predefined CALL statements
predefined specification, definition 243
predetermined data type definitions 9
preface iii-vi
prefixes for messages 29
primary, definition 243
principal diagonal, initializing 51
PRINT command, VM/370-CMS 94
PRINT loader option, OS/VS 127
printer files, VM/370-CMS 99
printing program results 18
PROC statement, modifying in cataloged procedures 82
procedure subprogram, definition 243
procedure, definition 243
PROCESS, @, statement 76
processing options, OS/VS linkage editor 125
program action after library messages 135
program constants
See constant
program design
complete FORTRAN programming 34-37
resources available and 34
simplified FORTRAN programming 6
top-down development and 34-37
typical structure and 34
program interrupt messages 137-138
program output
error free 29, 134
with errors 30, 134
program references, substring references valid in 49
PROGRAM statement, using 201
program stub 34
program switches, assigned GO TO statement 66
program termination, ends DO loop execution 65
program unit
definition 243
sharing storage between 193-201
program-name
program, coding
See coding your program
program, sample 31
programming, general logic structure 6
publications
current FORTRAN source programs 7
current language documented in this book 2
DOS/VSE publications vi
MVS publications v
old FORTRAN source programs 7
OS/VS publications v-vi
OS/VS1 publications v-vi
related systems publications v-vi
usage of 2-3
usage, illustration 3
VM/370-CMS publications v
VS FORTRAN Application Programming iv
VS FORTRAN Installation and Customization iv-v
PUNCH command, VM/370-CMS 94
punctuation, common errors in 68

Q

QUIT, EDIT subcommand, VM/370-CMS 94

R

READ statement
   asynchronous 165
   description 155
   direct files 172-173
   directly accessed VSAM direct files 178
   errors in 30
   FORMAT statement and 157
   internal files 168, 169
   internal, simplified FORTRAN programming 8
   list-directed 167
   sequential files 162
   sequentially accessed VSAM direct files 177
   simplified FORTRAN programming 7-8
   unformatted record size and 186
   VSAM considerations 176
real constant with real exponent, defining 43
real constant, definition 243
real data type
   constant 42
   constant, defining by value 43-44
   default names for 9
   defining 10
   definition 243
   description 39-41
   general description 9
   internal representation 233-234
   predetermined definition 9
   reference length in common 196
   subscripts 47
   valid in arithmetic expressions 55
   valid lengths 39
   variable in DO statement 65
real part of a complex constant, defining 44
RECFM=F required for direct files 173
record formats, direct access files 186
record formats, OS/VS 185-186, 187
record formatting, OPEN statement and 155
record length, OPEN statement and 155
record size, source program 20
record, definition 243
records, defining 185-188
recursive calls, common coding errors 69
reference number, data set (definition) 241
reference numbers, for DOS/VSE execution 132
reference numbers, for OS/VS execution 130
REFERENCES column in cross-reference dictionary 119, 120
references, substring 49
register usage, traceback map lists 136
relational expression
   character operands in 57
   description 56-57
   in subprograms 207-208
   relational operators in 53
relational expression, definition 243
relational operator, definition 243
relational operators and their meanings 56
relative record data set (RRDS), source language considerations 175-179
relative record number, in direct files 170
remainder, integer division truncates 55
RENAME command, VM/370-CMS 94
repeated format specifications 160
reply to PAUSE statement, operator 67
RES loader option, OS/VS 127
RETURN statement
   ends DO loop execution 65
   invalid as DO loop terminal statement 65
   simulated in assembler subprograms 232
REWIND statement
   invalid for directly accessed VSAM direct files 178
   sequential files 163
   sequentially accessed VSAM direct files 177, 178
   VSAM sequential file considerations 176
RLD record, in object module 123
routines, listed in traceback map 136
RRDS, under OS/VS2-TSO 111
RRDS, under VM/370-CMS 99-101
RRDS, VSAM direct files 174
rules for use 192
RUN command, VM/370-CMS 94

