

VS FORTRAN Version 2



Programming Guide for CMS and MVS

Release 6

VS FORTRAN Version 2



Programming Guide for CMS and MVS

Release 6

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page vii.

ReadMe!

Additional information, made available after publication of this edition, may be found in the VS FORTRAN Version 2 README file. This file is updated with new information pertaining to this product as the result of service updates, as well as any additional information provided in response to Reader Comment Forms. On CMS, this file name is named VSF2 README and is located on the disk on which the VS FORTRAN Version 2 product is installed. On MVS, this file is named SYS1.VSF2.README (note: your product installer may have chosen a name other than those given here; check with your local product support for the correct name used by your installation). You may browse or print this file using local procedures to view this information.

| Eighth Edition (November 1993)

| This edition replaces and makes obsolete the previous edition, SC26-4222-06.

| This edition applies to VS FORTRAN Version 2 Release 6, Program Numbers 5668-805, 5688-087, and 5668-806, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Changes" following "About This Book." Specific changes for this edition are indicated by a vertical bar to the left of the change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Canada Ltd. Laboratory Information Development 2G/345/1150/TOR 1150 Eglinton Avenue East North York, Ontario, Canada M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1986, 1993. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Programming Interface Information	vii
Trademarks and Service Marks	vii
About This Book	ix
How This Book Is Organized	ix
How to Use This Book	x
Syntax Notation	x
Publications	xi
Industry Standards	xiii
Documentation of IBM Extensions	xiii
Operating System Support	xiv
Summary of Changes	xv
Release 6, October 1993	xv
Major Changes to the Product	xv
Release 5, September 1991	xvii
Major Changes to the Product	xvii
Release 4, August 1989	xviii
Major Changes to the Product	xviii
Release 3, March 1988	xix
Major Changes to the Product	xix
Release 2, June 1987	xx
Major Changes to the Product	xx
Release 1.1, September 1986	xxi
Major Changes to the Product	xxi

Part 1. Introduction 1

Chapter 1. Overview of VS FORTRAN Version 2	3
Compiler	3
Run-Time Library	3
Interactive Debug	3

Part 2. Compiling and Running Your Program 5

Chapter 2. Compiling Your Program	7
Compiling Your Program under CMS	7
Compiling Your Program under MVS	11
Compiling Your Program under MVS with TSO	18
Chapter 3. Using the Compile-Time Options	21
Available Compile-Time Options	21
Conflicting Compile-Time Options	40
Using the Compiler Output Listing	41
Using the Terminal Output Display—TERMINAL and TRMFLG Options	49
Using the Standard Language Flagger—FIPS Option	50
Using the SAA Flagger—SAA Option	50
Using the Automatic Precision Increase Facility—AUTODBL Option	51

Chapter 4. Running Your Program	61
Running Your Program under CMS	62
Running Your Program under MVS	74
Running Your Program under MVS with TSO	91
Dynamically Loaded User Subprograms	97

Chapter 5. Using the Run-Time Options and Identifying Run-Time Errors	105
Available Run-Time Options	105
Establishing a Default Run-Time Options Table	117
Identifying Run-Time Errors	118
Identifying Coding Errors	129

Part 3. Performing Input/Output Operations 131

Chapter 6. I/O Concepts and Terminology	133
External and Internal Files	133
Units and File Connection	134
File Definitions and Dynamic File Allocation	134
Named Files	136
Unnamed Files	137
VS FORTRAN File Access Methods	138
Access Methods Used by the Operating System	139
Records As Seen by Fortran	140
Records As Seen by the Operating System	140
File Existence	141

Chapter 7. Connecting, Disconnecting, and Reconnecting Files for I/O	143
Preconnecting Data Files to Units	143
Connecting Files	150
Disconnecting Files	157
Reconnecting Files	159

Chapter 8. Using File Access Methods	163
Input/Output Operations for Sequential Access	163
Input/Output Operations for Direct Access	181
Input/Output Operations for Keyed Access	182
Input/Output Operations for Sequential, Direct, and Keyed Access	191

Chapter 9. Advanced I/O Topics	195
Dynamically Allocating a File	195
File Characteristic Defaults	197
Coding INQUIRE in Your Program	201

Chapter 10. Considerations for Double-Byte Data	205
Connecting A File Containing Double-Byte Data	205

Chapter 11. Using VSAM	209
VSAM Terminology	210
Organizing Your VSAM File	210
Defining VSAM Files	211
Using VSAM Keyed Files	215
Processing DEFINE Commands	219
Source Language Considerations—VSAM Files	220

Obtaining the VSAM Return Code—IOSTAT Option	226
Chapter 12. Considerations for Specifying RECFM, LRECL, and BLKSIZE	227
Chapter 13. What Determines File Existence	233
CMS File Existence Tables	235
MVS File Existence Tables	241

Part 4. High Performance Features 251

Chapter 14. Overview of High Performance Features	253
Overview of the Optimization Feature	253
Overview of the Automatic Vector Feature	256
Overview of the Automatic Parallel Feature	257
Terminology for Automatic Vector and Parallel Processing	258
Overview of the Parallel Language Feature	266
Chapter 15. Using the Optimization Feature	267
Increasing Optimization of Your Program	267
Debugging Optimized Programs	275
Chapter 16. Using the Vector Feature	277
Techniques for Improving Vectorization	277
Examples of Vectorization	286
Considerations and Restrictions for Vectorization	290
Chapter 17. Using the Parallel Feature	293
Overview of Parallel Programs	294
Using Parallel Processing within a Parallel Task	297
Using Parallel Processing between Parallel Tasks	310
Using Library Service Subroutines and Functions	317
Using the Parallel Trace Facility	325
Parallel Environment	331
Improving Parallel Processing	332
Using the Parallel Feature—Examples	333
Chapter 18. Aids for Tuning Your Program	343
Producing Compiler Reports	344
Using Parallel and Vector Directives	353
Gathering Run-Time Statistics	368

Part 5. Advanced Coding Topics 369

Chapter 19. Interprogram Communication	371
Passing Arguments to Subprograms	371
Using Common Areas	372
Intercompilation Analysis	380
Chapter 20. The Multitasking Facility (MTF)	397
Introducing MTF	398
Designing and Coding Applications for MTF	406
Compiling and Linking Programs That Use MTF	414

Running Programs That Use MTF	416
Converting MTF Programs to Parallel Programs	418

Chapter 21. Creating Reentrant Programs	423
Creating and Using a Reentrant Program under CMS	430
Creating and Using a Reentrant Program under MVS	439
Link-Editing and Running a Reentrant Program under MVS with TSO	444

Part 6. Appendixes	447
-------------------------------------	------------

Appendix A. Internal Limits in VS FORTRAN Version 2	449
--	------------

Appendix B. Assembler Language Considerations	451
Using Fortran Data in Assembler Subprograms	451
Linkage Conventions for VS FORTRAN Programs	453
Calling Fortran Subprograms from Assembler Programs	457
Calling a Fortran Main Program from an Assembler Subprogram	460
Calling Assembler Subprograms from Fortran Programs	460
Internal Representation of VS FORTRAN Version 2 Data	469
Requesting Compilation from an Assembler Program	473

Appendix C. Object Module Records	475
--	------------

Appendix D. Compatibility and Migration Considerations	479
Differences Among Certain IBM FORTRAN Compilers	481
Passing Character Arguments	483
Extended Common Blocks	484
Using the Current Library with Existing VS FORTRAN Load Modules	484
Migration Considerations	485

Appendix E. Compiler Report Diagnostic Messages	491
Compiler Report Diagnostic Messages—Vector	492
Compiler Report Diagnostic Messages—Parallel	528
Compiler Report Diagnostic Messages—Vector and Parallel	549

Index	581
------------------------	------------

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interface Information

This book is intended to help you to compile and run application programs using VS FORTRAN Version 2 on CMS and MVS. This book documents General-Use Programming Interface and Associated Guidance Information provided by VS FORTRAN Version 2.

General-Use programming interfaces allow the customer to write programs that obtain the services of VS FORTRAN Version 2.

Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

AIX
AIX/ESA
ESA/390
ES/3090
ES/9000
IBM
MVS/ESA
MVS/SP
MVS/XA
RACF
SAA
Systems Application Architecture
System/370
System/390
VM/ESA
VM/XA
3090

About This Book

How This Book Is Organized

This book is organized as follows:

Chapter 1, “Overview of VS FORTRAN Version 2,” gives an overview of the language, compiler, library, and debugger.

Chapter 2, “Compiling Your Program,” explains how to compile your program under CMS, MVS, and MVS with TSO.

Chapter 3, “Using the Compile-Time Options,” describes the compiler options and explains how to interpret the compiler listing.

Chapter 4, “Running Your Program,” explains how to run your program under CMS, MVS, and MVS with TSO.

Chapter 5, “Using the Run-Time Options and Identifying Run-Time Errors,” describes the run-time options and several VS FORTRAN Version 2 features to help you identify run-time errors.

Chapter 6, “I/O Concepts and Terminology,” explains the concepts and terms associated with Fortran files and input/output processing.

Chapter 7, “Connecting, Disconnecting, and Reconnecting Files for I/O,” describes preconnecting, connecting, disconnecting, and reconnecting Fortran files for I/O.

Chapter 8, “Using File Access Methods,” gives an overview of the access methods used by Fortran and explains the coding of the I/O statements specific to each access method.

Chapter 9, “Advanced I/O Topics,” describes how to dynamically allocate certain files by omitting file definition statements, and how to use and override the file characteristic defaults.

Chapter 10, “Considerations for Double-Byte Data,” explains how to connect files with double-byte data and shows how double-byte data appears in a sample compiler listing.

Chapter 11, “Using VSAM,” discusses considerations for using VSAM input/output files.

Chapter 12, “Considerations for Specifying RECFM, LRECL, and BLKSIZE,” describes how VS FORTRAN prioritizes record format, record length, and block size values from different sources and gives the IBM*-supplied defaults for these values.

Chapter 13, “What Determines File Existence,” describes the conditions that VS FORTRAN Version 2 uses to determine the existence of input/output files.

Chapter 14, “Overview of High Performance Features,” gives an overview of the optimization, vector, and parallel features.

* IBM is a trademark of the International Business Machines Corporation.

Chapter 15, “Using the Optimization Feature,” suggests ways to make your programs run faster and the best way to use the OPTIMIZE compiler option.

Chapter 16, “Using the Vector Feature,” explains how to code programs that make use of the vector facility.

Chapter 17, “Using the Parallel Feature,” explains how to code programs that use parallel processing.

Chapter 18, “Aids for Tuning Your Program,” describes compiler reports and compiler directives, which help you increase the parallel and/or vector performance of your programs.

Chapter 19, “Interprogram Communication,” explains how to use arguments and common data areas to associate data between calling and called programs. It also explains how to use the intercompilation analysis feature.

Chapter 20, “The Multitasking Facility (MTF),” explains how to use MTF under MVS.

Chapter 21, “Creating Reentrant Programs,” explains the advantages and limitations of reentrant programs and gives an overview of how to create and use them.

Appendix A, “Internal Limits in VS FORTRAN Version 2,” describes the specifications for the maximum sizes and lengths for various VS FORTRAN statements and constructs.

Appendix B, “Assembler Language Considerations,” explains how to call Fortran subprograms and main programs from assembler programs.

Appendix C, “Object Module Records,” describes the structure of the object module and contents of the SYM object module record.

Appendix D, “Compatibility and Migration Considerations,” discusses compatibility of VS FORTRAN Version 2 with VS FORTRAN Version 1 as well as with earlier IBM Fortran products.

Appendix E, “Compiler Report Diagnostic Messages,” describes the diagnostic messages that appear in the compiler report listing.

How to Use This Book

For application programming, you will need to use both this book and *VS FORTRAN Version 2 Language and Library Reference*. This book contains information on how to compile and run your VS FORTRAN Version 2 programs, as well as some information on advanced coding topics. *VS FORTRAN Version 2 Language and Library Reference* contains more detailed, supplementary information.

Syntax Notation

The following items explain how to interpret the syntax used in this manual:

- Uppercase letters and special characters (such as commas and parentheses) are to be coded exactly as shown, except where otherwise noted. You can mix lowercase and uppercase letters; lowercase letters are equivalent to their uppercase counterparts, except in character constants.
- Italicized, lowercase letters or words indicate variables, such as array names or data types, and are to be substituted.

- Underlined letters or words indicate IBM-supplied defaults.
- Ellipses (...) indicate that the preceding optional items can appear one or more times in succession.
- Braces ({ }) group items from which you must choose one.
- Square brackets ([]) group optional items from which you can choose none, one, or more.
- OR signs (|) indicate you can choose only one of the items they separate.
- Blanks in Fortran statements are used to improve readability; they have no significance, except when shown within apostrophes (' '). In non-Fortran statements, blanks can be significant. Code non-Fortran statements exactly as shown.
- The + at the end of a TSO command line indicates a continuation on the next line.




For example, given the following syntax:

CALL *name* [([*arg1* [,*arg2*] ...])]

these statements are among those allowed:

```
CALL ABCD
CALL ABCD ( )
CALL ABCD (X)
CALL ABCD (X, Y)
CALL ABCD (X, Y, Z)
```

For double-byte character data, the following syntax notation is used:

-  represents the shift-out character (X'0E'), which indicates the start of double-byte character data
-  represents the shift-in character (X'0F'), which indicates the end of double-byte character data
-  represents the left half of a double-byte EBCDIC character (X'42')
- kk represents a double-byte character not in the double-byte EBCDIC character set

Publications

Figure 1 on page xii lists the VS FORTRAN Version 2 publications and the tasks they support.

Figure 1. VS FORTRAN Version 2 Publication Library

Task	VS FORTRAN Version 2 Publication	Order Number
Evaluation and Planning	<i>General Information</i>	GC26-4219
	<i>Licensed Program Specifications</i>	GC26-4225
Installation and Customization	<i>Installation and Customization for CMS</i>	SC26-4339
	<i>Installation and Customization for MVS</i>	SC26-4340
Application Programming	<i>Language and Library Reference</i>	SC26-4221
	<i>Programming Guide for CMS and MVS</i>	SC26-4222
	<i>Interactive Debug Guide and Reference</i>	SC26-4223
	<i>Reference Summary</i>	SX26-3751
Library Reference	<i>Master Index and Glossary</i>	SC26-4603
Diagnosis	<i>Diagnosis Guide</i>	LY27-9516
Migration	<i>Migration from the Parallel FORTRAN PRPQ</i>	SC26-4686

Figure 2 lists additional publications that you might need while using VS FORTRAN.

Figure 2 (Page 1 of 2). Related Publications

Title	Order Number
<i>Access Method Services for the Integrated Catalog Facility</i>	SC26-4562
<i>IBM Enterprise Systems Architecture/370 and System/370 Vector Operations</i>	SA22-7125
<i>IBM System/360 and System/370 FORTRAN IV Language</i>	GC28-6515
<i>IBM System/370 Principles of Operation</i>	GA22-7000
<i>Interactive Storage Management Facility User's Guide</i>	SC26-4563
<i>MVS/ESA Application Development Guide</i>	GC28-1672
<i>MVS/ESA JCL Reference</i>	GC28-1829
<i>MVS/370 JCL Reference</i>	GC28-1350
<i>MVS/XA Integrated Catalog Administration: Access Method Services Reference</i>	GC26-4135
<i>MVS/XA JCL Reference</i>	GC28-1352
<i>MVS/XA Linkage Editor and Loader User's Guide</i>	GC26-4011
<i>MVS/XA Supervisor Services and Macro Instructions</i>	GC28-1154
<i>OS/VS Tape Labels</i>	GC26-3795
<i>OS/VS2 MVS JCL Reference</i>	GC28-0692
<i>Systems Application Architecture, An Overview</i>	GC26-4341
<i>Systems Application Architecture Common Programming Interface: FORTRAN Reference</i>	SC26-4357
<i>IBM High Level Assembler/MVS & VM & VSE: Language Reference</i>	SC26-4940
<i>IBM High Level Assembler/MVS & VM & VSE: Programmer's Guide</i>	SC26-4941
<i>Using Magnetic Tape Labels and File Structure</i>	SC26-4565
<i>VM/ESA: ESA/Extended Configuration Principles of Operation</i>	SC24-5594

Figure 2 (Page 2 of 2). Related Publications

Title	Order Number
<i>VM/SP Installation Guide</i>	SC24-5237
<i>VM/SP Planning Guide and Reference</i>	SC19-6201
<i>VM/SP System Programmer's Guide</i>	SC19-6203
<i>VM/SP CMS User's Guide</i>	SC19-6210

Industry Standards

The VS FORTRAN Version 2 compiler and library are designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of September 1991.

The following two standards are technically equivalent. In the publications, references to **FORTRAN 77** are references to these two standards:

- 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.

The bit string manipulation functions are based on ANSI/ISA-S61.1.

The following two standards are technically equivalent. References to **FORTRAN 66** are references to these two standards:

- American Standard FORTRAN, X3.9-1966
- International Organization for Standardization ISO R 1539-1972 Programming Languages—FORTRAN.

At both the FORTRAN 77 and the FORTRAN 66 levels, the VS FORTRAN Version 2 language also includes IBM extensions. References to **current FORTRAN** are references to the FORTRAN 77 standard, plus the IBM extensions valid with it. References to **old FORTRAN** are references to the FORTRAN 66 standard, plus the IBM extensions valid with it.

Documentation of IBM Extensions

In addition to the statements available in FORTRAN 77, IBM provides “extensions” to the language. In the *VS FORTRAN Version 2 Language and Library Reference*, these extensions are printed in color.

VS FORTRAN's implementation of parallel language tracks a subset of the PCF proposal (a precursor to ANSI committee X3H5—Parallel Processing Constructs for High Level Programming Languages) as it advances through industry acceptance and the standards proposal process. Any future changes made by these groups to the parallel language that is implemented in VS FORTRAN may result in corresponding changes in future releases of the VS FORTRAN product, even if these changes result in incompatibilities with previous releases of the product.

Operating System Support

In this manual, MVS refers to MVS/SP*, MVS/XA*, and MVS/ESA* systems; VM refers to VM/SP, VM/XA*, and VM/ESA* systems; and XA refers to VM/XA, VM/ESA, MVS/XA, and MVS/ESA systems.

The following are supported under MVS only:

- Multitasking facility
- Data-in-virtual
- Asynchronous input/output.

Programs run in parallel only on VM/XA, VM/ESA, MVS/XA, and MVS/ESA.

Support for extended common blocks requires execution of the compiled code under MVS/ESA or VM/ESA.

Programs with function unique to an operating system can be run only on that system.

Load modules that have been link-edited on one operating system cannot be run on another operating system.

* MVS/ESA, MVS/SP, MVS/XA, VM/ESA, and VM/XA are trademarks of the International Business Machines Corporation.

Summary of Changes

Release 6, October 1993

Major Changes to the Product

Support for AIX^{*}/370 is not included in VS FORTRAN Version 2 Release 6.
Support for AIX/ESA^{*} is found in the AIX VS FORTRAN/ESA product.

- Printable copies of certain of the VS FORTRAN Version 2 Release 6 publications are provided on the product tape.
- Support for additional language has been added, much of which address previous incompatibilities between XL FORTRAN and VS FORTRAN
 - Additional constant and operator support:
 - Quote-delimited literal character constants
 - Maximum negative integer literal value
 - Typeless constants (binary, octal and hexadecimal)
 - Named constants in complex constants
 - .XOR. logical operator
 - <> graphical relational operator
 - Storage classes:
 - STATIC and AUTOMATIC storage classification statements
 - IMPLICIT STATIC and IMPLICIT AUTOMATIC statement support
 - Additional data type support:
 - LOGICAL*2 and LOGICAL*8
 - INTEGER*1 and INTEGER*8
 - UNSIGNED*1
 - BYTE
 - DOUBLE COMPLEX and DOUBLE COMPLEX FUNCTION type specifications
 - XREF and MAP Processing has been improved to remove storage-ordering perturbations.
 - Additional intrinsic function support:
 - INTEGER*8, INTEGER*2, and INTEGER*1 support for existing functions
 - XOR, LSHIFT, RSHIFT, ISHFTC, IBITS, and LOC have been added.
 - HALT compiler option to reduce compilation times for unsuccessful compilations
 - MVBITS and ARGSTR service routines
 - Dynamic storage support, for allocating and deallocating storage at run-time
 - Integer POINTER data type

^{*} AIX and AIX/ESA are trademarks of the International Business Machines Corporation.

- Including dynamically dimensioned arrays
- ALLOCATED, DEALLOCATE, and NULLIFY statements
- Intrinsic function ALLOCATED
- Compile-time option DDIM
- Dynamically loaded module support (via DYNAMIC option)
- I/O features enhancements:
 - Support for dummy arguments in NAMELIST lists.
 - NML keyword for NAMELIST READ/WRITE statements
 - OPEN and INQUIRE statement support for the POSITION, PAD, and DELIM specifiers
 - B and O format control codes
 - Expansion of the Z format control code
 - \$ format control code
- Optimization improvements:
 - Optimization for pipelined instruction scheduling
 - Improved optimization for Inter-loop register assignments
- Message improvements:
 - All messages written to SYSTERM file
 - Information messages no longer marked as errors
- Run-time options improvements:
 - Abbreviations are now allowed.
 - Default I/O units are now user-specifiable with ERRUNIT, RDRUNIT, PRTUNIT, and PUNUNIT
- Vector support enhancements:
 - The VECTOR option defaults can be set at installation.
 - VECTOR suboptions MODEL and SPRECOPT.
 - Vectorized SIGN, ANINT, DNINT, NINT and IDNINT intrinsic functions
 - Vector performance improvements
- Parallel support enhancements:
 - LOCAL statement allowing arrays for Parallel Do / Parallel Sections
 - NAMELIST processing for parallel environment allows local variables
 - Support for user-generated parallel trace (including service routines PTWRIT and PTPARM and suboption TRACE)
 - Parallel execution controls (affinity control)
 - Load module support for routines invoked via SCHEDULE and PARALLEL CALL

Major Changes to the Product

- Support has been added in the compiler and library for the Advanced Interactive Executive/370 (AIX/370) operating system. In addition, programs designed to be run on AIX/370 may use the following enhancements to VS FORTRAN Version 2:
 - The **fvs** command to invoke the VS FORTRAN Version 2 compiler.
 - Use of tab characters in source programs.
 - Communication between Fortran programs and C programs.
 - Coding of indirectly recursive Fortran routines.
- Support for parallel programs¹ has been added for programs running under CMS and MVS. These enhancements support:
 - Automatically generating parallel code for DO loops using a compile-time option.
 - Explicit coding of parallel loops, sections, and calls with parallel language extensions.
 - Using lock and event services to control synchronization in parallel programs.
 - Directing the compiler to generate parallel or serial code using enhanced directives.
 - Determining the number of virtual processors available and specification of the number of virtual processors to use during run time.
 - Using I/O within parallel programs.
 - Calling subroutines within parallel loops and sections.
 - Obtaining information for tuning your parallel program using a compiler report listing.
- Support for extended common blocks has been added for programs running under MVS/ESA or VM/ESA. Compile-time and run-time enhancements and options have been added to support the three types of common blocks that now exist.
- Virtual storage constraint relief has been added for compiling under MVS/XA, MVS/ESA, VM/XA, or VM/ESA. This support allows the compiler to reside above the 16MB line and to process in 31-bit addressing mode. Therefore, larger programs can now be compiled.
- The default name for a main program has been changed from MAIN to MAIN#, which allows MAIN to be used as a user name for a common block or other global entity.

¹ The VS FORTRAN Version 1 standard math routines (VSF2MATH) are not supported for parallel processing. Interactive debug can be used only with non-parallel programs, and with serial portions of parallel programs when the run-time option PARALLEL(1) is specified.

- Array declarator expressions for object-time dimensions can now be of type Integer*2.

Release 4, August 1989

Major Changes to the Product

- Enhancements to the vector and optimization features of VS FORTRAN Version 2
 - Automatic vectorization of user programs is enhanced by improvements to the dependence analysis done by the compiler. Specifically, the following constructs are eligible for vectorization:
 - Loops containing simple READ, WRITE, and PRINT statements
 - Loops containing equivalenced arrays
 - Loops containing branches out of the loop
 - Loop bound appearing in a subscript expression
 - Loop nests that process a triangular matrix
 - Simple IF loops
 - Integer sum reduction.
 - Additional advanced vector optimization.
 - In programs optimized at either OPT(2) or OPT(3), arithmetic constants will be propagated globally for scalar variables.
 - In programs optimized at either OPT(2) or OPT(3), informational messages are issued when local scalar variables may be referenced before they have been initialized.
 - Improved performance when the CHAR, ICHAR, and LEN character intrinsic functions are used.
 - Single and double precision complex divide routines can be vectorized.
- Enhancements to input/output support in VS FORTRAN Version 2
 - Improved data transfer rate for sequential DASD and tape input/output on MVS systems.
 - Ability to perform data striping (parallel I/O) on sequential data sets.
 - Ability to detect input conversion and record length errors.
- Enhancements to the intercompilation analyzer (ICA)
- Enhancements to the language capabilities of VS FORTRAN Version 2
 - Ability to use graphic relational operators as an alternative to FORTRAN 77 relational operators.
- Enhancements to the pseudo-assembler listing provided by VS FORTRAN Version 2
- Enhancements to the math routines:
 - Single, double, and extended precision MOD library routines are more precise.
 - Single and double precision scalar complex divide routines are more precise and faster, and are vectorizable.

- The old complex divide and MOD routines are in the alternate math library.

Release 3, March 1988

Major Changes to the Product

- Enhancements to the vector feature of VS FORTRAN Version 2
 - Automatic vectorization of user programs is improved by relaxing some restrictions on vectorizable source code. Specifically, VS FORTRAN Version 2 can now vectorize MAX and MIN intrinsic functions, COMPLEX compares, adjustably dimensioned arrays, and DO loops with unknown increments.
 - Ability to specify certain vector directives globally within a source program.
 - Addition of an option to generate the vector report in source order.
 - Ability to collect tuning information for vector source programs.
 - Ability to record compile-time statistics on vector length and stride and include these statistics in the vector report.
 - Ability to record and display run-time statistics on vector length and stride. Two new commands, VECSTAT and LISTVEC, have been added to interactive debug to support this function.
 - Enhancements to interactive debug to allow timing and sampling of DO loops.
 - Inclusion of vector feature messages in the online HELP function of interactive debug.
 - Vectorization is allowed at OPTIMIZE(2) and OPTIMIZE(3).
- Enhancements to the language capabilities of VS FORTRAN Version 2
 - Ability to specify the file or data-set name on the INCLUDE statement.
 - Ability to write comments on the same line as the code to which they refer.
 - Support for the DO WHILE programming construct.
 - Support for the ENDDO statement as the terminal statement of a DO loop.
 - Enhancements to the DO statement so that the label of the terminal statement is optional.
 - Support for statements extending to 99 continuation lines or a maximum of 6600 characters.
 - Implementation of IBM's Systems Application Architecture* (SAA*) FORTRAN definition; support for a flagger to indicate source language that does not conform to the language defined by SAA.
 - Support for the use of double-byte characters as variable names and as character data in source programs and I/O, and for interactive debug input and output.

* Systems Application Architecture and SAA are trademarks of the International Business Machines Corporation.

- Support for the use of a comma to indicate the end of data in a formatted input field, thus eliminating the need for the user to insert leading or trailing zeros or blanks.
- Enhancements to the programming aids in VS FORTRAN Version 2
 - Enhancements to the intercompilation analysis function to detect conflicting and undefined arguments.
 - Support for the data-in-virtual facility of MVS/XA and MVS/ESA.
 - Ability to allocate certain commonly used files and data sets dynamically.
 - Enhancements to the multitasking facility to allow large amounts of data to be passed between parallel subroutines using a dynamic common block.
 - Support for named file I/O in parallel subroutines using the multitasking facility.
 - Ability to determine the amount of CPU time used by a program or a portion of a program by using the CPUTIME service subroutine.
 - Ability to determine the Fortran unit numbers that are available by using the UNTANY and UNTNOFD service subroutines.
- Enhancements to the full screen functions of interactive debug

Release 2, June 1987

Major Changes to the Product

- Support for 31-character symbolic names, which can include the underscore (_) character.
- The ability to detect incompatibilities between separately-compiled program units using an intercompilation analyzer. The ICA compile-time option invokes this analysis during compilation.
- Addition of the NONE keyword for the IMPLICIT statement.
- Enhancement of SDUMP when specified for programs vectorized at LEVEL(2), so that ISNs of vectorized statements and DO-loops appear in the object listing.
- The ability of run-time library error-handling routines to identify vectorized statements when a program interrupt occurs, and the ability under interactive debug to set breakpoints at vectorized statements.
- The ability, using the INQUIRE statement, to report file existence information based on the presence of the file on the storage medium.
- Addition of the OCSTATUS run-time option to control checking of file existence during the processing of OPEN statements, and to control whether files are deleted from their storage media.
- Under MVS, addition of a data set and an optional DD statement to be used during processing for loading library modules and interactive debug.
- Under VM, the option of creating during installation a single VSF2LINK TXTLIB for use in link mode in place of VSF2LINK and VSF2FORT.
- The ability to sample CPU use within a program unit using interactive debug. The new commands LISTSAMP and ANNOTATE have been added to support this function.

- The ability to automatically allocate data sets for viewing in the interactive debug source window.

Release 1.1, September 1986

Major Changes to the Product

- Addition of vector directives, including compile-time option (DIRECTIVE) and installation-time option (IGNORE)
- Addition of NOIOINIT run-time option
- Addition of support for VM/XA System Facility Release 2.0 (5664-169) operating system

Part 1. Introduction

Chapter 1. Overview of VS FORTRAN Version 2

Compiler	3
Run-Time Library	3
Interactive Debug	3

The VS FORTRAN Version 2 language is best suited to applications that involve mathematical computations and other manipulation of arithmetic data. The language consists of a set of characters, conventions, and rules that are used to convey information to the compiler. The basis of the language is a *statement* containing combinations of names, operators, constants, and words (keywords) whose meaning is predefined to the compiler. For complete information, see *VS FORTRAN Version 2 Language and Library Reference*.

The VS FORTRAN Version 2 product consists of a compiler, a run-time library of subprograms, and an interactive debugging facility.

Compiler

The compiler analyzes the source program statements and translates them into a machine language program called the object file. The object file can be combined with library subprograms to form a program that you can then run. When the compiler detects errors in the source program, it produces appropriate diagnostic messages.

The compiler operates under the control of an operating system that provides it with input, output, and other services. Object files generated by the compiler also operate under control of an operating system and depend on it for similar services. Information on how to compile your source programs is contained in this book.

Run-Time Library

The run-time library contains subprograms that can be combined with compiled source code.

You can compile and add your own subprograms to furnish any commonly used code sequences. These subprograms must reside in a private data set called at load or link-edit time. For complete information on the library, see *VS FORTRAN Version 2 Language and Library Reference*.

Interactive Debug

Interactive debug is a flexible and efficient tool that assists you in monitoring your VS FORTRAN programs as they run.

Interactive debug allows you to:

- Start, suspend, and continue program processing
- Examine, change, and display values of variables
- Gather and display program performance information
- Trace program transfers
- Control the action taken for run-time errors

- Save output in a file.

For information on how to use interactive debug, see *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Part 2. Compiling and Running Your Program

Chapter 2. Compiling Your Program

The compiler translates Fortran source statements into object code and creates an object module for processing.

This chapter explains how to compile your programs under CMS, MVS batch, and MVS with TSO.

If you are a CMS user, begin with the section that immediately follows.

If you are an MVS batch user, skip to page 11.

If you are a TSO user, skip to page 18.

Note: For information about restrictions on compiling your programs on one operating system for use on another, see “Operating System Support” on page xiv.

Compiling Your Program under CMS

The following sections discuss:

- Requesting compilation
- Specifying compile-time options
- Using INCLUDE files
- Compiler output.

Requesting Compilation

You cannot compile your programs in the CMS/DOS environment. If you have been running other programs in CMS/DOS mode, you must issue the command

```
SET DOS OFF
```

before attempting to compile your Fortran programs.

To compile a source program on disk, specify FORTVS2 followed by the filename of your program and, optionally, the filetype and filemode. For example:

```
FORTVS2 MYPROG
FORTVS2 MYPROG FORTRAN A
FORTVS2 ABC PROG1 *
FORTVS2 ABC PROG1 B
```

If you omit the filetype, the default is FORTRAN. If you omit the filemode or specify * for the filemode, VS FORTRAN refers to the first file name and file type that is found on any disk through the standard search from A-Z disks.

Note: For a source program on disk, a FILEDEF command is not required. However, if you issue a FILEDEF, and the filenames, filetypes, or filemodes specified on FORTVS2 and the FILEDEF command do not agree, those on FORTVS2 override those on FILEDEF.

Specifying Compile-Time Options

You can specify most compile-time options on either the FORTVS2 command or the @PROCESS statement. Options on the @PROCESS statement override the options specified when the compiler (FORTVS2) is invoked. However, if conflicting options exist between the options on FORTVS2 and another source, that is, a default option or @PROCESS option, the option assumed is that shown in

Figure 10 on page 40. For more information on the @PROCESS statement, refer to the *VS FORTRAN Version 2 Language and Library Reference*.

To specify options on FORTVS2, specify them within parentheses after the file identifier. For example:

```
FORTVS2 MYPROG (FREE FLAG(E) MAP)
```

Note: The final parenthesis, shown in the example above, is optional.

Listing Dispositions: The option for specifying the disposition of the listing file can be specified only on the FORTVS2 command. For example:

```
FORTVS2 MYPROG (PRINT)
```

If a value for this option is not specified, the default is DISK. The option values are:

DISK

The compiler places a copy of your LISTING file on a disk.

Abbreviation: DI

NOPRINT

No LISTING file is produced, and any existing LISTING file with the same name is erased.

Abbreviation: NOPRI

PRINT

The compiler prints your LISTING file on the spooled virtual printer.

Abbreviation: PRI

Using Source Files on Tape, on Punched Cards, or in Your Virtual Reader

If you have a source file on tape or punched cards, you must issue a FILEDEF command whose ddname is FORTRAN, and which specifies the appropriate device type. For example, for a source file on tape, issue the following FILEDEF:

```
FILEDEF FORTRAN TAPx (options
```

where x is a number from 0 through 7 that corresponds to virtual tape units 180 through 187 and from 8 through F that corresponds to tape units 288 through 28F, and options are the record format, the logical record length, and the block size.

To use a source file from your virtual reader, issue the following FILEDEF:

```
FILEDEF FORTRAN READER (RECFM F LRECL 80 BLKSIZE 80
```

Note: After a source file is compiled from the virtual reader, it is erased from the reader.

To invoke the compiler using the READER or TAPn as input, issue:

```
FORTVS2 dummy (options
```

where dummy is the filename for listing or object output (a filename is *required*), and options are the desired compile-time options.

For information on invoking the VS FORTRAN compiler from an assembler program see “Requesting Compilation from an Assembler Program” on page 473.

Loading from a DCSS

Multiple DCSS (Discontiguous Shared Segments) can be defined for the VS FORTRAN compiler, with one or more residing at an address above the 16mb line, others below the line.

If the CMS virtual machine is defined as a 370-mode machine, the compiler is loaded from the eligible DCSS at the lowest available storage address below the line that is outside the size of the virtual machine. A DCSS that is located at an address within the 370 virtual machine size cannot be used.

If the CMS virtual machine is not defined as a 370-mode machine, the compiler is loaded from the eligible DCSS at the most suitable storage address. A DCSS that is above the line and outside the size of the virtual machine is preferred over one that is within the size of the virtual machine or below the line. However, if a DCSS above the line is not available, one that is within the size of the virtual machine or below the line will be used.

If no eligible DCSS is found within the addressing range of the machine, an appropriate message is written to the user's console, and the compilation is terminated.

Using INCLUDE Files

INCLUDE files may reside on any accessed minidisk.

If your program contains an INCLUDE directive of the form:

```
INCLUDE char-constant
```

where *char-constant* is a CMS file identifier and, optionally, the name of a MACLIB member, you need not have a FILEDEF command in effect for the INCLUDE file. (In the case of a library member, you need not have a FILEDEF or GLOBAL command for the library). The record formats allowed are the same as those that are allowed for any VS FORTRAN source file.

If your program contains an INCLUDE directive of the form:

```
INCLUDE (member-name)
```

where *member-name* is the name of a MACLIB member, you need to take **one** of the following steps:

- Specify the macro library in a GLOBAL statement:

```
GLOBAL MACLIB filename ...
```

or:

- Define SYSLIB for use by the compiler:

```
FILEDEF SYSLIB DISK filename MACLIB A (PERM
```

For information on the INCLUDE directive and the conditional INCLUDE statements, see *VS FORTRAN Version 2 Language and Library Reference*. For information on how to create MACLIB members, see the CMS User's Guide for your operating system.

Compiler Output

Depending on your site's compile-time defaults and the options you select in your FORTVS2 command, you may get one or more of the following files as output placed in your mini-disk storage:

- TEXT file
- Compiler output listing
- Program information file
- Intercompilation analysis file.

TEXT File

The TEXT file contains the object code the compiler created from your source program.

If the OBJECT compile-time option is specified, the file is written to your disk with the filename of your source program and a filetype of TEXT. For example, the text for MYPROG is MYPROG TEXT.

You may want to direct the compiler object code to a file other than MYPROG TEXT. If so, you can use a FILEDEF command with a ddname of TEXT that specifies where you want to place the object code. To put an object file into MY FILE2 A, for example, issue the following FILEDEF:

```
FILEDEF TEXT DISK MY FILE2 A
```

If the DECK compile-time option is specified, the object code goes to the virtual punch. If you want to direct the object code to a different file, use a FILEDEF command with a ddname of SYSPUNCH that specifies where you want to place the object code. To put an object file into MY FILE3 A, for example, issue the following FILEDEF:

```
FILEDEF SYSPUNCH DISK MY FILE3 A
```

Compiler Output Listing

The LISTING file contains the compiler output listing, which may include a source program listing, an object module listing, and other listings, depending on the options in effect. For more information on the contents of the compiler output listing, see "Using the Compiler Output Listing" on page 41 and "Object Module Listing—LIST Option" on page 127.

The LISTING file has the filename of your source program, and the filetype LISTING.

You can display the LISTING file at your terminal, using an editor. Or, you can print a copy of the LISTING file by means of your virtual printer, using the PRINT command:

```
PRINT MYPROG LISTING
```

Note: A LISTING file containing double-byte characters can be displayed or printed only on a device with double-byte processing capability.

You may want to direct the compiler output listing to a file other than MYPROG LISTING. If so, you can use a FILEDEF command with a ddname of LISTING that specifies where you want the listing to be placed.

To put a listing into MY FILE A, for example, issue the following FILEDEF:

```
FILEDEF LISTING DISK MY FILE A
```

Program Information File

The program information file is produced if VECTOR(IVA) is in effect. This file is required by interactive debug for the interactive vectorization aid functions. For information about vector tuning, see *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

The program information file has the filename of your source program and the filetype PIF. For example, the file for MYPROG is MYPROG PIF. It also has the default ddname VSF2PIF.

To direct the program information file to a file other than MYPROG PIF, use a FILEDEF command with a ddname of VSF2PIF. For example:

```
FILEDEF VSF2PIF DISK MY FILE4 A
```

If you accept the default naming conventions for the file, Interactive Debug is able to identify the file without user intervention; otherwise, you must explicitly define the file when you invoke interactive debug.

Refer to *VS FORTRAN Version 2 Interactive Debug Guide and Reference* for more information on using the program information file.

Intercompilation Analysis File

The intercompilation analysis file is produced if you specify the UPDATE suboption of the ICA compile-time option. The report is placed in the compiler output LISTING file with other requested listings. For information on intercompilation analysis, see “Intercompilation Analysis” on page 380.

Compiling Your Program under MVS

Under MVS, you process a Fortran program by submitting batch *jobs* to the operating system. A *job* consists of one or more of the following *job steps*:

1. Compiling your program
2. Link-editing the compiled program
3. Running the program (GO step).

This chapter discusses only the compile step. Link-editing and running your program are discussed in Chapter 4, “Running Your Program” on page 61.

The following sections discuss:

- Coding and processing jobs
- Requesting compilation
- Specifying compile-time options
- Defining compiler data sets
- Using INCLUDE files
- Compiler output.

Coding and Processing Jobs

You define the job requirements to the operating system through *job control statements*. Job control statements provide a communication link between your program and the operating system. The statements define a job, a job step within a job, and data sets required by the job.

Some of the job control statements most often used are the JOB, PROC, EXEC, and DD statements. The following sections give an overview of these statements. For complete descriptions of these and other job control statements, see one of the following system publications:

OS/VS2 MVS JCL Reference

MVS/XA JCL Reference

MVS/ESA JCL Reference.

JOB Statement—Identifying a Job

The JOB statement begins each MVS job you submit to the system:

Syntax

```
//jobname JOB [parameters]
```

jobname

Identifies this job to the system. The jobname must conform to the standards defined in the the JCL manual for your system.

parameters

Give accounting and processing information that is specific to your site.

PROC Statement—Assigning Default Values

The PROC statement must mark the beginning of an in-stream procedure and, optionally, the beginning of a cataloged procedure. On the PROC statement, you can assign default values to symbolic parameters.

Syntax

```
//[name] PROC symbolic-parameter=value[,...]
```

name

Identifies a procedure. *Name* is required for an in-stream procedure and is optional for a cataloged procedure.

symbolic-parameter=value

Identifies the value(s) assigned to a symbolic parameter.

The name should correspond to a member name in the procedure library; for example, VSF2CLG in SYS1.PROCLIB.

You can modify a PROC statement parameter by specifying a change in the EXEC statement that calls the procedure. For more information, see the JCL manual for your system.

EXEC Statement—Executing Job Steps

Use the EXEC job control statement to invoke a program or procedure.

Syntax

```
//[stepname] EXEC [PROC=]procname [,PARM=' option[,option] ... ']  
                [,other parameters]
```

stepname

Identifies this job step.

procname

Is the name of a procedure you want executed. The names of the IBM-supplied cataloged procedures, which are in your system procedure library, are given in Figure 32 on page 84 and Figure 145 on page 441.

option

Is the name of a compile-time option.

other parameters

Specify accounting and processing information specific to your site.

DD Statement—Defining Files

To identify files you may need, you must specify a DD statement:

Syntax

```
//[ddname | procstep.ddname] DD [ data-set-name][other-parameters]
```

ddname

Identifies the data set defined by this DD statement to the compiler, linkage editor, loader, or to your program. The ddnames you can use for VS FORTRAN Version 2 are shown in Figure 3 on page 14, Figure 4 on page 16, and Figure 28 on page 80.

procstep

Identifies the procedure step.

data-set-name

Is the qualified name you've given the data set that contains your file or files.

other parameters

Specify additional information about the data set, such as its location and space allocation. One of these *other parameters* is the SPACE parameter. Note that for sequential I/O, the release (RLSE) subparameter of the SPACE parameter is ignored for any file for which you specify an ENDFILE or a BACKSPACE.

Requesting Compilation Only

The simplest way to request compilation only is to use the IBM-supplied catalog procedure VSF2C, as shown in the sample JCL below.

```
//jobname      JOB
//              EXEC VSF2C
//FORT.SYSIN DD DSN=USERID.MODULE-NAME.FORTRAN,DISP=SHR
/*
//
```

Note that other IBM-supplied cataloged procedures that combine compilation with other job steps are available. For example, VSF2CL compiles and link-edits your program, and VSF2CLG compiles, link-edits, and runs your program. For a list of all the cataloged procedures, see Figure 32 on page 84. The cataloged procedures should be located in your system procedure library.

The cataloged procedures, however, may not give you the programming flexibility you need for your more complex data processing jobs, in which case you may need to specify your own job control statements, or write your own cataloged procedures. For information on how to write your own job control statements or cataloged procedures, see the JCL manual for your operating system.

For information on invoking the VS FORTRAN compiler from an assembler program see “Requesting Compilation from an Assembler Program” on page 473.

Specifying Compile-Time Options

Use the PARM parameter of the EXEC statement to specify compile-time options for your program. The options must be enclosed in single quotes, as shown in the sample JCL below:

```
//jobname      JOB
//              EXEC VSF2C, PARM='LIST,VEC(IVA)'
//FORT.SYSIN DD DSN=MYPROG.MYFILE.FORTRAN,DISP=SHR
/*
//
```

You can specify most compile-time options on either the PARM parameter above or the @PROCESS statement. Options on the @PROCESS statement override those specified on the PARM parameter. However, if conflicting options exist between the options on the PARM parameter and another source, that is, a default option or @PROCESS option, the option assumed is that shown in Figure 10 on page 40. For more information on the @PROCESS statement, refer to the *VS FORTRAN Version 2 Language and Library Reference*.

Defining Compiler Data Sets

Figure 3 lists the required and optional data sets used by the compiler. Many of the data sets used by the compiler are defined in IBM-supplied cataloged procedures. For those that are not, as indicated in Figure 3, you must supply DD statements. (Cataloged procedures are discussed under “Requesting Compilation Only” on page 13.)

Figure 3 (Page 1 of 2). Compiler Data Sets

ddname	Function	Device Types	Device Class	Defined(1)
SYSIN	Reading input source data set (always required)	Direct access Magnetic tape Card reader	Input stream (defined as DD *, DD DSN=data-set-name, or DD DATA)	No

Figure 3 (Page 2 of 2). Compiler Data Sets

ddname	Function	Device Types	Device Class	Defined(1)
SYSPRINT	Writing source, object, and cross reference listings, storage maps, messages (always required)	Printer Magnetic tape Direct access	A SYSSQ SYSDA	Yes
SYSLIB	Reading INCLUDE data sets (required if INCLUDE statements without fully-qualified data set names are specified—see “Using INCLUDE Files” on page 16 for more information)	Direct access	SYSDA	No
SYSLIN(2)	Creating an object module data set as compiler output and linkage editor input (required if OBJECT is specified)	Direct access Magnetic tape Card punch	SYSDA SYSSQ SYSCP	Yes
SYSPUNCH(2)	Punching the object module deck (required if DECK is specified)	Card punch Magnetic tape Direct access	B SYSCP SYSSQ SYSDA	Yes
SYSTEM	Writing error message and compiler statistics (required if TERM or TRMFLG is specified)	Printer Magnetic tape Direct access	A SYSSQ SYSDA	Yes
icafile(3)	Saving intercompilation information (required if ICA suboption USE, UPDATE, or DEF is specified)	Direct access	SYSDA	No
VSF2PIF	Saving information needed by interactive debug for interactive vectorization aid functions (required if VECTOR(IVA) is specified)	Direct access	SYSDA	No

Notes:

1. The **Defined** column indicates whether or not the ddname is defined in cataloged procedures calling the compiler.
2. SYSLIN and SYSPUNCH may not be directed to the same card punch.
3. The ddname for an ICA file is user-specified. A DD statement is required for each ICA file.

DCB Default Values

The DCB subparameters define record characteristics of a data set. Figure 4 on page 16 lists the DCB default values for compiler data set characteristics.

Figure 4. Compiler Data Set DCB Default Values

ddname	LRECL	RECFM	BLKSIZE
SYSIN	80	–	–
SYSPRINT	137	VBA	3429(1)
SYSLIN	80	FB	3200(1)
SYSPUNCH	80	FB	3440(1)
SYSTEM	240	VS	–
icafile	2004	VB	6144
VSF2PIF	2004	VB	6144

Note:

1. These default block size values correspond to the BLKSIZE values specified on the DD statements in the distributed cataloged procedures. As a default, the compiler sets the BLKSIZE to be the longest record length (LRECL).

Naming Conventions for the Program Information File

VSF2PIF is the ddname for the program information file, a file needed by interactive debug for the interactive vectorization aid functions.

If you use the following naming conventions for the file, interactive debug is able to identify the file without user intervention; otherwise, you must explicitly define the file when you invoke interactive debug.

- If the input to the compiler is in a sequential data set named:
userid.module-name.FORTRAN
you should name the program information file:
userid.module-name.PIF
- If the input to the compiler is in a partitioned data set named:
userid.module-name.FORTRAN(member-name)
you should name the program information file:
userid.member-name.PIF

Using INCLUDE Files

If your program contains an INCLUDE directive of the form:

```
INCLUDE char-constant
```

where *char-constant* is a fully-qualified data set name or partitioned data set member, you need not code a DD statement for the INCLUDE file. The data set must be cataloged. The record formats allowed are the same as those that are allowed for any VS FORTRAN source file.

If your program contains an INCLUDE directive of the form:

```
INCLUDE (member-name)
```

where *member-name* is the name of a partitioned data set member, a DD statement with the ddname SYSLIB must be in effect for the partitioned data set. The record formats allowed are fixed blocked and fixed unblocked.

An example of a DD statement is:

```
//FORT.SYSLIB DD DSN=USER.LIB.FORT,DISP=SHR
```

where USER.LIB.FORT is the name of the partitioned data set of which MASKS is a member.

Note: Specify a maximum blocksize of 6000 to 7000 for use with INCLUDE files; larger blocksizes may cause an ABEND during compilation due to insufficient storage.

For additional information on coding rules, see *VS FORTRAN Version 2 Language and Library Reference*.

Compiler Output

The VS FORTRAN Version 2 compiler provides some or all of the following output, depending on the options in effect for your compilation:

- Object file
- Compiler output listing
- Program information file
- Intercompilation analysis file.

Object File

If you specified the OBJECT compile-time option, the object module is directed to the data set defined by the SYSLIN ddname. If you specified the DECK compile-time option, the object module is directed to the data set defined by the SYSPUNCH ddname. If you specified neither the OBJECT nor the DECK compile-time option, no object module is produced.

Compiler Output Listing

The output listing is written to the data set defined by the SYSPRINT ddname. The compiler output listing may include a source program listing, an object module listing, and other listings, depending on the options in effect. For more information on the contents of the compiler output listing, see “Using the Compiler Output Listing” on page 41 and “Object Module Listing—LIST Option” on page 127.

Program Information File

The program information file is produced if VECTOR(IVA) is in effect. This file is required by interactive debug for the interactive vectorization aid functions (see information about vector tuning in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*).

The program information file is directed to the data set defined by the ddname VSF2PIF.

Intercompilation Analysis File

The intercompilation analysis file is produced if you specify the UPDATE suboption of the ICA compile-time option. Do not use 'DISP=MOD' in the DD statement for the intercompilation analysis file. The report is placed in the compiler output LISTING file with other requested listings. For information on intercompilation analysis, see “Intercompilation Analysis” on page 380.

Compiling Your Program under MVS with TSO

Under TSO, you process a Fortran program by submitting commands to the operating system. This chapter discusses the commands you need to compile your program. Commands for link-editing and running your program are discussed in Chapter 4, “Running Your Program” on page 61.

The following sections discuss:

- Allocating compiler data sets
- Requesting compilation
- Specifying compile-time options
- Compiler output.

For additional information on TSO, see the User's Guide and Language Reference for your system.

Allocating Compiler Data Sets

Before compiling, link-editing, or running your program, you must allocate the files you'll need, using the ALLOCATE command. For example, you could allocate the files described below when processing a source program named MYPROG.

For the Source Program as Compiler Input: The following ALLOCATE command tells TSO that the file named MYPROG.FORT is an existing data set (OLD), described in the SYSIN DD statement.

```
ALLOCATE DATASET(MYPROG.FORT) FILE(SYSIN) OLD
```

If you are using the form of the INCLUDE directive that doesn't specify a fully-qualified data set name, a SYSLIB DD statement is required. The following ALLOCATE command tells TSO that the SYSLIB DD statement describes the library USER.LIB.FORT, which contains the library member specified on the INCLUDE directive.

```
ALLOCATE FILE(SYSLIB) DATASET('USER.LIB.FORT') SHR
```

For information on the INCLUDE directive, see “Using INCLUDE Files” on page 16.

For Compiler Output Listings: The following ALLOCATE command tells TSO that the file named MYPROG.LIST is a new file (NEW) described in the SYSPRINT DD statement. The line length is 133 characters; the primary space allocation is 60 lines.

```
ALLOCATE DATASET(MYPROG.LIST) FILE(SYSPRINT) NEW+  
          BLOCK(133) SPACE(60,10)
```

For an Object Deck: The following ALLOCATE command tells TSO that the file named MYPROG.OBJ is a new file (NEW) and is described in the SYSPUNCH DD statement. The record length and block size are both 80 characters.

```
ALLOCATE DATASET(MYPROG.OBJ) FILE(SYSPUNCH) NEW+  
          BLOCK(80) SPACE(120,20)
```

For the Object Module: The following ALLOCATE command tells TSO that the file named MYPROG.OBJ is a new file (NEW) described on the SYSLIN DD statement. The record size (and block size) must be 80 characters. The space you can specify as any size you need.

```
ALLOCATE DATASET(MYPROG.OBJ) FILE(SYSLIN)+
      NEW BLOCK(80) SPACE(100,10)
```

For a Program Information File: The following ALLOCATE command tells TSO that the file named USER1.MYPROG.PIF is a new file described on the VSF2PIF DD statement. The block size is 6144. For information on naming program information files, see “Naming Conventions for the Program Information File” on page 16.

```
ALLOCATE DATASET('USER1.MYPROG.PIF') FILE(VSF2PIF) NEW+
      BLOCK(6144) SPACE(25,5)
```

For an Intercompilation Analysis Data Set: The following ALLOCATE command tells TSO that the file named MYPROG.ICA is a new file described on the DD statement that has the ddname ICADD. (You may specify any ddname on the DD statement.) The block size is 6144. An ALLOCATE command and DD statement are required for each intercompilation analysis file.

```
ALLOCATE DATASET(MYPROG.ICA) FILE(ICADD) NEW+
      BLOCK(6144) SPACE(40,10)
```

For Terminal Output: The following ALLOCATE command tells TSO that the file identified by the asterisk (*) is described on the SYSTERM DD statement. You can then use the terminal to receive error message output. (The listing output is described on the SYSPRINT DD statement.)

```
ALLOCATE DATASET(*) FILE(SYSTERM)
```

For Program Data Sets: The following ALLOCATE command tells TSO that the file identified by USER1.MASS.DATA is available on the FT09F001 data set.

```
ALLOCATE DATASET('USER1.MASS.DATA') FILE(FT09F001)
```

Before you can load a direct data set, you must preformat it.

Requesting Compilation

To request compilation using the default compile-time options, issue the CALL command as follows:

```
CALL 'SYS1.VSF2COMP(FORTVS2)'
```

Specifying Compile-Time Options

To request one or more compile-time options, you may specify them on the CALL command. For example:

```
CALL 'SYS1.VSF2COMP(FORTVS2)' 'FREE,TERM,SOURCE,MAP,LIST,OBJECT'
```

You can specify most compile-time options on either the CALL command above or the @PROCESS statement. Options on the @PROCESS statement override those specified on the CALL command. However, if conflicting options exist between the options on the CALL command and another source, that is, a default option or @PROCESS option, the option assumed is that shown in Figure 10 on page 40.

For more information on the @PROCESS statement, refer to the *VS FORTRAN Version 2 Language and Library Reference*.

Compiler Output

The VS FORTRAN Version 2 compiler provides some or all of the following output, depending on the options in effect for your compilation:

- Object file
- Compiler output listing
- Program information file
- Intercompilation analysis file.

Object File

The data set containing the object module has the name of your source program and the qualifier OBJ. For example, the qualified name for MYPROG is MYPROG.OBJ.

Compiler Output Listing

The data set containing the compiler output listing has the name of your source program, and the qualifier LIST. For example, the qualified name for MYPROG is MYPROG.LIST. The listing may include a source program listing, an object module listing, and other listings, depending on the options in effect. For more information on the contents of the compiler output listing, see “Using the Compiler Output Listing” on page 41 and “Object Module Listing—LIST Option” on page 127.

Program Information File

The program information file is produced if VECTOR(IVA) is in effect. This file is required by interactive debug for the interactive vectorization aid functions (see information on vector tuning in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*).

You need to allocate the data set by using an ALLOCATE command that specifies FILE(VSF2PIF). This allocation will set up the proper ddname for the data set.

Intercompilation Analysis File

The intercompilation analysis file is produced if you specify the UPDATE suboption of the ICA compile-time option. The report is placed in the compiler output LISTING file with other requested listings. For information on intercompilation analysis, see “Intercompilation Analysis” on page 380.

Chapter 3. Using the Compile-Time Options

VS FORTRAN Version 2 compile-time options let you specify details about the input source program and request specific forms of compilation output. This chapter describes the options and the compiler output. For information on modifying compile-time options using the @PROCESS statement, see the *VS FORTRAN Version 2 Language and Library Reference*.

Available Compile-Time Options

Figure 5 lists the compile-time options, their abbreviations, and the IBM-supplied defaults. Your system administrator may have changed these defaults for your installation; there is a column in Figure 5 where you can note any change.

Figure 5 (Page 1 of 2). VS FORTRAN Version 2 Compile-Time Options

Option	Abbreviation	Installation Default
AUTODBL (<i>value</i> <u>NONE</u>)	AD	NONE(1)
CHARLEN (<i>number</i> <u>500</u>)	CL	
CI (<i>number1,number2,...</i>)	None	CI (<i>number1,number2,...</i>)(1)
DBCS <u>NODBCS</u>	None	
DC (* <i>name1,name2,...</i>)	None	NODC(1)
DDIM <u>NODDIM</u>	None	
DECK <u>NODECK</u>	D NOD	
DIRECTIVE (<i>trigger-constant</i>) <u>NODIRECTIVE</u> [(<i>trigger-constant</i>)]	DIR NODIR	NODIRECTIVE(1)
DYNAMIC (* <i>name1, name2,...</i>)	DY	
EC (* <i>name1,name2,...</i>)	None	NOEC(1)
EMODE <u>NOEMODE</u>	None	
FIPS (S F) <u>NOFIPS</u>	None	
FLAG (<u>I</u> W E S)	None	
FREE <u>FIXED</u>	None	
GOSTMT <u>NOGOSTMT</u>	GS NOGS	
HALT (I W E <u>S</u>)	None	
ICA [([USE (<i>name1,name2,...</i>)] [UPDATE (<i>name</i>)] [DEF (<i>nameA,nameB,...</i>)] [MXREF (S <u>L</u>) NOMXREF] [<u>CLEN</u> NOCLEN] [CVAR <u>NOCVAR</u>] [MSG ({ <u>NEW</u> NONE ALL })] [MSGON (<i>number1,number2,...</i>)] [MSGOFF (<i>number1,number2,...</i>)] [RCHECK <u>NORCHECK</u>])] <u>NOICA</u>	UPD MON MOFF	
IL (<u>DIM</u> NODIM)	None	
LANGVL (66 <u>77</u>)	LVL	
LINECOUNT (<i>number</i> <u>60</u>)	LC	

Figure 5 (Page 2 of 2). VS FORTRAN Version 2 Compile-Time Options

Option	Abbreviation	Installation Default
LIST <u>NOLIST</u>	L NOL	
MAP <u>NOMAP</u>	None	
NAME (<i>name</i> <u>MAIN#</u>)	None	
<u>OBJECT</u> NOOBJECT	OBJ NOOBJ	
OPTIMIZE (0 1 2 3) <u>NOOPTIMIZE</u>	OPT NOOPT	
PARALLEL [([REPORT [(<i>optionlist</i>)] <u>NOREPORT</u>] [<u>LANGUAGE</u> NOLANGUAGE] [<u>AUTOMATIC</u> NOAUTOMATIC] [<u>REDUCTION</u> NOREDUCTION] [<u>TRACE</u> NOTRACE] [<u>ANZCALL</u> NOANZCALL])] <u>NOPARALLEL</u>	PAR REP NOREP LANG NOLANG AUTO NOAUTO RED NORED NOPAR	NOPARALLEL(1)
PTRSIZE (4 8)		
RENT <u>NORENT</u>	None	
SAA <u>NOSAA</u>	None	
SC (* <i>name1, name2, ...</i>)	None	SC (*)(1)
<u>SDUMP</u> [(<u>ISN</u> SEQ)] NOSDUMP	SD NOSD	
<u>SOURCE</u> NOSOURCE	S NOS	
<u>SRCFLG</u> NOSRCFLG	SF NOSF	
SXM <u>NOSXM</u>	None	
SYM <u>NOSYM</u>	None	
<u>TERMINAL</u> NOTERMINAL	TERM NOTERM	
TEST <u>NOTEST</u>	None	
<u>TRMFLG</u> NOTRMFLG	TF NOTF	
VECTOR [([REPORT [(<i>optionlist</i>)] <u>NOREPORT</u>] [<u>INTRINSIC</u> NOINTRINSIC] [<u>IVA</u> NOIVA] [<u>REDUCTION</u> NOREDUCTION] [<u>SIZE</u> ({ <u>ANY</u> LOCAL <i>n</i> })] [<u>MODEL</u> (<u>ANY</u> VF2 LOCAL)] [<u>SPRECOPT</u> NOSPRECOPT] [<u>ANZCALL</u> NOANZCALL] [<u>CMPLXOPT</u> NOCMPLXOPT])] <u>NOVECTOR</u>	VEC REP NOREP INT NOINT RED NORED SIZ SOPT NOOPT CPLX NOCPLX NOVEC	
XREF <u>NOXREF</u>	X NOX	

Note:

1. This option cannot be changed at installation time. The value is always the IBM-supplied default.

AUTODBL (*value* | **NONE**)

Provides an automatic means of converting single-precision, floating-point calculations to double precision, and converting double-precision calculations to extended precision. For information on using AUTODBL, see “Using the Automatic Precision Increase Facility—AUTODBL Option” on page 51.

The value can be:

NONE

Indicates no conversion is to be performed.

DBL

Indicates that both single- and double-precision items are to be promoted. Items of REAL*4 and COMPLEX*8 types are converted to REAL*8 and COMPLEX*16. Items of REAL*8 and COMPLEX*16 types are converted to REAL*16 and COMPLEX*32.

DBL4

Indicates that only single-precision items are to be promoted.

DBL8

Indicates that only double-precision items are to be promoted.

DBLPAD

Indicates that single- and double-precision items are to be both promoted and padded. REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 types are promoted. Items of other types are padded if they share storage space with promoted items. The DBLPAD option thus ensures that the storage-sharing relationship that existed prior to conversion is maintained.

Note: No promotion or padding is performed on character data type.

DBLPAD4

Indicates that single-precision items, such as REAL*4 and COMPLEX*8 items, are to be promoted. Items of other types are padded if they share storage space with promoted items.

DBLPAD8

Indicates that double-precision items, such as REAL*8 and COMPLEX*16 items, are to be promoted. Items of other types are padded if they share storage space with promoted items.

value

Indicates that the program is to be converted according to the value of *value*, a five-position field. All five positions must be coded; if a function is not required, the corresponding position must be coded 0.

Each position is coded with a numeric value specifying how a given conversion function is to be performed. The leftmost position (position 1) specifies the promotion function, or whether promotion is to occur and, if so, which items are to be promoted. The second position from the left specifies the padding function, or whether padding is to occur and, if so, where within the program (such as in the common area or in argument lists) padding is to take place. The third, fourth, and fifth positions from the left specify whether padding is to occur for particular types (logical, integer, and real/complex, respectively) within the program entities specified in the second position. The values for each position are shown in Figure 6 on page 24.

Figure 6. Conversion Values for AUTODBL Compile-Time Option

Position 1 — Promotion Function

Value	Meaning
0	No promotion
1	Promote REAL*4 and COMPLEX*8 items only
2	Promote REAL*8 and COMPLEX*16 items only
3	Promote all REAL and COMPLEX items

Position 2 — Padding Function

Value	Meaning
0	No padding
1	Pad all COMMON statement variables and all argument list variables.
2	Pad EQUIVALENCE statement variables made equivalent to promoted variables
3	Pad all COMMON statement variables, pad EQUIVALENCE statement variables made equivalent to promoted variables, and pad all argument list variables
4	Pad EQUIVALENCE statement variables that do not relate to variables in COMMON statements
5	Pad all variables

Position 3 — Padding Logical Variables

Value	Meaning
0	Pad no LOGICAL variables
1	Pad LOGICAL*1 variables only
2	Pad LOGICAL*4 variables only
3	Pad LOGICAL*1 and LOGICAL*4 variables

Position 4 — Padding Integer Variables

Value	Meaning
0	Pad no INTEGER variables
1	Pad INTEGER*2 variables only
2	Pad INTEGER*4 variables only
3	Pad INTEGER*2 and INTEGER*4 variables

Position 5 — Padding Real and Complex Variables

Value	Meaning
0	Pad no REAL or COMPLEX variables
1	Pad REAL*4 and COMPLEX*8 variables
2	Pad REAL*8 and COMPLEX*16 variables
3	Pad REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 variables
4	Pad REAL*16 and COMPLEX*32 variables
5	Pad REAL*4, COMPLEX*8, REAL*16, and COMPLEX*32 variables
6	Pad REAL*8, REAL*16, COMPLEX*16, and COMPLEX*32 variables
7	Pad all REAL and COMPLEX variables

Note that promotion overrides padding. If the first position specifies that promotion is to occur for single precision items, REAL*4 and COMPLEX*8 items are promoted regardless of the padding function specified in position 5.

Figure 7 shows equivalent AUTODBL values.

Figure 7. Equivalent AUTODBL Values

AUTODBL Value	Is Equivalent To
AUTODBL(NONE)	AUTODBL(00000)
AUTODBL(DBL)	AUTODBL(30000)
AUTODBL(DBL4)	AUTODBL(10000)
AUTODBL(DBL8)	AUTODBL(20000)
AUTODBL(DBLPAD)	AUTODBL(33334)
AUTODBL(DBLPAD4)	AUTODBL(13336)
AUTODBL(DBLPAD8)	AUTODBL(23335)

Examples:

AUTODBL(12330)

All REAL*4 variables and arrays are promoted to REAL*8 and all COMPLEX*8 variables and arrays are promoted to COMPLEX*16. Padding is performed for all logical and integer type entities that are equivalenced to promoted variables.

AUTODBL(01001)

No promotion is performed, but padding is performed for all REAL*4 and COMPLEX*8 variables in common blocks and argument lists. This code setting permits a program not requiring double precision accuracy to link with a subprogram compiled with the option AUTODBL(DBL4).

AUTODBL(01337)

No promotion is performed, but padding is performed for all logical, integer, real, and complex variables that are in the common area or are used as subprogram arguments. This code setting permits a nonconverted program to link with a program converted with the option AUTODBL(DBLPAD).

CHARLEN (*number* | **500**)

Specifies the maximum length permitted for any character variable, character array element, or character function, where *number* is any number up to and including 32767. Within a program unit, you cannot specify a length for a character variable, array element, or function greater than the CHARLEN specified.

Note: The value specified for CHARLEN can affect the total size of your program. For more information about CHARLEN and program size, see “Character Manipulations” on page 269.

CI (*number1,number2,...*)

Specifies the identification numbers of the INCLUDE statements to be processed. Any conditional INCLUDE statements specified on the @PROCESS statement will be added to any others in effect. For information on INCLUDE statements, see the *VS FORTRAN Version 2 Language and Library Reference*.

number

Any integer from 1 to 255. *Number1, number2, ...* can be specified in any sequence and can be separated by blanks or commas.

DBCS | NODBCS

Specifies whether the source file may contain double-byte characters. When DBCS is specified, the codes X'0E' and X'0F' are interpreted as the shift-out and shift-in characters, which delimit double-byte characters from EBCDIC characters.

DC (* | *name1,name2,...*)

Defines the names of common blocks that are to be allocated at run time. This option allows the specification of common blocks that can reside in the additional storage space available through the XA environment. This option can be repeated; the lists of names are combined.

The "*" form indicates that all common blocks not named otherwise in the SC or EC option are to be dynamic.

On an @PROCESS statement, multiple names can be supplied as parameters to the DC option or on invocation of the compiler (EXECUTE options).

Compiling Under CMS: Each name specified in the DC option can be up to 31 characters in length if:

- The FORTVS2 command is part of an EXEC written in either EXEC2 or REXX, *or*
- You enter the FORTVS2 command on the command line at your terminal.

If the FORTVS2 command is part of an EXEC written in the EXEC language, only the first 8 characters immediately following the left parenthesis can be passed. No error message is generated if truncation occurs.

No checking is done to see if the names specified are valid names of common blocks. For more information on coding the DC, EC, and SC options, see "Coding Common Options" on page 380.

DDIM | NODDIM

Specifies whether pointee arrays that specify object-time dimensions are to be evaluated at each array element reference (DDIM) or at the time of entry into the subprogram (NODDIM).

For dynamically dimensioned arrays (DDIM), the array declarations can contain any integer variable of length 4 or 2. These variables are not restricted to appearing in a COMMON block or as dummy arguments, but the value of the integer variables must be known when the array element containing it is referenced.

Dynamically dimensioned arrays can be used in a Fortran main program.

DECK | NODECK

Specifies whether the compiler is to write the object module to the data set defined by the ddname SYSPUNCH. DECK causes the object module to be written to the data set.

DIRECTIVE (*trigger-constant*) | NODIRECTIVE [(*trigger-constant*)]

Specifies whether selected comments containing parallel and vector directive statements are to be processed. Refer to "Using Parallel and Vector Directives" on page 353 for more information on parallel and vector directive statements.

This option may be specified only once for each compilation unit. It is not possible to specify this option as a compiler invocation option; it can be specified only on an @PROCESS statement.

trigger-constant

Is a character constant whose value is used to identify directives in comment statements. The *trigger-constant* may contain alphameric characters. All lowercase letters are treated as uppercase.

The optional specification of a *trigger-constant* with the NODIRECTIVE option allows you to disable the processing of parallel and vector directive statements without deleting them from the source program.

DYNAMIC (* | name1,name2,...)

Provides dynamic loading of user subroutines or functions during program execution. Dynamic loading allows these subprograms to be contained in separate modules and to be loaded when they are needed during program execution. This option can be repeated; the lists of names are combined. See “Dynamically Loaded User Subprograms” on page 97 for more information.

EC (* | name1,name2,...)

Defines the names of common blocks that are to be dynamically allocated as extended common blocks. This option allows the specification of common blocks that can reside in the storage space available through the ESA environment.

The "*" form indicates that all common blocks not named otherwise in the SC or DC option are to be extended. The rules and syntax are identical to the DC compile-time option.

EMODE | NOEMODE

Indicates the code that is compiled for a subroutine or function can receive arguments that reside in an extended common block. The EMODE compile-time option is required for the compilation of a subroutine or function program when:

- The subroutine or function program accepts arguments, and
- Any of the parameters to the subroutine or function program can be data that resides in an extended common block.

If the EMODE option is not specified for such a compilation, the routine fails at run time when it is passed an argument that resides in an extended common block.

Note: The EMODE option is automatically in effect whenever a subroutine or function program explicitly references entities in an extended common block. EMODE applies only to subroutines or function programs that accept arguments, and is ignored for other program units.

Be aware that using extended common requires additional run time and the program must run in an ESA/390* environment.

Use EMODE to recompile all program units of an application that uses extended common. Use EMODE to compile all subroutines or functions to be used as a library routine among different applications.

* ESA/390 is a trademark of the International Business Machines Corporation.

FIPS (S | F) | NOFIPS

Specifies whether standard language flagging is to be performed. When FIPS is specified, the standard language flagging level is specified as *subset* or *full*.

Items not defined in ANSI X3.9-1978 are flagged. Flagging is valuable only if you want to write a program that conforms to FORTRAN 77. If you specify `LANGVL(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. FLAG allows you to suppress messages that are below the level desired. For example, if you want to suppress all messages that are warning or informational, specify `FLAG(E)`.

FREE | FIXED

Indicates whether the input source program is in free format or in fixed format. These formats are described in more detail in *VS FORTRAN Version 2 Language and Library Reference*.

GOSTMT | NOGOSTMT

Specifies whether internal statement numbers (for run-time error debugging information) are to be generated for a calling sequence to a subprogram or to the run-time library from the compiler-generated code.

HALT (I | W | E | S)

Causes termination of the compile after any phase if the compiler return code is at or above the specified level.

- I** Selects return code 0 (Informational diagnostic level).
- W** Selects return code 4 (Warning diagnostic level).
- E** Selects return code 8 (Error diagnostic level).
- S** Selects return code 12 (Severe error diagnostic level).

The `HALT(I)` option terminates the compile after the first compiler phase, which is the syntax analysis phase.

If a compile of a source file is terminated because of the `HALT` option, an object file is not produced.

ICA [(
 [USE (*name1, name2, ...*)]
 [UPDATE (*name*)]
 [DEF (*nameA, nameB, ...*)]
 [MXREF (S | L) | NOMXREF]
 [CLEN | NOCLN]
 [CVAR | NOCVAR]
 [MSG ({ NEW | NONE | ALL })]
 [MSGON (*number1, number2, ...*) |
 MSGOFF (*number1, number2, ...*)]
 [RCHECK | NORCHECK]
)]

| NOICA

Specifies whether intercompilation analysis is to be performed, specifies the files containing intercompilation analysis information to be used or updated, and controls output from intercompilation analysis. Specify ICA when you have a group of programs and subprograms that you want to process together and you need to know if there are any conflicting external references, mismatched

commons, and so on. For more information, see “Intercompilation Analysis” on page 380.

USE (*name1*,*name2*,...)

Specifies the names of the ICA files against which a program unit is to be checked. The number, type, and use of the program unit's arguments will be checked for consistency against the arguments in the called subprograms listed in the named ICA files. All of the arguments in the subprograms listed in the ICA files will then be checked for their consistency across calls. This option can be repeated any number of times as long as the total number of files specified in USE, UPDATE and DEF suboptions does not exceed forty (40). For more information about this option, see “Using the USE, UPDATE, and DEF Suboptions” on page 388.

name

The name of an ICA file containing entries describing interfaces between program units. The name must be a sequence of 1 to 8 alphameric characters, beginning with a letter. Commas or blanks can separate a list of names.

If the same *name* is specified for both USE and DEF, the name specified for DEF is used.

UPDATE (*name*)

Specifies the name of the intercompilation analysis file that is to be created or updated.

name

The intercompilation analysis file name must be a sequence of 1 to 8 alphameric characters, beginning with a letter. If the same *name* is specified for both UPDATE and DEF, the name specified for UPDATE is used.

For more information about this option, see “Using the USE, UPDATE, and DEF Suboptions” on page 388.

DEF (*nameA*,*nameB*,...)

Specifies the names of the ICA files against which a program unit is to be checked. The number, type, and use of the program unit's arguments will be checked for consistency against the arguments in the called subprograms listed in the named ICA files. This option can be repeated any number of times as long as the total number of files specified in USE, UPDATE and DEF suboptions does not exceed forty (40). For more information about this option, see “Using the USE, UPDATE, and DEF Suboptions” on page 388.

name

The name of an ICA file containing entries describing interfaces between program units. The name must be a sequence of 1 to 8 alphameric characters, beginning with a letter. Commas or blanks can separate a list of names.

If the same *name* is specified for both USE and DEF, the name specified for DEF is used. If the same *name* is specified for both UPDATE and DEF, the name specified for UPDATE is used.

MXREF (S | L) | NOMXREF

Specifies whether to produce external cross-reference listings. To produce the shortest MXREF listing, use the DEF suboption with MXREF(S).

S Eliminates from the list of references names that make no references and are not referenced by other subprograms.

L Includes in the list of references names that make no references and are not referenced by other subprograms.

CLEN | NOCLEN

Specifies whether to check the length of named common blocks.

CVAR | NOCVAR

Specifies whether usage information for variables in a named common block is to be collected.

MSG ({NEW | NONE | ALL})

Specifies the type of diagnostic messages to be printed.

NEW

Specifies that only messages about the new compilations will appear on the printout.

NONE

Specifies that only messages about deleting entries in an intercompilation analysis file will appear on the printout.

ALL

Specifies that all messages will be printed.

MSGON ([*number1,number2,...*]) | MSGOFF ([*number1,number2,...*])

Specifies the intercompilation analysis messages to be issued. MSGON and MSGOFF are mutually exclusive. With MSGON, the messages you specify are issued; all others are suppressed. With MSGOFF, the messages you specify are suppressed; all others are issued. If you specify neither MSGON nor MSGOFF, all messages are issued.

number

The message number; for example, 61 for message ILX00611.

See “Using the MSGON and MSGOFF Suboptions to Suppress Messages” on page 391 for a list of the message numbers and corresponding message texts.

RCHECK | NORCHECK

Specifies whether to produce a report of recursive subroutine calls or function invocations. If you specify RCHECK and recursion is detected, ICA will print the sequence of program calls that produced the recursion.

IL (DIM | NODIM)

Specifies whether the code for adjustably-dimensioned arrays is to be placed inline, IL(DIM), or called from the library, IL(NODIM). Inline code may result in faster processing, but it does not check for user dimensioning errors. The library call method may result in slower processing, but it does check for such errors. IL(NODIM) may also be specified as NOIL.

LANGLVL (66 | 77)

Specifies the language level in which the input source program is written: the FORTRAN66 language level, or the FORTRAN77 language level. The VS FORTRAN Version 2 manuals describe only the LANTLRVL(77) processing. LANTLRVL66 indicates that no LANTLRVL77 features may be used.

LINECOUNT (*number* | 60)

Specifies the maximum number of lines on each page of the printed source listing. The number may be in the range 7 to 32765. The advantage of using a large LINECOUNT number is that there are fewer page headings to look through if you are using only a terminal. Your output, if printed, will run together from page to page without a break.

LIST | NOLIST

Specifies whether the object module listing is to be written. The LIST option allows you to see the pseudo-assembly language code that is similar to what is actually generated. A full description of this output is given under "Object Module Listing—LIST Option" on page 127.

MAP | NOMAP

Specifies whether a table of source program variable names, named constants, and statement labels and their displacements is to be produced. A complete description of the output is given under "Source Program Map—MAP Option" on page 46.

NAME (*name* | MAIN#)

Specifies the name of the control section (CSECT) generated in the object module. It only applies to main programs. It can only be specified when LANTLRVL(66) is specified.

OBJECT | NOOBJECT

Under CMS, specifies whether the compiler is to write the object module to the file associated with the ddname TEXT.

Under MVS, specifies whether the compiler is to write the object module to the data set associated with the ddname SYSLIN.

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 partial optimization.

OPTIMIZE (2) specifies full optimization with interruption localizing.

OPTIMIZE (3) specifies full optimization without interruption localizing.

If you are debugging your program, it is advisable to use NOOPTIMIZE. To create more efficient code and, therefore, a shorter run time at the price of a longer compile time, use OPTIMIZE(2) or (3). The different levels of optimization are described under Chapter 15, "Using the Optimization Feature" on page 267.

PARALLEL [(
[REPORT [(*optionlist*)] | NOREPORT
[LANGUAGE | NOLANGUAGE]
[AUTOMATIC | NOAUTOMATIC]
[REDUCTION | NOREDUCTION]

```
[TRACE | NOTRACE]  
[ANZCALL | NOANZCALL]
```

```
) ]
```

| NOPARALLEL

Specifies suboptions to the compiler for generating code for DO loops, the PARALLEL DO, PARALLEL SECTIONS, and PARALLEL CALL constructs, and the task management statements. This option also allows you to request reports detailing the portions of your program that are chosen for parallel processing.

The PARALLEL compile-time option is required when you want to generate parallel code for parallel language constructs.

Parallel processing requires that the optimization level in effect be OPT(2) or OPT(3). If the optimization level is not OPT(2) or OPT(3), the compiler upgrades the optimization level to OPT(3) for you. However, the use of DEBUG or invalid Fortran statements downgrades the optimization level. As a result, the optimization level is downgraded to OPT(0) and no parallel processing occurs.

The parallel feature is described in detail in Chapter 17, “Using the Parallel Feature” on page 293.

The PARALLEL compile-time option includes the following suboptions:

REPORT [(*optionlist*)] | NOREPORT

Specifies whether to produce a report showing how parallel code was generated for DO, PARALLEL DO, and PARALLEL SECTIONS statements. REPORT instructs the compiler to display the report on a terminal screen, or provide the information in a printed report.

The format and content of the report varies, depending on the REPORT options specified. In general, the report consists of a listing of source statements. Additional information is supplied in the margins of the listing and in tables at the end of the listing. See “Producing Compiler Reports” on page 344 for examples and a description of the content of the various types of reports you can request.

If both PARALLEL and VECTOR are specified and the REPORT suboption is specified in one or both of the options, a single listing containing the vector and parallel reports is produced.

optionlist

The options specified are one or more of the strings, TERM, LIST, XLIST, SLIST, or STAT, separated by blanks or commas. Figure 8 shows the options, their abbreviations, and describes the output produced by each option.

Figure 8. PARALLEL(REPORT) Options

Option	Abbreviation	Description
TERM	TE	Produces a parallel report at the terminal when TRMFLG is specified.
LIST	LI	Produces a simple format of the parallel report on an output listing.
<u>XLIST</u>	XL	Produces an extended format of the parallel report on an output listing.
SLIST	SL	Produces a parallel report, in a format similar to that produced by the SOURCE compile-time option, on an output listing. Use SLIST in conjunction with the NOSOURCE option when you want to create a single listing for archive purposes.
STAT	ST	Produces a parallel statistics table on an output listing.

You can specify the options in any order; however, in the printed listing, the sections of the report appear in the following order: LIST, XLIST, SLIST, STAT.

The above report types can be requested individually or in any combination.

LANGUAGE | NOLANGUAGE

Specifies whether parallel code is generated for the following parallel language constructs:

PARALLEL DO
PARALLEL SECTIONS
PARALLEL CALL
ORIGINATE statements
TERMINATE statements
SCHEDULE
WAIT FOR statements

If you specify LANGUAGE, parallel code is generated for these constructs; if you specify the NOLANGUAGE suboption, serial code is generated for these constructs. If any of the above constructs appear in your program and you specify the NOLANGUAGE suboption, the resulting program may not be semantically or computationally equivalent to the parallel source program. Parallel code is always generated for parallel task management statements.

AUTOMATIC | NOAUTOMATIC

Specifies whether parallel code is automatically generated for DO loops. When AUTOMATIC is specified, parallel code is generated only when it is determined that the results will be computationally equivalent and potentially result in a more efficient running time.

REDUCTION | NOREDUCTION

Specifies whether reduction functions are to be processed in parallel mode.

Parallel processing can produce different numeric results than serial. Parallel processing cannot be performed in serial order, and some data may have results that are dependent upon the processing order.

TRACE | NOTRACE

Specifies programs that contain PARALLEL SECTIONS or PARALLEL DO loops (whether explicit via the PARALLEL DO statement, or implicit via the AUTOMATIC suboption of the PARALLEL compiler option) are enhanced to call the run-time library to trace the entry to, exit from, and the beginning of each unit processed for PARALLEL SECTIONS and PARALLEL DO loops.

When the NOTRACE suboption is specified, these calls are not generated and these activities are not traced by the Parallel Trace Facility.

ANZCALL | NOANZCALL

Specifies whether an analysis of CALL statements and user function references within DO loops is to be performed when the PARALLEL option is in effect. While greater parallelization might be achieved with ANZCALL, compilation time will increase for eligible loops.

NOPARALLEL

Specifies that the semantics of serial language constructs will be used for the following constructs:

PARALLEL DO
PARALLEL SECTIONS
PARALLEL CALL

If you specify NOPARALLEL, and the source code contains any of the following:

- A PARALLEL CALL
- Subroutine calls or function references within PARALLEL DO or PARALLEL SECTIONS
- Computational dependences

then the results of the serial program can be semantically or computationally different from the parallel program.

Task management statements are not allowed.

PTRSIZE (4|8)

Sets the default length for pointer variables to 4 or 8 bytes. This can be overridden by explicit typing. Be aware that changing the pointer size will alter any common or equivalence association.

RENT | NORENT

Specifies whether the object module generated is to be suitable for use in a shareable area. If you do not want to run your program in a shareable area, specify NORENT. Otherwise, see Chapter 21, "Creating Reentrant Programs" on page 423.

SAA | NOSAA

Specifies whether flagging of language elements that are not part of the Systems Application Architecture (SAA) is to be performed.

SC(* | *name1,name2,...*)

Defines the name of common blocks that are to be compiled as static common blocks. The "*" form indicates that all common blocks not named otherwise in the DC or EC option are to be static. The rules and syntax are identical to the DC compile-time option.

SDUMP [(ISN | SEQ)] | NOSDUMP

Specifies whether symbolic dump information is to be generated. The ISN|SEQ information can be specified with either the usual compiler-generated internal statement numbers, or the user-supplied sequence numbers in columns 73 through 80 of the statement line.

SDUMP(ISN)

Specifies SDUMP tables be generated using internal statement numbers.

SDUMP(SEQ)

Specifies SDUMP tables be generated using sequence numbers (columns 73-80 of fixed-form source).

Note: INCLUDE statements may make sequence numbers ambiguous because interactive debug takes the first statement it finds when searching the statement table.

Generation of the SDUMP table, as well as the line numbers (inserted along with the debugging hooks by the TEST option) are affected by the SEQ suboption. As a result, null sequence numbers are set to 1.

Make sure the numbers are in a sequence meaningful to you. The alphabetic portion of the sequence field is ignored and only the rightmost numeric field is used. For example, S2X00007 uses the rightmost five characters (00007) for sequencing.

SOURCE | NOSOURCE

Specifies whether the source listing is to be produced. By specifying NOSOURCE, you can decrease the size of your listing. If SRCFLG is specified with NOSOURCE, only the initial line of each source statement in error and its associated error messages are printed on the listings.

SRCFLG | NOSRCFLG

Controls insertion of error messages in the source listing. SRCFLG allows you to view error messages after the initial line of each source statement that caused the error, rather than at the end of the listing. If SRCFLG is specified with NOSOURCE, only the initial line of each statement in error and its associated error message are printed on the listings. The NOSRCFLG option causes error messages to appear at the end of the listing.

SXM | NOSXM

Formats XREF or MAP listing output to a 72-character width. The NOSXM option formats listing output for a printer.

SYM | NOSYM

Invokes the production of SYM cards in the object text file. The SYM cards contain location information for variables within a Fortran program. SYM cards are useful to MVS users. For more information about SYM cards, see Appendix C, "Object Module Records" on page 475.

TERMINAL | NOTERMINAL

Specifies whether error messages and compiler diagnostics are to be written on the SYSTERM data set and whether a summary of messages for all the compilations is to be written at the end of the listing.

Specify the NOTERMINAL and NOTRMFLG options if you are running batch jobs on MVS and do not want output sent to a SYSTERM data set. See "Using the Terminal Output Display—TERMINAL and TRMFLG Options" on page 49.

TEST | NOTEST

Specifies whether to override any optimization level above OPTIMIZE(0). TEST is required only for debugging a reentrant program in a shared area (DCSS or LPA) using interactive debug.

The TEST option adds run-time overhead.

TRMFLG | NOTRMFLG

Specifies whether to display the initial line of source statements in error and their associated error messages (formatted for the terminal being used) at the terminal. Specify the NOTERMINAL and NOTRMFLG options if you are running batch jobs on MVS and do not want output to a SYSTERM data set. See “Using the Terminal Output Display—TERMINAL and TRMFLG Options” on page 49.

VECTOR [(

[REPORT [(*optionlist*)] | NOREPORT]

[INTRINSIC | NOINTRINSIC]

[IVA | NOIVA]

[REDUCTION | NOREDUCTION]

[SIZE ({ ANY | LOCAL | *n* })]

[MODEL ({ ANY | VF2 | LOCAL })]

[SPRECOPT | NOSPRECOPT]

[ANZCALL | NOANZCALL]

[CMPLXOPT | NOCMPLXOPT]

)]

| NOVECTOR

Specifies whether to invoke the vectorization process, which produces programs that can use the speed of the vector facility. VECTOR instructs the compiler to transform eligible statements in loops into vector instructions. As much of a loop as possible is translated into vector object code and the remainder is translated into scalar object code.

Vectorization requires that the optimization level in effect be OPT(2) or OPT(3). If the optimization level is not OPT(2) or OPT(3), the compiler upgrades the optimization level to OPT(3) for you. However, the use of DEBUG or invalid Fortran statements downgrades the optimization level. As a result, the optimization level may be downgraded to OPT(0) and no vectorization occurs.

The vectorization process is described in detail in Chapter 16, “Using the Vector Feature” on page 277.

The VECTOR compile-time option includes the following suboptions:

REPORT [(*optionlist*)] | NOREPORT

Specifies whether to produce a report of vectorization information.

REPORT instructs the compiler to either display the report on a terminal screen, or provide the information in a printed report.

The format and content of the report varies, depending on the REPORT options specified. In general, the report consists of a listing of source statements. Additional information is supplied in the margins of the listing and in tables at the end of the listing; for example, brackets that indicate loop nesting, flags that identify vectorized and nonvectorized loops, and messages that give more detailed information about the vectorization process.

If both PARALLEL and VECTOR are specified and the REPORT suboption is specified in one or both of the options, a single listing containing the vector and parallel reports is produced. See “Producing Compiler Reports” on page 344 for examples and a description of the content of the various types of reports you can request.

optionlist

Where the options specified are one or more of the strings, TERM, LIST, XLIST, SLIST, or STAT, separated by blanks or commas.

Figure 9 shows the options, their abbreviations, and describes the output produced by each option.