S

S, severe error code 22, 114
SAM data sets, VSAM-managed sequential files 174
sample program 31
SAVE statement
   BLOCK DATA subprogram and 209
   in subprograms 208
SAVE, EDIT subcommand, OS/VS2-TSO 103
SC compiler option 73
scale factor, definition 243
secondary input, defining using VM/370-CMS 97
sequential access, VSAM direct files 177-178
sequential file processing
   ASCII considerations 186
   asynchronous input/output statements 164-166
   CMS FILEDEF command and 98
   complete FORTRAN programming 161-170
   concatenating, warning on 161
   devices valid for 161
   EBCDIC encoded records 186
   formatted, description 157-160
   internal files and 168
   list-directed 166-168
   READ, simplified FORTRAN programming 7-8
   source program considerations 161-166

unformatted, description 157
valid VSAM source statements, summary 175
VM/370-CMS 98-99
VSAM considerations 174-183
VSAM source language 175-179
WRITE, simplified FORTRAN programming 18
service routine library functions 222-223
severe error code 22
severity level, compiler messages 114
sharing data between programs 190-201
simplified FORTRAN programming
coding your program 7-19
common coding errors 19
compiling your program 20-21
description 5-31
designing your program 6
executing your program 26-28
fixing compile time errors 22-23
fixing execution time errors 29-30
general description 5
link-editing your program 24-25
single precision, conversions of 218
SIZE linkage editor option, OS/VS 126
SIZE loader option, OS/VS 127
SIZE parameter, VSAM file processing 183
slash (/), job control syntax 71
slash(/), format code 158
SOURCE compiler option 74, 112-113
SOURCE compiler option, OS/VS2-TSO considerations 105
source program
cataloging 90-91
direct file processing 171-173
efficient size 213
input/output 152-188
internal files in 168-170
sequential file considerations 161-168
using OS/VS2-TSO 102-104
using VM/370-CMS 93-95
source program elements, definition 7
source program listing
compiler default 89
description 112-113
simplified FORTRAN programming. 21
using MAP and XREF 115
source program map, using 115
source program optimization, detailed description 213-221
source program reordering, OPTIMIZE(3) 220
SP, format code 158
spanned records, description 186
specification statement, definition 243
specification subprogram, definition 243
SPF (Structure Programming Facility), using under TSO 102
square brackets ([]), job control syntax 72
SS, format code 158
standard file labels, direct access 184
standard file labels, tape files 183
standard user labels, direct access 185
standard user labels, tape files 183
standards
applicable for VS FORTRAN iii-iv
current FORTRAN iv
current standard, definition iv
FORTRAN 77, definition iv
old FORTRAN iv
old standard, definition iv
START command, VM/370-CMS 94
statement
definition 7, 38
specification, and data initialization 50
statement function
common coding error 69
cross reference dictionary lists 119
definition of 60
definition, placement of in program 61
description 60-61
references to in program 61
storage map lists 116
statement function definition, definition 243
statement function reference, definition 243
statement function, definition 243
statement labels 239
statement number
assembler subprograms and 232
ASSIGN statement sets 66
assigned GO TO statement list 66
cross reference dictionary lists 119
definition 243
dictionary, compiler output 119
END statement, valid in 19, 68
storage maps and 118
statement number, definition 243
statement number, description 12
statement options, input/output 152
STATUS command, OS/VS2-TSO 103
STEPLIB DD, using 131
STOP statement
causes program termination 139
description 67
ends DO loop execution 65
invalid as DO loop terminal statement 65
operator message and 139
storage map description 116-118
storage map, example 116
storage sharing
See EQUIVALENCE statement
Structure Programming Facility (SPF), using under TSO 102
structured programming, description 61
stub, program 34
SUBMIT command, OS/VS2-TSO 103
subprogram
arguments in, general rules 192
BLOCK DATA 208-209
coding 205-209
common coding errors 69
cross reference dictionary names 119
definition 189, 244
FUNCTION 205-207
function, general description 189
general description 189
intrinsic function, general description 190
intrinsic functions, using 201-203
invocation example 189
paired arguments in 190
storage map lists 116
SUBROUTINE 206-208
subroutine, general description 189
top-down development and 37
SUBROUTINE statement, in subprogram 206