Figure 9. VECTOR(REPORT) Options

Option	Abbreviation	Description
TERM	TE	Produces a vector report at the terminal when TRMFLG is specified.
LIST	LI	Produces a simple format of the vector report on an output listing. This suboption will help you understand how the statements and loops in the pseudo-assembler listing have been reordered.
<u>XLIST</u>	XL	Produces an extended format of the vector report on an output listing. Use XLIST when you want to tune your program to increase vectorization. It will help you understand why statements did or did not vectorize.
SLIST	SL	Produces a vector report, in a format similar to that produced by the SOURCE compile-time option, on an output listing. Use SLIST in conjunction with the VEC(IVA) option when you are trying to tune your program to improve overall performance. This allows you to use the vector tuning tools available in the IAD. Use SLIST in conjunction with the NOSOURCE option when you want to create a single listing for archive purposes.
STAT	ST	Produces a vector statistics table on an output listing. Use STAT when you are tuning a program for performance without the IAD; it will tell you about some of the assumptions made by the compiler that could have affected its economic decisions.

You can specify the options in any order; however, in the printed listing, the sections of the report appear in the following order: LIST, XLIST, SLIST, STAT.

The above reports can be requested individually or in any combination.

INTRINSIC | NOINTRINSIC

Specifies whether out-of-line intrinsic function references are to be vectorized.

The VS FORTRAN Version 2 math library routines (VSF2FORT) have been revised to be more accurate. The results generated by these new routines may be different from the results generated by the old VS FORTRAN Version 1 standard math routines (VSF2MATH).

For scalar out-of-line intrinsic function references in your program, you can choose which math library to use by accessing libraries in the desired order. If the intrinsic function references in your program are vectorized, however, the new VS FORTRAN Version 2 math library routines will always be used.

Therefore, if you wish to always use the old math routines for compatibility of results, you should not allow out-of-line intrinsic function references to vectorize.

If you use the INTRINSIC suboption, out-of-line intrinsic function references in your program are eligible for vectorization. If the new math library routines are used, the scalar results will be the same as the vector results. The new math routines yield the same results in both vector and scalar.

If you use the NOINTRINSIC suboption, out-of-line intrinsic function references in your program are vectorized. The scalar math routines of the library you have accessed first will be used.

IVA | NOIVA

Specifies whether to produce a program information file. This file is required by interactive debug if you use the interactive vectorization aid functions (see information about vector tuning in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*).

For information on the program information file produced under CMS, see “Compiler Output” on page 10. For information on the program information file produced under MVS, see “Defining Compiler Data Sets” on page 14.

REDUCTION | NOREDUCTION

Specifies whether reduction functions are to be vectorized.

The translation of vector operations from scalar to vector code may produce different results. For example, because floating-point addition is not associative, vector code operations may not be performed in scalar processing order. This causes the results to be dependent on the order of accumulation of the floating-point data.

For more information, see *IBM Enterprise Systems Architecture/370 and System/370 Vector Operations*.

SIZE ({ANY | LOCAL | *n*})

Specifies the section size to be used to perform vector operations.

Specifying SIZE(ANY) causes the compiler to generate object code to use the section size of the computer on which the routine is running. The code may be slightly less efficient as that generated by SIZE(LOCAL) or SIZE(*n*) but you can move the program to a computer with a different section size without recompiling.

When you specify SIZE(LOCAL) the compiler uses the specified section size of the computer that compiled the program. If the section size is 0 or -1, the compiler uses the IBM-supplied default, ANY.

SIZE(*n*) allows you to specify an explicit section size. Using LOCAL or *n* produces slightly more efficient object code. However, if the specified section size is different from that of the computer's actual section size the program terminates upon the first processing of the routine compiled with the incompatible section size. The library issues a diagnostic message with a return code of 16.

If the specified section size is invalid—not a power of 2, or less than 8—the compiler issues an informational message and uses the IBM-supplied default, ANY.

MODEL ({ANY | VF2 | LOCAL })

Identifies the level of the ES/9000* vector facility on which the compiled program will be executed. The compiler generates code to use the enhanced features of that level.

When a program has been compiled for a specific level of the vector facility, the VS FORTRAN run-time vector support will allow the program to be executed only on a processor that can support that level of the vector facility.

ANY

Specifies that any level of the vector facility can be used to execute this program. The program will run on any IBM ES/3090* or ES/9000 processor that has the vector facility installed.

VF2

Specifies that the program will be executed on an ES/9000 processor with the basic level of the enhanced vector facility. Code compiled with this option can execute faster than code generated with MODEL(ANY).

LOCAL

Specifies that the program will be compiled for the level of the vector facility installed on the processor where it is compiled.

- If the compiling processor
 - Does not have the vector facility installed,
 - Is an ES/3090 processor
 - Is an ES/9000 processor without an enhanced vector facility

MODEL(LOCAL) is equivalent to MODEL(ANY).

- If the compiling processor is an ES/9000 with the basic level of the enhanced vector facility, MODEL(LOCAL) is equivalent to MODEL(VF2).

SPRECOPT | NOSPRECOPT

Allows single-precision multiply-and-add and multiply-and-subtract sequences to be generated. If SPRECOPT is specified, the vectorization process will look for single-precision multiplications followed by single-precision additions or subtractions, and attempt to perform these sequences with a vector multiply-and-add or multiply-and-subtract instruction. These calculations give double-precision results, and use double-precision values for the intermediate product. Processing is faster and the result is more accurate than for two separate single-precision operations, but you should be aware that the result might be different from what you expected.

ANZCALL | NOANZCALL

Specifies whether an analysis of CALL statements and user function references within DO loops is to be performed when the VECTOR option is

* ES/3090 and ES/9000 are trademarks of the International Business Machines Corporation.

in effect. While greater vectorization might be achieved with ANZCALL, compilation time will increase for eligible loops.

CMPLXOPT | NOCMPLXOPT

Specifies whether vectorization is to optimize loading and storing of single-precision complex (COMPLEX*8) data using long-precision real vector instructions. See “Complex Data” on page 284 for more details.

XREF | NOXREF

Specifies whether a source cross-reference listing is to be produced. For a description of cross-reference output, see “Cross Reference—XREF Option” on page 45.

Conflicting Compile-Time Options

Figure 10 lists conflicting compile-time options, and the options that are assumed when conflicting compile-time options are specified. For information on how to modify compile-time options using the @PROCESS statement, refer to the *VS FORTRAN Version 2 Language and Library Reference*.

<i>Figure 10 (Page 1 of 2). Conflicting Compile-Time Options</i>			
Conflicting Compile-Time Options		Options Assumed	
DBCS	FIPS	DBCS	NOFIPS
DBCS	SAA	DBCS	NOSAA
DC	EC	Note 1	Note 1
DC	SC	Note 1	Note 1
EC	SC	Note 1	Note 1
EC	NOEMODE	EC	EMODE
FIPS	FLAG=I	FIPS	FLAG=I
FIPS	SAA	Installation default	Installation default
FREE	FIPS	FREE	NOFIPS
FREE	SAA	FREE	NOSAA
FREE	SDUMP(SEQ)	FREE	SDUMP(ISN)
LANGLVL(66)	DBCS	LANGLVL(66)	NODBCS
LANGLVL(66)	FIPS	LANGLVL(66)	NOFIPS
LANGLVL(66)	SAA	LANGLVL(66)	NOSAA
LANGLVL(77)	NAME	LANGLVL(77)	Ignore NAME
NODECK	SYM	NODECK	NOSYM
NOOBJ	SYM	NOOBJ	NOSYM
NOTRMFLG	VEC(REP(TERM...))	NOTRMFLG	VEC(REP(NOTERM...))
NOTRMFLG	PAR(REP(TERM...))	NOTRMFLG	PAR(REP(NOTERM...))
PAR	OPT(0) or OPT(1)	PAR	OPT(3)
PAR	TEST	NOPAR	TEST
PAR(NOREP)	VEC(REP(LIST))	PAR(REP(LIST))	VEC(REP(LIST))
PAR(NOREP)	VEC(REP(XLIST))	PAR(REP(XLIST))	VEC(REP(XLIST))
PAR(NOREP)	VEC(REP(STAT))	PAR(REP(STAT))	VEC(REP(STAT))
PAR(REP(LIST))	VEC(NOREP)	PAR(REP(LIST))	VEC(REP(LIST))
PAR(REP(XLIST))	VEC(NOREP)	PAR(REP(XLIST))	VEC(REP(XLIST))
PAR(REP(STAT))	VEC(NOREP)	PAR(REP(STAT))	VEC(REP(STAT))
PAR(REP)	VEC(REP(SLIST))	PAR(REP(SLIST XLIST))	VEC(REP(SLIST XLIST))
PAR(ANZCALL)	VEC(NOANZCALL)	PAR(NOANZCALL)	VEC(NOANZCALL)
PAR(NOANZCALL)	VEC(ANZCALL)	PAR(ANZCALL)	VEC(ANZCALL)

<i>Figure 10 (Page 2 of 2). Conflicting Compile-Time Options</i>			
Conflicting Compile-Time Options		Options Assumed	
TEST	OPT(1), OPT(2), OPT(3)	TEST	OPT(0)
TEST	NOSDUMP	TEST	SDUMP(ISN)
VEC	OPT(0) or OPT(1)	VEC	OPT(3)
VEC(IVA)	NOSDUMP	VEC(IVA)	SDUMP(ISN)
VEC(IVA)	PAR	VEC(NOIVA)	PAR
Note: 1. The SC, DC, and EC compile-time options use the last option indicated to resolve conflicts between them.			

Using the Compiler Output Listing

For details about the compiler output under your specific operating system, see Chapter 2, “Compiling Your Program” on page 7.

If you use the IBM-supplied default compile-time options, the compiler output listing will have the following sections in the order listed:

- The date of the compilation—plus information about the compiler and this compilation; for example, the release level of the compiler
- 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.

All messages can also be displayed on your terminal output device.

The following sections describe the listing and the compile-time options you can use to modify it. For examples of compiler output generated by the TERMINAL, VECTOR, or ICA compile-time options, see “Using the Terminal Output Display—TERMINAL and TRMFLG Options” on page 49, “Printing Compiler Reports” on page 345, and “Sample Programs Compiled with Intercompilation Analysis” on page 393 respectively.

Compilation Identification

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:

month day, year hour:minute:second

For example:

OCT 1, 1990 14:31:01

The TIME given is the time the job was started. The TIME is shown on a 24-hour clock; that is, 14:31:01 is the equivalent of 2:31:01 PM.

Note: The DATE installation option allows you to print the date as:

year month day

instead of in the format shown above.

The first page of the listing also shows the default and explicit compile-time options in effect for this compilation. Figure 11 is an example of how the options in effect are shown:

```

REQUESTED OPTIONS (EXECUTE):  ICA MAP XREF OPT(3) VECTOR PARALLEL

OPTIONS IN EFFECT:  NOLIST MAP XREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT SDUMP(ISN)
                    NOSXM VECTOR IL(DIM) NOTEST NODC ICA NODIRECTIVE NODBCS NOSAA PARALLEL NOSAVE NOTABS
                    OPT(3) LONGLVL(77) NOFIPS FLAG(I) AUTODBL(NONE) LINECOUNT(60) CHARLEN(500)
                    ICA: MXREF(L) CLEN MSG(NEW)
                    VECTOR: NOIVA INTRINSIC REDUCTION SIZE(ANY) NOREPORT
                    PARALLEL: AUTOMATIC LANGUAGE NOREDUCTION NOREPORT

```

Figure 11. Options-in-Effect Example

Source Program Listing—SOURCE Option

The statements printed in the source program listing are identical to the Fortran statements you submitted in the source program, with some additional information. Figure 12 and Figure 13 on page 44 show examples of the source program listing.

```

1 2
IF DO ISN *.....1.....2.....3.....4.....5.....6....
1      Subroutine ADDOP(func,A,B,C,N,M)
2      Real*4 A(N,M),B(N,M),C(N,M)
      *
3      Integer*4 func,N,M
      *
4      Include (codes)
      *
      * Definition of function codes.
      *
5 +      Integer*4 fadd,fsub,fmul,fdiv
6 +      Parameter(
+      *      fadd = 1
+ 3      *      ,fsub = 2
+      *      ,fmul = 3
+      *      ,fdiv = 4
+      *      )
      *
7      If (func .eq. fadd .or. func .eq. fsub) Then
1 8      If (func .eq. fadd) Then
2 9          sgn = 1.0
2 10         Else
2 11             sgn = -1.0
2 12         Endif
      * Process the matrices
1 13         Do 20 i = 1,N
1 1 14             Do 10 j = 1,M
1 2 15                 A(i,j) = B(i,j) + sgn*C(i,j)
1 2 16 10             Continue
1 1 17 20             Continue
1 18         Else
1 19             Call ErrMsg(' Invalid function specified for ADDOP')
1 20         Endif
21         Return
22         End

```

Figure 12. Source Program Listing Example—with IF, DO, and INCLUDE Flags

The listing format provides additional information on noncommentary source lines. These are interpretive and debugging aids for large or complex programs.

The brief example in Figure 12 illustrates how the source listing flags the IF and DO levels and the INCLUDE code.

- 1** An IF-level counter. The count under the IF heading indicates the level of nesting of block IFs (not logical or arithmetic IFs). The count is incremented for each IF-THEN statement encountered and decremented after each ENDIF statement. The IF-level help locating “open” block-IF statements (those missing an ENDIF), and aid understanding program logic.
- 2** A DO-level counter. The count under the DO heading, at the top of the listing page, indicates the level of DO loop nesting for this statement. The DO-level is defined as the number of DO loops that enclose a source statement as indicated below. (For purposes of counting, the labeled statement that ends the DO loop is considered within the loop.) A blank under this heading indicates the statement is not in a DO loop. Use the DO-level count as an aid in understanding complex programs where DO loops span several listing pages.
- 3** Flagging of source statements that have been read from a source library through the INCLUDE statement. A plus sign (+) to the immediate left of a source statement indicates that the statement was read from an INCLUDE file.

Source Program Listing—SRCFLG Option

The SRCFLG option lets you obtain error messages following the initial line of statements in error. If SRCFLG is specified with NOSOURCE, only the initial lines of the statements in error and their associated error messages will be printed. Figure 13 on page 44 shows an example of an error that occurred at ISN 14 because a variable name is misspelled and IMPLICIT NONE is specified.

```

IF DO  ISN  *.....1.....2.....3.....4.....5.....6....
        1      Real Function Gauss*8 (x,sigma,mu,range_lo,range_hi)
        * This function computes the value of a gaussian distribution
        * considered over a finite range.
        * Input:
        *      X - point to compute gauss(x)
        *      mu,sigma - mean and standard deviation of distribution
        *      Require sigma > 0.
        *      range_lo,range_hi - the range over which the gaussian is
        *      to be normalized. Require range_hi > range_lo.
        * Output:
        *      function value
        2      Implicit NONE
        3      Real*8  x,sigma,mu,range_lo,range_hi
        *      ,t_lo,t_hi,root_2,root_pi,x_sq,total_area
        4      Parameter(
        *      root_2 = 1.4142136
        *      ,root_pi= 1.77245385
        *      )
        5      If (sigma .eq. 0.0d0) Then
1       6      gauss = 0.0d0
1       7      Else
1       8      t_lo = (range_lo - mu)/(root_2*sigma)
1       9      t_hi = (range_hi - mu)/(root_2*sigma)
1      10      x_sq = (x - mu)**2/(2.0*sigma**2)
1      11      If (ABS(x_sq) .gt. 50.0D0) Then
2      12      gauss = 0.0d0
2      13      Else
2      14      guass = EXP(-x_sq)/(root_pi*root_2*sigma)
***ERROR 1828(E)***  THE NAME "GUASS" HAS NOT APPEARED IN AN EXPLICIT TYPE
                     STATEMENT.
2      15      total_area = (ERF(t_hi) - ERF(t_lo))*0.5D0
2      16      gauss = gauss/MAX(total_area,1.0D-06)
2      17      End If
1      18      End If
        19      Return
        20      End

```

Figure 13. Source Program Listing Example—SOURCE and SRCFLG Options

Note: When the program listed in Figure 13 is compiled, the diagnostic messages shown in Figure 16 on page 48 are produced.

Detecting Errors—MAP and XREF Options

The MAP and XREF compile-time options are useful in detecting compile-time and potential run-time errors. XREF is also useful in the debugging of processing errors.

A storage map and cross reference show the use for each variable, statement function, subprogram, or intrinsic function within a program. The cross reference shows the names and statement labels in the source program, together with the internal statement numbers in which they appear.

You can use the storage map and cross reference to cross-check for these common source program problems:

- Are all variables defined as you expected?
- Are variables misspelled?

If you've declared all variables, then check the following:

- Unreferenced variables
- Variables referenced in only one place.

- Are all referenced variables set? (Exceptions include variables in common, parameters, and initialized variables.)
- Are one or more variables unexpectedly equivalenced?
- Are there unreferenced labels? (If there are, you may have entered an incorrect label number.)
- Have you accidentally redefined one of the standard library functions? (For example, through a statement function definition or by using a variable with the same name as a library function.)
- Are the types and lengths of arguments correct across subroutine calls? (You'll need both listings for this.) See "Intercompilation Analysis" on page 380.
- Have you inadvertently altered a variable passed to the main entry of a subroutine? (For example, at a subordinate entry point.)

You can use the SXM compile-time option to improve the readability of the MAP and XREF listing output at a terminal. If you specify SXM, the MAP and XREF output is formatted to 72-character width.

Cross Reference—XREF Option

The XREF option produces cross references for symbols and statement labels used in the source program. Figure 14 shows a cross reference for symbols. The format of the cross reference for labels is essentially the same.

SYMBOL CROSS REFERENCE DICTIONARY

PROGRAM NAME: GAUSS

TAGS:

A-ARRAY	I-INTRINSIC FUNCTION	S-ASSIGNED
C-COMMON	K-NAMED CONSTANT	T-EXPLICITLY TYPED
D-DUMMY ARGUMENT	N-ENTRY	V-INITIAL VALUE
E-EQUIVALENCED	P-PROMOTED	X-EXTERNAL SUBPROGRAM
F-STATEMENT FUNCTION	Q-PADDED	Y-DYNAMIC COMMON
G-GENERIC NAME	R-SUBPROGRAM NAME	Z-EXTENDED COMMON

1 NAME	2 TYPE	3 TAG	4 DECLARED	5 REFERENCED					
ABS		GI		11					
ERF		GI		15	15				
EXP		GI		14					
GAUSS	R*8	RT	1	6	12	16	16		
GUASS	R*4			14					
MAX		GI		16					
MU	R*8	DT	1	3	8	9	10		
RANGE_HI	R*8	DT	1	3	9				
RANGE_LO	R*8	DT	1	3	8				
ROOT_PI	R*8	KT	3	4	14				
ROOT_2	R*8	KT	3	4	8	9	14		
SIGMA	R*8	DT	1	3	5	8	9	10	14
T_HI	R*8	T	3	9	15				
T_LO	R*8	T	3	8	15				
TOTAL_AREA	R*8	T	3	15	16				
X	R*8	DT	1	3	10				
X_SQ	R*8	T	3	10	11	14			

Figure 14. Example of Cross Reference—XREF Option

The columns in the cross reference give you the following information:

- 1** Names are listed in alphabetic order. Names containing double-byte characters are listed in binary value order before EBCDIC names.
- 2** The second column is headed TYPE—it gives the type and (except for character items) length of each name, in the format:

type*length

where the type can be:

C complex
CHAR character (length not displayed)
I integer
L logical
R real or double precision

- 3** The tag for each name shows its usage. The symbols used in the TAG column are defined at the top of the cross reference.
- 4** The DECLARED column gives the internal statement number where the data item is defined.
- 5** The REFERENCED field gives the internal statement number(s) of each statement in the source program in which the data item is referenced. If there are no references within the program, this column contains the word UNREFERENCED.

If OPT(1) or higher is specified for the compilation, the use of a variable within a statement is indicated in the REFERENCED field. The Flag — F for fetched, S for set, or B for both set and fetched — follows each internal statement number.

A listing of variables referenced, but not set, follows the normal variable cross-reference listing.

Source Program Map—MAP Option

Figure 15 shows the storage map.

STORAGE MAP - TAGS DEFINED IN MAP ARE:							
A-ARGUMENT				I-INTRINSIC FUNCTION	S-SET		
C-COMMON VARIABLE				K-NAMED CONSTANT	T-STATEMENT FUNCTION		
E-EQUIVALENCED				N-ENTRY NAME	V-DATA VALUE(S)		
F-REFERENCED				P-PROGRAM NAME	X-SUBPROGRAM		
G-ASSIGNED				R-SUBPROGRAM NAME			

PROGRAM NAME: GAUSS				PROGRAM SIZE: 554				HEX BYTES.			
---------------------	--	--	--	-------------------	--	--	--	------------	--	--	--

1	2	3	4				
NAME	TYPE	TAG	DISPL.	NAME	TYPE	TAG	DISPL.
ABS		I		ERF		I	
EXP		I		GAUSS	R*8	SFR	0001C8
GUASS	R*4	SF	0001F0	MAX		I	
MU	R*8	F	000200	RANGE_HI	R*8	F	000204
RANGE_LO	R*8	F	000208	ROOT_PI	R*8	FK	20888C
ROOT_2	R*8	FK	0001C0	SIGMA	R*8	FA	00020C
T_HI	R*8	SFA	0001D0	T_LO	R*8	SFA	0001D8
TOTAL_AREA	R*8	SF	0001E0	X	R*8	F	000210
X_SQ	R*8	SF	0001E8				

**** NO USER LABELS ****

Figure 15. Example of a Storage Map—MAP Option

- 1** The first column is headed NAME—it shows the name of each item (for example, variable, array, or subprogram) in the program. Note that names containing double-byte characters are listed in binary value order before EBCDIC names.
- 2** The second column is headed TYPE—it gives the type and (except for character items) length of each name, in the format:

$$\text{type} * \text{length}$$
 where the type can be:
C complex
CHAR character (length not displayed)
I integer
L logical
R real or double precision
- 3** The third column is headed TAG—it displays use codes for each name and variable.
- 4** The fourth column is headed DISPL.—it gives the relative address assigned to a name. For unreferenced variables, or for intrinsic functions that are expanded inline, this column contains the letters UNREFD instead of a relative address. For constants that are referenced at compile time, but for which no storage is required to be allocated, this column contains the letters NO STG 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 EQUIVALENCE 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 which is a table of statement labels used in the program. It also gives you the internal statement number (ISN) for the statement in which the label is defined and the address assigned to the label.

Diagnostic Message Listing—FLAG Option

If the severity level of the message is greater than or equal to what you've specified in the FLAG option, and there are errors in your VS FORTRAN Version 2 source program, the compiler detects them and gives you a message. The messages are self-explanatory, making it easy to correct your errors before recompiling your program. Examples of VS FORTRAN Version 2 messages are shown in Figure 16 on page 48, which is produced when the program listed in Figure 13 on page 44 is compiled.

NUMBER	MODULE	LEVEL	ISN	VS FORTRAN ERROR MESSAGES
ILX1828I	DICP	8(E)	14	THE NAME "GUASS" HAS NOT APPEARED IN AN EXPLICIT TYPE STATEMENT.
ILX0023I	CNTL	0(I)	20	COMPILATION ERRORS HAVE CAUSED OPTIMIZATION TO BE DOWNGRADED. FIX ERRORS AND RECOMPILE.

Figure 16. Examples of Compiler Messages—FLAG Option

All VS FORTRAN Version 2 compiler messages are in the following format:

ILXnnnnI mmmm level [isn] message-text

where the areas have the following meanings:

- ILX** Is the message prefix identifying all VS FORTRAN Version 2 compiler messages
- nnnn** Is the unique number identifying this message
- mmmm** Identifies the compiler module issuing the message
- level** Is the *severity level* of the condition flagged by the compiler. Compiler messages are assigned severity levels as follows:
 - 0(I)** Indicates an informational message; it gives you information about the source program and how it was compiled.
The severity level is 0 (zero).
 - 4(W)** Is a warning message; it indicates a possible error or gives information about the program that you should consider more carefully than that given by a level 0 message.
The severity level is 4.
If no messages are produced that exceed this level, you can safely link-edit and run the compiled object module.
 - 8(E)** Is an error message; it indicates a definite error, but the compiler makes a corrective assumption and completes the compilation.
The severity level is 8.
 - 12(S)** Is a serious error message; it indicates an error for which the compiler can make no corrective assumption.
The severity level is 12.
During processing, if and when this statement in the program is reached, an error message that includes the internal statement number of the statement in error is produced, and the program is terminated.
 - 16(U)** Is an abnormal termination message; it indicates an error that stopped the compilation before it could be completed.
The severity level is 16.
- [isn]** Gives the internal statement number of the statement in which the error occurred, if the internal statement number can be determined.
- message-text** Explains the condition that was detected.

Statistics and End-of-Compilation Message

The last entries of the compiler output listing are the statistics and the end-of-Compilation message. The statistics and end-of-Compilation message shown in Figure 17 are from the source program listing in Figure 12 on page 42.

```
*STATISTICS*  SOURCE STATEMENTS: 22, PROGRAM SIZE: 1612 BYTES, PROGRAM NAME: ADDOP, PAGE: 1
*STATISTICS*  NO DIAGNOSTICS GENERATED.

**ADDOP** END OF COMPILATION 1 *****
```

Figure 17. Statistics and End-of-Compilation Message

ADDOP is the program name and 1 is the number identifying this program's sequence in a compilation.

Using the Terminal Output Display—TERMINAL and TRMFLG Options

TERMINAL specifies output to the terminal and produces a summary of messages and statistics, for all the compilations, at the end of the listing. The error message line includes the error number, the name of the compiler module which detected the error, the severity level, the ISN of the statement in error, and the message text. The format and content are shown in Figure 16 on page 48.

TRMFLG specifies output to the terminal and produces the initial lines of source statements in error and their associated error messages. The error message line includes only the severity level and the message text. No summary is produced at the end of the listing. Figure 18 shows output produced with the TRMFLG option specified.

```
VS FORTRAN VERSION 2 ENTERED.  09:29:12

ISN   14:      guass = EXP(-x_sq)/(root_pi*root_2*sigma)
(E) THE NAME "GUASS" HAS NOT APPEARED IN AN EXPLICIT TYPE STATEMENT.
ISN   20:      End
(I) COMPILATION ERRORS HAVE CAUSED OPTIMIZATION TO BE DOWNGRADED.  FIX ERRORS
AND RECOMPILE.

**GAUSS** END OF COMPILATION 1 *****

VS FORTRAN VERSION 2 EXITED.    09:29:13
```

Figure 18. Example of Compile-Time Messages—TRMFLG Option

Note: If you specify NOTERM, you will not see messages—such as those for invalid ICA files—that are generated between compilations, since they would only appear in the summary.

Figure 19 shows the results of different combinations of the TERMINAL|NOTERMINAL and TRMFLG|NOTRMFLG options.

Figure 19. Results of the *TERMINAL|NOTERMINAL* and *TRMFLG|NOTRMFLG* Options

Result	TERM TRMFLG	NOTERM TRMFLG	TERM NOTRMFLG	NOTERM NOTRMFLG
ENTERED and EXITED messages on terminal	X	X	X	—
Full error messages on terminal	—	—	X	—
Source, truncated error messages on terminal	X	X	—	—
Summary of messages and statistics at end of listing	X(1)	—	X(1)	—
Statistics on terminal	—	—	X	—

Note:

1. Not produced for a single compilation.

Note: For output containing double-byte characters in message text, a terminal capable of displaying double-byte characters must be used.

Using the Standard Language Flagger—FIPS Option

The FIPS option helps to 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 at either the full language level or 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.

FIPS warning messages are all in the same format as other diagnostic messages described under “Diagnostic Message Listing—FLAG Option” on page 47.

Using the SAA Flagger—SAA Option

By using the SAA option, you can identify language elements that are not a part of the Systems Application Architecture. If you specify this option, a warning is issued for the language elements that do not conform to Systems Application Architecture. For general information about VS FORTRAN diagnostic messages, see “Diagnostic Message Listing—FLAG Option” on page 47. For information about the Systems Application Architecture interface for Fortran, see *Systems Application Architecture Common Programming Interface: FORTRAN Reference*.

Using the Automatic Precision Increase Facility—AUTODBL Option

The AUTODBL compile-time option provides an automatic means of converting single precision, floating-point calculations to double precision and/or double precision calculations to extended precision. It is designed to be used to convert programs where this extra precision may be of critical importance.

No recoding of source programs is necessary to take advantage of the facility. Conversion is requested by means of the AUTODBL compile-time option at compilation time. The automatic precision increase facility should be considered as a tool for automatic precision conversion, but not as a substitute for specifying the desired precision in the source program.

See “Available Compile-Time Options” on page 21 for the syntax of the AUTODBL option and an explanation of each suboption.

The precision conversion process includes two functions: promotion and padding.

Promotion

Promotion is the process of converting items from one precision to a higher precision; for example, from single precision to double-precision.

You may request either or both of the following promotion conversions:

- Single precision items to be promoted to double precision items, for example, REAL*4 to REAL*8 and COMPLEX*8 to COMPLEX*16.
- Double precision items to be promoted to extended precision items, for example, REAL*8 to REAL*16 and COMPLEX*16 to COMPLEX*32.

Note that single precision items cannot be increased directly to extended precision items, and only real and complex items can be promoted.

Constants, variables and arrays, and intrinsic functions are promoted as follows:

Constants: Single precision real and complex constants are promoted to double precision. Double precision real and complex constants are promoted to extended precision. Logical and integer constants are not affected.

Examples of promoted constants are:

Constant	Promoted Form of Constant
3.7	3.7D0
3.5E2	3.5D2
4.5D2	4.5Q2
(3.2,3.14E0)	(3.2D0,3.14D0)
(3,4)	(3.D0,4.D0)
(3.2D1,4.2D0)	(3.2Q1,4.2Q0)

Variables and Arrays: REAL*4 and COMPLEX*8 variables and arrays are promoted to REAL*8 and COMPLEX*16, respectively. REAL*8 and COMPLEX*16 variables and arrays are promoted to REAL*16 and COMPLEX*32, respectively.

Examples of promoted variables are:

Variable	Promoted Form of Variable
REAL A,B,C	REAL*8 A,B,C
IMPLICIT REAL*8 (S-U)	IMPLICIT REAL*16 (S-U)
COMPLEX*16 Q(10)	COMPLEX*32 Q(10)

Intrinsic Functions: The correct higher precision, Fortran-supplied function is substituted when a program is converted; that is, if an argument to a Fortran-supplied function is promoted, the higher precision Fortran function is substituted.

For example, a reference to SIN causes the DSIN function to be used if promotion from REAL*4 to REAL*8 is invoked; similarly, a reference to DINT causes the QINT function to be used if the promotion from REAL*8 to REAL*16 is invoked. When a substitution of an intrinsic function is made in order to honor a promotion option, the actual name substituted is an alias; in the example above, D#SIN is the name actually substituted. This ensures that, if the source program contains an actual reference to a variable name such as DSIN, no conflict arises as a result of the substitution of the promoted name.

If a valid intrinsic function name is passed as an argument, and if the AUTODBL option is specified, then:

- The promoted function name (if it exists) is passed, if the AUTODBL option specifies *promotion* for the resulting precision mode of the intrinsic function name that is passed. If there is no function name of higher precision corresponding to the original intrinsic function name, the original intrinsic function name is used and an informational message is issued.
- The intrinsic function name is passed without being changed, if the AUTODBL option specifies *padding* only for the given precision mode.

See Figure 20 on page 57 and Figure 21 on page 59 for promotion of single and double precision intrinsic functions with LANGLVL(77) and LANGLVL(66), respectively.

User Subprograms: Previously compiled subprograms must be recompiled to convert them to the correct precision. If a calling program is compiled with the option to promote REAL*4 to REAL*8 (and COMPLEX*8 to COMPLEX*16), and this calling program also references a user-defined function, say FCT, whose precision is also to be increased, then the function FCT must also be compiled with the promote option.

Padding

Padding is the process of increasing the size of nonpromoted items so as to maintain the same storage relationships. Padding helps you to preserve the relationships between promoted and nonpromoted items sharing storage.

Integer and logical items (and nonpromoted real or complex items) are padded if they share storage space with promoted items in order to ensure that the storage-sharing relationship that existed prior to conversion is maintained.

Note: No promotion or padding is performed on character data type.

The major use of the padding option is for programs whose precision does not have to be increased, but which call or reference subprograms with increased precision. The communication between these programs is by argument lists and/or the common area. Therefore, you can pad all argument references and all common variables in the nonpromoted program, and be assured that the proper storage-sharing relationships will be maintained in the promoted program.

Programming Considerations with AUTODBL

This section describes how use of the AUTODBL facility affects program processing.

Common or EQUIVALENCE Data Values

Promotion and padding operations preserve the storage sharing relationships that existed before conversion. However, in storage-sharing items, data values are preserved only for variables having the same length, and for real and complex variables having the same precision.

For example, the following items retain value-sharing relationships:

```
LOGICAL*4 and INTEGER*4 (same lengths)
REAL*4 and COMPLEX*8    (same precision)
```

The following items do not retain value-sharing relationships:

```
INTEGER*2 and INTEGER*4 (different lengths)
REAL*8 and COMPLEX*8    (different precision)
```

Note that the character data type is not affected by the AUTODBL option; it is neither promoted nor padded, but promoted or padded entities of other data types may be equivalenced to character type variables and the inherent value sharing is, therefore, **not** maintained.

Initialization with Character Constants

Be careful when specifying character constants as data initialization values for promoted or padded variables, as subprogram arguments, or in NAMELIST input. For example, literals should be entered into arrays on an element-by-element basis rather than as one continuous string.

Consider the following statements (compiled with LANTLR(66)):

```
DIMENSION A(2), B(2)
DATA A/'ABCDEFGH' /, B(1) / 'IJKL' /, B(2) / 'MNOP' /
```

Array B is initialized correctly, but array A is not because padding takes place at the end of each element; therefore, no spill occurs if array A is padded or promoted. 'ABCDEFGH' initializes A(1) only.

Initialization with Hexadecimal Constants

Care should be exercised when using hexadecimal constants for initialization of promoted or padded entities.

Consider the following example:

```
DIMENSION RAR5(4)
DATA RAR5 /Z4DF1E76B,ZC6F1F04B,ZF46BF2E7,Z6BC9F55D/
A = 1.2345
I = 25
3 PRINT RAR5,A,I
```

This example initializes the array RAR5 with hexadecimal constants so that the contents of the array contain a valid format specification. In this case, the format is (1X,F10.4,2X,I5) and the array RAR5 is used in statement 3 to print the variables A and I.

However, if an AUTODBL option (such as AUTODBL(DBL)) were used for this program, the array RAR5 would be promoted to a REAL*8 array and the initialization performed by the DATA statement would affect only the low-order portion of each element of the array. The high-order portions would, in fact, be initialized with zeros, which are not valid for a format specification.

Therefore, you should not use an AUTODBL option in such a case and expect results similar to those obtained without the AUTODBL option.

If the DATA statement were changed to:

```
DATA RAR5 /Z4DF1E76B40404040,ZC6F1F04B40404040,
X ZF46BF2E740404040,Z6BC9F55D40404040/
```

the program would compile and run correctly for the AUTODBL option given. In this case, the format specification would be:

```
(1X, F10. 4,2X ,I5)
```

Called Subprograms

Fortran main programs and subprograms must be converted so that variables in the common area retain the same relationship, to guarantee correct linkage during processing. The recommended procedure is to compile all such program units with AUTODBL(DBLPAD). If an option other than DBLPAD is selected, be careful if the common area variables in one program unit differ from those in another; common area variables not to be promoted should be padded.

Any non-Fortran external subprogram called by a converted program unit should be recoded to accept padded and promoted arguments.

Mode-Changing Intrinsic Functions

Care should be exercised when using intrinsic functions whose functional types are different from their argument types; for example, the SNGL function expects a REAL*8 argument and returns a REAL*4 result. If the argument to SNGL was a promoted REAL*8 item, the function SNGLQ would be used, but the functional result would still be a REAL*4.

The following example calls for the promotion of all REAL*4 items to REAL*8, and all COMPLEX*8 items to COMPLEX*16. REAL*8 items are not promoted.

```
@PROCESS AUTODBL(DBL4)
REAL*8 D
COMPLEX*8 C
1 A = SNGL(D)
2 C = CMPLX(B,SNGL(D))
```

At statement 1, the function SNGL returns the high-order portion of its REAL*8 argument; that is, returns a REAL*4 result. This functional value is then expanded with zeros and set into the promoted variable A.

At statement 2, the CMPLX intrinsic function is used. This function requires that the modes of its two arguments must be the same (if two arguments are given). In the above example, however, the first argument, B, is promoted to a REAL*8, but the second argument is a REAL*4, because SNGL always returns a REAL*4 result.

Therefore, although this program would compile correctly if the AUTODBL option were not used, a compilation error would result if AUTODBL(DBL4) were specified.

If statement 2 were changed to:

```
2  C = CMPLX(B,A)
```

the program would compile correctly for the AUTODBL option given in the example.

Argument Padding on Arrays

When padding is requested with the second position of the AUTODBL option set as either 1 or 3, then all nonpromoted arguments of the type indicated by positions 3, 4, and 5 are padded. Note that this must include all nonpromoted arrays of the types indicated, because the compiler is not aware of the use of an array name or an array element as an argument until it encounters such a use. In other words, if such an array is used as an argument, all references to that array are calculated on the basis that the array is padded.

Consider the following example:

```
@PROCESS AUTODBL(11030)
      INTEGER I(20), N/3/, L/10/
1     K = I(5)
2     C = FCT(I(N),L)
```

In this case, when statement 1 is encountered and the displacement to the fifth element of the array, I, is calculated, it is not known whether or not the array will be used as an argument. The AUTODBL option calls for the promotion of all REAL*4 and COMPLEX*8 and the padding of all arguments of the integer type. Therefore, the array, I, is padded and the calculation of the displacement for the reference at statement 1 is made in terms of the padded array. Note that the array would be padded even if it did not appear as an argument reference as it does here in statement 2.

CALL DUMP or CALL PDUMP

The AUTODBL option has no effect on the parameter specifying the requested format for the DUMP/PDUMP subroutine. For example, if a CALL DUMP or CALL PDUMP statement requests a dump format of variables of types REAL*4 or COMPLEX*8, output from a converted program is shown in single precision format. Each item is displayed as two single precision numbers rather than as one double precision number.

For variables that are promoted, the first number is *approximately* the value of the stored variable; the second number is meaningless.

For variables that are padded, the first number is *exactly* the value of the variable; the second number is meaningless.

Direct Access Input/Output Processing

When an OPEN statement has been specified (or a DEFINE FILE statement for LONGLVL(66)), any record exceeding the maximum specified record length causes record overflow to occur.

For converted programs, you should check the record size coded in the defining statement to determine if it can handle the increased record lengths. If not sufficient, the size should be increased appropriately.

Asynchronous Input/Output Processing

Extreme care should be exercised in using AUTODBL for programs containing asynchronous input/output statements.

The asynchronous input/output operation transmits the number of bytes as specified by the transmitting or receiving areas. These areas for any given data set must have the same characteristics regarding both promotion and padding; that is, both must be padded or both must be promoted.

Formatted Input/Output Data Sets

The AUTODBL option has **no** effect on the FORMAT statement. Formatted input/output is controlled by the specifications in the FORMAT statement, and does not reflect the increased size and precision of any promoted variable.

Unformatted Input/Output Data Sets

Unformatted input/output data sets which have not been converted are not directly acceptable to converted programs if the I/O list contains promoted variables.

To make an unconverted data set accessible to the converted program, you should code BFALN=F in the DCB parameter at run time. (This can be used only with MVS systems.)

The effect of writing promoted or padded items to a data set with the BFALN=F parameter is to write the items as if they were not promoted or padded; that is, only the most significant portion of the promoted item is written and only the unpadded portion of the padded item is written.

The effect of reading into promoted or padded items from such a data set is the reverse; that is, the unformatted data is read into the most significant portion of the promoted item, and the least significant portion is skipped. For padded items, the unformatted data is read into the nonpadded portion and the padded portion is skipped.

The BFALN parameter should not be used for:

- Programs and data sets having the same conversion characteristics.
- Formatted data sets regardless of the conversion characteristics; the FORMAT statement controls the transmission of data.

Promotion of Single and Double Precision Intrinsic Functions

Figure 20 and Figure 21 on page 59 show the promotion of single- and double-precision intrinsic functions for LANTLRVL(77) and LANTLRVL(66). In the figures, the data types in parentheses are the arguments for the functions.

Figure 20 (Page 1 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANTLRVL(77)

Generic Name	Single Precision Function	Corresponding Double-Precision Function	Corresponding Extended-Precision Function
LOG	ALOG (REAL*4) CLOG (COMPLEX*8)	DLOG (REAL*8) CDLOG (COMPLEX*16)	QLOG (REAL*16) CQLOG (COMPLEX*32)
LOG10	ALOG10 (REAL*4)	DLOG10 (REAL*8)	QLOG10 (REAL*16)
EXP	EXP (REAL*4) CEXP (COMPLEX*8)	DEXP (REAL*8) CDEXP (COMPLEX*16)	QEXP (REAL*16) CQEXP (COMPLEX*32)
SQRT	SQRT (REAL*4) CSQRT (COMPLEX*8)	DSQRT (REAL*8) CDSQRT (COMPLEX*16)	QSQRT (REAL*16) CQSQRT (COMPLEX*32)
SIN	SIN (REAL*4) CSIN (COMPLEX*8)	DSIN (REAL*8) CDSIN (COMPLEX*16)	QSIN (REAL*16) CQSIN (COMPLEX*32)
COS	COS (REAL*4) CCOS (COMPLEX*8)	DCOS (REAL*8) CDCOS (COMPLEX*16)	QCOS (REAL*16) CQCOS (COMPLEX*32)
TAN	TAN (REAL*4)	DTAN (REAL*8)	QTAN (REAL*16)
ATAN2	ATAN2 (REAL*4)	DATAN2 (REAL*8)	QATAN2 (REAL*16)
COTAN	COTAN (REAL*4)	DCOTAN (REAL*8)	QCOTAN (REAL*16)
SINH	SINH (REAL*4)	DSINH (REAL*8)	QSINH (REAL*16)
COSH	COSH (REAL*4)	DCOSH (REAL*8)	QCOSH (REAL*16)
TANH	TANH (REAL*4)	DTANH (REAL*8)	QTANH (REAL*16)
ASIN	ASIN (REAL*4)	DASIN (REAL*8)	QARSIN (REAL*16)
ACOS	ACOS (REAL*4)	DACOS (REAL*8)	QARCOS (REAL*16)
ATAN	ATAN (REAL*4)	DATAN (REAL*8)	QATAN (REAL*16)
ABS	ABS (REAL*4) CABS (COMPLEX*8)	DABS (REAL*8) CDABS (COMPLEX*16)	QABS (REAL*16) CQABS (COMPLEX*32)
ERF	ERF (REAL*4)	DERF (REAL*8)	QERF (REAL*16)
ERFC	ERFC (REAL*4)	DERFC (REAL*8)	QERFC (REAL*16)
GAMMA	GAMMA (REAL*4)	DGAMMA (REAL*8)	Note 1
LGAMMA	ALGAMA (REAL*4)	DLGAMA (REAL*8)	Note 1
INT	INT (REAL*4) Note 2 (COMPLEX*8) IFIX (REAL*4) HFIX (REAL*4)	IDINT (REAL*8) Note 3 (COMPLEX*16) IDINT (REAL*8) Note 4 (REAL*8)	IQINT (REAL*16) IQINT (REAL*16)
	FLOAT (REAL*4) Note 5	DFLOAT (REAL*8)	QFLOAT (REAL*16)
REAL	Note 2 (REAL*4) Note 2 (COMPLEX*8)	SNGL (REAL*8) DREAL (COMPLEX*16)	SNGLQ (REAL*16) QREAL (COMPLEX*32)
DBLE	DBLE (REAL*4) Note 2 (COMPLEX*8)	Note 2 (REAL*8) Note 3 (COMPLEX*16)	DBLEQ (REAL*16)
QEXT	QEXT (REAL*4)	QEXTD (REAL*8)	Note 6 (REAL*16)
CMPLX	CMPLX (REAL*4) Note 2 (COMPLEX*8)	DCMPLX (REAL*8) Note 3 (COMPLEX*16)	QCMPLX (REAL*16)
IMAG	AIMAG (COMPLEX*8)	DIMAG (COMPLEX*16)	QIMAG (COMPLEX*32)

Figure 20 (Page 2 of 2). Promotion of Single and Double Precision Intrinsic Functions for `LANGVL(77)`

Generic Name	Single Precision Function	Corresponding Double-Precision Function	Corresponding Extended-Precision Function
CONJG	CONJG (COMPLEX*8)	DCONJG (COMPLEX*16)	QCONJG (COMPLEX*32)
AINT	AINT (REAL*4)	DINT (REAL*8)	QINT (REAL*16)
ANINT	ANINT (REAL*4)	DNINT (REAL*8)	Note 1
NINT	NINT (REAL*4)	IDNINT (REAL*8)	Note 1
MOD	AMOD (REAL*4)	DMOD (REAL*8)	QMOD (REAL*16)
SIGN	SIGN (REAL*4)	DSIGN (REAL*8)	QSIGN (REAL*16)
DIM	DIM (REAL*4)	DDIM (REAL*8)	QDIM (REAL*16)
—	DPROD (REAL*4)	Note 7 (REAL*8)	
MAX	AMAX1 (REAL*4) AMAX0 (INTEGER*4) MAX1 (REAL*4)	DMAX1 (REAL*8) Note 8 Note 9	QMAX1 (REAL*16)
MIN	AMIN1 (REAL*4) AMIN0 (INTEGER*4) MIN1 (REAL*4)	DMIN1 (REAL*8) Note 10 Note 11	QMIN1 (REAL*16)

Notes:

1. The extended precision equivalent of this function does not exist. In promoting REAL*8 to REAL*16, the double precision function is used. A warning message is issued.
2. There is no specific function name corresponding to this argument value.
3. The corresponding double precision function does not exist by name. In promoting COMPLEX*8 to COMPLEX*16, the single precision function is expanded as though the double precision function existed.
4. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the single precision function is used. A warning message is issued.
5. The argument mode for this function is integer, which is not promotable. Alternate function names are chosen depending upon the required mode of the function result (listed in this figure).
6. The extended precision equivalent of this function does not exist. In promoting REAL*8 to REAL*16, the double precision function is expanded as though the extended precision function existed.
7. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the single precision function is expanded as though the double precision function existed.
8. The argument mode for this function is integer, which is not promotable. In promoting REAL*4 to REAL*8, the functional result is promoted; that is, DFLOAT is used to float the maximum of the integer arguments.
9. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the double precision function IDINT is used to fix the maximum of the REAL*8 arguments.
10. The argument mode for this function is integer, which is not promotable. In promoting REAL*4 to REAL*8, the functional result is promoted; that is, DFLOAT is used to float the minimum of the integer arguments.
11. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the double precision function IDINT is used to fix the minimum of the REAL*8 arguments.

Figure 21 (Page 1 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(66)

Generic Name	Single Precision Function	Corresponding Double-Precision Function	Corresponding Extended-Precision Function
LOG Note 1	ALOG (REAL*4) CLOG (COMPLEX*8)	DLOG (REAL*8) CDLOG (COMPLEX*16)	QLOG (REAL*16) CQLOG (COMPLEX*32)
LOG10 Note 2	ALOG10 (REAL*4)	DLOG10 (REAL*8)	QLOG10 (REAL*16)
EXP	EXP (REAL*4) CEXP (COMPLEX*8)	DEXP (REAL*8) CDEXP (COMPLEX*16)	QEXP (REAL*16) CQEXP (COMPLEX*32)
SQRT	SQRT (REAL*4) CSQRT (COMPLEX*8)	DSQRT (REAL*8) CDSQRT (COMPLEX*16)	QSQRT (REAL*16) CQSQRT (COMPLEX*32)
SIN	SIN (REAL*4) CSIN (COMPLEX*8)	DSIN (REAL*8) CDSIN (COMPLEX*16)	QSIN (REAL*16) CQSIN (COMPLEX*32)
COS	COS (REAL*4) CCOS (COMPLEX*8)	DCOS (REAL*8) CDCOS (COMPLEX*16)	QCOS (REAL*16) CQCOS (COMPLEX*32)
TAN	TAN (REAL*4)	DTAN (REAL*8)	QTAN (REAL*16)
COTAN	COTAN (REAL*4)	DCOTAN (REAL*8)	QCOTAN (REAL*16)
SINH	SINH (REAL*4)	DSINH (REAL*8)	QSINH (REAL*16)
COSH	COSH (REAL*4)	DCOSH (REAL*8)	QCOSH (REAL*16)
TANH	TANH (REAL*4)	DTANH (REAL*8)	QTANH (REAL*16)
ASIN Note 3	ARSIN (REAL*4)	DARSIN (REAL*8)	QARSIN (REAL*16)
ACOS Note 4	ARCOS (REAL*4)	DARCOS (REAL*8)	QARCOS (REAL*16)
ATAN	ATAN (REAL*4)	DATAN (REAL*8)	QATAN (REAL*16)
ATAN2	ATAN2 (REAL*4)	DATAN2 (REAL*8)	QATAN2 (REAL*16)
ABS	ABS (REAL*4) CABS (COMPLEX*8)	DABS (REAL*8) CDABS (COMPLEX*16)	QABS (REAL*16) CQABS (COMPLEX*32)
ERF	ERF (REAL*4)	DERF (REAL*8)	QERF (REAL*16)
ERFC	ERFC (REAL*4)	DERFC (REAL*8)	QERFC (REAL*16)
GAMMA	GAMMA (REAL*4)	DGAMMA (REAL*8)	Note 5
LGAMMA Note 6	ALGAMA (REAL*4)	DLGAMA (REAL*8)	Note 5
INT	INT (REAL*4) IFIX (REAL*4) HFIX (REAL*4)	IDINT (REAL*8) IDINT (REAL*8) Note 7 (REAL*8)	IQINT (REAL*16)
	FLOAT (REAL*4) Note 8	DFLOAT (REAL*8)	QFLOAT (REAL*16)
REAL	REAL (COMPLEX*8)	DREAL (COMPLEX*16)	QREAL (COMPLEX*32)
SNGL	Note 9	SNGL (REAL*8)	SNGLQ (REAL*16)
DBLE	DBLE (REAL*4)	Note 10 (REAL*8)	DBLEQ (REAL*16)
QEXT	QEXT (REAL*4)	QEXTD (REAL*8)	Note 11 (REAL*16)
CMPLX	CMPLX (REAL*4)	DCMPLX (REAL*8)	QCMPLX (REAL*16)
IMAG Note 12	AIMAG (COMPLEX*8)	DIMAG (COMPLEX*16)	QIMAG (COMPLEX*32)
CONJG	CONJG (COMPLEX*8)	DCONJG (COMPLEX*16)	QCONJG (COMPLEX*32)
AINT	AINT (REAL*4)	DINT (REAL*8)	QINT (REAL*16)
MOD	AMOD (REAL*4)	DMOD (REAL*8)	QMOD (REAL*16)

Figure 21 (Page 2 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANTLRVL(66)

Generic Name	Single Precision Function	Corresponding Double-Precision Function	Corresponding Extended-Precision Function
SIGN	SIGN (REAL*4)	DSIGN (REAL*8)	QSIGN (REAL*16)
DIM	DIM (REAL*4)	DDIM (REAL*8)	QDIM (REAL*16)
MAX	AMAX1 (REAL*4) AMAX0 (INTEGER*4) MAX1 (REAL*4)	DMAX1 (REAL*8) Note 13 Note 14	QMAX1 (REAL*16)
MIN	AMIN1 (REAL*4) AMIN0 (INTEGER*4) MIN1 (REAL*4)	DMIN1 (REAL*8) Note 15 Note 16	QMIN1 (REAL*16)

Notes:

1. LOG is also the specific name of the single precision function (corresponding to ALOG).
2. LOG10 is also the specific name of the single precision function (corresponding to ALOG10).
3. ASIN is also the specific name of the single precision function (corresponding to ARSIN).
4. ACOS is also the specific name of the single precision function (corresponding to ARCOS).
5. The extended precision equivalent of this function does not exist. In promoting REAL*8 to REAL*16, the double precision function is used. A warning message is issued.
6. LGAMMA is also the specific name of the single precision function (corresponding to ALGAMA).
7. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the single precision function is used. A warning message is issued.
8. The argument mode for this function is integer, which is not promotable. Alternate function names are chosen depending upon the required mode of the function result (listed in this figure).
9. There is no intrinsic function for LANTLRVL(66) for a REAL*4 argument.
10. There is no specific function name corresponding to this argument value.
11. The extended precision equivalent of this function does not exist. In promoting REAL*8 to REAL*16, the double precision function is expanded as though the extended precision function existed.
12. IMAG is also the specific name of the single precision function (corresponding to AIMAG).
13. The argument mode for this function is integer, which is not promotable. In promoting REAL*4 to REAL*8, the functional result is promoted; that is, DFLOAT is used to float the maximum of the integer arguments.
14. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the double precision function IDINT is used to fix the maximum of the REAL*8 arguments.
15. The argument mode for this function is integer, which is not promotable. In promoting REAL*4 to REAL*8, the functional result is promoted; that is, DFLOAT is used to float the minimum of the integer arguments.
16. The double precision equivalent of this function does not exist. In promoting REAL*4 to REAL*8, the double precision function IDINT is used to fix the minimum of the REAL*8 arguments.

Chapter 4. Running Your Program

Running Your Program under CMS	62
Selecting Link Mode or Load Mode	62
Available Libraries	63
Creating an Executable Program	63
Creating a Temporary Copy of Your Program	64
Creating a Module	65
Creating a Module in a Member of a Library	66
Running Your Program and Specifying Run-Time Options and User Parameters	67
VM/XA and VM/ESA Considerations	68
Program Attributes—AMODE and RMODE	69
Creating New XA-Mode or XC-Mode Programs	69
Overriding the Compile-Time Value for RMODE	70
Overriding the Compile-Time Value for AMODE	71
AMODE/RMODE Compatibility with Other Programs	73
Extended Architecture Hints for Fortran Users	74
Running Parallel Programs under VM/XA or VM/ESA	74
Running Your Program under MVS	74
Selecting Load Mode or Link Mode	75
Available Libraries	75
Specifying Load Mode	76
Specifying Link Mode	77
Link-Editing Your Program	77
Link-Editing Your Program Using the Linkage Editor	78
Link-Editing Your Program Using the Loader	81
Running Your Program and Specifying Run-Time Options and User Parameters	84
Overlaying Programs—System Considerations	85
Migration of VS FORTRAN Load Modules	85
MVS/XA and MVS/ESA Considerations	88
Program Attributes—AMODE and RMODE	88
Specifying MVS/XA or MVS/ESA Linkage Editor and OS Loader Attributes	88
VS FORTRAN and MVS/XA or MVS/ESA Linkage Editor and Loader Interaction	89
Overriding AMODE/RMODE Attributes	90
Using Dynamic Common Above the 16-Megabyte Line	91
XA Hints for Fortran Users	91
Running Your Program under MVS with TSO	91
Selecting Link Mode or Load Mode	92
Specifying Load Mode	92
Specifying Link Mode	93
Link-Editing Your Program—LINK Command	93
Link-Editing and Running Your Program—LOADGO Command	94
Allocating Data Sets with LOADGO	94
Using LOADGO to Specify Loader and Run-Time Options	95
Running Your Program and Specifying Run-Time Options—CALL Command	95
Required Library Modules	95
Using the CALL Command to Run the Load Module	95
Specifying Run-Time Options and User Parameters	96
Using CLISTS	96

CLISTS for Foreground Processing	96
CLISTS for Background Processing	97
Dynamically Loaded User Subprograms	97
Creating Dynamic Modules Under MVS	98
Multiple Names	98
Creating Dynamic Modules Under CMS	99
Loading TEXT or TXTLIB files	100
Execution Considerations	100
Common blocks	101
Multiple Copies of Subprograms	101
Using Dynamically Loaded Modules	101

When you run a program, you may need other files. If your program requires input or output files, see Chapter 7, “Connecting, Disconnecting, and Reconnecting Files for I/O” on page 143. For information about other files, see the section listed below that explains considerations for your operating system.

If you are a CMS user, begin with the section that immediately follows.

If you are an MVS batch user, skip to page 74.

If you are an MVS with TSO user, skip to page 91.

If you are interested in dynamically loading modules, see page 97.

Note: For information about restrictions on compiling and running your programs on different operating systems, see “Operating System Support” on page xiv.

Running Your Program under CMS

You cannot run your programs in the CMS/DOS environment. If you have been running other programs in this mode, you must issue the command:

```
SET DOS OFF
```

before attempting to run your VS FORTRAN Version 2 programs.

Under VM, you can run parallel programs only on VM/XA and VM/ESA. For special considerations you need to follow when running a parallel program, see “Running Parallel Programs under VM/XA or VM/ESA” on page 74.

The following sections discuss:

- Selecting link mode or load mode

- Creating an executable program

- Running your program and specifying run-time options

- Considerations for VM/XA and VM/ESA.

Note: Attempting to run a Fortran main program that uses extended common blocks under CMS when the user's virtual machine is not defined as an XC virtual machine can result in a system abend, and CMS message "DMSITP141T program interrupt X'13' occurred at..." is issued.

Selecting Link Mode or Load Mode

All parallel library routines and all library modules, other than the mathematical routines, can either be included as part of your executable program along with the compiler-generated code, or loaded dynamically when your program is run. Mathematical routines are included as part of your executable program; parallel programs are dynamically loaded. Run-time loading has the advantages of

reducing the time required to create an executable program, and of reducing the auxiliary storage space required for your executable program. Including the library modules with your program has the advantages of improving run-time performance, and of being able to run the program when the libraries are not available on the system.

If you choose to have the necessary service subroutines included within your executable program, you are operating in *link mode*. If, on the other hand, you choose to have the service subroutines loaded when your program is run, you are operating in *load mode*. You make the choice of link mode or load mode by making the appropriate combination of libraries available when you create your executable program from your TEXT files.

If you choose load mode, you can further reduce the time to create your program and the required auxiliary storage space by using the DYNAMIC compiler option to have your own subroutines and functions dynamically loaded.

Available Libraries

Your system programmer must have made the following libraries available to you:

VSF2FORT	The principle text library that contains library modules used for creating a program to operate in link mode, and for creating a program that is to operate in load mode.
VSF2LOAD	The load mode text library; that is, a CMS LOADLIB that contains the library modules to be loaded into virtual storage when your program runs, and which contains the VS FORTRAN Version 2 interactive debug modules.
VSF2LINK	The link mode text library that contains library modules used for creating a program to operate in link mode. Depending on how VS FORTRAN Version 2 is installed at your site, you may be able to use this library as a combined link mode library, or you may have to use it in conjunction with VSF2FORT.
VSF2MATH	An alternate math library that contains math routines from VS FORTRAN Version 1 Release 4.

Your system programmer must tell you which CMS minidisk contains these libraries so you can gain access to this minidisk. In addition, your system programmer may have given these libraries names different from the standard names listed above; the examples below assume that the standard names are used.

Creating an Executable Program

You can use one of the following three methods to create an executable program:

- **Create a temporary copy of your program** within virtual storage by using the LOAD, and possibly INCLUDE, CMS commands. No permanent copy of the executable program is made.
- **Create a module** by using the LOAD, possibly the INCLUDE, and the GENMOD CMS commands. A module is an executable program that is stored as a file with a file type of MODULE on a CMS disk.

- **Create a module in a member of a library** by using the LKED CMS command. This method link-edits an executable program that is stored as a load module in a member of a CMS LOADLIB.

You may select either link mode or load mode for any of the above methods. The following sections show how to use each of these three methods for creating executable programs.

Creating a Temporary Copy of Your Program

Use the LOAD and INCLUDE commands to create a temporary copy of your executable program in virtual storage. Your object code from which the executable program is built may be either in a TEXT file or in a member of a text library. Use the GLOBAL command to access the appropriate VS FORTRAN Version 2 text libraries and your own text libraries.

Specifying Link Mode: If you are running a program in link mode, use one of the following commands:

- If VSF2LINK and VSF2FORT have been installed as separate libraries at your site, use:

```
GLOBAL TXTLIB VSF2LINK VSF2FORT userlib ...
```

- If VS FORTRAN Version 2 has been installed at your site with the combined link mode library, you do not need to specify VSF2FORT in the GLOBAL TXTLIB command. You can use the following:

```
GLOBAL TXTLIB VSF2LINK userlib ...
```

Specifying Load Mode: If you are running a program in load mode, you must use the following command to use the LOAD library:

```
GLOBAL TXTLIB VSF2FORT userlib ...
```

You need to specify *userlib* only if any of your object code (that is, your main program or any of the subprograms that you call) is stored as a member of a text library rather than as a TEXT file.

If you are going to run your program in load mode, issue the following GLOBAL command before running the temporary copy of your executable program:

```
GLOBAL LOADLIB VSF2LOAD userlib ...
```

You need to specify *userlib* only if any of your dynamically loaded subprograms is stored as a member of a load library rather than as a MODULE file.

For your convenience, you may issue the above GLOBAL command before or after issuing the LOAD and INCLUDE commands.

Issuing LOAD and INCLUDE Commands: To create the temporary copy of your executable program in virtual storage, issue one LOAD command, followed optionally by one or more INCLUDE commands as follows:²

```
LOAD myprog ...  
INCLUDE subprog ...
```

² If myprog contains a common area that is initialized by a BLOCKDATA statement, then the BLOCKDATA statement must be within the file myprog or explicitly loaded using a LOAD or INCLUDE command.

The LOAD command and each INCLUDE command may specify the names of TEXT files or of members of your text libraries that are to comprise your executable program in virtual storage. You must specify a name that refers to a main program, unless you are creating a dynamically loadable module. You need not list subprograms if the filenames of any TEXT files or the member names in the text libraries are identical to the names of the subprograms; in this case, these subprograms are included automatically.

When loading the main program, you should not list subprograms which are to be dynamically loaded.

If the user-supplied subprogram name for an EXTERNAL statement is the same as a VS FORTRAN intrinsic function, you need to issue the following command to ensure that your subprogram, not the VS FORTRAN intrinsic function, is invoked.

```
LOAD myprog subprogram1 subprogram2 ...
```

Creating a Module

Use a series of LOAD, INCLUDE, and GENMOD commands to create an executable program that is stored as a file with a file type of MODULE on your CMS disk. First, you must create a temporary copy of your program, by following the instructions given in the above section "Creating a Temporary Copy of Your Program" on page 64. After issuing the LOAD and INCLUDE commands, issue the GENMOD command.

Special Conditions for Using the LOAD Command: If your main program starts at a place other than the first byte of the load module, then you must use the RESET

option to force CMS to point to your main program. For example:

```
LOAD RTN1 RTN2 RBEGIN (RESET RBEGIN ...
```

where RBEGIN is the main program, that is, the routine to get control from CMS. RBEGIN will also be the routine to pass control to the Fortran library to initialize.

If the CSECT name in RBEGIN TEXT is other than the file name, then the CSECT name must be used. For example:

RBEGIN has no Fortran PROGRAM statement so the default name is MAIN. The LOAD command to use would be:

```
LOAD RTN1 RTN2 RBEGIN (RESET MAIN ...
```

RBEGIN has a Fortran PROGRAM statement that specified HERO as the program name. The LOAD command would be:

```
LOAD RTN1 RTN2 RBEGIN (RESET HERO
```

The GENMOD statement would then follow the LOAD statement used.

This action is necessary because the LOAD command finds the first main program and marks it as the main entry point. If you have an assembler routine that has the CSECT name on the END statement and that CSECT appears before the Fortran main program, then the module generated with GENMOD would not work.

Issuing the GENMOD Command: To create a file with a file type of MODULE on your CMS disk, issue the following GENMOD command:

```
GENMOD modname (FROM entry1
```

This command builds a file with a filename of `modname` and a filetype of `MODULE`. This program can be run at any time, if it includes a main program (rather than dynamically loadable subprograms). If you have any external references before your `START` or `CSECT` instructions, you must specify the `FROM` entry1 operand on the `GENMOD` command to load your program properly.

Creating a Module in a Member of a Library

Use the `LKED` command to create—that is, to link-edit—an executable program as a load module in a member of a CMS `LOADLIB`.

Including Files: Your `TEXT` file is input to the `LKED` command. Your program may call a subprogram with object code stored as a separate `TEXT` file or as a member of a text library. If so, the `TEXT` file must contain `INCLUDE` or `LIBRARY` statements that specify the locations of the object code for these subprograms.

For example:

```
FILEDEF SCORE1 DISK FTM TXTLIB A
FILEDEF SCORE2 DISK TC1 TEXT B
LKED GRADE (LIBE GRADE NOTERM
```

where `GRADE TEXT A` contains:

```
INCLUDE SCORE2
INCLUDE SCORE1(SUB1)
INCLUDE SCORE1(SUB2)
INCLUDE SCORE1(SUB3)
NAME CLASS(R)
```

The `INCLUDE` statement has two forms:

```
INCLUDE libdef(memname,...)
INCLUDE textdef
```

The first form causes the members listed as `memname` to be included in the load module from the text library referred to by the `ddname libdef`. The second form causes the `TEXT` file referred to by the `ddname textdef` to be included in the load module.

If the user-supplied subprogram name for an `EXTERNAL` statement is the same as a VS FORTRAN intrinsic function, you need to specify the above `INCLUDE` statements for your main program and all the referred `EXTERNAL` subprograms. This ensures that the subprograms defined by you, not the VS FORTRAN intrinsic functions, are invoked.

The `LIBRARY` statement has the following form:

```
LIBRARY libdef(memname, ...)
```

This causes the library (`libdef`) to be searched for the members (`memname`) if the members' subprograms are not already included in the load module. (The subprograms from members are included in the load module either by the `TEXT` file input to the `LKED` command, or by being specifically included with `INCLUDE` statements).

Prior to issuing the LKED command, you must have issued FILEDEF commands as follows to correspond to the forms of the INCLUDE or LIBRARY statement shown above:

```
FILEDEF libdef DISK libname TXTLIB fm
FILEDEF textdef DISK textname TEXT fm
```

Specifying Load Mode: For a program to be run in load mode, issue the following FILEDEF command before issuing the LKED command:

```
FILEDEF SYSLIB DISK VSF2FORT TXTLIB fm
```

where fm is the filemode of the CMS disk that contains the principal text library VSF2FORT.

Before running a program that was created with the LKED command in load mode, you must issue the following GLOBAL command:

```
GLOBAL LOADLIB VSF2LOAD libname1 libname2 ...
```

where libname1 is the filename of the CMS LOADLIB into which your load module was placed as a member by the LKED command, and libname2 ... are filenames of the CMS LOADLIBs into which your dynamically loadable modules were placed. For your convenience, you may issue the above GLOBAL command before or after issuing the LKED command.

Specifying Link Mode: If the combined VSF2LINK was installed and the program is to be run in link mode, use the following FILEDEF command:

```
FILEDEF SYSLIB DISK VSF2LINK TXTLIB fm
```

where fm is the filemode of the CMS disk that contains the principal text library VSF2LINK.

Note: To use LKED for a program to run in link mode, the combined VSF2LINK must be installed. If it is not installed, the program can run only in load mode.

Issuing the LKED command: After issuing the appropriate FILEDEF commands, issue the LKED command:

```
LKED myprog (LIBE libname NAME memname
```

In this command,

myprog	Is the filename of the TEXT file that contains your object code.
libname	Is the filename of the LOADLIB file into which the resulting load module is to be placed as a member.
memname	Is the name of the member in the LOADLIB file designated by libname, above, into which the resulting load module is to be placed.

Running Your Program and Specifying Run-Time Options and User Parameters

When you run a program, you can run it after issuing the LOAD or LKED commands, or specify that it be called from a library of modules.

If your virtual machine is simulating extended precision (REAL*16 or COMPLEX*32) floating-point instructions (not running in XA mode), you must include the name CMSLIB in the GLOBAL TXTLIB command.

You can use one of the following three methods to run your program and specify run-time options and user parameters. The available run-time options are listed in “Available Run-Time Options” on page 105.

You can run your program in either link mode or load mode.

- You can use the following START command to run a program that was loaded into storage using a LOAD command:

```
START * [option ...] [/user_parameters]
```

- You can use the file name of a MODULE file that was created with a GENMOD command:

```
modname [option ...] [/user_parameters]
```

where modname is the name of your program.

- You can use the following GLOBAL LOADLIB and OSRUN commands to run a program that is stored as a member of a CMS LOADLIB:

```
GLOBAL LOADLIB libname
```

```
OSRUN memname PARM='[option,option... [/user_parameters ] ]'
```

where libname is the name of the load library that contains memname, and memname is the name of the member that contains the module created with the LKED command.

Any substring you supply after the slash (/) in the above commands are recognized as parameters to a subprogram, but not as run-time options. User parameters can be accessed in your Fortran program with the use of the ARGSTR service subroutine. Note that programs invoked from an old-style EXEC will have parameters subjected to CMS 8-character tokenization. See *VS FORTRAN Version 2 Language and Library Reference* for more information.

Note: For information on using the current library with existing VS FORTRAN load modules, see page 484.

VM/XA and VM/ESA Considerations

The extended architecture support described in this section is available under VM/XA System Product with bimodal CMS or VM/ESA. Under VM/XA or VM/ESA, you can create VS FORTRAN programs in either a 370-mode, XA-mode, or XC-mode virtual machine. However, if you want the programs to make use of storage above the 16-megabyte line, you must run them in an XA-mode or XC-mode virtual machine.

Note that in an XA-mode or XC-mode machine, CMS does *not* support the following files:

- Files connected for keyed access
- Files connected for sequential access that refer to VSAM entry-sequenced data sets or VSAM relative record data sets
- Files connected for direct access that refer to VSAM relative record data sets.

The following sections discuss:

- Creating new programs under VM/XA or VM/ESA with VS FORTRAN Version 2
- Compatibility with existing programs compiled under earlier releases of VS FORTRAN and CMS
- Running parallel programs on VM/XA or VM/ESA.

Program Attributes—AMODE and RMODE

Every program that runs under VM/XA or VM/ESA is assigned two new attributes: AMODE (addressing mode) and RMODE (residence mode).

AMODE Is a program attribute that indicates which addressing mode can be supported at a particular entry into a program. *Addressing mode* refers to the length of an address, either 24 bits or 31 bits, used by the processor; indicated by the high-order bit of the program status word (PSW). Generally, the program is also designed to run only in the AMODE specified, although an assembler language program can switch the addressing mode. There are three possible values for AMODE: 24, 31, and ANY.

RMODE Is a program attribute that indicates which residence mode can be supported at a particular entry into a program. *Residence mode* refers to where a program resides in virtual storage in an XA environment above or below 16 megabytes. There are two possible values for RMODE: 24 and ANY.

Creating New XA-Mode or XC-Mode Programs

When you compile your program, it is assigned an AMODE value of ANY, which means your program can run in either 24-bit or 31-bit addressing mode. It is also assigned an RMODE value of ANY, which means your program can reside above or below the 16-megabyte line.

When you create an executable program using LOAD and START commands, the LOAD and GENMOD commands, or the LKED command, you can either accept the compile-time values for AMODE and RMODE or override them, as explained in the following sections.

Programs that reside below the 16-megabyte line can run in either 24-bit or 31-bit addressing mode, and in either link mode or load mode. Programs residing above the line must run in 31-bit addressing mode and in load mode. Figure 22 shows the valid combinations of AMODE and RMODE values.

Figure 22. Valid Combinations of AMODE and RMODE Values

	RMODE=24	RMODE=ANY
AMODE=24	Valid	Invalid
AMODE=31	Valid	Valid
AMODE=ANY	Valid	Valid

If you create a load module to be run in link mode, the module must reside below the 16-megabyte line. This is because the required library routines become part of the load module and several I/O service subroutines in the library must reside below the line in order to run in 24-bit addressing mode. To make use of storage above the line for your program as well as most of the library routines, run your program in load mode.

Obtaining Storage for Dynamic Common Blocks: Whether the storage for a dynamic common block is obtained above or below the 16-megabyte line depends on the addressing mode of your program. The addressing mode in effect upon the invocation of any program unit that refers to a given dynamic common block determines the location of that block. If the program unit is entered in 31-bit addressing mode, the storage for the dynamic common block is obtained above the

16-megabyte line; if it is entered in 24-bit addressing mode, the storage is obtained below the 16-megabyte line.

After the storage is obtained, the dynamic common block remains at the same location until the program has finished running. Therefore, if storage for a given dynamic common block is obtained above the 16-megabyte line, all subsequent program units that refer to that block must run in 31-bit addressing mode.

In order to use the extra storage available with VM/XA or VM/ESA, the load module must run in 31-bit addressing mode. In particular, the module cannot contain subroutines compiled under FORTRAN G1, HX or prior to VS FORTRAN Version 1, Release 2.

The linkage editor limits the size of a load module to 16 megabytes. To overcome this limit, VS FORTRAN Version 2 named common areas can be declared so that they will occupy storage outside of the load module. The storage is obtained dynamically and made available to the object code by the VS FORTRAN Version 2 Library at run time. For more information concerning dynamic common areas, see "Using Blank and Named Common Blocks" on page 378.

Example:

```
@PROCESS DC(CMN1,CMN2)
COMMON /CMN1/XARRAY(1000,1000,1000)
COMMON /CMN2/YARRAY(5000000)
COMMON /CMN3/ZARRAY(100,100,100)
```

Storage for common areas CMN1 and CMN2 is obtained dynamically at run time. The storage for COMMON CMN3 is part of the load module, and takes up part of the 16-megabyte maximum module size.

Overriding the Compile-Time Value for RMODE

To override the RMODE value that VS FORTRAN assigned to your program at compile-time, you can specify:

- The RMODE or ORIGIN option on the LOAD command
- The RMODE option on the GENMOD command
- The RMODE option on the LKED command
- Options on the SET LOADAREA command.

On the RMODE option, you can specify 24 or ANY, as shown in Figure 23.

Figure 23. RMODE Values

RMODE Value	Program Residence
RMODE 24	Below the 16-megabyte line
RMODE ANY	<ul style="list-style-type: none"> • Below the 16-megabyte line in a 370-mode virtual machine • Above the 16-megabyte line in an XA-mode or XC-mode virtual machine (unless the virtual machine has less than 16 megabytes of storage, in which case the program residence is below the 16-megabyte line)

For example, to specify residence mode above the 16-megabyte line, you may code one of the following:

```
LOAD MYPROG ...(RMODE ANY other_options...
```

```
LOAD MYPROG ...(ORIGIN 1030000 other_options...
```

```
GENMOD MYPROG ...(RMODE ANY other_options...
```

```
LKED MYPROG (LIBE libname NAME memname RMODE ANY other_options
```

```
SET LOADAREA RESPECT
```

Note: To run a 370-mode program in an XA or XC environment without having to regenerate the program, specify the RLDSAVE option on the LOAD command. This allows the program to be relocated above the 16-megabyte line, if the proper AMODE and RMODE have been specified. If you use the LOAD and GENMOD commands in a 370-mode machine, you must also specify the RLDSAVE option on the LOAD command in order for the program to reside above the 16-megabyte line.

The RMODE and ORIGIN options are mutually exclusive. The RMODE option on the GENMOD command overrides that of the LOAD command.

On the ORIGIN option, you can specify an address above or below the 16-megabyte line. If you specify an address above the 16-megabyte line, RMODE ANY results; if you specify an address below the 16-megabyte line, RMODE 24 results.

To allow compatibility with existing LOAD processing, you can use the SET LOADAREA command. This command determines the default RMODE in the case where you use LOAD and START to create a program and do not specify the RMODE or ORIGIN option on the LOAD command.

On the SET LOADAREA command, you can specify one of the following:

20000 This causes the LOAD command to start loading at address 20000. It overrides the RMODE value assigned at compile time.

RESPECT In an XA-mode or XC-mode virtual machine, this causes the LOAD command to respect the RMODE assigned at compile time.

In a 370-mode virtual machine, this causes the LOAD command to start loading below 16-megabyte.

Loading begins at the largest contiguous area above or below the 16-megabyte line, depending on the AMODE and RMODE values. This area may not start at 20000 when loading below the 16-megabyte line because of the way CMS organizes storage. This applies for both XA-mode and 370-mode virtual machines.

Overriding the Compile-Time Value for AMODE

You can override the AMODE value that was assigned to your program at compile time by specifying the AMODE option on the LOAD, GENMOD, or LKED command. (For general information about these commands, see “Creating an Executable Program” on page 63.)

Figure 24 on page 72 shows the values you can specify for the AMODE option.

Figure 24. AMODE Values

AMODE Value	Addressing Mode of Program
AMODE 24	24-bit addressing mode
AMODE 31	31-bit addressing mode in an XA-mode virtual machine 24-bit addressing mode in a 370-mode virtual machine
AMODE ANY	31-bit addressing mode in an XA-mode virtual machine 24-bit addressing mode in a 370-mode virtual machine

For example, to specify 31-bit addressing mode, you may code one of the following:

```
LOAD MYPROG ...(AMODE 31 other_options...
```

```
GENMOD MYPROG ...(AMODE 31 other_options...
```

```
LKED MYPROG (LIBE libname NAME memname AMODE 31 other_options
```

The AMODE option of the GENMOD command overrides that of the LOAD command. If you don't specify AMODE on the GENMOD command, the default is determined by the RMODE you specify on the GENMOD command, and what AMODE, if any, you specified on the LOAD command, as shown in Figure 25.

Figure 25. Default Values for AMODE on the GENMOD Command

RMODE Value	AMODE Value Default
RMODE ANY	31
RMODE 24	Value determined by LOAD command

If you don't specify AMODE or RMODE on the GENMOD command, the default is determined by the LOAD command.

If you don't specify the AMODE option on the LOAD or LKED command, the default is determined by the RMODE you specify on the LOAD or LKED command, or the ORIGIN option you specify on the LOAD command, and the value that VS FORTRAN assigns at compile time. The default values are shown in Figure 26 and Figure 27 on page 73.

Figure 26. Default Values for AMODE on the LOAD Command

RMODE or ORIGIN Value	AMODE Value Default
RMODE ANY	31
RMODE 24	Compile-time value assigned by VS FORTRAN
ORIGIN with an address above 16-megabyte	31
ORIGIN with an address below 16-megabyte	Compile-time value assigned by VS FORTRAN
None (RMODE or ORIGIN not specified)	Compile-time value assigned by VS FORTRAN

Figure 27. Default Values for AMODE on the LKED Command

RMODE Value	AMODE Value Default
RMODE ANY	31
RMODE 24	Compile-time value assigned by VS FORTRAN
None (RMODE not specified)	Compile-time value assigned by VS FORTRAN

AMODE/RMODE Compatibility with Other Programs

In order for your program to run in 31-bit addressing mode, all of its program units must be capable of running in 31-bit addressing mode. Therefore, if a particular unit must run in 24-bit addressing mode (for example, if you call a subprogram that was compiled by the FORTRAN H Extended Compiler), you must invoke the main program in 24-bit addressing mode.

Alternatively, you can switch the addressing mode while the program is running by calling user-coded assembler language subroutines. However, if a subprogram must run in 24-bit mode, the entire module containing the subprogram must reside below the 16-megabyte line. In addition, all the data areas, including dynamic common blocks, that the subprogram uses must also reside below the 16-megabyte line.

All executable programs created with releases of VS FORTRAN before Version 1 Release 2 can run in only 24-bit addressing mode and must reside below the 16-megabyte line.

The extent of extended architecture support for programs created under CMS before VM/XA SP Release 1 depends on how you stored the programs:

- Programs stored as text files or text library members that were compiled under VS FORTRAN Version 1 Release 2 or later have assigned to them the values AMODE ANY and RMODE ANY. Therefore, they can run in 24-bit or 31-bit addressing mode and reside above or below the 16-megabyte line.
- Programs stored as module files or LOADLIB members can run in only 24-bit addressing mode and reside below the 16-megabyte line. To take advantage of 31-bit addressing mode or program residence above the 16-megabyte line, you must recreate these programs under VM/XA or VM/ESA.

Shareable Load Modules: Shareable load modules created under VM/SP Release 4 or 5 run in the same addressing mode as their corresponding nonshareable parts. However, they always reside below the 16-megabyte line.

If the nonshareable parts of your program run in 31-bit addressing mode, you can recreate the shareable modules under VM/XA or VM/ESA so that they reside above the 16-megabyte line. To do this, use the LKED command, assigning the shareable modules an RMODE value of ANY (this is the default).

If the nonshareable parts run in 24-bit addressing mode, the shareable modules must reside below the 16-megabyte line. Therefore, if you recreate the shareable modules under VM/XA or VM/ESA, you must assign an RMODE value of 24 or an ORIGIN address below the 16-megabyte line.

For more information on using shareable load modules in a VM/XA environment, see “Special Considerations for VM/XA” on page 435.

Extended Architecture Hints for Fortran Users

Unless you specifically force an AMODE value of 24, do not mix object modules compiled with VS FORTRAN Version 1 Release 2 and later with:

- Object modules compiled with compilers prior to VS FORTRAN Version 1 Release 2
- Assembler code with 24-bit dependencies.

Running Parallel Programs under VM/XA or VM/ESA

You can run parallel programs on VM/XA or VM/ESA, which allows multiple virtual processors within a virtual machine. The directory of your virtual machine specifies the numeric limit of virtual processors you may define. If you need to increase your limit, see your system programmer.

Parallel programs run only in load mode. Follow the standard procedures for using the LOAD and INCLUDE commands as described in “Issuing LOAD and INCLUDE Commands” on page 64, but, in addition, specify the RLDSAVE option on both the LOAD and INCLUDE commands.

Nonrelocatable modules are not supported for parallel programs. Use the GENMOD command to create a relocatable module after you issue the LOAD and INCLUDE commands. You cannot use the START command to begin running a parallel program.

Running Your Program under MVS

Under MVS, you process a Fortran program by submitting batch *jobs* to the operating system. A *job* consists of one or more of the following *job steps*:

1. Compiling your program
2. Link-editing your program
3. Running the program (GO step).

This chapter discusses only the link-edit and run steps. How to compile your program is discussed in “Compiling Your Program under MVS” on page 11. For instructions on coding job control statements, see “Coding and Processing Jobs” on page 12.

The following sections discuss:

- Selecting load mode or link mode
- Link-editing your program using the linkage editor
- Link-editing and running your program using the loader
- Running your program and specifying run-time options
- Cataloging and overlaying programs
- Migrating load modules
- Considerations for MVS/XA and MVS/ESA.

Note: Attempting to run a Fortran main program that uses extended common blocks under MVS/370 or MVS/XA can result in a system abend, and abend code 0C1 or 0E0 and reason code 00000013 are issued.

Selecting Load Mode or Link Mode

All parallel library routines and all library modules, other than the mathematical routines, can either be included as part of your executable program along with the compiler-generated code, or loaded dynamically when your program is run. Mathematical routines are included as part of your executable program; parallel programs are dynamically loaded. Including the library modules in your executable program improves the run-time performance. Run-time loading reduces the time required to create an executable program, and the auxiliary storage space required for your executable program. Run-time loading allows some service subroutines to be placed in the extended link pack area.

If you choose to have the necessary service subroutines included within your executable program, you are operating in *link mode*. If, on the other hand, you choose to have the service subroutines loaded when your program is run, you are operating in *load mode*. You make the choice of link mode or load mode by making the appropriate combination of libraries available to the linkage editor.

If you choose load mode, you can further reduce the time to create your program and the required auxiliary storage space by using the DYNAMIC compiler option to have your own subroutines and functions dynamically loaded.

Available Libraries

Your system programmer must have made the following libraries available to you:

SYS1.VSF2FORT The principle load module library that contains library modules used for creating a program to operate in link mode, and for creating a program that is to operate in load mode.

SYS1.VSF2LOAD The load mode library that contains the library modules to be loaded into virtual storage when your program runs, and which contains the VS FORTRAN Version 2 interactive debug modules.

SYS1.VSF2LINK The link mode library that contains library modules used for creating a program to operate in link mode.

Depending on how VS FORTRAN Version 2 is installed at your site, you may be able to use this library as a combined link mode library, or you may have to use it in conjunction with SYS1.VSF2FORT.

SYS1.VSF2MATH An alternate math library that contains math routines from VS FORTRAN Version 1 Release 4.

Note: If you are running a parallel program, you cannot use the SYS1.VSF2MATH library.

Your system programmer must tell you whether your system is set up for link mode, load mode or both. In addition, your system programmer may have given these libraries names different from the standard names listed above; the examples below assume that the standard names are used.

Your program must have access to the VS FORTRAN Version 2 library when you are:

- Operating in load mode
- Using a load module that was created from a version of VS FORTRAN prior to Version 1 Release 4 and used the reentrant I/O library facility
- Using any load module linked at the Version 1 Release 3 or 3.1 level.

The procedures for specifying libraries in link mode or load mode are described below. Note that parallel programs and programs with dynamically loaded modules can be run only in load mode.

Note: If you have compiled with the RENT option and separated your text, and have link-edited the nonreentrant part for link mode operation, the reentrant parts modules still must be loaded, either from the link pack area or from a partitioned data set.

Specifying Load Mode

Parallel programs and programs with dynamically loaded modules can be run only in load mode.

For operating in load mode, you must perform *both* of the following:

1. For the link-editing step, provide SYS1.VSF2FORT but *not* SYS1.VSF2LINK to the linkage editor (or loader) to use when including VS FORTRAN Version 2 library modules. Specify only SYS1.VSF2FORT in the DD statement for SYSLIB:

```
//SYSLIB DD DSN=SYS1.VSF2FORT,DISP=SHR
```

The SYSLIB data set supplies the run-time library modules that your load module needs. It should be from the most recent level of VS FORTRAN Version 2 that has been installed on your system.

If you are running a parallel program, do not include SYS1.VSF2MATH in the SYSLIB concatenation.

2. For running your program (the GO step), make SYS1.VSF2LOAD or SYS1.VSF2FORT available by performing *one* of the following steps.
 - To provide access to the VS FORTRAN Version 2 library for all of the job steps, place the following JOBLIB DD statement for the load module immediately after the JOB statement:

```
//JOBLIB DD DSN=SYS1.VSF2LOAD,DISP=SHR
```

or

```
//JOBLIB DD DSN=SYS1.VSF2FORT,DISP=SHR
```

- To provide access to the VS FORTRAN Version 2 library for a single job step, include the following STEPLIB DD statement in the DD statements for that job step:

```
//STEPLIB DD DSN=SYS1.VSF2LOAD,DISP=SHR
```

```
// DD DSN=MY.USERLIB,DISP=SHR
```

or

```
//STEPLIB DD DSN=SYS1.VSF2FORT,DISP=SHR
```

```
// DD DSN=MY.USERLIB,DISP=SHR
```

If your load module was created using a version of VS FORTRAN before Version 1, Release 4 and used the reentrant I/O library facility, replace the SYS1.VRENTLIB in the original STEPLIB DD statement with SYS1.VSF2LOAD.

- To provide access to the VS FORTRAN Version 2 library for a single job step, include a FORTLIB DD statement in the DD statements for the job step.

```
//FORTLIB DD DSN=SYS1.VSF2LOAD,DISP=SHR
//          DD DSN=MY.USERLIB,DISP=SHR
```

or

```
//FORTLIB DD DSN=SYS1.VSF2FORT,DISP=SHR
//          DD DSN=MY.USERLIB,DISP=SHR
```

Note: If both FORTLIB and STEPLIB are specified, FORTLIB is searched first. If you are using IAD and have specified the libraries to be searched in FORTLIB, you must also specify the libraries in ISPLLIB.

If you are using dynamically loaded modules, add the libraries containing those modules to the appropriate DD concatenation.

This technique does not let you use reentrant modules that are in the link pack area, because step libraries and job libraries are searched before the link pack area. (Refer to your operating system's supervisor services manual.)

Specifying Link Mode

Parallel programs and programs using dynamic loading can be run only in load mode.

For operating in link mode, you must perform the following:

1. For the link-editing step, concatenate SYS1.VSF2LINK ahead of SYS1.VSF2FORT for use by the linkage editor (or loader) when it includes VS FORTRAN Version 2 library modules. Specify both SYS1.VSF2LINK and SYS1.VSF2FORT in the DD statement for SYSLIB:

```
//SYSLIB DD DSN=SYS1.VSF2LINK,DISP=SHR
//          DD DSN=SYS1.VSF2FORT,DISP=SHR
```

Note: A program link-edited in link mode does not require any VS FORTRAN Version 2 libraries at run time.

2. If you are using shareable modules, you will have to access them with either a FORTLIB DD statement or a STEPLIB DD statement.

Link-Editing Your Program

Before you can run your program, you must link-edit the object module with other object modules to construct an executable load module or a dynamically loadable module.

Your input to the link-edit step is the object module produced by the compiler using the DECK or OBJECT option. The object module consists of dictionaries, text, and an end-of-module indicator. (For additional details, see Appendix C, "Object Module Records" on page 475.)

Note: FORTRAN 66 object programs are link-edited exactly the same as FORTRAN 77 object programs.

For information on migrating load modules, see “Migration of VS FORTRAN Load Modules” on page 85.

You may use one of two methods to perform the link-edit: using the linkage editor or using the loader.

Linkage Editor Use the linkage editor if you want to: link-edit your program only, reduce storage requirements through overlays, use additional libraries as input, or define the structural segments of the program.

Loader Use the loader if you want to link-edit and run your program in one step. Your input is a small object module that doesn't require overlay, and doesn't require additional linkage editor control statements,

Note: The loader cannot be used to link-edit and run parallel programs.

VS FORTRAN Version 2 supplies you with cataloged procedures that let you link-edit or load your programs easily. For details, see Figure 32 on page 84.