SUBROUTINE subprogram
   arguments in 191-192
   CALL OPSYS loads, DOS/VSE 211
   coding 206-208
   definition 244
   ENTRY statement in 206-207
   general description 189
   invoking 204
   paired arguments in 190
   RETURN statement in 207-208
   SAVE statement as documentation 208
   top-down development and 37
   using 203-204
subscript
   complete FORTRAN programming 47
   defining 12-13
   definition 244
   description 46-49
   explicit lower bounds and 48
   implicit lower bounds and 47
   initializing array elements and 50
   initializing character array elements and 50
   integer variable as 13
   invalid values for, common coding error 69
   multidimensional arrays and 47
   one-dimensional arrays and 12, 47
   order of processing 47
   programming considerations 49
   signed 48-49
subscript quantity, definition 244
subset FIPS flagging 121
substring notation 49
substring, internal file unit 168
substrings of character items 49, 60
subtraction, evaluation order 15, 55
summary of errors, in traceback map 136
symbolic debugging example 141
symbolic parameter 77-84
syntactical unit, definition 72
syntax errors, compile-only runs find 20
syntax errors, MAP option helps find 116
syntax, job control statements 71-72
SYSCLB, DOS/VSE link-edit logical unit 128
SYSIN, data set for OS/VS execution 130
SYSIN, logical unit for DOS/VSE execution 132
SYSIN, required OS/VS compilation data set 77
SYSIPT, logical unit for OS/VS execution 132
SYSLIB required for CMS 95
SYSLIB, optional for OS/VS linkage editor 126
SYSLIB, optional OS/VS compilation data set 77
SYSLIB, optional OS/VS loader data set 127
SYSLIB, OS/VS
   catalogs OS/VS object module 92
   catalogs source program 90
SYSLIN, optional OS/VS compilation data set 77
SYSLIN, OS/VS loader required data set 127
SYSLIN, required for OS/VS linkage editor 126
SYSLMOD, cataloging load modules and 125
SYSLMOD, required for OS/VS linkage editor 126
SYSLNK, DOS/VSE link-edit logical unit 128
SYSLOUT, optional OS/VS loader data set 127
SYSLST, DOS/VSE link-edit logical unit 128
SYSPRINT, required for OS/VS linkage editor 126
SYSPRINT, required OS/VS compilation data set 77
SYSPRINT, required OS/VS loader data set 127
SYSPUNCH, optional OS/VS compilation data set 77
SYSRES, DOS/VSE link-edit logical unit 128
SYSRLB, DOS/VSE link-edit logical unit 128
system considerations, direct files 173-174
system considerations, sequential files 170
system considerations, TSO 111
SYSTERM, optional for OS/VS linkage editor 126
SYSTERM, optional OS/VS compilation data set 77
SYSUT1, required for OS/VS linkage editor 126
SYS001, DOS/VSE link-edit logical unit 128

T

TAG column in cross reference dictionary 119
TAG column, in storage map 117
tape files
   ASCII considerations 186
   CMS FILEDEF command and 98-99
tape labels
   description of use 183-184
   processing 183-184
TERM compiler option, OS/VS2-TSO considerations 105
terminal
   free-form input useful with 38
   source input device 20
   using with OS/VS2-TSO 102
   using with VM/370-CMS 93
TERMINAL compiler option 74
terminal files, VM/370-CMS FILEDEF command and 99
TEST command, OS/VS2-TSO 103
TEXT file, VM/370-CMS 96
Time Sharing Option
   See TSO (Time Sharing Option)
time, in output listing header 112
TL, format code 158
TLBL control statement, tape labels and 184
top-down design and development
   advantages of 35
   description 34-37
   example 36
   FORTPAN considerations 36-37
   INCLUDE statement and 36-37
   subprograms and 37

TR, format code 158
TRACE ON|OFF statements, description 140
TRACE ON/OFF statements, description 140
traceback map
    description 135-136
trailer labels, when processed 184
transfer of control, ends DO loop execution 65
truncation, common coding error 68
TSO (Time Sharing Option)
    ALLOCATE command 110-111
    background command procedures 108
    command procedures under 108
    compilation 104-105
    creating source programs 102-104
    description of use 102-111
    executing 105-107
    file naming conventions 109
    foreground command procedures 108
    free form source and 105
    linkage editor listings 106
    LIST compiler option and 105
    loader program and 107
    loading 105-107
    logoff 102
    logon 102
    MAP compiler option and 105
    OBJECT compiler option and 105
    SOURCE compiler option and 105
    system considerations 111
    TERM compiler option and 105
    terminal usage 102
    using commands 102
    VSAM data files 111
TSO command procedures 108
TSO commands, using 102
TXT record, in object module 123
TYPE command, VM/370-CMS 94
type declaration, definition 244
type statement
    See explicit type statement