Link-Editing Your Program Using the Linkage Editor

You can use a number of processing options and optional data sets when you use the linkage editor rather than the loader. For information on JCL statement syntax for the following sections, see “Coding and Processing Jobs” on page 12.

Requesting Linkage Editor Processing Options: 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 the load module to run, even when abnormal conditions have been detected that could cause the program to fail.

NCAL

Specifies that the linkage editor is not to attempt to resolve external references.

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, segments of the program share the same storage at different times during processing. (For more details, see Chapter 19, “Interprogram Communication” on page 371.)

SIZE

Specifies the amount of virtual storage to be used for this link-edit job.

Defining Linkage Editor Data Sets: On DD statements, you can request required and optional data sets for running your program. The linkage editor generally uses five data sets. Others may be necessary if you have specified secondary input by using an INCLUDE or LIBRARY linkage editor control statement. Cataloged procedures do not supply secondary input DD statements.

Required Linkage Editor Data Sets: For any link-edit job, you must make certain that at least the following data sets are available:

SYSLIB

Direct access data set (in partitioned data set format) that makes the automatic call library (SYS1.VSF2FORT or SYS1.VSF2MATH or both libraries, and perhaps others) 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: In addition, you can optionally specify the following data set:

SYSTEM

Used for writing error messages. This data set can be on a direct access, magnetic tape, or printer device.

Data Set Characteristics: Figure 28 on page 80 lists the function, device types, and allowable device classes for each linkage editor data set.

Figure 28. Linkage Editor Data Sets

ddname	Function	Device Types	Device Class	Defined(1)
SYSLIN	Primary input data, generally output of the compiler	Direct access Magnetic tape Card reader	SYSDA SYSSQ input stream (defined as DD * or DD DATA)	Yes
SYSLIB	Automatic call library (SYS1.VSF2FORT)	Direct access	SYSDA	Yes
SYSLMOD	Link-edit output (load module)	Direct access	SYSDA	Yes
SYSPRINT	Writing listings, messages	Printer Magnetic tape Direct access	A SYSSQ SYSDA	Yes
SYSUT1	Linkage, editor work, data set	Direct access	SYSDA	Yes
User-defined	Additional libraries and object modules	Direct access Magnetic tape	SYSDA SYSSQ	No

Note:

1. The **Defined** column indicates whether or not the ddname is defined in cataloged procedures.

Using Linkage Editor Control Statements: Linkage editor control statements specify an operation with one or more operands.

The first column of a control statement must be left blank. The operation field begins in column 2 and specifies the name of the operation to be performed. The operand field must be separated from the operation field by at least one blank. The operand field specifies one or more operands separated by commas. No embedded blanks may appear in the field. Linkage editor control statements may be placed before, between, or after either modules or secondary input data sets.

The INCLUDE and LIBRARY control statements specify secondary input. You can use the following INCLUDE and LIBRARY linkage editor control statements:

The INCLUDE Linkage Editor Control Statement: is used to specify additional object modules you want included in the output load module.

Operation	Operand
INCLUDE	ddname[(member-name [,member-name],...)] [,ddname[(member-name [,member-name],...)]]

ddname Indicates the name of a DD statement specifying a library or a sequential data set.

member-name Indicates the name of the member to be included. When sequential data sets are specified, member-name is omitted.

The LIBRARY Linkage Editor Control Statement: is used to specify additional libraries to be searched for object modules to be included in the load module.

Operation	Operand
LIBRARY	<i>ddname</i> [(<i>member-name</i> [, <i>member-name</i>],...)] [, <i>ddname</i> [(<i>member-name</i> [, <i>member-name</i>],...)]]

ddname Indicates the name of a DD statement specifying a library.

member-name Indicates the name of a member of the library.

The LIBRARY statement differs from the INCLUDE statement in that libraries specified in the LIBRARY statement are not searched until all other references (except those reserved for the automatic call library) are completed by the linkage editor. A module specified in the INCLUDE statement is included immediately.

Example of Using Linkage Editor Control Statements: LIBRARY and INCLUDE are specified in the input stream to the linkage editor. You may place them at the end of a file.

```
//SYSLIN DD DSN=&TEMP,DISP=(OLD,PASS)    COMPILER OUTPUT
//      DD *
    INCLUDE X(Y)          GET ROUTINE Y FROM DDNAME X
    LIBRARY Z(A)          GET ROUTINE A FROM LIBRARY Z
/*
```

Linkage Editor Output: Output from the linkage editor is in the form of load modules 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.

Link-Editing Your Program Using the Loader

You choose the loader when you want to combine link-editing into one job step with load module processing. The loader combines your object module with other modules into one load module, and then places the load module into main storage and runs it. The loader cannot be used to run a parallel program.

The loader options you can use, and the loader data sets, are described in the following paragraphs.

Requesting Loader Options: When you run 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 to run, even when abnormal conditions that could cause the program to fail have been detected.

CALL|NCAL

Specifies whether or not the loader is to attempt to resolve external references.

EP

Allows you to 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.

Defining Loader Data Sets: The loader generally uses six system data sets; other data sets may be defined to describe libraries and load module data sets. For any loader job, you must make certain that at least the SYSLIN data set (used for compiler output) is available.

In addition, depending on what you want the loader to do for you, you can, optionally, specify the data sets in Figure 29. This figure lists the function, device types, and allowable device classes for each data set.

Figure 29. Loader Data Sets

ddname	Function/Description	Device Types	Device Class	Defined(1)
SYSLIN	Input data to linkage function, normally output of the compiler	Direct access Magnetic tape Card reader	SYSDA SYSSQ Input stream (defined as DD *)	Yes
SYSLIB	Automatic call library (SYS1.VSF2FORT)	Direct access	SYSDA	Yes
SYSLOUT	Writing listings	Printer Magnetic tape Direct access	A SYSSQ	Yes

Note:

1. The **Defined** column indicates whether or not the ddname is defined in cataloged procedures.

Defining Load Module Execution Data Sets: The load module may be passed directly from a preceding link-edit job step, it may be called from a library of programs, or it may form part of the loader job step. The load module processing job step may use many data sets. Figure 30 on page 83 lists the IBM-supplied default function and device types for each data set.

Figure 30. Load Module Execution Data Sets

Fortran Unit Number	ddname	Function	Device Type	Device Class	Defined
5	FT05Fmmm	Input data set to load module	Card reader Magnetic tape Direct access	SYSDA	No
6	FT06Fmmm	Printed output data	Printer Magnetic tape Direct access	SYSDA	No
7	FT07Fmmm	Punched output data	Card punch Magnetic tape Direct access	B	Yes
0-4, 8-99	FTnnFmmm	Sequential data set	Unit record Magnetic tape Direct access	SYSSQ A,B SYSDA	No
0-4, 8-99	FTnnFmmm	Direct access data set	Direct access	SYSDA	No
0-4, 8-99	FTnnFmmm	Partitioned data set member using sequential access	Direct access	SYSDA	No
0-4, 8-99	FTnnPmmm	Unnamed striped sequential data set	Direct access Magnetic tape	SYSDA	No
100-2000	User-defined	Named data set	All	All	No

DCB Default Values for Sequential Data Sets: Figure 31 lists the DCB default values for load module execution **sequential** data sets. These default values also apply to dummy data sets.

Figure 31. Load Module Execution Sequential Data Set DCB Default Values

ddname	RECFM(1)	LRECL(2)	BLKSIZE	DEN	BUFNO
FT05Fmmm	F	80	80	–	2
FT06Fmmm	UA	133	133	–	2
FT07Fmmm	F	80	80	–	2
FTnnPmmm	U	–	800	2	The greater value of 7 or UAT default
all others	U	–	800	2	2

Notes:

1. For records not under FORMAT control, the default is VS. When a file is opened by a Fortran ENDFILE statement, the default is U.
2. For records not under FORMAT control, the default is 4 less than shown.

DCB Default Values for Direct Access Data Sets: For the DCB default values for all direct access data sets during load module run time, the record form (RECFM) is F, the buffer number (BUFNO) is 1, and the blocksize (BLKSIZE) or longest record length (LRECL) is the value specified as the maximum size of a record in the OPEN statement.

Running Your Program and Specifying Run-Time Options and User Parameters

How you run the load module depends on the kind of job you're running:

- Run only
- Link-edit and run
- Compile, link-edit, and run.

IBM-supplied cataloged procedures are available that let you easily compile, link-edit or load, and/or run. A list of all the nonreentrant cataloged procedures is given in Figure 32. The cataloged procedures should be located in your appropriate system procedure library.

Figure 32. IBM-Supplied Nonreentrant Cataloged Procedures

Action	Procedure Name
Compile only	VSF2C
Compile and link-edit	VSF2CL
Compile, link-edit, and run	VSF2CLG
Link-edit and run	VSF2LG
Run only	VSF2G
Compile and load	VSF2CG
Load only	VSF2L

A complete list of the available run-time options is in "Available Run-Time Options" on page 105. To specify run-time options, use the following method:

```
//GO EXEC PGM=main,PARM='option[,option...] [/user_parameters]'
```

Any substring you supply after the slash (/) in the above commands are recognized as parameters to a subprogram, but not as run-time options. User parameters can be accessed in your Fortran program with the use of the ARGSTR service subroutine. See *VS FORTRAN Version 2 Language and Library Reference* for more information.

Note: For information on using the current library with existing VS FORTRAN load modules, see page 484.

Requesting an Abnormal Termination Dump: Program interrupts causing abnormal termination produce a 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
```

You can find information on interpreting dumps in the appropriate debugging guide for your system.

Overlaying Programs—System Considerations

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 while the program runs.

You must specify linkage editor control statements to indicate the relationship of segments within the overlay structure.

Specifying Overlays: Overlay is initiated at run 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.

When you use the overlay features, consider the following:

- Although overlays reduce storage, they also may drastically increase program run time.
- Modules compiled with the RENT compile-time option are not executable in MVS as overlays.
- The SAVE statement has no effect on overlaid programs; that is, variable values in an overlaid program become undetermined when the program is overlaid by another.

For information on overlaying your program, see the linkage editor manual for your operating system.

Migration of VS FORTRAN Load Modules

When you link-edit a load module, your primary linkage editor input usually consists of the object modules that you have just compiled. Another linkage editor input, the SYSLIB data set, supplies the run-time library modules that your load module needs. (The SYSLIB data set that you use should be from the most recent level of VS FORTRAN Version 2 that has been installed at your installation.) Link-editing your program in this manner satisfies the following two requirements:

1. All of the run-time library modules in a load module must be at the same release and modification level.
2. The run-time library modules in a load module must be at a release level that is at least as high as the highest level of the compiler that was used to create any of the object modules.

You may have to create a new load module by using one of your existing load modules (rather than only the object modules) as input to the linkage editor. This can occur when:

- You need to recompile some, but not all, of your own Fortran routines that are within one of your load modules.
- You need to upgrade one of your existing load modules so it contains the run-time library modules at the latest release or maintenance level. A new release or corrective service to these library modules is always installed in your product data sets, but the changes are not reflected in any of your own load modules unless you link-edit them again using the updated data sets.
- You need to change the mode from link mode to load mode (or vice versa).

You might have to use your original load module rather than only your object modules as linkage editor input in these cases, either because you don't have all of your routines available in source form for recompilation or because you didn't retain the object modules. A problem occurs when you use your previous load module as linkage editor input: the linkage editor retains the run-time library modules that are in your original load module while including others from the current SYSLIB input; this may violate either or both of the two requirements listed on page 85.

Library Module Replacement Tool: When you must use one of your own load modules as linkage editor input, the following procedures assist you in link-editing your load modules. VS FORTRAN Version 2 supplies a set of linkage editor REPLACE statements. Use these to replace all of the run-time library modules when your input to the linkage editor is an existing load module containing the library modules.

If, on the other hand, you are able to recompile all of your own routines to create new object modules or if you still have all of the object modules available, then you can create your new load module from these object modules. In this case, you do not have to use the set of REPLACE statements.

The set of linkage editor REPLACE statements is in the member AFBVLKED in SYS1.SAMPLIB. In your linkage editor primary input data set, SYSLIN, include this member immediately before you include the load module in which the replacement is to occur. The run-time library modules from the data set pointed to by the SYSLIB DD statement can then replace the previous ones and become part of your new load module.

Examples: Figure 33 on page 87 illustrates the replacement of all of the VS FORTRAN Version 1 or Version 2 run-time library modules in one of your load modules without replacing any of your own modules. You might do this to incorporate the corrective service that has been applied to these library modules in your product data sets into your load module. In Figure 33 on page 87, `mypls.load` is the name of your load module library that contains the load module with the name `mylmod`.

```
//RELINK EXEC PGM=IEWL,PARM='LIST,MAP,XREF'
//SYSPRINT DD SYSOUT=A
//SYSLIB DD DSN=SYS1.VSF2FORT,DISP=SHR
//SAMPLIB DD DSN=SYS1.SAMPLIB,DISP=SHR
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD DD DSN=mypds.load,DISP=OLD
//SYSLIN DD *
INCLUDE SAMPLIB(AFBVLKED)
INCLUDE SYSLMOD(my1mod)
NAME my1mod(R)
/*
```

Figure 33. Replacing VS FORTRAN Run-Time Library Modules

In Figure 33, only SYS1.VSF2FORT is provided in the SYSLIB DD statement; therefore, the resulting load module runs in load mode.

Figure 34 illustrates the use of the cataloged procedure VSF2CL to recompile one of your modules, to retain all of your other modules from an existing load module, and to replace all of the run-time library modules with the current ones. In this example, mypds.load is the name of your load module library that contains the load module with the name my1mod, and mysrce is the name of the data set that contains the Fortran source program that you want to recompile.

```
//RECOMP EXEC VSF2CL,
//          PGMLIB='mypds.load',PGMNAME=my1mod
//FORT.SYSIN DD DSN=mysrce,DISP=SHR
//LKED.SYSLMOD DD DISP=OLD
//LKED.SAMPLIB DD DSN=SYS1.SAMPLIB,DISP=SHR
//LKED.SYSIN DD *
INCLUDE SAMPLIB(AFBVLKED)
INCLUDE SYSLMOD(my1mod)
/*
```

Figure 34. Using the VSF2CL Cataloged Procedure

In Figure 34, the selection of link mode or load mode is controlled by the SYSLIB DD statement that is in the linkage editor step of the cataloged procedure VSF2CL.

Notes:

1. You can use the REPLACE statements that are in the member AFBVLKED in SYS1.SAMPLIB with input load modules that were created with any release level of VS FORTRAN Version 1 or of VS FORTRAN Version 2.
2. The data sets that you supply in your linkage editor step control whether the resulting load module runs in link or load mode, and whether it uses the standard or the alternative mathematical routines. For example, your original load module may have been created so that it runs in link mode, but, using the REPLACE statements, you can link-edit it again so that it operates in load mode (and vice versa). Similarly, you can change between the standard and the alternative mathematical routines. This requires only that the proper data sets be specified as the SYSLIB input to the linkage editor.
3. Do not attempt to create and use your own set of REPLACE statements based upon the run-time library modules that are in your load module. (Certain modules must be replaced by using the form of the REPLACE statement that replaces a CSECT; others must be replaced by using the form of the REPLACE statement that deletes a CSECT and by allowing the module to be included by the automatic library-call mechanism.)

4. If you have previously tried to replace the run-time library modules using your own set of REPLACE statements and if the resulting load module did not work properly, then use the supplied set as described above, and create your load module so that it runs in load mode. You will probably not be able to link-edit it again to run successfully in link mode, but, for load mode, you can often create a usable load module.

MVS/XA and MVS/ESA Considerations

Program Attributes—AMODE and RMODE

Every program that runs under MVS/XA or MVS/ESA is assigned two attributes: AMODE (addressing mode) and RMODE (residence mode).

AMODE Is a program attribute that indicates which addressing mode can be supported at a particular entry into a program. *Addressing mode* refers to the length of an address, either 24 bits or 31 bits, used by the processor; indicated by the high-order bit of the program status word (PSW). Generally, the program is also designed to run only in the AMODE specified, although an assembler language program can switch the addressing mode. There are three possible values for AMODE: 24, 31, and ANY.

RMODE Is a program attribute that indicates which residence mode can be supported at a particular entry into a program. *Residence mode* refers to where a program resides in virtual storage in an XA or ESA environment above or below 16 megabytes. The boundary line is called the 16-megabyte line, which pertains to the range addressable by a 24-bit address. There are two possible values for RMODE: 24 and ANY.

Program units compiled by VS FORTRAN Version 1, Release 2 and later, or with Version 2, can run in 24- or 31-bit addressing mode in the MVS/XA or MVS/ESA operating system. These program units can reside either above the 16-megabyte line or below the 16-megabyte line. With 31-bit addressing, there is more freedom to define or reference larger data areas, files, tables, and to create a larger overall program. The program unit and its data are no longer constrained to fit in a 16-megabyte address space, but can refer to addresses anywhere in virtual storage, up to the 2-gigabyte maximum address.

Program units compiled by FORTRAN G1, HX, HX (Enhanced), F, or VS FORTRAN Version 1, Releases 1 and 1.1, have addressing and residence dependencies which allow only 24-bit addressing mode (AMODE=24), and can reside only below the 16-megabyte line (RMODE=24) when running under MVS/XA or MVS/ESA. These program units can still be used by themselves or link-edited with VS FORTRAN Version 2 subprograms to be run under MVS/XA or MVS/ESA. The resulting load module can run only with an addressing mode of 24-bit (AMODE=24), and must reside below the 16-megabyte line (RMODE=24).

Specifying MVS/XA or MVS/ESA Linkage Editor and OS Loader Attributes

To take advantage of 31-bit addressing, a program must be link-edited by the MVS/XA or MVS/ESA linkage editor or OS loader and have no 24-bit addressing dependencies. The MVS/XA or MVS/ESA linkage editor or OS loader provides the means for changing the addressing mode (AMODE) and residence mode (RMODE)

specification. The valid linkage editor AMODE and RMODE specifications are listed below.

Attribute	Meaning
AMODE=24	24-bit data addressing mode
AMODE=31	31-bit data addressing mode
AMODE=ANY	Either 24-bit or 31-bit addressing mode
RMODE=24	The module must reside in virtual storage below 16 megabytes. Use RMODE=24 for 31-bit programs that have 24-bit dependencies.
RMODE=ANY	Indicates that the module can reside anywhere in virtual storage.

The linkage editor validates the combination of the AMODE value and the RMODE value when specified in either the PARM field of the EXEC statement, or the linkage editor MODE control statement, according to Figure 35.

Figure 35. Valid AMODE and RMODE Values

	RMODE=24	RMODE=ANY
AMODE=24	Valid	Invalid
AMODE=31	Valid	Valid
AMODE=ANY	Valid	Invalid

VS FORTRAN and MVS/XA or MVS/ESA Linkage Editor and Loader Interaction

VS FORTRAN Compiler Version 1 Release 2 or later creates object code that is given the attributes AMODE=ANY and RMODE=ANY in each CSECT produced. By default, all previous Fortran object code CSECTs are given the attributes AMODE=24 and RMODE=24. These attributes are then modified at link-edit time by default values, or by values set in the PARM field of the EXEC statement or the linkage editor MODE control statement, as discussed under “Specifying MVS/XA or MVS/ESA Linkage Editor and OS Loader Attributes” on page 88.

The default action of the linkage editor is to check each CSECT of the entire load module, and set the RMODE to the lowest mode encountered. It then checks the AMODE of the main entry point, and sets the AMODE for the entire load module to the AMODE of the entry point CSECT. This means that:

- All Fortran main programs compiled prior to VS FORTRAN Version 1, Release 2 have the default AMODE and RMODE of 24. The linkage editor will set the AMODE and RMODE of the load module to 24 by default. The created load module resides below the 16-megabyte line, and is invoked in 24-bit addressing mode.
- All VS FORTRAN main programs compiled with VS FORTRAN Version 1, Release 2 and later have the AMODE set to ANY. The linkage editor sets the AMODE of the load module to ANY by default.

The load module will be entered in the AMODE that the linkage editor stored it in. You can force AMODE to be any valid value that you wish, but if there are any dependencies, your program will fail. There is no compile-time option that can change the AMODE value in the input to the linkage editor.

If the main routine is originally RMODE=24, AMODE=31, and calls a VS FORTRAN subroutine compiled by VS FORTRAN Version 1, prior to Release 2, or a Fortran subroutine compiled by any other Fortran compiler or an Assembler routine with 24-bit addressing dependencies, the program may abnormally terminate while running. To prevent this, the default AMODE attribute of the subroutine must be overridden in the link-edit step to set AMODE=24.

- The RMODE of the load module is based upon whether the load module is created to be run in link mode or load mode. For complete details concerning link mode and load mode of the VS FORTRAN Version 2 Library, see “Selecting Load Mode or Link Mode” on page 75.

A program that is link-edited to operate in link mode is always given an RMODE of 24. Overriding this value to ANY is not permissible because there are some service subroutines in the created load module that must reside below the 16-megabyte line.

A program that is link-edited to operate in load mode can, except for the cases noted above, have any valid combination of AMODE and RMODE values. The service subroutines that are loaded during run time are loaded either above or below the 16-megabyte line, based upon their individual residence mode requirements. Because of the scattered loading of individual VS FORTRAN Version 2 Library modules, the run-time library always switches to 31-bit addressing mode while in the service subroutines, and to the addressing mode of the caller of the service subroutine upon return.

The control program invokes the load module created by the linkage editor according to its AMODE, and places the module above or below the 16-megabyte line according to its RMODE. For more information about AMODE and RMODE, see *MVS/XA Supervisor Services and Macro Instructions*.

Overriding AMODE/RMODE Attributes

To override the default link-edit attributes, use *one* of the following to specify AMODE and/or RMODE:

- The linkage editor or loader EXEC statement

```
//LKED EXEC PGM=programname,  
//          PARM='AMODE=xxx,RMODE=yyy'
```

where xxx is 24, 31 or ANY and yyy is 24 or ANY. Valid combinations of xxx and yyy are described in “Specifying MVS/XA or MVS/ESA Linkage Editor and OS Loader Attributes” on page 88.

For additional detail, see *MVS/XA Linkage Editor and Loader User's Guide*.

- The linkage editor MODE control statement

```
MODE AMODE(xxx),RMODE(yyy)
```

For additional detail, see *MVS/XA Linkage Editor and Loader User's Guide*.

- The TSO commands LINK or LOADGO

```
LINK (dsn-list) AMODE(xxx) RMODE(yyy)
```

or

```
LOADGO (dsn-list) AMODE(xxx) RMODE(yyy)
```

Using Dynamic Common Above the 16-Megabyte Line

In order to use the extra storage available with MVS/XA or MVS/ESA, the load module must run in 31-bit addressing mode. In particular, the module cannot contain subroutines compiled under FORTRAN G1, HX or prior to VS FORTRAN Version 1, Release 2. The storage for dynamic common areas is obtained above the 16-megabyte line only when the program is running in 31-bit addressing mode (regardless of the residence mode) and storage is available; storage is obtained below the 16-megabyte line when the program is running in 24-bit addressing mode.

The linkage editor limits the size of a load module to 16 megabytes. To overcome this limit, VS FORTRAN Version 2-named common areas can be declared so that they will occupy storage outside of the load module. The storage is dynamically obtained and made available to the object code by the VS FORTRAN Version 2 Library at run time. For details concerning dynamic common areas, see “Using Static, Dynamic, and Extended Common Blocks” on page 379.

Example:

```
@PROCESS DC(CMN1,CMN2)
COMMON /CMN1/XARRAY(1000,1000,1000)
COMMON /CMN2/YARRAY(5000000)
COMMON /CMN3/ZARRAY(100,100,100)
```

Storage for common areas CMN1 and CMN2 is dynamically obtained at run time. The storage for COMMON CMN3 is part of the load module, and takes up part of the 16-megabyte maximum module size.

XA Hints for Fortran Users

Unless you specifically force an AMODE value of 24, do not mix object modules compiled with VS FORTRAN Version 1 Release 2 and later with:

- Object modules compiled with compilers prior to VS FORTRAN Version 1 Release 2
- Assembler code with 24-bit dependencies.

Running Your Program under MVS with TSO

Under MVS with TSO, you process a Fortran program by performing the following steps:

1. Compiling your program
2. Link-editing your program
3. Running your program.

This section discusses only the link-edit and run steps. Compiling your program is discussed in “Compiling Your Program under MVS with TSO” on page 18.

To link-edit your program under TSO, use the LINK command to create a load module from one or more object modules (plus any needed VS FORTRAN Version 2 library modules). To run your program, use the CALL command. To link-edit and run your program in one step, use the LOADGO command.

The input object modules must be OBJ data sets; for example:

```
userid.name.OBJ
```

The following sections discuss:

Selecting link mode or load mode
Link-editing your program using the linkage editor (LINK command)
Link-editing and running your program using the loader (LOADGO command)
Running your program and specifying run-time options and user parameters
(CALL command)
Using CLISTs.

For information on using XA-mode files, see “MVS/XA and MVS/ESA Considerations” on page 88.

Selecting Link Mode or Load Mode

As in MVS, you can run your program in either link mode or load mode. See “Selecting Load Mode or Link Mode” on page 75 for a description of link mode and load mode.

Specifying Load Mode

For operating in load mode, you must perform *both* of the following:

1. For the link-editing step, provide VSF2FORT but *not* VSF2LINK to the linkage editor (or loader) to use when including VS FORTRAN Version 2 library modules.
 - If you are using the linkage editor to link-edit the OBJ data sets for myprog and its subprogram subprog, you specify:
For using standard mathematical routines:
`LINK (myprog,subprog) LOAD(myprog) LIB('SYS1.VSF2FORT')`
For using alternate mathematical routines:
`LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VSF2MATH','SYS1.VSF2FORT')`
 - If you are using the loader to link-edit and run the OBJ data sets for myprog and its subprogram subprog, you specify:
For using standard mathematical routines:
`LOADGO (myprog,subprog) LIB('SYS1.VSF2FORT')`
For using alternate mathematical routines:
`LOADGO (myprog,subprog) LIB('SYS1.VSF2MATH','SYS1.VSF2FORT')`
The LIB operand provides access to the required data sets. The loader resolves any external references in myprog and loads the required object modules.
2. For running your program, make VSF2LOAD available by performing *one* of the following:
 - To provide access to the VS FORTRAN Version 2 library, associate the data set SYS1.VSF2LOAD with FORTLIB in an ALLOCATE command, as follows:
`ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD' userlib) SHR`
where *userlib* is the dataset containing your dynamically loadable modules.

- To provide access to the VS FORTRAN Version 2 library, ask your system programmer to do one of the following:

Add SYS1.VSF2LOAD to your system link list.

Concatenate SYS1.VSF2LOAD with the data sets named in the STEPLIB DD statement in your logon procedure.

Note: The system programmer might also install certain VS FORTRAN modules in the link pack area. If a different maintenance level is installed in the link pack area than is installed in the library you allocate to FORTLIB, unpredictable results might occur. For example, unpredictable result might occur if you first run a program using no allocation to FORTLIB, and then run a program allocating FORTLIB during the same TSO session. This can occur for programs that use asynchronous I/O or the parallel feature. To avoid this possibility, do not change the way VSF2LOAD is made available in a TSO session when running programs which use these functions.

Specifying Link Mode

For the link-editing step, concatenate VSF2LINK ahead of VSF2FORT for the linkage editor (or loader) to use when including VS FORTRAN Version 2 library modules.

- If you are using the linkage editor to link-edit the OBJ data sets for myprog and its subprogram subprog, you specify:

For using standard mathematical routines:

```
LINK (myprog,subprog) LOAD(myprog) LIB('SYS1.VSF2LINK','SYS1.VSF2FORT')
```

For using alternate mathematical routines:

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VSF2LINK','SYS1.VSF2MATH','SYS1.VSF2FORT')
```

- If you are using the loader to link-edit and run the OBJ data sets for myprog and its subprogram subprog, you specify:

For using standard mathematical routines:

```
LOADGO (myprog,subprog) LIB('SYS1.VSF2LINK','SYS1.VSF2FORT')
```

For using alternate mathematical routines:

```
LOADGO (myprog,subprog) LIB('SYS1.VSF2LINK','SYS1.VSF2MATH','SYS1.VSF2FORT')
```

The LIB operand provides access to the required data sets. The loader resolves any external references in myprog and loads the required object modules.

Note: A program link-edited in link mode does not require VS FORTRAN Version 2 libraries at run time.

Link-Editing Your Program—LINK Command

You use the LINK command to create a load module. The input you use consists of your object module, VS FORTRAN Version 2 service subroutines, and any other secondary input (such as OBJ data sets of called subprograms).

For example, if you want to link-edit the OBJ data sets for myprog and its subprogram subprog, (in load mode), you specify:

```
LINK (myprog,subprog) LOAD(myprog) LIB('SYS1.VSF2MATH','SYS1.VSF2FORT')
```

When the commands are run, the OBJ data sets for myprog and subprog are link-edited together into a load module.

You must request the linkage editor to search the library to resolve external references. In the above example, you are, therefore, requesting a search of SYS1.VSF2MATH and SYS1.VSF2FORT.

Requesting Linkage Editor Options: You can use the LINK command to specify linkage editor options. 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('SYS1.VSF2FORT') LOAD(myprog) PRINT
```

The qualified name of the data set to be sent to the system printer is *userid.myprog.linklist*. To print the data set, you must use a print command, or the ISPF HARDCOPY command.

At Your Terminal

```
LINK (myprog,subprog) LIB('SYS1.VSF2FORT') LOAD(myprog) PRINT(*)
```

When you specify PRINT(*), the linkage editor listings are displayed at your terminal.

Link-Editing and Running Your Program—LOADGO Command

Depending on the version of VS FORTRAN you used to link-edit the original load module, your TSO load module must have access to the library modules mentioned in “Required Library Modules” on page 95 in order to run your program.

The LOADGO command invokes the system loader program to load and run your compiled program. This load function is equivalent to the link-edit and run function, providing the capability for link-editing and running your program in one step. When the program has run, TSO automatically deletes the load module created by LOADGO. The LOADGO command cannot be used to run a parallel program.

Allocating Data Sets with LOADGO

Use the following procedure to allocate the data sets you want to use with LOADGO.

1. Allocate the required data sets as described in “Allocating Compiler Data Sets” on page 18.
2. Specify link mode or load mode as indicated in “Selecting Link Mode or Load Mode” on page 92.
3. Run the LOADGO command.

Example 1: You can use the LOADGO command to link-edit and run an object module myprog:

```
LOADGO (myprog) LIB('SYS1.VSF2LINK','SYS1.VSF2FORT')
```

Example 2: You can also use LOADGO to run a link-edited load module:

```
LOADGO myprog(tempname)
```

Note: You can use load mode if you allocated the FORTLIB ddname earlier.

Using LOADGO to Specify Loader and Run-Time Options

You can use LOADGO to specify loader options. The following form of LOADGO specifies that a load module map and listings are to be printed, either on the system printer or at your terminal:

On the System Printer

```
LOADGO (myprog) MAP PRINT
```

The output data set sent to the system printer has the qualified name MYPROG.LOADLIST.

At Your Terminal

```
LOADGO (myprog) MAP PRINT(*) ABSDUMP
```

Specifying PRINT(*) causes the loader listings to display on your terminal.

Running Your Program and Specifying Run-Time Options—CALL Command

Before you run your program, you may need other library modules. Depending on the version of VS FORTRAN you used to link-edit the original load module, your TSO load module must have access to the library modules listed below.

Required Library Modules

If you are using a VS FORTRAN Version 1, Release 2 or Release 3 or 3.1 load module with the VS FORTRAN Version 2 library (that is, you have not link-edited your object files with the Version 1, Release 4 library or with the Version 2 library), then

- AFBVASUB (with alias of IFYVASUB) and AFBVPOST (with alias of IFYVPOST) must be installed in a library on the system link list (see SYS1.PARMLIB member LNKLIST00 for the libraries on the list), or use the FORTLIB ddname.
- If you are using the IFYVRENT routines, AFBVRENT (with alias of IFYVRENT) must also be in a library on the system link list, or use the FORTLIB ddname.
- If you are using the IFYVRENT routines and if AFBVRENT (with alias of IFYVRENT) is in the link pack area (LPA), then AFBVPOST and AFBVASUB (AFBVASUB may go into the LPA) must be in a library on the system link list, or use the FORTLIB ddname.

If you are uncertain whether your executable module has access to the correct library modules, see your system programmer.

Using the CALL Command to Run the Load Module

The following example assumes that you have a load module named MYPROG.LOAD. You can run the load module with the following set of TSO commands. The ALLOCATE commands identify the input and output data sets, as well as any work data sets used by the program. The CALL command causes TSO to run program tempname from the file MYPROG.LOAD.

```
ALLOCATE DATASET(myprog.indata) FILE(FT05F001) (as needed)
ALLOCATE DATASET(myprog.outdata) FILE(FT06F001) (as needed)
ALLOCATE DATASET(myprog.workfil) FILE(FT10F001) (as needed)
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD' userlib) SHR
CALL myprog
```

where *userlib* is the dataset containing your dynamically loadable modules.

After the program has run, you should delete any data sets that you won't be using again. Use the DELETE command to delete unneeded data sets named in the ALLOCATE and CALL commands. For example:

```
DELETE (myprog.indata myprog.outdata myprog.workfil)
```

Considerations for Terminal Files: For terminal files, the end-of-file signal is *'/*'*. Another ALLOCATE command or an explicit OPEN is required to continue processing.

When you use a terminal for list-directed I/O, do not specify the IN parameter with the ALLOCATE command. The IN parameter forces the terminal data set to be opened for input only; its use may cause input records to be skipped.

Specifying Run-Time Options and User Parameters

The same run-time options that are available in MVS are available in TSO. "Available Run-Time Options" on page 105 contains a list of these options and describes how to use each option. In TSO, you specify run-time options with the CALL command. This command has the following form:

```
CALL pgmname 'option[,option... [/user_parameters] ]
```

where *pgmname* is the name of your VS FORTRAN Version 2 program, and *option* is a run-time option.

Any substring you supply after the slash (/) in the above commands are recognized as parameters to an assembler subprogram, but not as run-time options.

Using CLISTS

You can use a CLIST to write a set of TSO option commands to be used whenever you want to run a VS FORTRAN Version 2 program under TSO. Running the CLIST runs your program with all of the options specified in the CLIST. You can use CLISTS to process your VS FORTRAN Version 2 programs either in the foreground or background.

CLISTS for Foreground Processing

The following CLISTS will link-edit and run a VS FORTRAN Version 2 program in the foreground.

CLIST 1:

```
PROC 1 NAME  
ALLOCATE FILE(FT06F001) DA(*) REUSE  
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD')SHR REUSE  
LOADGO &NAME LIB('SYS1.VSF2FORT')
```


CLIST 2:

```

PROC 1 NAME
LINK &NAME LOAD(temp(MAIN)) LIB('SYS1.VSF2FORT') LET MAP
ALLOCATE FILE(FT06F001) DA(*) REUSE
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD') SHR REUSE
CALL &NAME(MAIN)
DELETE temp

```

CLISTS for Background Processing

Although foreground processing can be convenient if you are running a small source program, running long source programs can take a long time. You may want to batch process such programs so that the CPU is free for other tasks. For information about cataloged procedures for background processing, see Figure 32 on page 84.

Dynamically Loaded User Subprograms

Executable VS FORTRAN programs usually exist as a single program module containing all user subprograms, intrinsic routines, and, if executing in link mode, all the necessary VS FORTRAN library routines to allow for the successful execution of the VS FORTRAN program.

To avoid creating very large program modules, VS FORTRAN provides load mode execution, whereby the necessary VS FORTRAN library routines are not included in the VS FORTRAN program module, but are dynamically loaded during execution. However, this can still result in large user program modules, especially when many user subprograms are involved.

Dynamic loading of user subroutines and functions provides a means to reduce the size of the program module containing the main VS FORTRAN program; using the dynamic loading capabilities of VS FORTRAN Version 2, each called user subprogram can reside in a separate loadable module that is loaded during execution. Once loaded, these dynamic modules remain resident for the duration of the execution of the VS FORTRAN program.

In order for a subroutine or function to be defined as a dynamically loaded module, the subprogram must be referenced from a VS FORTRAN Version 2 program, and that VS FORTRAN Version 2 program must specify the DYNAMIC compiler option, naming the subprogram in that option (or specifying the option as DYNAMIC(*) to cause all called user subprograms to be dynamically loaded).

In order to use dynamically loaded modules, the VS FORTRAN Version 2 program must execute in load mode, and each dynamic module must also be created to execute in load mode; dynamic modules may not be used when executing in link mode.

Example: The following is an example of a VS FORTRAN program that uses a dynamically loaded subroutine, SUBRTN:

```
@PROCESS DYNAMIC(SUBRTN)
PROGRAM MAIN
...
CALL SUBRTN
...
STOP
END
```

In separate load module named SUBRTN is this Fortran program unit:

```
SUBROUTINE SUBRTN
...
RETURN
END
```

Creating Dynamic Modules Under MVS

Under MVS, dynamically loadable modules are created using the linkage editor and placed in a library that is defined via a JOBLIB, STEPLIB, or FORTLIB DD statement during execution of the VS FORTRAN main program. See “Specifying Load Mode” on page 76 for a description of how to specify these DD statements.

Creation of dynamically loaded modules occurs in the same manner as for VS FORTRAN main program modules operating in load mode; see “Specifying Load Mode” on page 76 for a description of how to create modules operating in load mode.

Multiple Names

MVS allows a load module to have multiple names, called aliases. Each alias name defines an additional entry point in the load module at which the load module can be entered and from which execution can begin. The number of these additional entries is dependent on the level of MVS being used; early levels limit the number of alias names for a load module to 15.

In the VS FORTRAN environment, each alias name for a dynamically loaded module may be the name of a dynamically referenced subprogram. This allows multiple dynamically referenced subprograms to be packaged together into a single load module. However, to be effective and eliminate the need to load multiple copies of this load module, the VS FORTRAN program units should be compiled using the REENTRANT compiler option, and the resulting load modules containing the static program segment and the reentrant program segment built as for standard reentrant program units. See Chapter 21, “Creating Reentrant Programs” on page 423 for a description of reentrant programs in VS FORTRAN.

The following is an example of a dynamically loaded module with multiple names; each name is an entry into the module and the name of a dynamically called subprogram.

Example 1

```

@PROCESS DYNAMIC(SUBRTN,SUBRTN2)      SUBROUTINE SUBRTN
PROGRAM MAIN                          ...
...                                  RETURN
CALL SUBRTN                          END
...                                  SUBROUTINE SUB2
CALL SUB2                            ...
...                                  RETURN
STOP                                END
END

```

The program load modules are organized as follows:

```

MAIN                                SUBRTN
                                   SUB2 (alias)

```



Example 2: An alternative exists when the subprogram contains an additional entry using a Fortran ENTRY statement. Our previous example is modified to define SUBRTN as having SUB2 as an alternate entry:

```

@PROCESS DYNAMIC(SUBRTN,SUBRTN2)      SUBROUTINE SUBRTN
PROGRAM MAIN                          ...
...                                  RETURN
CALL SUBRTN                          ...
...                                  ENTRY SUB2
CALL SUB2                            ...
...                                  RETURN
STOP                                END
END

```

The program load modules are now organized as follows:

```

MAIN                                SUBRTN
                                   SUB2 (alias)

```



Creating Dynamic Modules Under CMS

Under CMS, dynamically loadable modules exist in one of four forms:

1. As a RELOCATABLE load module stored in a MODULE file. A RELOCATABLE load module is one that was created using the RLDSAVE option on the LOAD command.
2. As a load module stored in a LOADLIB created by the LKED command and accessed by a GLOBAL LOADLIB command.
3. As an object module stored in a TEXT file.
4. As an object module stored in a TXTLIB file and accessed by a GLOBAL TXTLIB command.

These four forms are not mutually exclusive; the same dynamically called module may exist in two or more of these forms simultaneously. Therefore, a search order exists by which these different files are examined to find each dynamically called module. This order is as listed above:

1. First, a search is made for a MODULE file containing a RELOCATABLE module, and if found, this file is used to load the dynamically called module.
2. If the module was not loaded from a MODULE file, the files specified in the last GLOBAL LOADLIB command are searched in order to find the module. If found, the module is loaded from the LOADLIB file.
3. If the module was not loaded from a LOADLIB file, a search for a TEXT file having the same name as the dynamically called module is made, and if found, this file is used to load the dynamically called module.
4. If the module was not loaded from a TEXT file, the files specified in the last GLOBAL TXTLIB command are searched in order to find the module; if found, the module is loaded from the TXTLIB file.

Loading TEXT or TXTLIB files

Dynamically called modules that are obtained from a TEXT or TXTLIB file are loaded using the CMS loader (the same process as the LOAD command).

If the dynamically loaded module statically references additional routines, these routines must also exist as, first, a TEXT file, or, second, in a GLOBAL TXTLIB file. Failure to find one of these routines in either a TEXT or TXTLIB file will cause CMS to abort loading the dynamically loaded module and to terminate execution of the VS FORTRAN program.

Similarly, if the dynamically loaded module references VS FORTRAN library routines or functions, then a GLOBAL TXTLIB command specifying the VSF2FORT TXTLIB file must have been issued prior to executing the VS FORTRAN program; otherwise the necessary VS FORTRAN library routines will not be found, and CMS will abort loading the dynamically loaded module and will terminate execution of the VS FORTRAN program.

Execution Considerations

All modules in the same VS FORTRAN executable program share the same copy of the VS FORTRAN Version 2 execution environment. Dynamically loaded subprograms do not have their own copy of the execution environment; they do however have their own copies of the mathematical intrinsic routines and of certain service subroutines.

Common blocks and any allocatable storage created within the VS FORTRAN program as the result of issuing an ALLOCATE statement are shared among all static and dynamically loadable subprograms in the same VS FORTRAN environment. Once loaded, dynamically loadable modules remain loaded for the duration of the execution of the main VS FORTRAN Version 2 program; no provision is provided to delete or reload modules.

Common blocks

VS FORTRAN Version 2 Release 5 causes all common blocks, whether static, dynamic or extended, to be shared among all program units in the VS FORTRAN program. Program units compiled by a level of VS FORTRAN prior to VS FORTRAN Version 2 Release 5 do not contain the necessary support to share access to static common blocks among dynamic modules and must be recompiled with VS FORTRAN Version 2 Release 6 to avoid unpredictable results. If a static common block is referenced in a program unit residing in a dynamically loaded module, all program units that reference that static common block must have been compiled by VS FORTRAN Version 2 Release 5 or later; results are unpredictable if a program unit compiled before Release 5 references a static common block that is also referenced by a dynamically loaded module.

Multiple Copies of Subprograms

It is advisable to have each dynamic subprogram contained in a separate module. This will ensure that only one copy of the subprogram is available for execution. If the same subprogram is contained in more than one dynamically loaded module, you may not obtain the results you expect, since different values of local data may be contained in each copy. To ensure that you have the same results as would occur when dynamically loaded modules are not used, create a separate module for each subprogram which is to be dynamically loaded.

Using Dynamically Loaded Modules

The name of the dynamically loaded module must be the name used in the CALL statement or function reference in the VS FORTRAN program to access the subroutine or function.

For example, given the program unit:

```
SUBROUTINE SUBRTN
...
RETURN
END
```

then a load module with the name SUBRTN can be created; when a VS FORTRAN program calls SUBRTN (and has specified the appropriate DYNAMIC compiler option), the load module SUBRTN will be loaded and the subroutine SUBRTN will be executed.

A program module may contain more than one subprogram. In the prior example, if SUBRTN referenced additional subprograms, those subprograms, if not themselves dynamically referenced, would be included in the load module SUBRTN. For example, if SUBRTN references two additional subprograms, subroutine SUB2 and function FCN3, e.g.:

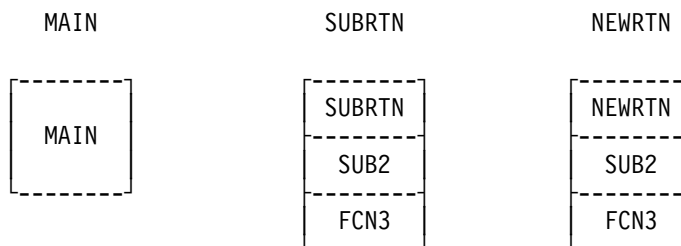
```
SUBROUTINE SUBRTN
...
CALL SUB2
...
R2 = FCN3(R)
...
RETURN
END
```

then the load module SUBRTN would also contain the subprograms SUB2 and FCN3.

If another subprogram, NEWRTN, were defined similarly to SUBRTN, e.g.:

```
SUBROUTINE NEWRTN
...
CALL SUB2
...
RZ = FCN3(RX)
...
RETURN
END
```

and NEWRTN were also a dynamically loaded subprogram, the load module NEWRTN would also contain copies of the SUB2 and FCN3 subprograms. This would result in three load modules, MAIN, SUBRTN and NEWRTN, organized as follows:



Note that the main program module may also have copies of SUB2 and FCN3; this would then result in three separate and unique versions of SUB2 and FCN3 to exist during the execution of the VS FORTRAN program.

Often the existence of multiple copies of subprograms is not of concern. However, if a subprogram (SUB2 or FCN3) were to contain static variables that are initialized and then modified on subsequent executions, or a variable that is expected to retain a value over multiple execution invocations of the subprogram, then a problem arises with multiple copies of the subprogram existing in the VS FORTRAN program execution. Each copy would have different occurrences of the static variables; the static variables would not be shared among all copies of the subprogram.

For example:

<pre>@PROCESS DYNAMIC(SUBRTN,NEWRTN) PROGRAM MAIN ... CALL SUBRTN ... CALL NEWRTN ... CALL SUB2 ... STOP END</pre>	<pre>SUBROUTINE SUBRTN ... CALL SUB2 ... RETURN END</pre>	<pre>SUBROUTINE NEWRTN ... CALL SUB2 ... RETURN END</pre>
--	---	---

where subroutine SUB2 which maintains a counter, ICNT, of the number of times it has been called, is defined as:

```

SUBROUTINE SUB2
  INTEGER*4 ICNT /0/
  ...
  ICNT = ICNT + 1
  ...
  RETURN
END

```

then the static variable ICNT would be unique to each existence of SUB2, and would **not** contain the total number of times SUB2 was called during the actual execution of all parts of the VS FORTRAN program.

In order for a variable to be shared among dynamically loaded modules, including any subprograms included in those modules, the variable must reside in a Common block.

Chapter 5. Using the Run-Time Options and Identifying Run-Time Errors

Available Run-Time Options 105

Abbreviations for Run-Time Options 106

Establishing a Default Run-Time Options Table 117

Identifying Run-Time Errors 118

 Using the Optional Traceback Map 119

 Using Program Interrupt Messages 121

 Requesting an Abnormal Termination Dump 122

 Operator Messages 122

Extended Error Handling 123

 Extended Error Handling by Default 123

 Controlled Extended Error Handling—CALL Statements 123

 Effects of VS FORTRAN Version 2 Interactive Debug on Error Handling . . 125

Static Debug Statements 125

Object Module Listing—LIST Option 127

Formatted Dumps 129

Identifying Coding Errors 129

This chapter describes the run-time options and how to identify run-time errors.

Available Run-Time Options

Figure 36 lists the run-time options and the IBM-supplied defaults. There is a column in Figure 36 where you can note the installation-wide defaults set up by your system programmer, if they differ from those supplied by IBM. You can set the default values for the options for a single program by establishing default option tables. See “Establishing a Default Run-Time Options Table” on page 117 for instructions on how to set up this table for a single program. If you code conflicting run-time options (for example, SPIE and NOSPIE), the last value specified takes precedence. For more information on using the run-time options, see:

- “Running Your Program and Specifying Run-Time Options and User Parameters” on page 67 for CMS
- “Running Your Program and Specifying Run-Time Options and User Parameters” on page 84 for MVS
- “Running Your Program and Specifying Run-Time Options—CALL Command” on page 95 for MVS with TSO
- The *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Figure 36 (Page 1 of 2). VS FORTRAN Version 2 Run-Time Options

Option	Installation Default	Notes
ABSDUMP NOABSDUMP		
AUTOTASK(loadmod,numtasks) NOAUTOTASK)	NOAUTOTASK(1)	Available on MVS only
CNVIOERR NOCNVIOERR		
DEBUG NODEBUG		

Figure 36 (Page 2 of 2). VS FORTRAN Version 2 Run-Time Options

Option	Installation Default	Notes
DEBUNIT(<i>s1</i> [, <i>s2</i> ,...]) NODEBUNIT (MVS) DEBUNIT(<i>s1</i> [<i>s2</i> <i>s3</i>]) NODEBUNIT (CMS)		
ECPACK NOECPACK		
ERRUNIT(<i>number</i>)		
FAIL (ABEND RC ABENDRC)		Not supported for parallel processing
FILEHIST NOFILEHIST		
INQPCOPN NOINQPCOPN		
IOINIT NOIOINIT		
OCSTATUS NOOCSTATUS		
PARALLEL [(<i>numprocs</i>)] NOPARALLEL	NOPARALLEL(1)	Available on VM/XA and MVS only
PRTUNIT(<i>number</i>)		
PTRACE[(<i>options</i>)]		
PUNUNIT(<i>number</i>)		
RDRUNIT(<i>number</i>)		
RECPAD NORECPAD		
SPIE NOSPIE		
STAE NOSTAE		
XUFLOW NOXUFLOW		

Note:

1. These options cannot be changed at installation time. The value is always the IBM-supplied default.

Abbreviations for Run-Time Options

Abbreviations for the run-time options are accepted. In general, only a minimum portion of the option name need be specified to identify the option.

The following list shows each option and its minimum abbreviation. The portion of the option name in capital letters is the minimum part that must be specified. If more than the minimum portion is specified, it must form a valid abbreviation of the option.

<u>ABS</u> dump	<u>NOABS</u> dump
<u>AUTO</u> task	<u>NOAUTO</u> task
<u>CNV</u> ioerr	<u>NOCNV</u> ioerr
<u>DEBUG</u>	<u>NODEBUG</u>
<u>DEBU</u> nit	<u>NODEBU</u> nit
<u>EC</u> Pack	<u>NOEC</u> Pack
<u>ERR</u> unit	
<u>FAIL</u>	
<u>FI</u> lehist	<u>NOFI</u> lehist
<u>INQ</u> pcopn	<u>NOINQ</u> pcopn
<u>IO</u> init	<u>NOIO</u> init
<u>OC</u> status	<u>NOOC</u> status
<u>PAR</u> allel	<u>NOPAR</u> allel
<u>PT</u> Race	<u>NOPT</u> Race
<u>PRT</u> unit	
<u>PUN</u> unit	
<u>RDR</u> unit	
<u>REC</u> pad	<u>NOREC</u> pad
<u>SP</u> ie	<u>NO</u> SPie
<u>STA</u> e	<u>NO</u> STAe
<u>XU</u> flow	<u>NOXU</u> flow

For example, CNVIO is an accepted abbreviation for the CNVIOERR option because it includes the minimum characters CNV and is a proper subset of the complete option name.

ABSDUMP | NOABSDUMP

Specifies whether post-ABEND symbolic dump information will be printed in the event of an abnormal termination.

AUTOTASK(*loadmod,numtasks*) | NOAUTOTASK (MVS only)

Specifies whether the multitasking facility (MTF) is enabled for your program. See Chapter 20, “The Multitasking Facility (MTF)” on page 397 for more information on the multitasking facility.

AUTOTASK(*loadmod,numtasks*)

Enables the multitasking facility (MTF) for your program.

loadmod

Is the name of the load module that contains the concurrent subroutines that will be run in the subtasks created by MTF.

AUTOTASK and PARALLEL are conflicting run-time options; if both are specified, the last one specified is used. If AUTOTASK is active the following are detected as errors when the program runs:

- Parallel language constructs
- DO loops compiled for automatic parallel processing
- Library event services
- Library lock services

numtasks

Is the number of subtasks created by MTF. This value may range from 1 through 99.

NOAUTOTASK

Nullifies the effects of previous specifications of AUTOTASK parameters so that MTF will not be enabled.

CNVIOERR | NOCNVIOERR

Specifies whether input conversion errors 206, 212, 213, 215, 225, 226, and 238 will be treated as I/O errors. (See *VS FORTRAN Version 2 Language and Library Reference* for additional information.)

CNVIOERR

Causes the ERR and IOSTAT specifiers on the READ statement to be honored, for the errors listed above. (ERR and IOSTAT are also honored on formatted WRITE statements for error 212.)

NOCNVIOERR

Causes the ERR and IOSTAT specifiers not to be honored for the errors listed above.

DEBUG | NODEBUG

Specifies whether VS FORTRAN Version 2 interactive debug will be invoked. It can be specified by CMS and TSO users at run time. For more information, see *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Note: If you specify the DEBUG run-time option and the PARALLEL run-time option with more than one processor, your program terminates with return code 16 and an error message is issued.

DEBUNIT(s1[,s2,...]) | NODEBUNIT (MVS Format)

DEBUNIT(s1[s2 s3]) | NODEBUNIT (CMS Format)

Specifies a list of Fortran units that are to be treated like terminal units for debugging. VS FORTRAN Version 2 interactive debug is able to capture terminal input and output and merge it with the debugging input and output. However, in batch mode (on MVS), units cannot be allocated to the terminal. DEBUNIT, therefore, provides the means for selecting certain units to be considered as terminal units for debugging purposes.

The list of units may consist of a single Fortran unit number and/or a range of unit numbers (x, or yy-zz).

The VS FORTRAN Version 2 interactive debug TERMIO command specifies whether I/O for these units are to be handled by interactive debug or by the VS FORTRAN Version 2 library in its normal manner. See *VS FORTRAN Version 2 Interactive Debug Guide and Reference* for details.

NODEBUNIT nullifies the effects of the previous specifications of DEBUNIT parameters.

ERRUNIT(number)

Identifies the unit number to which run-time error information is to be directed.

number

Is a valid unit number in the range: 0 <= number <= limit (where limit is either 99 or the highest allowed unit number, as set by the installation when VS FORTRAN Version 2 is installed).

The default error unit number is 6 unless it is changed by the installation when VS FORTRAN Version 2 is installed.

ECPACK | NOECPACK

Specifies whether a data space should be filled with as many extended common blocks as possible before a new data space is allocated.

ECPACK

Packs extended common blocks into the fewest number of data spaces to reduce some of the overhead associated with referencing data spaces.

NOECPACK

Places each extended common block in a separate data space. As a result, reference errors made beyond the bounds of an extended common block might be detected, with a minimal increase in run time.

FAIL (ABEND | RC | ABENDRC)

Indicates how unsuccessfully executed programs are to be terminated.

ABEND

Causes a program to end by abnormal termination with a user completion code of 240.

RC

Causes a program to end normally, but with a return code of 16.

ABENDRC

Causes a program to end by abnormal termination if the operating system detects a failure, and causes a program to end with a return code of 16 if VS FORTRAN detects a failure.

This option is not supported for parallel processing.

FILEHIST | NOFILEHIST

Specifies whether to allow the file definition of a file referred to by a ddname to be changed during run time. This option is intended for use with mixed language applications. (See "Ignoring File History" on page 155.)

FILEHIST

Causes the history of a file to be used in determining its existence. It checks to see whether:

- The file was ever internally opened (in which case it exists)
- The file was deleted by a CLOSE statement (in which case it does not exist).

When FILEHIST is specified, you cannot change the file definition of a file during run time and have the same results produced as in previous VS FORTRAN releases.

NOFILEHIST

Causes the history of a file to be disregarded in determining its existence.

If you specify NOFILEHIST you should consider:

- **If you change file definitions during run time:** the file is treated as if it were being opened for the first time. Note that before the file definition can be changed, the existing file must be closed.
- **If you do not change file definitions during run time:** you must use STATUS=NEW to re-open an empty file that has been closed with STATUS=KEEP, because the file does not appear to exist to Fortran.

INQPCOPN | NOINQPCOPN

Controls whether the OPENED specifier on an INQUIRE by unit can be used to determine whether a preconnected unit has had any I/O statements directed to it.

INQPCOPN

Causes the execution of an INQUIRE by unit to provide the value *true* in the variable or array element given in the OPENED specifier if the unit is connected to a file. This includes the case of a preconnected unit, which can be used in an I/O statement without executing an OPEN statement, even if no I/O statements have been executed for that unit.

NOINQPCOPN

Causes the execution of a INQUIRE by unit to provide the value *false* for the case of a preconnected unit for which no I/O statements other than INQUIRE have been executed.

IOINIT | NOIOINIT

Specifies whether the normal initialization for I/O processing will occur during initialization of the run-time environment. If you choose NOIOINIT:

- The error message unit will not be opened during initialization of the run-time environment. However, this does not prevent I/O from occurring on this or on any other unit. (Such I/O may fail if proper DD statements or FILEDEF statements are not given.)
- Under CMS, the CMS FILEDEF commands for the reader, punch, and printer will not be issued. Should subsequent I/O be directed to these units, the default FILEDEFs that are provided by CMS, not by VS FORTRAN, will be used.

OCSTATUS | NOOCSTATUS

Controls whether the OPEN and CLOSE status specifiers will be verified.

OCSTATUS

Specifies that file existence will be checked with each OPEN statement to verify that the status of the file is consistent with STATUS='OLD' and STATUS='NEW'; and specifies that file deletion will occur with each CLOSE statement with STATUS='DELETE' for those devices which support file deletion. Preconnected files are included in these verifications. OCSTATUS consistency checking applies to DASD files, PDS members, VSAM files, MVS labeled tape files, and dummy files only. For dummy files, the consistency checking occurs only if the file was successfully opened previously in the current program.

In addition, when a preconnected file is disconnected by a CLOSE statement, an OPEN statement is required to reconnect the file under OCSTATUS. Following the CLOSE statement, the INQUIRE statement parameter OPENED will indicate that the unit is disconnected.

NOOCSTATUS

Bypasses file existence checking with each OPEN statement and bypasses file deletion with each CLOSE statement.

If STATUS='NEW', a new file is created; if STATUS='OLD', the existing file is connected.

If STATUS='UNKNOWN' or 'SCRATCH', and the file exists, it is connected; if the file does not exist, a new file is created.

In addition, when a preconnected file is disconnected by a CLOSE statement, an OPEN statement is *not* required to reestablish the connection under NOOCSTATUS. A sequential READ, WRITE, BACKSPACE, REWIND, or ENDFILE will reconnect the file to a unit. Before the file is reconnected, the INQUIRE statement parameter OPENED will indicate that the unit is disconnected; after the connection is reestablished, the INQUIRE statement parameter OPENED will indicate that the unit is connected.

PRTUNIT(*number*)

Identifies the unit number that is to be used for PRINT and WRITE statements that do not specify a unit number.

number

Is a valid unit number in the range: $0 \leq \text{number} \leq \text{limit}$ (where limit is either 99 or the highest allowed unit number, as set by the installation when VS FORTRAN Version 2 is installed).

The default print unit number is 6 unless changed by the installation when VS FORTRAN Version 2 is installed.

PTRACE[*(options)*] | NOPTTRACE

Enables the Parallel Trace Facility and causes the Trace Log File to be initialized. To control the activity of the Parallel Trace Facility and which actions are recorded in the Trace Log File, suboptions of the PTRACE option can be specified.

The values for the *options* specification, with the default value indicated by underlining, are:

ALL | NONE
COMMON | NOCOMMON
EVENTS | NOEVENTS
THREADS | NOTHREADS
LOCKS | NOLOCKS
PCALL | NOPCALL
PDO | NOPDO
PSECTION | NOPSECTION
PROGRAM | NOPROGRAM
TASKS | NOTASKS
USER | NOUSER
TIMES | NOTIMES

where:

ALL | NONE

ALL indicates that trace records for all categories are to be included in the trace file.

NONE indicates that no trace records are included in the trace file.

COMMON | NOCOMMON

COMMON indicates that trace records for all common block activity are to be included in the trace file.

NOCOMMON indicates that no trace records for common block activity are to be included in the trace file.

EVENTS | NOEVENTS

EVENTS indicates that trace records for all user-initiated event activity are to be included in the trace file.

NOEVENTS indicates that no trace records for user-initiated event activity are to be included in the trace file.

THREADS | NOTHREADS

THREADS indicates that trace records for all internal processing for execution thread activity are to be included in the trace file.

NOTHREADS indicates that no trace records for internal processing for execution thread activity are to be included in the trace file.

LOCKS | NOLOCKS

LOCKS indicates that trace records for all user-initiated lock activity are to be included in the trace file.

NOLOCKS indicates that no trace records for user-initiated lock activity are to be included in the trace file.

PCALL | NOPCALL

PCALL indicates that trace records for all parallel call activity are to be included in the trace file.

NOPCALL indicates that no trace records for parallel call activity are to be included in the trace file.

PDO | NOPDO

PDO indicates that trace records for all parallel loop activity are to be included in the trace file.

NOPDO indicates that no trace records for parallel loop activity are to be included in the trace file.

PSECTION | NOPSECTION

PSECTION indicates that trace records for all parallel section activity are to be included in the trace file.

NOPSECTION indicates that no trace records for parallel section activity are to be included in the trace file.

PROGRAM | NOPROGRAM

PROGRAM indicates that trace records for all user program entry and exit activity are to be included in the trace file.

NOPROGRAM indicates that no trace records for user program entry and exit activity are to be included in the trace file.

TASKS | NOTASKS

TASKS indicates that trace records for all user-initiated task activity are to be included in the trace file.

NOTASKS indicates that no trace records for user-initiated task activity are to be included in the trace file.

USER | NOUSER

USER indicates that trace records resulting from all user calls to the PTWRIT service routine are to be included in the trace file.

NOUSER indicates that no trace records resulting from user calls to the PTWRIT service routine are to be included in the trace file.

TIMES | NOTIMES

TIMES indicates that trace records should have the current cpu time for the virtual processor included in trace records.

NOTIMES indicates that trace records are not to have the current cpu time for the virtual processor included in trace records.

The ALL and NONE options are allowed together with the other options. This allows you to ensure that all options are set to a particular state before using a specific option to alter a certain tracing category, and frees you from having to specify each option individually.

For a description of the specific trace categories controlled by each suboption, see “Program Tracing Categories” on page 328.

The following abbreviations are allowed:

Suboption	Abbreviation	Suboption	Abbreviation
COMMON	CO	NOCOMMON	NOCO
EVENTS	EV	NOEVENTS	NOEV
THREADS	TH	NOTHEADS	NOTH
LOCKS	LO	NOLOCKS	NOLO
PCALL	PC	NOPCALL	NOPC
PDO	PD	NOPDO	NOPD
PSECTION	PS	NOPSECTION	NOPS
PROGRAM	PR	NOPROGRAM	NOPR
TASKS	TA	NOTASKS	NOTA
USER	US	NOUSER	NOUS
TIMES	TI	NOTIMES	NOTI

PUNUNIT(*number*)

Identifies the unit number that is to be used for PUNCH statements that do not specify a unit number.

number

Is a valid unit number in the range: $0 \leq \text{number} \leq \text{limit}$ (where limit is either 99 or the highest allowed unit number, as set by the installation when VS FORTRAN Version 2 is installed).

The default punch unit number is 7 unless changed by the installation when VS FORTRAN Version 2 is installed.

PARALLEL [(*numprocs*)] | NOPARALLEL

Specifies whether your program will run in the parallel processing environment.

PARALLEL [(*numprocs*)]

Identifies a program to be run in parallel mode and specifies the number of virtual processors to be used for parallel processing.

PARALLEL and AUTOTASK are conflicting run-time options; if both are specified, the last one specified is used. If neither PARALLEL nor AUTOTASK is specified and the program contains parallel code, the program will run in parallel mode on one virtual processor and a warning message will be issued.

numprocs

Is the number of virtual processors to be used for parallel processing and must be between 1 and 99. The default value for *numprocs* is 1. Before choosing any number of processors on which to run your parallel program, you should check with your system programmer.

When *numprocs* is specified as 1, different parallel threads will not run concurrently. However, running your parallel program on a single virtual processor is not the same as running a program serially, because of the existence of parallel language statements and synchronization subroutines in your program.

Note: If you specify the PARALLEL run-time option with more than one processor and the DEBUG run-time option, your program terminates with return code 16 and an error message is issued. For information about debugging parallel programs, see page 296.

NOPARALLEL

Identifies a program to be run in serial mode.

If you specify the NOPARALLEL run-time option and your program contains parallel language constructs and was compiled with the PARALLEL(LANG) compile-time option, it will run PARALLEL(1) and a warning message will be issued.

RECPAD | NORECPAD

Specifies whether a formatted input record is padded with blanks.

RECPAD

Causes a formatted input record within an internal file or a varying/undefined length record (RECFM=U or V) external file to be padded with blanks when an input list and format specification require more data from the record than the record contains. Blanks added for padding are interpreted as though the input record actually contains blanks in those fields.

NORECPAD

Specifies that an input list and format specification must not require more data from an input record than the record contains. If more data is required, error message AFB212I is issued.

Note: The PAD specifier of the OPEN statement can be used to indicate padding for individual files. See *VS FORTRAN Version 2 Language and Library Reference* for information about using the PAD specifier and its interaction with RECPAD.

RDRUNIT(number)

Identifies the unit number that is to be used for READ statements that do not specify a unit number unit.

number

Is a valid unit number in the range: $0 \leq \text{number} \leq \text{limit}$ (where limit is either 99 or the highest allowed unit number, as set by the installation when VS FORTRAN Version 2 is installed).

The default read unit number is 5 unless changed by the installation when VS FORTRAN Version 2 is installed.

SPIE | NOSPIE

Specifies whether the run-time environment will take control in the event of program interrupts.

SPIE

Causes a SPIE (or ESPIE) macro instruction to be run during initialization of the run-time environment in order to allow the run-time environment to take control in the event of program interrupts.

NOSPIE

Suppresses execution of the SPIE (or ESPIE) macro instruction. If NOSPIE is specified, various run-time functions that are dependent on control being returned for a program interrupt are not available. These include:

- Messages and corrective action for a floating-point overflow
- Messages and corrective action for a floating-point underflow interrupt (unless the underflow is to be handled by the hardware based on the XUFLOW option)
- Messages and corrective action for a floating-point or fixed-point divide exception
- Simulation of extended precision floating-point operations on processors that do not have these instructions
- Realignment of vector operands which are not on the required storage boundaries and the re-running of the failing instruction.

Instead of the corrective action indicated, abnormal termination results. In this case, either the STAE or NOSTAE option, whichever is in effect, governs whether the run-time environment gains control at the time of the ABEND.

VS FORTRAN Version 2 interactive debug requires a SPIE exit for some of its operations and will continue to use a SPIE exit even though NOSPIE has been specified; however, unpredictable results may occur.

STAE | NOSTAE

Specifies whether the run-time environment takes control in the event of an abnormal termination.

STAE

Causes a STAE (or ESTAE) macro instruction to be run during initialization of the run-time environment in order to allow the run-time environment to take control in the event of abnormal termination.

NOSTAE

Suppresses running of the STAE (or ESTAE) macro instruction. If NOSTAE is specified, abnormal termination is handled by the operating system rather than by the run-time environment. In this case:

- Message AFB240I, which shows the PSW and register contents at the time of the ABEND, is not printed. However, some of this information may be provided by the operating system.
- Internal sequence number (ISN), or the sequence number of the last-run Fortran statement, is not printed.
- The traceback of called routines is not printed.
- The post-ABEND symbolic dump is not printed, even with the ABDUMP option in effect.

- Certain exceptional conditions that are handled by the run-time environment or by the debugging device causes system ABENDs rather than VS FORTRAN Version 2 messages. For example, some errors that occur during the processing of an OPEN statement result in a system ABEND rather than the printing of message AFB219I, which allows possible continuations of program processing.
- While using interactive debug to debug your program, if you use the QUIT command to terminate the program following an attention interrupt, a user ABEND 500 occurs instead of the normal termination of the run-time environment.
- An MTF subtask that terminates unexpectedly causes a user ABEND 922 in the main task, rather than message AFB922I.

There is an exception to the above list of items to be printed during termination of the run-time environment. If the NOSTAE and SPIE options are both in effect and a program interrupt occurs, then the following items are printed:

- Message AFB240I
- Internal sequence number (ISN) or sequence number of the last-run Fortran statement
- Traceback of called routines
- Post-ABEND symbolic dump (if allowed by the ABSDUMP option).

XUFLOW | NOXUFLOW

Specifies whether an exponent underflow should cause a program interrupt. (An exponent underflow is produced when a floating-point number becomes too small to be represented.)

XUFLOW

Allows an exponent underflow to cause a program interrupt, followed by a message from the VS FORTRAN Version 2 run-time library, followed by standard fix-up. For information about how XUFLOW behaves in the parallel environment see “Considerations for Existing Service Subroutines” on page 324.

NOXUFLOW

Suppresses the program interrupt caused by an exponent underflow. The hardware provides the fix-up. Using this option reduces processing time.

Both the standard fix-up done by the run-time library and the fix-up done by the hardware for the exponent underflow set the result to zero.

VS FORTRAN Version 2 provides a subroutine that allows you to suppress or enable the program interrupt that occurs because of exponent underflow. This subroutine, XUFLOW, can be used at any point in your program. Calling XUFLOW as follows:

```
CALL XUFLOW (0)
```

suppresses the program interrupt and is equivalent to the action taken by the NOXUFLOW option. Calling XUFLOW as follows:

```
CALL XUFLOW (1)
```

allows the program interrupt to occur and is equivalent to the action taken by the XUFLOW option.

Establishing a Default Run-Time Options Table

There are three ways to specify run-time options. They are listed in the order of the priority taken when there are conflicting options present. Run-time options can be specified as follows:

1. On the control statement that runs your program:
 - For CMS, see “Running Your Program and Specifying Run-Time Options and User Parameters” on page 67
 - For MVS, see “Running Your Program and Specifying Run-Time Options and User Parameters” on page 84
 - For TSO, see “Running Your Program and Specifying Run-Time Options—CALL Command” on page 95.
2. In the default run-time options table (AFBVLPRM) used for a single program, as assembled by the VSF2PARM macro.
3. In the default run-time options table established for your installation.

Only the default run-time options table for a single program, AFBVLPRM, will be discussed in detail in this book. The installation-wide default options table may have been established by your system programmer.

You can create a default run-time options table for a single program by coding the desired options in a VSF2PARM macro instruction. When assembled, this produces an object module called AFBVLPRM. This object module is to be included with your program. To generate and use this options table, use the following steps:

1. Code the macro instruction in the following format:


```
VSF2PARM option[,option...]
```

One or more run-time options can be specified if you wish to override your installation's default values. The possible options are listed in the preceding pages.
2. Assemble the AFBVLPRM module using the VSF2PARM macro instruction you have written in step 1. To assemble the module, make the VS FORTRAN Version 2 library that contains the VSF2PARM macro definition available to the assembler.

If you are running under MVS and have Assembler H Version 2 available, code the assembly as follows:

```
//ASSEM EXEC PGM=IEV90,PARM='OBJECT,NODECK'  
//SYSPRINT DD SYSOUT=A  
//SYSLIB DD DSN=SYS1.AFBLBS,DISP=SHR  
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))  
//SYSLIN DD DSN=&&VLPRM1,DISP=(NEW,PASS),  
// UNIT=SYSDA,SPACE=(TRK,(1,1)),  
// DCB=BLKSIZE=3200  
//SYSIN DD *  
VSF2PARM option[,option]  
/*
```

If you are running on a non-XA version of MVS and do not have Assembler H Version 2 available, use IFOX00 as the name of the assembler rather than IEV90. In this case, also add DD statements to define work files for SYSUT2 and SYSUT3.

Under CMS, code your VSF2PARM macro instruction in a file with a file name of your choice and a file type of ASSEMBLE. If, for example, your file is called VLPRM1 ASSEMBLE, then you can code the assembly as follows:

```
GLOBAL MACLIB VSF2MAC  
HASM VLPRM1
```

This produces a text file called VLPRM1 TEXT.

Note: For access to VSF2MAC, the VS FORTRAN macro library, see your system programmer.

3. Include the object module produced from the assembly of the VSF2PARM macro instruction when you create your executable program.

Under MVS, you must provide the object module as link editor input so that it will be included in your load module. For example, continuing the example in step 2 above, you can perform a compile and link-edit as follows:

```
//CL EXEC PROC=VSF2CL  
//FORT.SYSIN DD Your source program  
//LKED.SYSIN DD DSN=&&VLPRM1,DISP=(OLD,DELETE)
```

Under CMS, include the text file of the options table when you issue the LOAD command.

For example, if you have a text file called VLPRM1 as in step 2 above, then issue the following command:

```
LOAD myprog VLPRM1
```

where myprog is the name of the text file that contains your program.

Identifying Run-Time Errors

VS FORTRAN Version 2 has a number of features that help you find errors. One feature, VS FORTRAN Version 2 interactive debug, is described in the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*. Other debugging aids are described in the following sections. For information on run-time messages (AFB001-AFB971), see *VS FORTRAN Version 2 Language and Library Reference*.

Using the Optional Traceback Map

Whenever you receive a library diagnostic message, you can, optionally, request a traceback map. Your site may have set this as the default whenever a library message is generated. If not, you can request a traceback map for any message, using the ERRSET subroutine. You can also get a traceback map at any point in your source program by using the ERRTRA subroutine.

For more information on these subroutines, see “Controlled Extended Error Handling—CALL Statements” on page 123.

To cause ISNs or offsets to appear in a traceback map, you must have compiled with the SDUMP compile-time option (see “Available Compile-Time Options” on page 21).

The traceback map is a tool to help you find where an error occurred in your program. The information in the map starts from the most recent instruction run and ends with the origin of the program. The traceback routine processes up to 12 programs or subroutines, and can detect a bad save area or a loop in the save area chain.

The sample traceback map in Figure 37 lists the names of called routines, internal statement numbers (ISNs) within routines, and the arguments received by each subroutine.

```

1 AFB230I VSERH : SOURCE ERROR AT ISN 2 - EXECUTION TERMINATED. THE PROGRAM NAME IS "SUBB".
    VSERH : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM SUBB AT ISN 2 (OFFSET 0001CA).

TRACEBACK OF CALLING ROUTINES; MODULE ENTRY ADDRESS = 020000 2
-----
3 VSERH# (02097A) 4 CALLED BY SUBB (020458) 5 AT ISN 2 6 AT OFFSET 0001CA.
-----
SUBB (020458) CALLED BY SUBA (020208) AT ISN 2 AT OFFSET 0001BE.
7 ARGUMENT LIST AT 020390.
  ARG. NO.  ADDRESS      INTEGER      REAL      CHAR      HEXADECIMAL
          1  00020188 :           0  0.000000E+00  '....'  00000000
          2  80020188 :           0  0.000000E+00  '....'  00000000
-----
SUBA (020208) CALLED BY MAIN (020000) AT ISN 1 AT OFFSET 0001BE.
  ARGUMENT LIST AT 02018C.
8 ARG. NO.  ADDRESS      INTEGER      REAL      CHAR      HEXADECIMAL
          1  00020188 :           0  0.000000E+00  '....'  00000000
          2  80020188 :           0  0.000000E+00  '....'  00000000
-----
MAIN (020000) CALLED BY OPERATING SYSTEM.
-----

AFB900I VEMG2 : EXECUTION TERMINATING DUE TO ERROR COUNT FOR ERROR NUMBER 230

MESSAGE SUMMARY: MESSAGE NUMBER - COUNT

                        230      1

```

Figure 37. Sample Traceback Map

1 Error Messages

Lists the errors produced while running your program. The summary of errors printed at the end of the listing tells you how many times an error occurred.

The control program runs its own routine to attempt to recover from the error, and sometimes displays the following message:

STANDARD CORRECTIVE ACTION TAKEN, EXECUTION CONTINUING

If your program uses its own error recovery routine, the word USER replaces STANDARD in this message. After the error recovery, the program continues to run.

2 MODULE ENTRY ADDRESS = *address*

Shows the entry point of the earliest routine entered.

3 routine (*address*)

Lists the names of all routines entered in the current calling sequence with the routine entry address. In Figure 37 on page 119, the final routine that ran is SUBB, which begins at hexadecimal address 00020458.

Names are shown with the last routine called at the top and the first routine called at the bottom of the listing.

**4 CALLED BY routine (*address*)
CALLED BY OPERATING SYSTEM**

Lists the calling routine. The starting address of the calling routine follows the routine name. In Figure 37 on page 119, SUBA, which began at address 00020208, called routine SUBB, which began at address 00020458. Calls to the main program from the operating system are indicated by the CALLED BY OPERATING SYSTEM format.

5 AT ISN *number*

Lists the FORTRAN internal statement number (ISN) of the calling statement in the CALLED BY routine. ISN information is only available if a program unit has been compiled with SDUMP.

6 OFFSET *address*

Lists the hexadecimal offset within the routine that made the call.

7 ARGUMENT LIST AT *address*

Shows the address of the argument list passed to the called routine or the message, NO ARGUMENT PASSED TO SUBROUTINE.

8 ARG. NO. ADDRESS INTEGER REAL CHAR HEXADECIMAL

Lists the arguments by number, address, and content. A maximum of 99 arguments can be displayed in a traceback map. The contents of the first four bytes of each argument is displayed in four types of notation.

integer

0

real

0.000000E+00

character

'....'

hexadecimal

00000000

Traceback Map Procedure: To use the traceback map for error detection:

1. Look at the message text in the first line of the AFB message. If you need more explanation than the text provides, see *VS FORTRAN Version 2 Language and Library Reference*.
2. Find the last routine called by the program. It should be the first item under the traceback heading.
3. Use the ISN, SEQ. NO. or OFFSET on the same line to locate the statement within the CALLED BY routine in your source code.
4. Investigate the statement for proper use, and continue by analyzing the arguments within the routine.

If the statement is still not found, go through this procedure again, using the next oldest routine and so on, until the error is found.

The traceback map lists the internal statement number (ISN) calling each routine. For an example using ISNs, see Figure 13 on page 44. Using the ISN, you can locate the source statement within the calling module.

Using Program Interrupt Messages

During program execution, messages are generated whenever the program is interrupted. Program interrupt messages, which indicate the cause of the interrupt, are written in the compiler output.

The program interrupt message for the VS FORTRAN Version 2 library indicates the true exception that caused the termination; however, the completion code from the system always indicates the job termination resulted from a specification or operation exception.

When a program interrupt occurs in a program unit that was compiled with the SDUMP or TEST option, symbolic dumps of program data are automatically provided for your use in determining the cause of the interrupt. For more information on symbolic dumps, see "Requesting an Abnormal Termination Dump" on page 122.

Exception codes appear in the eighth digit of the PSW and indicate the reason for the interruption. Their meanings are as follows:

Code Meaning

- 1 Operation exception: the operation is not defined to the operating system.
- 2 Privileged-operation exception: the processor encounters a privileged instruction in the problem state.
- 3 Execute exception: the subject instruction of EXECUTE is another EXECUTE.
- 4 Protection exception: an illegal reference is made to an area of storage protected by a key.
- 5 Addressing exception: a reference is made to a storage location outside the range of storage available to the job.
- 6 Specification exception: for example, a unit of information does not begin on its proper boundary.

- 7** Data exception: the arithmetic sign or the digits in a number are incorrect for the operation being performed.
- 8** Fixed-point-overflow exception: a carry occurs out of the high-order bit position in fixed-point arithmetic operations, or high-order significant bits are lost during the algebraic left-shift operations.
- 9** Fixed-point-divide exception: an attempt is made to divide by zero.
- A** Decimal-overflow exception: one or more significant high-order digits are lost because the destination field in a decimal operation is too small to contain the result.
- B** Decimal-divide exception: the divisor in the decimal division is zero or the quotient exceeds the specified data field size.
- C** Exponent-overflow exception: a floating-point arithmetic operation produces a positive number that is too large for a register. A standard corrective action is taken and execution continues after generating the additional message:
REGISTER CONTAINED *number*
- D** Exponent-underflow exception: a floating-point arithmetic operation generates a number with a negative exponent too small for a register. A standard corrective action is taken and execution continues after generating the additional message:
REGISTER CONTAINED *number*
- E** Significance exception: the resulting fraction in floating-point addition or subtraction is zero.
- F** Floating-point-divide exception: an attempt is being made to divide by zero in a floating-point operation. A standard corrective action is taken and execution continues after generating the additional message:
REGISTER CONTAINED *number*
- 19** Operation exception for vector instructions: occurs when trying to run a vectorized program on a machine on which the vector facility is disabled.
- 1E** Unnormalized operands: a floating-point number in which the leading fraction digit is zero.

Requesting an Abnormal Termination Dump

How you request an abnormal termination dump depends on the system you're using.

For system considerations when requesting a dump, see "Requesting an Abnormal Termination Dump" on page 84 (MVS).

Information on interpreting dumps is in the appropriate debugging guide for your system.

Operator Messages

Operator messages are generated when your program issues 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 run.

The operator message may take the following forms:

Character	Meaning
<i>n</i>	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 run. (Nothing is printed for a similar STOP statement.)

A PAUSE message causes the program to stop running pending operator response. The format of the operator's response to the message depends on the system being used.

A STOP message causes program termination.

Extended Error Handling

Extended error handling can operate with default values, or you can control the values using service subroutines.

Extended Error Handling by Default

Your installation has a default value preset in the VS FORTRAN Version 2 error option table for the following run-time conditions associated with each error:

- The number of times an error can occur before the program is terminated
- The maximum number of times a run-time message is printed
- Whether a traceback map is to be printed with the message
- Whether a user error exit routine is to be called.

The actions of error handling are controlled by these settings in the error option table. IBM provides a standard set of option table entries; your system administrator may have provided additional entries for your site. The data in error (or some other associated information) is printed as part of the message text. For more information on run-time messages, see *VS FORTRAN Version 2 Language and Library Reference*.

Controlled Extended Error Handling—CALL Statements

To make changes to the option table dynamically at load module run time, you can use the service subroutines, summarized here.

The service subroutines let you change the extended error handling information in your copy of the option table, so that you get control that you specify over load module errors while your program runs:

- The ERRMON subroutine calls the error monitor routine, the same routine used by VS FORTRAN Version 2 when it detects an error.
- The ERRTRA subroutine dynamically requests a traceback and continued processing (see "Using the Optional Traceback Map" on page 119).
- The ERRSAV subroutine copies an option table entry into an area accessible to your program.
- The ERRSTR subroutine stores an entry in the option table.

- The ERRSET subroutine allows you to control processing when an error condition occurs.

For detailed reference documentation about the error option table and the service subroutines, see *VS FORTRAN Version 2 Language and Library Reference*.

Usage Notes for User-Controlled Error Handling:

1. The default settings of the error option table may be permanently changed in the VS FORTRAN Version 2 library by reassembling a macro and replacing the table in the library. Also, entries may be added to the table for installation-designated errors. If this has been done for your installation, your system administrator has information about it.
2. When you set option table entries, allow no more than 255 occurrences of any error; otherwise, infinite program looping can result.
3. If an error entry is set to allow no corrective action (neither standard nor user-exit-provided), the entry must also allow only one occurrence of the error before program termination.
4. Warning: Incorrect responses may result when changing the values of any variables in the common area while in a user error handling routine under optimization levels of 1, 2, or 3. Certain control flow and variable usage information are not known since the user error handling routine is called indirectly, not directly, when an error is encountered.

For example:

```
COMMON /A/ RETCDE
EXTERNAL ERRSUB
INTEGER RETCDE
CALL ERRSET(215,0,-1,1,ERRSUB)
1  READ(5,*,END=100)I
   IF (RETCDE.GT.0) GO TO 100
   WRITE(6,*)I
   GOTO 1
100 STOP
END
```

In the above example, RETCDE may be changed in ERRSUB when an invalid data error is encountered in the READ statement; however, this fact is hidden in the context of the program. Therefore, it is assumed that RETCDE is not changed between the READ and the GOTO 1 in the above example. Incorrect results may occur if it is changed to ERRSUB. If you specify IOSTAT in the READ statement, as follows:

```
1  READ(5,*,END=100,IOSTAT=RETCDE) I
```

it can be handled correctly.

5. Note that VS FORTRAN Version 2 does not detect fixed-point overflow. If an integer overflow condition occurs while evaluating an arithmetic expression that is used in an arithmetic IF statement, a branch is taken, although which branch is taken is uncertain.

If an integer overflow condition occurs while evaluating a DO loop iteration, the results may be unpredictable.

Effects of VS FORTRAN Version 2 Interactive Debug on Error Handling

If you are running with VS FORTRAN Version 2 interactive debug, error handling is modified as follows:

- Traceback maps are not produced for any error.
- The interactive debug error routine operates instead of the library error monitor (ERRMON).
- If your program calls ERRSET to provide a user exit routine, that user exit routine operates instead of the interactive debug error routine.
- If you are debugging in ISPF or line mode, unlimited error occurrences are allowed for all errors and error counts are not maintained.

For more information on error handling by interactive debug, see information on the ERROR command in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Static Debug Statements

The debug statements help you locate errors in your source program that are not diagnosed by the library. The debug statements, when used, must be the first statements in your program. If debug statements are used, the RENT compile-time option is ignored.

If you use a debug packet in your source program and compile it using OPTIMIZE(1), OPTIMIZE(2), OPTIMIZE(3), or VECTOR, the compiler changes the optimization parameter to NOOPTIMIZE and NOVECTOR.

Note: The static debug cannot be used with extended common. For more information on extended common, see “Extended Common” on page 379.

Figure 38 on page 126 shows how you can use VS FORTRAN Version 2 debug statements to obtain information for determining the cause of an error.

Figure 38. Using Static Debug Statements

1	DEBUG SUBCHK(ARRAY1), TRACE, INIT(ARRAY1)
2	AT 10
3	TRACE ON
4	(procedural code for debugging)
5	AT 40
6	TRACE OFF
7	DISPLAY I, J, K, L, M, N, ARRAY1
8	END DEBUG
	:
10	DO ... (program tracing begins here; procedural debugging code run)
	:
30	CONTINUE
40	WRITE ... (program tracing ends here; values of I, J, K, L, M, N, and ARRAY1 are displayed)

- 1** The DEBUG statement precedes the first debug packet and specifies the following:
 - SUBCHK(ARRAY1) requests validity checking for the values of ARRAY1 subscripts.
 - TRACE specifies that tracing is to be allowed within debug packets.
 - INIT(ARRAY1) specifies that ARRAY1 is to be displayed when values within it change.
- 2** AT 10 begins the first debug packet.
- 3** TRACE ON turns on program tracing at statement label 10, so that subsequent statements with Fortran labels are traced.
- 4** (Procedural debugging code contains valid Fortran statements to aid in debugging; for example, to initialize variables.)
- 5** AT 40 ends the first debug packet and begins the second.
- 6** TRACE OFF turns off program tracing at statement label 40, so that subsequent statements after 40 are not traced.
- 7** The DISPLAY statement writes the values of I, J, K, L, M, N, and ARRAY1.
- 8** END DEBUG ends the second (and last) debug packet.

In debug packets you can use the following statements:

```
DEBUG
AT
TRACE ON | TRACE OFF
DISPLAY
END DEBUG
```

In addition to these specific debug statements (valid only in a debug packet), you can also use most Fortran procedural statements to gather information about what's happening while the program runs.

Object Module Listing—LIST Option

The object module listing shows you (in pseudo-assembler format) the machine code the compiler generated from your source statements.

A sample object module listing is shown in Figure 39. (Some of the information in the listing has been realigned to fit the dimensions of the page.)

1	2	3	4	5
PROGRAM CODE				
000944 58F0 D848		1.000 L	G15,2120(0,G13)	VSPIL#
000948 4110 D10C		LA	G1,268(0,G13)	
00094C 05EF		BALR	G14,G15	
00094E 5870 D838		L	G7,2104(0,G13)	MAIN+000001E8
000952 4190 0258		LA	G9,600	600
000956 58B0 C860		2.000 L	G11,2144(0,G12)	2.000#
* ISN 2				
00095A 58A0 D0E4		L	G10,228(0,G13)	1
00095E 89A0 0002		SLL	G10,2	
000962 5830 D0D8		L	G3,216(0,G13)	4
000966 4C30 D830		MH	G3,2096(0,G13)	.V.VSS
00096A 5030 D85C		ST	G3,2140(0,G13)	.Q0002
00096E 4160 0096		LA	G6,150	150
000972 1883		LR	G8,G3	.Q0002
000974 A645 0060		1.107 VLVCU	G6	.Q0001
000978 410A 726C		LA	G0,620(G10,G7)	B
00097C A409 0000		VLE	V0,G0(0)	.Z.000,B
000980 410A 74C4		LA	G0,1220(G10,G7)	A
000984 A400 0000		VAE	V0,V0,G0(0)	.Z.001,.Z.000,A
000988 410A 7014		LA	G0,20(G10,G7)	C
00098C A40D 0000		VSTE	V0,G0(0)	C,.Z.001
* ISN 3				
000990 87A8 B01E		2.107 BXLE	G10,G8,30(G11)	1.107#
000994 58B0 C864		2.002 L	G11,2148(0,G12)	2.002#
* ISN 4				
000998 58F0 D844		L	G15,2116(0,G13)	VFWSL#
00099C 4110 D0FC		LA	G1,252(0,G13)	
0009A0 05EF		BALR	G14,G15	
0009A2 58F0 D840		L	G15,2112(0,G13)	VFIXL#
0009A6 4110 D104		LA	G1,260(0,G13)	
0009AA 05EF		BALR	G14,G15	
* ISN 5				
0009AC 58F0 D83C		L	G15,2108(0,G13)	VFEE#
0009B0 1B11		SR	G1,G1	
0009B2 05EF		BALR	G14,G15	
PROLOGUE CODE				
0009B4 58C0 3048		L	G12,72(0,G3)	
0009B8 1B11		SR	G1,G1	
0009BA 18D3		LR	G13,G3	
0009BC 58F0 384C		L	G15,2124(0,G3)	VFEIM#
0009C0 05EF		BALR	G14,G15	
0009C2 47F0 D868		BC	M15,2152(0,G13)	

Figure 39. Object Module Listing Example—LIST Compile-Time Option

If the LIST option has been specified the ISN number is printed before each source statement in the pseudo-assembler listing, including statements in vectorized loops.

Each line of the listing is formatted (from left to right) as follows:

- 1** The relative address of the instruction or the data item in hexadecimal.
- 2** The storage representation of the instruction or initialized data item, in hexadecimal format.
- 3** The 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*). Values are not always present in this area of the listing.
- 4** The pseudo-assembler language format for each statement. The following designators identify the register type:

G = general register
F = floating-point register
V = vector register
M = mask

For example:

G0 identifies general register 0.
F1 identifies floating-point register 1.
V3 identifies vector register 3.
M5 identifies a mask constant of 5.

Binary (hexadecimal) constants are printed as:

DC XLn'...'

Character constants are printed as:

DC CLn'...'

Adcons are printed as:

DC AL4(cccccccc)

These constants are displayed in the appropriate character set for the particular operating system. Unprintable data are printed as hex characters in the listing file.

- 5** The source program items referred to by the instruction, such as entry points of subprograms, variable names, or other statement labels.

Figure 39 on page 127 shows an example of an object module listing for which the reentrant feature has not been invoked.

The object module listing with the reentrant feature contains the same sections as those shown in Figure 39 on page 127; in addition, there can be one table in the reentrant listing that is not in the listing for nonreentrant: ADCONS (address constants) FOR REENTRANT RELOCATION.

Formatted Dumps

You can request various dumps while your program runs using the VS FORTRAN Version 2 dump subroutines: PDUMP, DUMP, CPDUMP, CDUMP, and SDUMP. For descriptions of these subroutines, see *VS FORTRAN Version 2 Language and Library Reference*.

Identifying Coding Errors

The VS FORTRAN compiler cannot identify all possible coding errors. If your program compiles successfully but does contain an error, it may not run successfully or it may provide an erroneous result. The following list identifies several coding errors, not detected by the compiler, that are likely to result in run-time problems:

1. Forgetting to assign values to variables and arrays before using them in your program.
2. Specifying subscript values that are not within the bounds of an array. In particular, if you assign data outside the array bounds you may inadvertently destroy data and instructions.
3. Moving data into an item that is too small for it, resulting in truncation.
4. Making invalid data references to EQUIVALENCE items of differing types (for example, integer and real).
5. Transferring control into the range of a DO loop from outside the range of the loop. The compiler will issue a warning message for all such branches if you specify OPT(2), OPT(3), or VECTOR.
6. Using arithmetic variables and constants that are too small to give the precision you need in the result. For example, if you want to obtain more than six decimal digit floating-point results, you should use double precision.
7. Failing to initialize the run-time environment when your main program is not a Fortran program. See Appendix B, “Assembler Language Considerations” on page 451.
8. Concatenating character strings in such a way that overlap can occur.
9. Trying to access services not available on the run-time operating system or hardware.

The MAP and XREF compile-time options and intercompilation analysis (ICA) are VS FORTRAN features that can help you identify many coding errors. See “Detecting Errors—MAP and XREF Options” on page 44 and “Intercompilation Analysis” on page 380.

Part 3. Performing Input/Output Operations

Chapter 6. I/O Concepts and Terminology

External and Internal Files	133
Units and File Connection	134
File Definitions and Dynamic File Allocation	134
Named Files	136
Unnamed Files	137
VS FORTRAN File Access Methods	138
Sequential Access	139
Direct Access	139
Keyed Access	139
Access Methods Used by the Operating System	139
Records As Seen by Fortran	140
Records As Seen by the Operating System	140
File Existence	141

This chapter explains several concepts and terms associated with Fortran files and I/O processing that are used within this section.

External and Internal Files

To your Fortran program, a *file* is a sequence of records that can be processed as a single entity. Fortran deals with two types of files: external files and internal files.

An *external file* is a file that is stored on an input/output device, such as a tape, disk, or terminal. However, a file from the Fortran point of view doesn't necessarily correspond to what the operating system views as a file or even to what is stored on some physical storage medium.

For example, your Fortran program could read records from a file for which you enter the data at a terminal. The records aren't really stored on the terminal, but since the data that you enter is available to your program as a sequence of records, that data is seen as a Fortran file. Similarly, your program could read data from a file that is a concatenation of several collections of records that the operating system might view as individual files (or data sets). Because all of the records are available to your program as a sequence of records, they constitute a Fortran file.

An *internal file* is located in main storage and is a character variable, character array element, character array, or character substring. It can contain either data that you create with your program or data that you transfer from an external file. Internal files are useful when you don't know the arrangement of data within the records of an external file. Transferring the data into an internal file allows your program to examine a part of a record and, based on some condition within the record, process the rest of the record accordingly. Furthermore, the record can be reread many times, if required, without the need to BACKSPACE and READ from the physical device again.

Units and File Connection

Fortran uses *units* as a means of referring to files. Before you can write data to or read data from a file, the file must be associated with—that is, *connected to*—a unit. There are two ways to connect a file to a unit. You can connect a file to a unit within your program by coding an OPEN statement or you can preconnect a file, in which case it is automatically connected when your program begins to run.

A file may be connected to only one unit at a time, and a unit may be connected to only one file at a time. You can, however, disconnect a file from a unit and reconnect it to a different unit in the same program.

Once a file has been connected to a unit, you refer to that file indirectly by referring to the unit. Every I/O statement, except INQUIRE, must have a UNIT specifier. The UNIT specifier has the form UNIT=*un*, where *un* is the unit identifier.

For external files, *un* is either an integer expression or an asterisk (*). When an integer expression is given, its value is the unit number. The asterisk (*) specifies the standard input or output unit. For a READ statement, the IBM-supplied default for the standard input unit is unit 5. For a WRITE statement, the IBM-supplied default for the standard output unit is unit 6.

For internal files, *un* is the name of an internal storage area containing the file.

In the following example, the external file TAXRATES is connected to unit 14 with the OPEN statement. It is then referred to only by the unit number in the READ statement.