U

U, unrecoverable error code 22, 115
unconditional GO TO, invalid as DO loop terminal statement 65
undefined length records, description 186
unformatted input/output 157
unformatted record, definition 244
unformatted records, EBCDIC encoded files 186
unit
    definition 244
    external, in input/output 153
    external, in READ 8
    external, in WRITE 18
    internal, in READ 8
    internal, in READ and WRITE 168
unit identifier, definition 244
UNIT option, input/output 153
unit record files, VM/370-CMS FILEDEF command and 99
unit, INQUIRE statement and 156
unrecoverable error code 22
upper bound, in substring notation 49
upper bounds, in arrays 48
uppercase items, job control syntax 71
user standard labels, direct access 185
user standard labels, tape files 183
using terminals 93, 102

V

variable
    accumulator usage 218
    and assembler subprograms 231
    as actual arguments 192
    assignment statement 58
    character, hexadecimal constants initialize 45
    character, substrings of 49
    definition 244
    description 42
    dummy, for alignment in common 195
    efficient common arrangement 195-197
    EQUIVALENCE statement and 51
    example in WRITE statement 18
    expressions and 53
    fixed order alignment in common 195
    integer or real, in DO statement 65
    internal file unit 168
    internal representation 232-235
    optimization limitations 218
    READ statement example 8
    recognition when constant 217
    storage map lists 116
    subscript 13
    subscripts 47
variable format specifications 160
variable-length records, description 185
VM/370-CMS considerations
    compilation 95-96
    compiler options and 72
    creating source programs 93-95
    description of use 93-101
    executing 97
    file identifier 98
    FILEDEF command for data files 98-99
    filemode 98
    filename 98
    filetype 98
    free form source and 96
    library availability 96-97
    load module creation 97
    loading 97
    logoff 93
    logon 93
    terminal usage 93
    VSAM data files 99-101
    VSAM file creation 101
    VSAM file definition under 100
    VSAM file processing 101
volume labels, direct access 184
volume labels, tape files 183
VS FORTRAN
    coding form 7
    common coding errors 68-70
    complete FORTRAN programming 33-150
    current FORTRAN language level 1
    current language documented in this book 2
    extensions, how documented 3
    features 1-2
    general description 1-3
    input/output features 152-188

old FORTRAN language level 1
overview 1-3
reference numbers for DOS/VSE execution 132
reference numbers for OS/VS execution 130
simplified FORTRAN programming 5-31
source input format rules 20
source program elements 7
special features 101, 111, 151
VSAM file processing
catalog entry creation 179-181
complete FORTRAN programming 174-183
creation of files 182-183
IOSTAT option obtains return code 179
processing of files 182-183
source language considerations 175-179
under OS/VS2-TSO 111
under VM/370-CMS 99-101
valid source statements, summary 175
VSAM-managed sequential files (DOS/VSE) 179
VSCOM, common errors using 69

W

W, warning error code 22, 114
WAIT statement, asynchronous input/output 165-166
waits, errors in 29
warning error code 22
WRITE statement
asynchronous 164-165
description 155
direct files 171-172
directly accessed VSAM direct files 178
errors in 30
FORMAT statement and 157
internal files 168, 169
list-directed 168
sequential files 161-162
sequentially accessed VSAM direct files 177
simplified FORTRAN programming 18
unformatted record size and 186
VSAM considerations 176

X

XREF compiler option 74, 115
XREF compiler option, detailed description 120
XREF compiler option, output 90
XREF linkage editor option, OS/VS 125

Y

yy, operator message identifier 139

0

0, in operator message 139
0, informational code 22, 114

1

12, severe error code 22, 114
16, abnormal termination code 22, 115

4

4, warning error code 22, 114

8

8, error code 22, 114

SC26-3985-2

VS FORTRAN Application Programming: Guide (File No. S370-25) Printed in U.S.A. SC26-3985-2

IBM®