```
OPEN (UNIT=14, FILE='TAXRATES')  
READ (UNIT=14, FMT=100) A, B, C
```

To disconnect a file, you usually code a CLOSE statement. A file is automatically disconnected from a unit if you connect a different file to the same unit using the OPEN statement with the FILE specifier. In addition, all files that remain connected at program termination are automatically disconnected.

File Definitions and Dynamic File Allocation

A *file definition* must be in effect to establish the linkage between your program and the file. Named files are specified by including the ddname of the file on the FILE specifier of the OPEN statement. Unnamed files do not use the FILE specifier of the OPEN statement, however, a file definition is in effect for the unit specified on the UNIT specifier. For example:

Named file not dynamically allocated:

```
OPEN (UNIT=10, FILE='MYDD1')
```

How you provide a file definition depends on your operating system:

- For CMS, you use the FILEDEF command for non-VSAM data sets and the DLBL command for VSAM data sets. The DLBL command is explained in *VM/SP CMS User's Guide*.

- For non-VSAM data sets, you can depend on a default file definition when you don't provide one of your own. For example, for a named file specified on the OPEN statement:

```
OPEN (UNIT=11, FILE='MYINPUT', STATUS='NEW')
```

the following default FILEDEF command is used:

```
FILEDEF MYINPUT DISK FILE MYINPUT A1
```

- For an unnamed file, the file definition default depends and the type of access you specify on the ACCESS specifier. For example:

```
OPEN (UNIT=11)
```

the following default FILEDEF command for sequential access is used:

```
FILEDEF FT11F001 DISK FILE FT11F001 A1
```

For information on:

- Default ddnames, see “Named Files” on page 136
- Specifying your own file definitions, see “Defining Files under CMS” on page 144.
- For MVS, you use the DD statement. For information about how to code a DD statement, see the *MVS JCL Reference*. For specific information about coding the DD statement for use with VS FORTRAN Version 2, see “DD Statement—Defining Files” on page 13 of this manual. Note that for sequential I/O, the release (RLSE) subparameter of the SPACE parameter on the DD statement is ignored for any file for which you specify an ENDFILE or a BACKSPACE.

For named files under MVS, you must explicitly provide a file definition for the file, or an error message will be issued.

Parallel tasks have their own set of default file definitions. For information on default file definitions for parallel processing, see “Using I/O between Parallel Tasks” on page 315.

- For MVS with TSO, you use the ALLOCATE command as explained under “Allocating Compiler Data Sets” on page 18.

For some files, you can omit coding file definitions. This is called *dynamic file allocation*. With dynamic file allocation, file characteristics are determined from several possible sources: parameters you supply by means of the FILEINF service subroutine, the existing data set, or installation and other defaults.

Besides eliminating the need for coding file definitions, dynamic file allocation provides the additional advantage of allowing you to allocate, that is assign resources to, files as they are required by your program, rather than at the time the program is loaded into storage.

Dynamically allocated named files must be explicitly specified on the FILE specifier of the OPEN statement. Unnamed files must specify STATUS='SCRATCH' while no file definition is in effect.

For example:

Dynamically allocated named file:

```
OPEN (UNIT=10, FILE='/MYFILE')
```

Dynamically allocated unnamed file:

```
OPEN (UNIT=10, STATUS='SCRATCH')
```

Dynamic file allocation is discussed in detail under Chapter 9, “Advanced I/O Topics” on page 195.

Named Files

A *named* file is a file whose name you supply in the FILE specifier of the OPEN statement. For dynamically allocated files, the file name must be the data set name or file identifier. For example:

Open statement with CMS file identifier:

```
OPEN (UNIT=9, FILE='/WRITER DATA')
```

Open statement with MVS data set name:

```
OPEN (UNIT=9, FILE='/WRITER.DATA')
```

For files not dynamically allocated, the file name must be the ddname of the corresponding file definition for that file. Following is an example of an OPEN statement for a file that is not dynamically allocated. Also shown are the corresponding CMS and MVS file definitions. The ddname is MYINPUT.

OPEN statement with ddname:

```
OPEN (UNIT=9, FILE='MYINPUT')
```

CMS file definition:

```
FILEDEF MYINPUT DISK WRITER DATA A
```

MVS file definition:

```
//MYINPUT DD DSN=WRITER.DATA,DISP=OLD
```

When an I/O statement in your program operates on a named file, it normally refers to the same physical file each time the same ddname is used in an OPEN statement. However, when the file is not connected to any unit, you can, through some non-Fortran process (such as an assembler language program or an IAD SYSCMD command) change the file definition so that it refers to some different physical file. In this case, when you use an OPEN statement with the same ddname on the FILE specifier, your program will refer to a different physical file. Your program's behavior will reflect the properties of the file referred to by the new file definition.

It is possible for two different file definitions, that is, file definitions with different ddnames, to refer to the same physical file. In this case, they will appear to be

different files as far as your program is concerned. For an example of this, see Chapter 9, “Advanced I/O Topics” on page 195.

Unnamed Files

An *unnamed* file is a file that you never refer to by name in an OPEN statement in your program (that is, you omit the FILE specifier). For such files, instead of referring to a file name or ddname, you refer only to a particular unit.

For unnamed files that are not dynamically allocated, the access method used for the file, and other factors, determine which ddname is used. Access methods are discussed under “VS FORTRAN File Access Methods” on page 138.

The ddname for an unnamed file takes one of the following forms:

- For sequential and direct access: *FTnnF001*, where *nn* is the 2-digit unit number. (For a file connected for sequential access, there can be additional subfiles; the forms *FTnnF002*, *FTnnF003*, and so on, are used in this case.)
- For striped sequential files: *FTnnPmmm*, where *nn* is the 2-digit unit number and *mmm* is the stripe number. There must be as many file definitions as there are stripes to be used, with ddnames *FTnnP001*, *FTnnP002*, and so on.
- For keyed access: *FTnnKkk*, where *nn* is the 2-digit unit number and *kk* is a 2-digit number that represents one of the keys that you intend to use. There must be as many file definitions as there are keys to be used, with ddnames *FTnnK01*, *FTnnK02*, and so on.
- For the error message unit, or the standard output unit if it is the same as the error message unit, for an MTF subtask: *FTERRsss*, where *sss* is the MTF subtask number.
- For the standard output unit, if it is different from the error message unit, for an MTF subtask: *FTPRTsss* where *sss* is the MTF subtask number.
- For an originated task created with the ORIGINATE ANY TASK statement:

Error Message Unit: FTSExxxx
Standard Print Unit: FTSPxxxx
Standard Punch Unit: FTSQxxxx
Standard Reader Unit: FTSRxxxx

where *xxxx* is a number from 1 to 9999, and is the absolute value of the *rtaskid* specified on the ORIGINATE ANY TASK statement.

- For an originated task created with the ORIGINATE TASK statement:

Error Message Unit: FTUExxxx
Standard Print Unit: FTUPxxxx
Standard Punch Unit: FTUQxxxx
Standard Reader Unit: FTURxxxx

where *xxxx* is a number from 1 to 9999, and is the absolute value of the *ptaskid* specified on the ORIGINATE TASK statement.

Thus, if you code the following OPEN statement:

```
OPEN (UNIT=9, ACCESS='SEQUENTIAL')
```

the ddname *FT09F001* is used if a file definition statement has not been provided for *FT09P001*.

When an I/O statement in your program operates on an unnamed file, it normally refers to the same physical file each time the same ddname applies to the unit. This means, for example, that different OPEN statements with the same values given for all of their specifiers will refer to the same unnamed file. However, after the file has been disconnected from the unit, you can, through some non-Fortran process (such as an assembler language program or an IAD SYSCMD command) change the file definition so that it refers to some different physical file. In this case, when you reconnect the unit to the unnamed file by using an OPEN statement without a FILE specifier, your program will refer to a different physical file even though the OPEN statement doesn't specify anything different. Your program's behavior will reflect the properties of the file referred to by the new file definition.

It is possible for two different file definitions, that is, file definitions with different ddnames, to refer to the same physical file. In this case, they will appear to be different files as far as your program is concerned.

Unnamed files that can be dynamically allocated include only the following:

- Temporary files (that is, STATUS='SCRATCH' is coded on the OPEN statement) that are connected for sequential or direct access, and which do not have a file definition specified.

For these files, the default ddname FTnnF001 is used. You may also supply file characteristics by means of the FILEINF service subroutine. Under MVS, VS FORTRAN creates a file definition with file characteristics that are determined by installation defaults. Under CMS, the normal CMS default file definition is used.

- Under CMS, preconnected files directed to any of the standard I/O units (see "Units and File Connection" on page 134 for an explanation of connecting files to units).

For these files, the default ddname FTnnF001 is used. VS FORTRAN creates a file definition with file characteristics that are determined by installation defaults.

- Under MVS, preconnected files directed to the error message unit (and standard output unit for WRITE and PRINT statements if different).

For these files, the default ddname FTnnF001 is used. VS FORTRAN creates a file definition with file characteristics that are determined by installation defaults.

VS FORTRAN File Access Methods

The term *file access method* refers to the means used by Fortran to find a specific record in a file to be read or written. VS FORTRAN supports three file access methods: sequential access, direct access, and keyed access. You specify the file access method for a file in the ACCESS specifier of the OPEN statement as SEQUENTIAL, DIRECT, or KEYED. Preconnected files are always connected for sequential access.

This section provides a brief description of each file access method. Chapter 8, "Using File Access Methods" on page 163 describes the file access methods in detail.

Sequential Access

In a file connected for *sequential access*, records are read or written consecutively, from the first record in the file to the last. Files for tapes, terminals, printers, card readers, and punches are always accessed sequentially; in addition, many disk files can be accessed sequentially.

To read records in a sequential file, you could code the following:

```
22      READ (UNIT=14, END=99) A, B, C
      :
      GO TO 22
```

The READ statement will read from the file connected to unit 14. When this group of statements is processed, all of the records in the file—beginning with the first one—will be read, one at a time. When the endfile record is read, control transfers to the Fortran statement labeled 99.

Direct Access

Records in a file connected for *direct access* are arranged in the file according to their relative record numbers. All records are the same size and each record occupies a predefined position in the file, determined by its record number. You can access any record in the file directly by specifying its number in a READ or WRITE statement. For example:

```
READ (UNIT=14, REC=28) A, B, C
```

This statement retrieves record number 28.

Files connected for direct access must be stored on disks.

Keyed Access

In a file connected for *keyed access*, each record contains a *primary key* whose value uniquely identifies that record. In addition, a record can contain one or more *alternate keys* whose values identify the record. You must use keyed access I/O statements to access records in a file organized for keyed access. “Input/Output Operations for Keyed Access” on page 182 explains how to use keyed access.

An example of a READ statement for reading a keyed access file is:

```
READ (UNIT=14, KEY='SMITHBROOK', NOTFOUND=88) A, B, C
```

This statement retrieves the record with a key of SMITHBROOK. If there is no such record in the file, control transfers to the Fortran statement labeled 88.

Access Methods Used by the Operating System

Related to the three file access methods that you specify in Fortran, but not the same, are the access methods used by the operating system. Examples include the virtual storage access method (VSAM), basic direct access method (BDAM), and basic sequential access method (BSAM).

Normally you need not be concerned with these, except for certain considerations you must take into account with VSAM files and striped data sets. For information about VSAM files, see Chapter 11, “Using VSAM” on page 209.

Records As Seen by Fortran

Fortran recognizes three types of records: formatted records, unformatted records, and endfile records.

A *formatted record* is a sequence of characters. It is written with a formatted output statement and is read with a formatted input statement. (Formatted input/output statements will be discussed later in this chapter.) Because data is usually not in character form within main storage, it must be converted to character form when a formatted record is written, and converted to its internal form when a formatted record is read. The main purpose of formatting data is so that you can easily examine it, as for example in a listing.

If you don't need to examine the records, for example when you write records and subsequently read them within the same program, you can save file space and the processing time required by data conversion by using unformatted records. An *unformatted record* is a sequence of data in its internal form and involves no conversion of data between main storage and the file. It is written with an unformatted output statement and, similarly, read with an unformatted input statement.

A file cannot contain both formatted and unformatted records.

An *endfile record* is a special record that contains no data and occurs only as the last record of a file. When a file is read sequentially (see "VS FORTRAN File Access Methods" on page 138), the endfile record allows your program to easily determine when the end of the file has been reached.

Records As Seen by the Operating System

The system term *record format* unfortunately has nothing to do with the Fortran term *formatted record*. Instead, it is concerned with the length of the records and their arrangement within a non-VSAM file. (For information about VSAM files, see Chapter 11, "Using VSAM" on page 209.)

The record format of a file can be fixed-length, variable-length, or undefined. In a file with *fixed-length* records, all the records in the file have the same length. In a file with *variable-length* records, the records may have different lengths. The length of each record is stored along with the data. Similarly, in a file with *undefined* records, the records may have different lengths, but the length of each record is *not* stored with the data.

For a tape file, if the record length for an undefined record is less than 18, the additional bytes are padded up to 18 bytes to ensure that no data is lost.

Records may also be blocked or unblocked. *Blocking* is the process of grouping records before they are written to a file. The group of records that is written to the file is called a *block*. Only fixed-length and variable-length records can be blocked.

A variable-length record can be *spanned*; that is, it can be contained in more than one block.

In most cases, you need not specify a record format and instead can accept the default values for files supplied by VS FORTRAN. These are shown in Figure 40 on page 141.

Figure 40. VS FORTRAN-Supplied File Default Values

Direct Access	Sequential Access		
Fixed-length unblocked	Formatted	Unformatted	
	Variable	MVS (including TSO)	CMS
		Undefined	Fixed length or undefined, depending on which was chosen when VS FORTRAN was installed

If you don't want to accept these defaults, specify the record format as follows:

- For CMS, use the RECFM option on the FILEDEF command.
- For MVS, use the DCB parameter on the DD statement.
- For MVS with TSO, use the RECFM option on the ALLOCATE or ATTRIBUTE command.
- For dynamically allocated files, use the RECFM parameter on the FILEINF service subroutine.

The following restrictions apply:

- Only files with fixed-length unblocked records (RECFM specifies F) can be connected for direct access.
- Files with variable-length spanned records (RECFM specifies VS or VBS) cannot be used with formatted input/output.
- Only files that allow unblocked variable-length spanned records (RECFM specifies VS) can be used for asynchronous I/O.

File Existence

At a very simple conceptual level, those external files that are available to your program for reading or that have been created within your program are said to *exist* for your program. However, in reality a number of factors, such as what kind of device the file is on and whether the file contains any data, play a role in determining a file's existence.

Before referring to a particular file in your program, you can use the INQUIRE statement to determine whether that file exists. More details about existence and the use of INQUIRE are discussed in Chapter 13, "What Determines File Existence" on page 233 and *VS FORTRAN Version 2 Language and Library Reference*.

Chapter 7. Connecting, Disconnecting, and Reconnecting Files for I/O

This chapter explains how to preconnect, connect, disconnect and reconnect your data files. It also describes special considerations for reading and writing data to your files.

Preconnecting Data Files to Units

Running a VS FORTRAN program may require reading and writing to files. Chapter 8, "Using File Access Methods" on page 163 explains how to use VS FORTRAN I/O statements to process data files. The discussion in this section is limited specifically to relating physical files to Fortran data files.

Before you can read or write a Fortran file, the file must be associated with—that is, connected to—a unit. Some files are already defined and connected when the program begins to run. These files are referred to as being preconnected.

Before the OPEN statement was introduced in FORTRAN 77, all files were preconnected. To maintain compatibility, preconnection is still allowed, but only for unnamed non-VSAM files that are accessed sequentially. For information on defining file characteristics for preconnected files, see "Defining Files under CMS" on page 144 (CMS) or "DD Statement—Defining Files" on page 13 (MVS).

By default, preconnected files are connected for sequential access and formatted input/output. In addition, all blanks, other than leading blanks, within arithmetic formatted input fields are treated as zeros. If you wish, you can change these connection properties, as explained under "Changing Connection Properties" on page 154.

To preconnect a nonstriped file, you provide a file definition with a ddname of FTnnF001, where nn is the 2-digit number of the unit to which you want to connect the file. For CMS, unless you have explicitly provided a FILEDEF with the ddname FTnnP001 for a striped file, you can rely on a default FILEDEF command with the ddname FTnnF001.

To preconnect a file for striped sequential access, each stripe must have a separate file definition. The file definition must have the ddname FTnnP001 for the first stripe, FTnnP002 for the second stripe, and so on. When both ddnames FTnnF001 and FTnnP001 are specified in file definition statements, only FTnnP001 is preconnected.

That's all you have to do. When your program begins to run, the file will be automatically connected to the unit.

Preconnecting Standard I/O Units

A subset of preconnected files are those that are read from the *standard input unit* or written to the *standard output unit*. The standard input unit is a unit defined during installation of VS FORTRAN to be used as the default unit for input. During installation, either two or three standard output units are defined:

- Two Standard Output Units

Preconnecting Files

1. One standard unit for the PUNCH statement
 2. One standard unit for all other output (including VS FORTRAN error messages, WRITE statements and PRINT statements). This unit is often called the error message unit.
- Three Standard Output Units
 1. One standard unit for the PUNCH statement
 2. One standard unit for VS FORTRAN error messages
 3. One standard unit for WRITE and PRINT statements

Each standard input/output unit has a fixed unit number. The IBM-supplied defaults for the standard input/output units are:

Unit 5 READ statement, with * as the unit identifier.

Unit 6 WRITE statement with * as the unit identifier, PRINT statement, and VS FORTRAN error messages.

Unit 7 PUNCH statement.

You can define your own file characteristics using a file definition for any of the standard input/output units. The file characteristics you provide are used instead of the default. For example, for CMS terminal input, to inhibit the folding of alphabetic character data to uppercase, issue the command:

```
FILEDEF FT05F001 TERMINAL (LOWCASE
```

Only files connected for sequential access can be directed to the standard input/output units. In addition, only formatted I/O is allowed for the error message unit.

Also, under CMS, if you don't provide a file definition for a file that is not dynamically allocated, VS FORTRAN supplies defaults.

Defining Files under CMS

The form of the FILEDEF command you use varies, depending on the type of file you're processing: sequential or direct, tape, terminal, or unit record.

Defining Sequential and Direct Files: To define sequential and direct files on disk, specify the FILEDEF command as follows:

```
FILEDEF FTnnFmmm DISK filename filetype [filemode] [(options)]
```

or

```
FILEDEF nn DISK filename filetype [filemode] [(options)] if mmm = 001
```

You specify the *FTnnFmmm* field to agree with the Fortran unit numbers in the source program.

- For the *nn* field, see Figure 30 on page 83.
- For the *mmm* field, specify 001 if you're not using multiple subfiles. If you are using multiple subfiles, you can specify 001 through 999.

If you have specified the FILE specifier in the OPEN statement, specify the FILEDEF command as follows:

```
FILEDEF fn DISK filename filetype [filemode] [(options)]
```

where *fn* is the name specified in the FILE specifier.

If you need to map a CMS file to an MVS data set, make sure that the file mode number is 4 (for example, A4) or that the record format is undefined or fixed unblocked.

For direct files with the UPDATE-IN-PLACE attribute (direct files to which you write new records over existing records), specify the file mode number as 6; for example, C6.

The options are any FILEDEF options valid for disk files. In particular, the maximum LRECL and BLKSIZE that can be specified is 32760.

Defaults for the XTENT, LRECL, BLKSIZE, and RECFM Options: If you omit the XTENT option, which applies only to files connected for direct access, the system provides a default of 50 for the number of records. For dynamically allocated files, the MAXREC parameter of the FILEINF routine determines the value. For information on dynamic file allocation, see Chapter 9, “Advanced I/O Topics” on page 195.

For information on defaults for LRECL, BLKSIZE, and RECFM, see Chapter 12, “Considerations for Specifying RECFM, LRECL, and BLKSIZE” on page 227.

Warning: A FILEDEF command should not define a file for output on a unit that VS FORTRAN Version 2 predefines for input (for example, terminal input). Likewise, a FILEDEF command should not define an existing file for input on a unit that VS FORTRAN Version 2 predefines for output. In this situation, running the program could cause undesirable results, including destruction of data on an existing file or loss of the file from the CMS directory.

Defining Tape Files: To define tape files, you specify the FILEDEF command as follows:

```
FILEDEF FTnnFmmm TAPx [(options)]
    or
FILEDEF nn TAPx [(options)]      if mmm = 001
    or
FILEDEF fn TAPx [(options)]
```

You specify the FTnnFmmm field to agree with the Fortran unit numbers in the source program:

- For the *nn* field, see Figure 30 on page 83.
- For the *mmm* field, specify 001 if you are not using multiple files. If you are using multiple files, you can specify 001 through 999.
- For the *x* field, you specify any valid tape unit (0 through F).
- For the *fn* field, specify the file name you specified on the OPEN statement FILE specifier.
- Options are any FILEDEF options valid for tape files.

Defining Terminal Files: To define terminal files, you specify the FILEDEF command as follows:

```
FILEDEF FTnnF001 TERMINAL [(options)]
      or
FILEDEF nn TERMINAL [(options)]      if mmm = 001
      or
FILEDEF fn TERMINAL [(options)]
```

You specify the *FTnnF001* field to agree with the Fortran unit numbers in the source program.

- For the *nn* field, see Figure 30 on page 83.
- For the *fn* field, you specify the file name you specified on the OPEN statement FILE specifier.
- The options are any FILEDEF options valid for terminal files.

Note: You can use the LOWCASE option to inhibit the folding of alphabetic character data to uppercase.

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.

For terminal files, a null entry in response to a prompt is taken to be an end-of-file. If you want to continue processing, a FILEDEF or an explicit OPEN is required.

Defining Unit Record Files: To define unit record files, you specify the FILEDEF command as follows:

For Card Reader Files:

```
FILEDEF FTnnF001 READER [(options)]
      or
FILEDEF nn READER [(options)]
      or
FILEDEF fn READER [(options)]
```

For Card Punch Files:

```
FILEDEF FTnnF001 PUNCH [(options)]
      or
FILEDEF nn PUNCH [(options)]
      or
FILEDEF fn PUNCH [(options)]
```

For Printer Files:

```
FILEDEF FTnnF001 PRINTER [(options)]
      or
FILEDEF nn PRINTER [(options)]
      or
FILEDEF fn PRINTER [(options)]
```

You specify the *FTnnF001* field to agree with the Fortran unit numbers in the source program:

- For the *nn* field, see Figure 30 on page 83.
- For the *fn* field, you specify the file name you specified on the OPEN statement FILE specifier.

- Options are any FILEDEF options valid for the type of unit record file you're processing.

Input/Output System Considerations for MVS

Tape Labels

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.

For more information about job control statements, see “Coding and Processing Jobs” on page 12. For additional detail on magnetic tape label processing, see *OS/VS Tape Labels*.

Direct Access Labels

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.

For additional details on direct access label processing, see the Data Management Services Guide for your operating system.

Fortran Record Definition

Your Fortran programs must define the characteristics of the data records it processes: their formats, their record length, their blocking, and the type of device upon which they reside.

You can define data record characteristics through the DCB parameter of the DD statement or, for certain dynamically allocated files, through the FILEINF routine. For information on the FILEINF routine, see “Overriding File Characteristic Defaults” on page 198. VS FORTRAN also supplies defaults for I/O data sets, as described under “Installation Defaults for I/O Data Set Characteristics” on page 149.

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 ISCII/ASCII character set
- Special information for tape files
- Special information for direct access files
- Information to be used from another data set.

Record Formats: Under VS FORTRAN Version 2, 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. The maximum LRECL and the maximum BLKSIZE is 32760.

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.

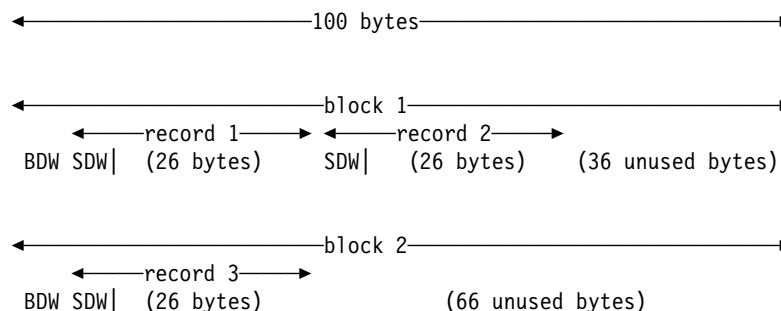
Each record contains a segment-descriptor word, and each block contains a block-descriptor word. These descriptor fields are used by the system; they are not available to Fortran programs. The maximum BLKSIZE is 32760, the maximum LRECL is 32756, and, assuming one record per block, the maximum amount of data is 32752.

When variable-length records are blocked, the blocks may not be filled to the maximum block size specified, even though it appears that another record can be contained in the block. The block-descriptor word (BDW) occupies the first 4 bytes (word) of a block. A segment-descriptor word (SDW) occupies the first word of each variable-length record. Both must be considered when defining BLKSIZE and LRECL parameters. If the remainder of the block is not large enough to contain another complete record, as defined by the record size (LRECL), the current buffer is written and a new block is started for the next record.

Example (all numbers are given in decimal):

RECFM=VB LRECL=50 BLKSIZE=100

In the above example, if you write three records, each of length 30, you might expect all three records to be written in one block. However, Fortran writes records 1 and 2 in block 1, after the BDW, for a length of 64 bytes. Record 3 is written in block 2. Although the third record of length 30 will fit in the first block, it is not included because the test for record length is done using LRECL (length 50). VS FORTRAN Version 2 does not know the actual length of the record until after the data is transferred. The following diagram shows how the records are stored in the blocks:



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 word, and each block includes a block-descriptor word. These descriptor 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-descriptor, block-descriptor, or segment-descriptor words.

Sequential EBCDIC Data Sets

You can define Fortran records in an EBCDIC data set (which may also contain double-byte character data) as formatted or unformatted. List-directed I/O statements are considered to be formatted.

If you're processing records using asynchronous input/output, the records must be variable spanned and unblocked.

Use blocked records wherever possible, since blocked records substantially reduce processing time.

Formatted Records: You can specify formatted records as fixed length (blocked or unblocked), variable length (blocked or unblocked), or undefined length.

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. Unformatted records can be specified as fixed, fixed block, undefined, variable, and spanned.

Sequential ISCI/ASCII Data Sets

ISCI/ASCII data sets may have sequential organization only. For system considerations, see the documentation for the system you're using.

Fortran records in an ISCI/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.

Installation Defaults for I/O Data Set Characteristics

When you code the data set characteristics on the DD statement, the values you specify override the existing values for that file. When you omit any characteristics, the values are obtained from the old file if they are available.

When values are not available from an old file, or if you are creating a new file, the missing values are obtained from installation defaults and other, fixed defaults, based on the information available. The IBM-supplied installation default for the device for units other than 5, 6, and 7 is SYSDA. For information on the defaults for record format, record length, and block size, see Chapter 12, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 227.

Connecting Files

A program cannot read or write to a file unless that file has been connected to a unit. A file can be connected to a unit while the program runs by means of an OPEN statement, or it can be preconnected, that is, automatically defined and connected when your program begins to run. *In either case, if a file is not dynamically allocated, a file definition must be in effect for it.* A file may be connected to only one unit at a time and, similarly, a unit may be connected to only one file at a time.

A subset of preconnected files are those that are read from the *standard input unit* or written to the *standard output unit*. The standard input unit is a unit defined during installation of VS FORTRAN to be used as the default unit for input. On your READ statement, you can specify the standard input unit by an asterisk rather than its unit number. For example, if unit 5 is the standard input unit,

```
READ (*, FMT=100) A,B,C
```

is equivalent to

```
READ (5, FMT=100) A,B,C
```

Similarly, you can specify a standard output unit (on a WRITE or PUNCH³ statement) by using an asterisk rather than the unit number. (The PRINT statement is always directed to a standard output unit.) For information on preconnecting and defining files, see “Preconnecting Data Files to Units” on page 143.

Identifying the Unit and File Using the OPEN Statement

For any file that is not preconnected, you must use the OPEN statement to connect it to a unit before you can read from it or write to it.

For information on the specifiers available on the OPEN statement, see *VS FORTRAN Version 2 Language and Library Reference*.

For named files, you identify the unit and file to be connected by specifying the unit number with the UNIT specifier and the file name with the FILE specifier.

The unit number may be any valid unit number, which is determined at installation time. An exception is that you cannot specify the error message unit (usually unit 6), unless you are running the OPEN statement just to change the CHAR specifier. (Changing the CHAR specifier is discussed under “Changing Connection Properties” on page 154.)

What you specify for the file name depends on how the file is allocated. For dynamically allocated files, the file name must be the CMS file identifier or the MVS data set name, preceded by a slash (/). For example, the following OPEN statement connects the file having the CMS file identifier MYDATA OUTPUT A4 to unit 9:

```
OPEN (UNIT=9, FILE='/MYDATA OUTPUT A4')
```

³ The PUNCH statement is a FORTRAN 66 statement that is still supported by VS FORTRAN, but only under the compile-time option LONGLVL(66). It is described in *IBM System/360 and System/370 FORTRAN IV Language*.

The following OPEN statement connects the file having the MVS data set name MYDATA.ON.MVS to unit 9:

```
OPEN (UNIT=9, FILE='/MYDATA.ON.MVS')
```

For files not dynamically allocated, the file name must be the ddname of the file definition. For example the following OPEN statement connects the file referred to by the ddname REPORT to unit 9:

```
OPEN (UNIT=9, FILE='REPORT')
```

For unnamed files, you omit the FILE specifier. If the unit is being connected for sequential or direct access and the ddname FT nn P001 has not been provided (serial processing or root task only), the ddname FT nn F001 is used.

If the unit is being connected for sequential access (serial processing or root task only) and the ddname of the form FT nn P001 has been provided in a file definition statement, ddnames of the form FT nn P mmm are used (where mmm is 001, 002, and so on, for each stripe). Direct access cannot be performed on unit nn if ddname FT nn P001 has been provided in a file definition statement.

If the file is being connected for keyed access, the ddnames of the form FT nn K kk are used (where kk is 01, 02, etc., for each key specified in the KEYS specifier). To indicate the access method, you code the ACCESS specifier. If you don't code the ACCESS specifier, sequential access is the default.

In the following example, the file referred to by the ddname FT09F001 is connected to unit 9. If FT09P001 is provided in a file definition statement, the file referred to by the ddname FT09P001, is connected to unit 9 for striped sequential access. If both FT09F001 and FT09P001 are provided, FT09P001 is used:

```
OPEN (UNIT=9)
```

When connecting an unnamed file, you must first disconnect any file that is already connected to the unit; otherwise, the OPEN statement will refer to the file that is already connected. Disconnecting files is discussed in “Disconnecting Files” on page 157.

Indicating Whether the File Exists: The STATUS specifier helps prevent accidental overwriting of existing data. For a file that does not exist, you specify STATUS='NEW', in which case the file will be created and connected to the unit. For a file that does exist, you specify STATUS='OLD', in which case the existing file will be connected to the unit. If you don't know whether the file exists, you can either omit the STATUS specifier or specify STATUS='UNKNOWN'; in either case the file's existence will be checked, and it will be connected as NEW or OLD, accordingly.

The fact that a file doesn't exist does not necessarily mean that you will be able to create it. For example, the file or device that you refer to in your file definition may have some characteristic that prevents you from writing on it.

- Under CMS, an example of this would be a minidisk that you have been linked to in read-only status.

The one exception to this deals with read-only extensions to the read-write disk. You can create a file on the read-write disk that has the same name as a file that already exists on the read-only extension in one of the following two ways:

1. Explicitly code `STATUS='NEW'` or `ACTION='WRITE'` on the `OPEN` statement
 2. A `WRITE` statement is the first statement that affects the file in your Fortran program. If a `READ` statement has been processed, the file on the read-only extension is explicitly pointed at and an output statement is not allowed.
- Under MVS, the `LABEL` parameter of your `DD` statement could specify input-only processing, or the file might have RACF* protection that prevents you from writing on it.

If the run-time option `OCSTATUS` is in effect, the consistency between file existence and `STATUS='NEW'` or `STATUS='OLD'` is verified. (For more information on run-time options, see “Available Run-Time Options” on page 105.) If `STATUS='NEW'` is coded for an existing file or `STATUS='OLD'` is coded for a nonexistent file, the `OPEN` statement will fail. However, be aware that this verification is done only for the following:

- DASD files, including:
 - PDS members under MVS
 - TXTLIB, MACLIB, and LOADLIB members under CMS
 - VSAM files
- Labeled tape files under MVS
- Files whose file definitions specify `DUMMY`. For these files, the verification is done only if the file was successfully opened previously in the current program.

The verification is *not* done when you specify `STATUS='SCRATCH'` or `STATUS='UNKNOWN'`, or omit the `STATUS` specifier.

If `NOOCSTATUS` is in effect, consistency between existence and the `STATUS` specifier is not verified for any file.

On MVS, you may ignore any non-Fortran messages that are issued from the system.

Connecting Temporary Files: To connect a temporary file (that is, a file that can be used only within the current program and will be deleted when the file is disconnected), you specify `STATUS='SCRATCH'`. File existence will be checked to determine whether the file already exists or should be created.

The `FILE` specifier is not allowed with `STATUS='SCRATCH'`.

Choosing the Access Method: To indicate whether you want the file to be connected for sequential, direct, or keyed access, you code the `ACCESS` specifier. Sequential access is the default.

* RACF is a trademark of the International Business Machines Corporation.

If you choose direct access, you must also code the RECL specifier to indicate the record length.

For keyed access, the KEYS specifier allows you to give the starting and ending positions of the primary and alternate keys to be used. This is discussed in more detail under “Input/Output Operations for Keyed Access” on page 182.

Choosing Formatted or Unformatted I/O: To specify formatted or unformatted I/O, use the FORM specifier. FORM='FORMATTED' specifies formatted I/O and FORM='UNFORMATTED' specifies unformatted I/O. The default is formatted I/O for sequential access and unformatted I/O for direct access and keyed access.

Choosing How Input Blanks Will Be Treated: For a formatted input file, you have the option of specifying how blanks in arithmetic fields will be treated. If you specify BLANK='NULL', all blanks are ignored, except that a field of all blanks has a value of zero. If you specify BLANK='ZERO', all blanks, other than leading blanks, are treated as zeros. For preconnected files, the default is BLANK='ZERO'. For files connected with the OPEN statement, the default is BLANK='NULL'.

Indicating the Processing To Be Done: Specify ACTION='READ' to indicate that the file is being connected for reading only, ACTION='WRITE' to indicate writing only, or ACTION='READWRITE' for both reading and writing. (For more specific meanings of ACTION='WRITE' and ACTION='READWRITE' for files connected for keyed access, see “Connecting Files” on page 183. For information on using the ACTION specifier in an MTF environment, see Chapter 20, “The Multitasking Facility (MTF)” on page 397.)

Under CMS, ACTION='WRITE' and ACTION='READWRITE' are not allowed for TXTLIB, MACLIB, and LOADLIB members.

The default is READ for keyed access and READWRITE for sequential access and direct access.

Specifying VSAM Passwords: For VSAM files that are password-protected, you must specify a password with the PASSWORD specifier. If ACTION='READ' is specified, the file's read password is required; otherwise, its update password is required.

Error Checking: The IOSTAT and ERR specifiers allow for error checking during processing of the OPEN statement. How to use these specifiers is discussed in *VS FORTRAN Version 2 Language and Library Reference*.

Indicating Whether the File Contains Double-Byte Characters: If a file may contain data in a language, such as Japanese, that has characters belonging to the double-byte character set, specify CHAR='DBCS'; otherwise, specify CHAR='NODBCS'. The default is CHAR='NODBCS'. For more information on double-byte characters, see Chapter 10, “Considerations for Double-Byte Data” on page 205.

Changing Connection Properties

After a file has been connected, either by preconnection or by the OPEN statement, use the OPEN statement to change certain connection properties.

Preconnected Files: Preconnected files are connected with the following defaults:

```
ACCESS='SEQUENTIAL'
FORM='FORMATTED'
BLANK='ZERO'
CHAR='NODBCS'
```

Figure 41 shows which of these defaults you can override, depending on which I/O statement you use first.

Figure 41. Overriding Defaults for Preconnected Files

If your first I/O statement for the pre-connected file is:	The access method will be:	The form will be:	Blanks will be:	The file may contain double-byte characters:	But with subsequent I/O statements, you can change:
OPEN	Sequential or direct access, as given in the ACCESS specifier. Sequential is the default.	Formatted or unformatted, as given in the FORM specifier. Unformatted is the default for direct access. Formatted is the default for sequential access.	Ignored or treated as zeros, as given in the BLANK specifier. The default is for blanks to be ignored.	Yes or no as given in the CHAR specifier.	How blanks are treated, by issuing an OPEN statement. Whether the file may or may not contain double-byte characters, by issuing an OPEN statement.
PRINT, formatted READ, or formatted WRITE	Sequential.	Formatted.	Treated as zeros.	No	How blanks are treated, by issuing an OPEN statement. Whether the file may or may not contain double-byte characters, by issuing an OPEN statement.
Unformatted READ or unformatted WRITE	Sequential.	Unformatted.	N/A	N/A	Nothing.
BACKSPACE or REWIND	Sequential.	Formatted.	Treated as zeros.	No	The form, by issuing an OPEN, unformatted READ, or unformatted WRITE statement. How blanks are treated, by issuing an OPEN statement. Whether the file may or may not contain double-byte characters, by issuing an OPEN statement.
ENDFILE	Sequential.	Formatted.	N/A	N/A	Nothing.

The following example shows how to change the access method from sequential to direct, using the OPEN statement. As shown in Figure 41, this can be done only if no I/O statements have referred to the preconnected file.

```
OPEN (UNIT=12, ACCESS='DIRECT', RECL=80)
```

Note that you cannot use the OPEN statement to change the access method to keyed access. If you specify ACCESS='KEYED', the statement will refer to a different file. Remember that for files connected for sequential or direct access, the ddname FTnnF001 is used; for unnamed files connected for striped sequential access, the ddname FTnnP001 is used; and for files connected for keyed access, the ddname FTnnKkk is used.

Files Connected with OPEN: After you've connected a file with an OPEN statement, you can change only the BLANK and CHAR specifiers. You can change the specifier by coding another OPEN statement, such as the following:

```
OPEN (UNIT=12, BLANK='NULL')
```

Note that for a named file, you do not have to code the FILE specifier on this OPEN statement.

Ignoring File History

The IGNFHU and IGNFHDD service subroutines cause the history of a file to be disregarded in determining file existence. These routines should be used when a non-Fortran subroutine is being used to change the file definition unknown to the Fortran run time. These service subroutines will notify the VS FORTRAN Version 2 run-time environment that you have changed the file definition for a given unit or ddname, and that the previous usage of the file to which the unit or ddname was connected should be disregarded. The file must not be currently open at the time the service subroutine is called. (The interfaces for these service subroutines are described in the VS FORTRAN Version 2 Language and Library Reference.)

The IGNFHU and IGNFHDD service subroutines give the same behavior for the file identified as specifying the NOFILEHIST run-time option. They are supplied for users who are not able to specify the NOFILEHIST run-time option and to give the flexibility of specifying this behavior dynamically, on an individual file basis.

Example: Valid IGNFHU Calls.

```
C* Call non-fortran subroutine ALLOC1 to allocate (using SVC 99)
C* a file and connect it to unit 8
    CALL ALLOC1 (parmlist)
    OPEN (8,access='sequential',action='read')
    CLOSE(8)
C* Call non-fortran subroutine ALLOC2 to deallocate (using SVC 99)
C* the file and disconnect it from unit 8
C*
    CALL ALLOC2 (parmlist)
C*
C* Must call IGNFHU here to tell Fortran that we have switched
C* the file that UNIT 8 is connected to, and not to look at the
C* previous history of the file that unit 8 was connected to.
C*
    CALL IGNFHU(8,irc)
C*
C* Call non-fortran subroutine ALLOC3 to allocate (using SVC 99)
C* a different new (empty) file to unit 8.
C* If IGNFHU had not been used, the _old_ history would cause
C* the Fortran run-time to believe the file existed, and the
C* status of NEW would _not_ have been allowed; the direct
C* access file would not have been formatted.
C*
    CALL ALLOC3 (parmlist)
    OPEN (8,access='direct',status='new',action='readwrite')
    CLOSE(8)
```

Example: Valid IGNFHDD Calls.

```
C* Call non-fortran subroutine ALLOC1 to allocate (using SVC 99)
C* a file and use the ddname of MYFILE01
    CALL ALLOC1 (parmlist)
    OPEN (8,file='MYFILE01',access='sequential',action='read')
    CLOSE(8)
C* Call non-fortran subroutine ALLOC2 to deallocate (using SVC 99)
C* the file and disconnect it from the ddname MYFILE01
C*
    CALL ALLOC2 (parmlist)
C*
C* Must call IGNFHDD here to tell Fortran that we have switched
C* file that MYFILE01 is connected to, and not to look at the
C* previous history of the file MYFILE01 was connected to.
C*
    CALL IGNFHDD("MYFILE01",irc)
C*
C* Call non-fortran subroutine ALLOC3 to allocate (using SVC 99)
C* a different new (empty) file to the same ddname, MYFILE01.
C*
    CALL ALLOC3 (parmlist)
    OPEN (8,file='MYFILE01',access='direct',status='new',
1      action='readwrite')
    CLOSE(8)
```

Disconnecting Files

You can disconnect a file from a unit in more than one way. The usual way is to use a CLOSE statement for the particular unit. Another way is to use an OPEN statement for a different, named file, which causes the file already connected to the unit to be disconnected. Or, you can allow a file to be automatically disconnected at program termination.

Disconnecting Files with the CLOSE Statement

For information on the specifiers available on the CLOSE statement, see *VS FORTRAN Version 2 Language and Library Reference*.

Identifying the Unit: On the UNIT specifier, specify the number of the unit to be disconnected. For example:

```
CLOSE (UNIT=14)
```

The unit number may be any valid unit number, which is determined at installation time, except for that of the error message unit (usually unit 6).

Do not specify a file on the CLOSE statement; whichever file is connected to the unit at the time will be disconnected.

Retaining Files After Disconnection: The STATUS specifier allows you to specify what should happen to the file after it is disconnected. You can specify either STATUS='KEEP' or STATUS='DELETE'. In most cases, if you specify KEEP, the file will be retained and will continue to exist in the VS FORTRAN environment during the remainder of the current program and after program termination.

However, in the case of an empty DASD file (including VSAM files, but excluding PDS members), the file will be seen as existing *only* during the remainder of the current program. Therefore, if you want to reconnect it during the current program, and OCSTATUS is in effect, you must specify STATUS='OLD' or STATUS='UNKNOWN' on the OPEN statement. But if you want to reconnect it during a subsequent program you must specify STATUS='NEW' or STATUS='UNKNOWN'.

Note: Under MVS, a CLOSE with STATUS='KEEP' for a dynamically-allocated file causes a FREE; that is, the physical data set is not allocated. If the file is connected again by a Fortran statement, the data set is reallocated.

The default for STATUS is KEEP, except for those files connected with a status of SCRATCH, in which case the default is DELETE. If OCSTATUS is in effect or the file is dynamically allocated, even if you specify KEEP for a temporary file, the file will be deleted and an error will be detected.

Deleting Files After Disconnection: If you specify STATUS='DELETE', and the run-time option OCSTATUS is in effect or the file is dynamically allocated, the file will be deleted and will cease to exist within the VS FORTRAN environment. For files that are not dynamically allocated, if NOOCSTATUS is in effect and you specify DELETE, no file will be deleted; the file will only be disconnected, as though you had specified KEEP.

DELETE is not allowed for files connected with ACTION='READ'. In addition, you can specify DELETE for only certain types of files. Figure 42 lists these files and explains how they are actually deleted.

Figure 42. Files That Can Be Deleted

Type of File	How It Is Deleted
DASD file under CMS (excluding files with filetype TXTLIB, LOADLIB, or MACLIB)	The file is erased from the minidisk.
DASD file under MVS (excluding VSAM files and PDS members)	If dynamically allocated: The file is removed from the volume and uncataloged. If not dynamically allocated: The file is emptied, making it appear as though it were just allocated, without any records in it.
PDS member	The member's name is removed from the PDS directory.
Reusable VSAM file	The file is emptied, making it appear as though it were just defined, without any records in it.
Labeled tape file under MVS	The file is recreated with no data records and with a block size and record size of 0 in the header label. Any files that follow the deleted file on the tape are lost.
File whose file definition specifies DUMMY	Deletion is recorded internally by VS FORTRAN for the duration of time the program runs.
Striped files	If an MVS file has any stripes defined on an unlabeled tape, or a striped file is defined under CMS, it is disconnected but not deleted, and an error is detected. Striped files can be deleted if all of the stripes referred to by the ddnames belonging to the same striped file can be deleted. They will be deleted by a method(s) above that apply to the stripe(s).

Note that once you delete a file, it will not exist unless you reconnect it by issuing an OPEN statement with STATUS='NEW' or STATUS='UNKNOWN'.

An exception, which applies to all files but those whose file definitions specify DUMMY, occurs when a routine, such as an assembler language routine, that is unknown to the VS FORTRAN environment, writes records into the file in the interim between the CLOSE and OPEN statements. This causes the file to exist again, which in turn will cause an OPEN with STATUS='NEW' to fail if OCSTATUS is in effect. In order to successfully reconnect the file, you must use an OPEN statement that specifies STATUS='OLD' or STATUS='UNKNOWN', or omits the STATUS specifier.

Only files listed in Figure 42 can be deleted. For example, you cannot delete a file whose file definition refers to one of the following:

- An in-stream (DD * or DD DATA) data set (MVS only)
- A system output (sysout) data set (MVS only)
- A unit record device
- A terminal
- A nonreusable VSAM data set
- A tape file (CMS only)
- An unlabeled tape data set
- A file with the file type TXTLIB, LOADLIB, or MACLIB (CMS only)
- A file on a read-only disk, or on a CMS read-only extension.

Nor can you delete the following:

- A concatenation of data sets within a single ddname (MVS only)
- A set of subfiles, that is, files referred to by ddnames FTnnF001, FTnnF002, and so on.

If you attempt to delete a set of subfiles and you have not read or written data beyond the first subfile (ddname FTnnF001) during the current connection, only the first subfile is deleted and no error is detected. This is discussed in more detail under “Processing Subfiles” on page 170.

If OCSTATUS is in effect or the file is dynamically allocated and you specify DELETE for an existing file that cannot be deleted, the file will only be disconnected, as if you had specified KEEP, and an error will be detected.

Error Checking: The IOSTAT and ERR specifiers allow for error checking during processing of the CLOSE statement. These specifiers are discussed in *VS FORTRAN Version 2 Language and Library Reference*.

Disconnecting Files with the OPEN Statement

If you use an OPEN statement for a named file, and a different file is already connected to the specified unit, that file will be disconnected. In such case, you need not use a CLOSE statement for the file that is automatically disconnected, although it is good documentation practice to do so.

In the following example, the file named ATOM1 is disconnected when the OPEN statement for ATOM2 is used.

```
OPEN (13, FILE='ATOM1')
READ (13, FMT=10) A, B
OPEN (13, FILE='ATOM2')
WRITE (13, FMT=10) A, B
```

Files are disconnected with the status of KEEP, except for temporary files, which are disconnected with the status of DELETE.

Disconnecting Files at Program Termination

When your program terminates, all files that remain connected are disconnected.

Files are disconnected with the status of KEEP, except for temporary files, which are disconnected with the status of DELETE.

Reconnecting Files

Sometimes it is necessary to reconnect a file that you have disconnected in the current program. How you reconnect a file depends on whether the file is named or unnamed. For preconnected files, it also depends on whether OCSTATUS or NOOCSTATUS is in effect.

Reconnecting Named Files

To reconnect a named file, you simply use another OPEN statement with the same file name given in the FILE specifier. The same rules that you followed for coding the original OPEN statement apply.

Keep in mind, however, that the status of the file might have changed. If you did not delete the file when you disconnected it, you should code STATUS='OLD' on the OPEN statement; but if you did delete it, you should code STATUS='NEW'. If OCSTATUS is in effect, an inconsistency between actual existence and what you code on the STATUS specifier will cause the processing of the OPEN statement to fail. If you are unsure of the existence of a file, you can code STATUS='UNKNOWN' or omit the STATUS specifier.

Reconnecting Unnamed Files

As with a named file, you reconnect an unnamed file by using the OPEN statement. In this case, however, you must not code the FILE specifier.

Before reconnecting an unnamed file, you must first use a CLOSE statement to disconnect any file that is already connected to the unit. The following examples illustrate what happens if you do *not* use the CLOSE statement.

```
OPEN (8, BLANK='NULL')
READ (8, FMT=10) A, B, C
```

Assuming that no previous I/O statements have been directed to unit 8, the above READ statement is directed to the unnamed file associated with unit 8.

```
OPEN (8, FILE='MYDATA', STATUS='OLD', ACCESS='SEQUENTIAL')
READ (8, FMT=10) D, E
```

The second OPEN statement breaks the connection and associates unit 8 with the file named MYDATA.

```
OPEN (8, BLANK='ZERO')
READ (8, FMT=10) F, G, H
```

The third OPEN acts only on the connected file, which is MYDATA, and the file position remains unchanged; that is, the file does not get repositioned to the beginning. Therefore, the last READ is directed to MYDATA, and not to the unnamed file.

The following example is the same as the above except that a CLOSE statement disconnects the named file. Therefore, the final OPEN statement reconnects the unnamed file and the last READ statement is directed to that file:

```
OPEN (8, BLANK='NULL')
READ (8, FMT=10) A, B, C
OPEN (8, FILE='MYDATA', STATUS='OLD', ACCESS='SEQUENTIAL')
READ (8, FMT=10) D, E
CLOSE (8)
OPEN (8, BLANK='ZERO')
READ (8, FMT=10) F, G, H
```


Reconnecting Preconnected Files

If OCSTATUS is in effect, you must use the OPEN statement with no FILE specifier to reconnect a preconnected file. However, if NOOCSTATUS is in effect, you can reconnect a preconnected file by issuing any of the following statements:

```
OPEN
READ
WRITE
BACKSPACE
ENDFILE
```

As with all unnamed files, you must first disconnect any file already connected to the unit before using any of the above statements.

The example below illustrates how to reconnect a preconnected file using an OPEN statement. In this case, OCSTATUS or NOOCSTATUS can be in effect.

```
1  READ (2, '(BN, 3E10.3)', END=2) A, B, C
   GO TO 1
2  CLOSE (2)
C
   OPEN (2)
3  READ (2, '(BN, 3E10.3)', END=4) A, B, C
   GO TO 3
4  CLOSE (2)
   END
```

The following example illustrates how to reconnect a preconnected file using a READ statement. In this case, NOOCSTATUS must be in effect.

```
1  READ (2, '(BN, 3E10.3)', END=2) A, B, C
   GO TO 1
2  CLOSE (2)
C
3  READ (2, '(BN, 3E10.3)', END=4) A, B, C
   GO TO 3
4  CLOSE (2)
   END
```


Chapter 8. Using File Access Methods

This chapter gives an overview of the file access methods used by Fortran and explains coding of the I/O statements specific to each access method. Figure 43 shows the input/output statements available in VS FORTRAN, the access methods to which they apply, and the operations they perform.

Figure 43. VS FORTRAN Input/Output Statements

I/O Statement	Access Method	Operation
BACKSPACE	Nonstriped sequential and Keyed	For sequential access, positions a file at the beginning of the last record that was written or read. For keyed access, positions a file at a point before the current record.
CLOSE	Sequential, Direct, and Keyed	Disconnects a file from a unit
DELETE	Keyed	Removes a record from a file.
ENDFILE	Sequential	Writes an endfile record on an external file.
INQUIRE	Sequential, Direct, and Keyed	Provides information about a file or unit.
OPEN	Sequential, Direct, and Keyed	Connects a file to a unit; creates a file that is preconnected; creates a file and connects it to a unit; changes certain specifiers of the connection between a file and a unit.
PRINT	Sequential	Transmits data from an area of main storage to an external file.
READ	Sequential, Direct, and Keyed	Transmits data from an external or internal file to an area of main storage.
REWIND	Sequential and Keyed	For sequential access, positions a file at the beginning of the first record in the file. For keyed access, positions a file at the first record with the lowest value of the key of reference.
REWRITE	Keyed	Replaces a record in a file.
WAIT	Sequential	Completes the data transmission begun by the corresponding asynchronous READ or WRITE statement.
WRITE	Sequential, Direct, and Keyed	Transmits data from an area of main storage to an external or internal file.

Input/Output Operations for Sequential Access

This section gives an overview of the sequential access method used by VS FORTRAN, and explains coding I/O statements specific to sequential access.

In a file connected for *sequential access*, records are read or written consecutively, from the first record in the file to the last.

The types of physical files you can connect for sequential access are:

- Non-VSAM files. (For example, files on tape, terminals, printers, card readers, and punches are always accessed sequentially. In addition, many disk files are

Sequential Access

accessed sequentially. An exception is a disk file created by a language other than Fortran and that was not written sequentially.)

- VSAM entry-sequenced data sets
- VSAM relative-record data sets
- Striped files. Striped I/O can be performed only on unnamed files of all record formats, and for all forms of formatted and unformatted I/O.

In many of the above types of files, the records may vary in length.

The following sections discuss special processing that applies to files connected for sequential access.

Connecting Files

You can connect a file using an OPEN statement. For information on preconnecting files, see “Preconnecting Data Files to Units” on page 143.

Note: You cannot use POSITION=APPEND on an OPEN statement for other than a DASD or tape file.

Reading Data

You can use sequential access to read formatted or unformatted data. Reading formatted data is described below; for information about reading unformatted data, see “Reading Unformatted Data” on page 194.

On the READ statement, the optional END specifier is available to branch to another statement in the same program when the endfile record is encountered. For example, after the following READ statement is processed and the end of file is reached, control transfers to the statement labeled 200.

```
READ (*, *, END=200) R4VAR, I2ARR, CH4
```

You can use the END specifier on all forms of the READ statement, except for asynchronous I/O. For information about asynchronous I/O, see “Performing Asynchronous I/O” on page 179.

To format data for input or output, you can rely on either list-directed or NAMELIST formatting, or you can specify your own format.

Formatted records are always in character form. Because data is usually not in character form within main storage, it must be converted to its internal form when you read formatted records.

Note: A file cannot contain both formatted and unformatted records.

Using List-Directed Formatting: With list-directed formatting, the records read consist of a series of constants. The format of the constants is controlled by the type of data that you read.

To read data using list-directed formatting, you use the READ statement. For READ statement coding rules, see *VS FORTRAN Version 2 Language and Library Reference*.

When you use a list-directed READ statement at a terminal, you will be prompted for input with a question mark (?), or a question mark followed by the statement label of the READ statement (for example, ? 00020).

Input records contain a series of constants with separators between them. The first constant is placed in the first item of your input list, the second constant in the second item, and so forth.

Following are the rules for specifying constants and separators in the input records:

- Each constant must agree in type with its corresponding item in the input list, as shown below:

If the data type in the input list is:	The constant must be:
Integer	Integer
Real	Real or Integer
Complex	Complex
Character	Character
Logical	Logical
Byte	Integer
Unsigned	Unsigned Integer

Character constants must be entered within apostrophes. To include an apostrophe within the character data, use two consecutive apostrophes. Note that in a file written with list-directed formatting, character constants are not written within apostrophes. Therefore, you cannot read character constants that have been written with list-directed formatting.

A noncharacter constant must not contain any embedded blanks.

- Each separator can be:

One or more blanks
A comma (,)
A slash (/)

If more than one record is read, there is an implied separator between records.

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 the current READ operation:

- Any items following the slash are not retrieved during the current 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, data from subsequent records, if any, are used to satisfy this list.
- If there are more items in the record than in the data list, the excess items are ignored.

- A null constant is represented by two successive commas. For example:

1, , 3

Note that when you specify a null constant, if the corresponding item in your input list has a value before the READ statement is run, that value is not changed.

- A repetition factor can be specified for any constant, including a null constant. For example, specifying

3*2.6

is equivalent to specifying

2.6, 2.6, 2.6

For a null constant, specifying

2*, 3

is equivalent to specifying

, , 3

Following is an example of reading data using list-directed formatting. If your input is:

b-12.5E12, 33, -44, 'TWO'

running a program with these statements:

```
REAL*4      R4VAR
INTEGER*2    I2ARR(2)
CHARACTER*4  CH4
```

```
READ (11, *)  R4VAR, I2ARR, CH4
```

causes the variables and array elements to be set to the values shown:

```
R4VAR      -0.125E+14
I2ARR(1)    33
I2ARR(2)    -44
CH4         TWO
```

Using NAMELIST Formatting: With NAMELIST formatting, the records read consist of a series of constants, each preceded by the name of the corresponding variable or array in your program. The format of the constants is controlled by the type of data that you read.

An advantage of NAMELIST formatting is that you can specify input items in any order, regardless of the order in which you declare them in your NAMELIST statements. Furthermore, you don't need to code anything for input items that you don't supply.

To read data with NAMELIST formatting, you use a NAMELIST statement in addition to a READ statement. For NAMELIST and READ statement coding rules, see *VS FORTRAN Version 2 Language and Library Reference*.

The NAMELIST name, which you code in the NAMELIST statement, identifies an I/O list. The READ statement that you code then refers to the NAMELIST name on the FMT specifier, instead of providing an I/O list.

The following example shows a NAMELIST statement and a READ statement that refers to it:

```
NAMELIST / MYDATA / R4VAR, I2ARR, CH4
```

```
READ (9, MYDATA)
```

The input/output records for NAMELIST formatting must be in a particular form. For coding rules regarding NAMELIST I/O records, see *VS FORTRAN Version 2 Language and Library Reference*.

For example, if your input data is the following:

```
b&MYDATA I2ARR= 33, -44, CH4='TWO', R4VAR=-12.5E12 &END
```

and you run a program with these statements:

```
REAL*4      R4VAR
INTEGER*2    I2ARR(2)
CHARACTER&smasxt.4  CH4
NAMELIST     / MYDATA / R4VAR, I2ARR, CH4
```

```
READ (12, MYDATA)
```

the variables and array elements are set to the values shown:

```
R4VAR      -0.125E+14
I2ARR(1)    33
I2ARR(2)    -44
CH4         TWO
```

Writing Data

You can use sequential access to write formatted or unformatted data. Writing formatted data is described below; for information about writing unformatted data, see “Writing Unformatted Data” on page 193.

When you write a record to a file connected for sequential access, that record becomes the *last* record in the file. If any records previously existed after this last written record, they are lost.

To write an endfile record, you use the ENDFILE statement. The specifiers for the ENDFILE statement are shown in *VS FORTRAN Version 2 Language and Library Reference*.

The following ENDFILE statement:

```
ENDFILE (UNIT=10, IOSTAT=INT, ERR=300)
```

performs these actions:

- Writes an endfile record on the file connected to unit 10
- Returns a positive or zero value in INT to indicate failure or success
- Transfers control to statement label 300 if an error occurs.

After you use the ENDFILE statement, the endfile record becomes the last record in the file. The file remains connected to the unit. If you are processing a striped file, you cannot use a READ or WRITE statement after the ENDFILE statement. If you are processing a set of subfiles, and you write additional records after the ENDFILE statement, the records are written to the next subfile. Subfiles are discussed under “Processing Subfiles” on page 170.

Note: When processing partitioned data sets, after a member is written, a CLOSE or REWIND statement (not an ENDFILE statement) must be specified for the unit representing the member. This must be done before another member is written. For considerations for using partitioned data sets (PDS), see “Using MVS Partitioned Data Sets” on page 180.

Writing Formatted Data

To format data for input or output, you can rely on either list-directed or NAMELIST formatting, or you can specify your own format.

Formatted records are always in character form. Because data is usually not in character form within main storage, it must be converted to character form when you write formatted records.

Note: A file cannot contain both formatted and unformatted records.

Using List-Directed Formatting: With list-directed formatting, the records written consist of a series of constants. The format of the constants is controlled by the type of data that you write.

To write data using list-directed formatting, you can use either the PRINT or WRITE statement. The PRINT statement is the simpler of the two statements. For PRINT and WRITE statement coding rules, see *VS FORTRAN Version 2 Language and Library Reference*.

The WRITE statement can perform the same functions as the PRINT statement and more. While PRINT has certain defaults built into it, the WRITE statement gives you more control and flexibility with additional specifiers.

For an external file, the UNIT specifier indicates the unit to which the file is connected. (Connecting files to units is discussed under Chapter 7, “Connecting, Disconnecting, and Reconnecting Files for I/O” on page 143.)

With the PRINT statement, you do not specify the unit because the standard output unit is always used by default. However, with the WRITE statement, you can specify the standard output unit as well as others. To specify the standard output unit for the WRITE statement, you code UNIT=*. To specify another unit, you code the unit number; for example, UNIT=2.

Note that the following PRINT and WRITE statements are equivalent:

```
PRINT *, R4VAR, I2ARR, CH4
```

```
WRITE (*, *) R4VAR, I2ARR, CH4
```

Using NAMELIST Formatting: With NAMELIST formatting, the records written consist of a series of constants, each preceded by the name of the corresponding variable or array in your program. The format of the constants is controlled by the type of data that you write.

An advantage of NAMELIST formatting is that you can specify input items in any order, regardless of the order in which you declare them in your NAMELIST statements. Furthermore, you don't need to code anything for input items that you don't supply.

To write data with NAMELIST formatting, you use a NAMELIST statement in addition to a WRITE or PRINT statement. When you format data using NAMELIST, the data is written in a form that can be read using NAMELIST.

For example, if you run a program with these statements:

```
REAL*4      R4VAR      / -12.5E+12 /
INTEGER*2   I2ARR(2)   / 33, -44 /
CHARACTER*4  CH4        / 'TWO' /
NAMELIST     / MYDATA / R4VAR, I2ARR, CH4
```

```
WRITE (2, MYDATA)
```

the following records are generated:

```
b&MYDATA
bR4VAR=-0.125000000E+14,I2ARR=    33,    -44,CH4='TWO'
b&END
```

The constants are formatted in the same way as they are for list-directed formatting.

Using the REWIND statement

To reposition a file to the beginning of its first record, use the REWIND statement. When processing a set of subfiles, the REWIND statement repositions the file to the beginning of the first subfile.

The specifiers for the REWIND statement are shown in *VS FORTRAN Version 2 Language and Library Reference*.

The following REWIND statement:

```
REWIND (UNIT=11, IOSTAT=INT, ERR=300)
```

performs these actions:

- Positions the file connected to unit 11 to its beginning point
- Returns a positive or a zero value in INT to indicate failure or success
- Transfers control to statement label 300 if an error occurs.

After you use the REWIND statement, the file remains connected to the unit.

Using the BACKSPACE statement

To reposition a file to the beginning of the previous record, use the BACKSPACE statement.

Note: The BACKSPACE statement must not be used for the following:

- List-directed formatting
- NAMELIST formatting
- Internal files
- Striped files
- Units connected for PDS members
- Units connected to concatenated data sets
- Units connected to multivolume data sets

The release (RLSE) subparameter of the SPACE parameter on the DD card for a new dataset is ignored if the Fortran program issues a BACKSPACE or ENDFILE instruction to the unit which is connected to the dataset.

Sequential Access

The BACKSPACE statement specifiers are shown in *VS FORTRAN Version 2 Language and Library Reference*.

The following example shows how to use the BACKSPACE statement to reprocess a record that was just written. The first READ statement retrieves the record from the file. The BACKSPACE statement positions the file at the beginning of the record just retrieved. The second READ statement retrieves the same record again for reprocessing.

```
READ (UNIT=11, FMT=500) A, B
BACKSPACE (UNIT=11)
READ (UNIT=11, FMT=600) C, D
```

The following example shows how to use the BACKSPACE statement to replace a record in a file on tape or DASD. The READ statement retrieves the record to be replaced. The BACKSPACE statement positions the file at the beginning of the record just retrieved. The WRITE statement writes the new record. ***After this WRITE is processed, no records exist in the file following this record. Any records that did exist are lost.***

```
READ (UNIT=11, FMT=500) A, B
BACKSPACE (UNIT=11)
WRITE (UNIT=11, FMT=500) A, B
```

A file becomes positioned after the endfile record when you use an ENDFILE statement or use a READ statement that encounters the endfile record. When the file is so positioned, you must take the endfile record into account when backspacing. A single BACKSPACE statement positions the file only to the beginning of the endfile record. At this point, you can extend the file by using a WRITE statement. For example:

```
      READ (8, END=30) A, B, C
      :
30    BACKSPACE (8)
      WRITE (8) D, E, F
```

If you want to position the file to the beginning of the last record containing data, you must use another BACKSPACE statement, as shown below:

```
      READ (8, END=30) A, B, C
      :
30    BACKSPACE (8)
      BACKSPACE (8)
      WRITE (8) D, E, F
```

In the above example, the WRITE statement replaces the last record containing data.

Processing Subfiles

Subfiles are separate physical files that, when appended one to another, form a single file as seen by Fortran. Each subfile has its own file definition with a separate ddname. The file definition for the first subfile must have the ddname FTnnF001, the file definition for the second subfile must have the ddname FTnnF002, and so on.

Do not confuse a set of subfiles with a set of concatenated data sets under MVS. In either case, VS FORTRAN treats each set as a single file. Subfiles, however, require special coding when you write your program; concatenated data sets do not.

An unnamed file connected for nonstriped sequential access can be composed of subfiles. (If file definitions for a striped file are provided, they will override the file definitions for the subfiles.) Use of the POSITION=APPEND specifier of the OPEN statement is not supported for subfiles.

To read or write data to the first subfile, you use READ or WRITE statements in the normal manner. To write data to the second subfile, you use an ENDFILE and a WRITE statement. For example:

```
WRITE (8) A, B, C  
ENDFILE (8)  
WRITE (8) D, E, F
```

Note that the ENDFILE statement only writes an endfile record and does not position the file to the next subfile. It is the WRITE statement that positions the file to the next subfile (see Figure 44 on page 172).

Note: The release (RLSE) subparameter of the SPACE parameter on the DD card for a new dataset is ignored if the Fortran program issues a BACKSPACE or ENDFILE instruction to the unit which is connected to the dataset.

Figure 44. The WRITE Statement Positions the File to the Next Subfile

Statements	Subfile at Which File is Positioned Before Statement	Subfile Acted On
⋮		
WRITE (8) A, B, C	001	001
ENDFILE (8)	001	001
WRITE (8) A, B, C	001	002
⋮		

To read data from the second subfile, you must first read through all the records in the first subfile. When the endfile record is encountered, control passes to the statement indicated by the END specifier. Now, if you use a second READ statement, you will read data from the second subfile. For example:

```

10      READ (8, END=20) A, B, C
      ⋮
      GO TO 10
20      READ (8, END=30) A, B, C

```

The second READ statement positions the file to the second subfile and reads data from it. To read from and write to subsequent subfiles, you follow the same procedures.

To reposition the file to the beginning of the first subfile, you can use a REWIND statement.

To determine which subfile the file is positioned to, you can use the OPENED specifier on the INQUIRE statement. On the INQUIRE statement, you must code the ddname of the subfile on the FILE specifier. Figure 45 on page 173 shows, for a sample program, how the change in file position affects the value assigned to the variable given in the OPENED specifier.

Figure 45. Values Returned for OPENED When You Code FILE='FTnnFmmm'

Statements		Subfile at Which File Is Positioned	Value Returned for OPNn
111	OPEN (8)	—	
	READ (8, FMT=100, END=10) A	001	
	GO TO 111		
C			
10	CONTINUE		
222	READ (8, FMT=100, END=20) B	002	
	GO TO 222		
C			
20	CONTINUE		
333	READ (8, FMT=100, END=30) C	003	
	GO TO 333		
C			
30	CONTINUE		
	INQUIRE (FILE='FT08F001', OPENED=OPN1)	003	FALSE
	INQUIRE (FILE='FT08F002', OPENED=OPN2)	003	FALSE
	INQUIRE (FILE='FT08F003', OPENED=OPN3)	003	TRUE
	INQUIRE (FILE='FT08F004', OPENED=OPN4)	003	FALSE
C			
	REWIND (UNIT=8)	001	
C			
	INQUIRE (FILE='FT08F001', OPENED=OPN5)	001	TRUE
	INQUIRE (FILE='FT08F002', OPENED=OPN6)	001	FALSE
	INQUIRE (FILE='FT08F003', OPENED=OPN7)	001	FALSE
	INQUIRE (FILE='FT08F004', OPENED=OPN8)	001	FALSE
	:		

Finally, if you have read or written data beyond the first subfile you cannot delete the file, or any subfile, during the current connection. If you do, however, an error is detected and the file is disconnected as though STATUS=KEEP were specified. If you have *not* read or written data beyond the first subfile, only the first subfile is deleted and no error is detected.

For example, in the code shown in Figure 46, an error is detected for the first CLOSE statement because data has been read from the second subfile. However the second CLOSE statement causes the first subfile to be deleted and no error is detected.

Figure 46. Result of Attempting to Delete A Set of Subfiles for which Deletion Is Not Allowed

Statements		Subfile Acted On	Result
5	OPEN (8)	001	
	READ (8, '(2I9)', END=10) I, J	001	—
	GO TO 5		
10	READ (8, '(2I9)') I, J	002	—
	CLOSE (8, STATUS='DELETE')	—	Error
	OPEN (8)	001	
	READ (8, '(2I9)', END=20) I, J	001	—
	CLOSE (8, STATUS='DELETE')	—	First subfile deleted
	:		

Note that when the first subfile is deleted, a subsequent INQUIRE statement inquiring about the existence of FTnnF001 will indicate that it does not exist.

However, if the other subfiles existed before the CLOSE statement was processed, subsequent INQUIRE statements for these files will indicate that they do exist.

Using Internal Files

As explained earlier under Chapter 6, “I/O Concepts and Terminology” on page 133, an *internal file* is located in main storage and is a character variable, character array element, character array, or character substring.

Internal files allow you to move data from one internal storage area to another while converting the data from one format to another. This gives you a convenient and standard method of making such conversions.

To read from and write to internal files, you use the READ and WRITE statements. Internal files always contain formatted data. To format the data, you can use list-directed or NAMELIST formatting, or you can specify your own format.

The only difference in coding the READ and WRITE statements is that in the UNIT specifier, instead of coding a unit identifier, you code the name of the character variable, character array element, character array, or character substring.

The following is an example of writing data to an internal file using list-directed formatting.

If you run a program with these statements:

```
REAL*4      R4VAR      / -12.5E+12 /  
INTEGER*2    I2ARR(2)   / 33, -44 /  
CHARACTER*4  CH4        / 'TWO' /  
CHARACTER*35 OLSTDIR
```

```
WRITE (OLSTDIR, *) R4VAR, I2ARR, CH4
```

the internal file, that is, the character variable OLSTDIR, is set to this value:

```
b-0.125000000E+14      33      -44 TWO
```

Internal files are especially useful when you don't know the arrangement of data within the records of an external file. Transferring the data into an internal file allows your program to examine a part of a record and, based on some condition within the record, process the rest of the record accordingly. Furthermore, the record can be reread many times, if required, without the need to backspace and read from the physical device again.

For example, in the sample program in Figure 47 on page 175, an unformatted record is read from an external file. The data is placed in the character variable RECORD. The first character of the variable is then examined to determine the type of data that follows. If the data type is integer and the record is long enough to contain five integer values, the data is transferred to the array INTEGERS by means of a READ statement, which treats the variable RECORD as an internal file. If the data type is real and the record is long enough to contain five real values, the data is transferred to the array REALS.

```

      CHARACTER*80  RECORD
      REAL*4        REALS(5)
      INTEGER*4     INTEGERS(5)
      INTEGER*4     LEN

1     READ (1, NUM=LEN, END=2) RECORD
C
      IF (RECORD(1:1) .EQ. 'I' .AND. LEN .GE. 26) THEN
          READ (RECORD, '(1X, BN, 5I5)') INTEGERS
C
C          Code to process integer values goes here
C
      ELSE IF (RECORD(1:1) .EQ. 'R' .AND. LEN .GE. 51) THEN
          READ (RECORD, '(1X, BN, 5E10.3)') REALS
C
C          Code to process real values goes here
C
      ELSE
C
C          Code to process other types of records goes here
C
      ENDIF
      GO TO 1
2     CONTINUE
      END

```

Figure 47. Sample Program—Transferring Data to an Internal File

Specifying the Number of Buffers for DASD and Tape Input/Output (MVS Only)

You may want to improve the performance of sequential I/O by increasing the I/O data transfer rate. You can do this by increasing the number of DASD or tape I/O buffers that your program uses. You may specify between 1 and 255 buffers.

You can specify how many DASD I/O buffers to use for a specified unit by using either JCL or the FILEINF service subroutine. If you do not use either FILEINF or JCL to specify the number of buffers, then the VS FORTRAN installation default value will be used. The following describes how to specify the number of buffers to be used for unnamed and named files.

For Unnamed Files: For unnamed files, FTnnF001, you must use JCL to specify a BUFNO value. For example, you may have specified the following JCL:

```
//FT90F001 DD UNIT=SYSDA,DSN=MYDSN,DCB=(BUFNO=100)
```

for the following OPEN statement:

```
OPEN (UNIT=90,ACCESS='SEQUENTIAL',FORM='UNFORMATTED')
```

For Unnamed Temporary Files: For unnamed files that are temporary files (STATUS='SCRATCH'), which are obtained by dynamic allocation, you must use FILEINF to specify a BUFNO value. For example:

```
CALL FILEINF (IRC,'BUFNO',10)
OPEN (UNIT=91,STATUS='SCRATCH',...)
```

For Named Files that are Dynamically Allocated: For dynamically allocated named files, you must use FILEINF to specify a BUFNO value. For example:

```
CALL FILEINF (IRC,'BUFNO',100)
OPEN (UNIT=90,FILE='/MYDSN',...)
```

For Named Files that are Not Dynamically Allocated: For named files that are not dynamically allocated, you must use JCL to specify a BUFNO value. For example, you may have specified the following JCL:

```
//MYDD DD DSN=MYDSN,DCB=(BUFNO=20),...
```

for the following OPEN statement:

```
OPEN (UNIT=92,FILE='MYDD',...)
```

Notes:

1. Dynamic allocation is available for DASD I/O only.
2. BSAM allows up to 255 buffers. If you specify a BUFNO value greater than 255, the value 255 will be used.⁴ There is never a situation where more than 255 buffers are used.

Processing Striped Files

A *striped file* is a file as seen by Fortran in which the data is to be interleaved from several physical files (or stripes). Each stripe has a separate file definition.

The use of striped files is a unique feature of VS FORTRAN Version 2 and can only be used with VS FORTRAN programs.

Depending on the availability of storage and other system considerations, you may be able to speed up the data transfer rates for I/O by processing the file as a striped file (MVS only). This allows portions of the file to be read or written simultaneously on multiple channels.

A file can be striped across multiple disks only or tapes only. On MVS, a file can also be striped across a mixture of disks and tapes. Striped files can be processed on CMS, but there will be no increase in the data transfer rate.

Before you can preconnect the striped file you must decide on the data set names or file types for the stripes to be used. The following describes the considerations for determining the names.

Choosing Stripe Names

The MVS data set name, CMS file type, or CMS file identifier (tape) of a stripe must end in the form xxxPy, where xxx is 1 to 3 digits indicating the total number of stripes in the file, and yy is 1 to 3 digits indicating the stripe number. The number of digits in yy must be the same as in xxx.

On MVS, the data set names for the stripes of a file must all have the same qualifiers, except for the stripe number, to uniquely identify the stripes as belonging to the same file. MVS DASD data set names can be from 4 to 44 characters long. MVS tape data set names can be from 4 to 17 characters long. The minimum data set name that can be specified is axPy, where a is any single alphabetic character.

On CMS, the file names for DASD must be the same. The file types or file identifiers except for the value yy, must also be the same. The minimum file type or file identifier that can be specified is xPy. The CMS file type can be from 3 to 8 characters long, and the file identifier can be from 3 to 17 characters long.

⁴ If you are using a release of DFSMS earlier than 1.1, the number of buffers is limited to 99

Sample MVS Data Set Names: The data set names for a file with 10 stripes could take the form:

```
DATAFILE.QUAL1.BN10P01
DATAFILE.QUAL1.BN10P02
:
DATAFILE.QUAL1.BN10P10
```

Sample CMS File Identifiers: The file identifiers for a file with 9 stripes could take the form:

DASD	TAPE
DATAFILE ABC9P1 A1	ABC9P1
DATAFILE ABC9P2 A1	ABC9P2
:	:
DATAFILE ABC9P9 A1	ABC9P9

Defining Striped Files

To preconnect a file for striped sequential access, each stripe must have a separate file definition. The file definition must have the ddname *FTnnP001* for the first stripe, *FTnnP002* for the second stripe, and so on. When both ddnames *FTnnF001* and *FTnnP001* are specified, *FTnnP001* is used.

The stripe number *mmm* of the ddname *FTnnPmmm* must match the stripe number *yyy* on the file type. Note that the stripe number *mmm* must have 3 digits, where the stripe number *yyy* can have 1 to 3 digits.

Specifying Multiple ddnames on MVS: The following shows the required JCL to process a striped file with 6 stripes:

```
//MYJOB JOB ...
:
//FT10P001 DD DSN=MYNAME.MYFILE.ABC6P1
//FT10P002 DD DSN=MYNAME.MYFILE.ABC6P2
//FT10P003 DD DSN=MYNAME.MYFILE.ABC6P3
//FT10P004 DD DSN=MYNAME.MYFILE.ABC6P4
//FT10P005 DD DSN=MYNAME.MYFILE.ABC6P5
//FT10P006 DD DSN=MYNAME.MYFILE.ABC6P6
:
```

Specifying Multiple ddnames on CMS: The following shows the required CMS FILEDEF statements to process a striped file with 9 stripes on disk:

```
FILEDEF FT10P001 DISK FILE1 STP9P1 A4 (RECFM F LRECL 800)
FILEDEF FT10P002 DISK FILE1 STP9P2 A4 (RECFM F LRECL 800)
FILEDEF FT10P003 DISK FILE1 STP9P3 A4 (RECFM F LRECL 800)
FILEDEF FT10P004 DISK FILE1 STP9P4 A4 (RECFM F LRECL 800)
FILEDEF FT10P005 DISK FILE1 STP9P5 A4 (RECFM F LRECL 800)
FILEDEF FT10P006 DISK FILE1 STP9P6 A4 (RECFM F LRECL 800)
FILEDEF FT10P007 DISK FILE1 STP9P7 A4 (RECFM F LRECL 800)
FILEDEF FT10P008 DISK FILE1 STP9P8 A4 (RECFM F LRECL 800)
FILEDEF FT10P009 DISK FILE1 STP9P9 A4 (RECFM F LRECL 800)
```

The following shows the required CMS FILEDEF statements to process a striped file with 4 stripes on tape:

```
FILEDEF FT10P001 TAP1 NL (RECFM F LRECL 800  
LABELDEF FT10P001 FID STP9P1  
FILEDEF FT10P002 TAP2 NL (RECFM F LRECL 800  
LABELDEF FT10P002 FID STP9P2  
FILEDEF FT10P003 TAP3 NL (RECFM F LRECL 800  
LABELDEF FT10P003 FID STP9P3  
FILEDEF FT10P004 TAP4 NL (RECFM F LRECL 800  
LABELDEF FT10P004 FID STP9P4
```

Considerations for Defining Striped Files: The following parameters for each file definition of a striped file must be consistently defined for RECFM, LRECL and BLKSIZE. Values specified on the first stripe for these parameters are automatically used for subsequent stripes. If you define values on the subsequent stripes, the values must be equal to the values that you specified on the first stripe.

When you provide a file definition under MVS, the BUFNO option allows you to specify the number of buffers to use per stripe of a file. The value specified on the first stripe for BUFNO is used for subsequent stripes. If you define BUFNO values on the subsequent stripes, the values are ignored. The number of simultaneous I/O operations performed is one less than the value of BUFNO.

If BUFNO is not specified for a striped file, the number of buffers to be used for I/O defaults to the greater value of 7 or the UAT default. If the UAT default is 10, BUFNO defaults to 10. If the UAT default to 4, BUFNO defaults to 7.

Creating a Striped File from an Existing File

1. Identify the unit you want to use for the striped file.
2. Decide how many stripes to create.
3. Code and compile a program to read the existing file into the striped file.
4. Provide a file definition for each stripe.
5. Run the program to read the existing file into the striped file.

Note: Existing programs do not need to be changed to use the new striped file.

Considerations for Using Striped Files

For the greatest improvement in the data transfer rate:

- Run the striped files under MVS. While striped files run correctly under CMS, the data transfer rate is not improved.
- Ensure that you use only large primary allocations. Using secondary allocations affects the data transfer rates.
- Ensure that the I/O paths are independent, that is, that no two stripes share a control unit with cross-connect.
- Ensure that the amount of virtual storage to be used for the file corresponds to an equal amount of real storage. Increased paging requirements may negate the gains in the data transfer rates.

Striped sequential access may be performed only on DASD or tape, and may not be performed on the following:

- VSAM files
- Partitioned data sets

The error message unit

Named files (unnamed files cannot be used with MTF)

In-stream (DD * or DD DATA) data sets (MVS only)

System output (SYSOUT) data sets (MVS only)

Files processed using BACKSPACE statements, asynchronous I/O or random I/O.

Use of the POSITION=APPEND specifier of the OPEN statement is also not supported for striped files.

Performing Asynchronous I/O

Under MVS, you can transfer unformatted data between external files and arrays in main storage, and while the transfer is taking place, continue other processing within your program. The files must be preconnected for sequential access and must allow variable-length spanned records. Asynchronous I/O is allowed only for nonstriped sequentially accessed files, and is not allowed in the MTF environment. Arrays located in extended common areas cannot participate in asynchronous I/O.

To transfer data from an external file to an array, you code the READ statement in addition to a WAIT statement. Between the READ and the WAIT statements, you can use any other statements so long as they do not refer to the input items specified on the READ statement.

Similarly, to transfer data from an array to an external file, you code a WRITE and a WAIT statement. Any statements that you use between the WRITE and WAIT statements can refer to the output items, but should not modify them.

Each READ or WRITE statement transfers only one record. For coding rules for asynchronous READ, WRITE and WAIT statements, see *VS FORTRAN Version 2 Language and Library Reference*.

In the *list* option of the READ statement, you can specify an entire array or part of an array. To specify an entire array, you code the array name. In the following example, the entire array named DATA is retrieved.

```
READ (9, ID=1) DATA
```

You can specify part of an array in three ways, as shown in the examples below. In the first example, the array element DATA(5) and all the array elements up to and including DATA(30) are retrieved. In the second example, the array element DATA(30) and all the array elements following it are retrieved. In the third example, the array element DATA(30) and all the elements preceding it are retrieved. **The ellipsis (...) is an integral part of the syntax and must appear in the positions shown.**

```
READ (9, ID=1) DATA(5) ... DATA(30)
```

```
READ (9, ID=1) DATA(30) ...
```

```
READ (9, ID=1) ... DATA(30)
```

To retrieve just one array element, you must specify it in the first way shown above. For example:

```
READ (9, ID=1) DATA(5) ... DATA(5)
```

Sequential Access

On the WAIT statement, you specify the same unit and identifier that you specified on the pending READ or WRITE statement. For instance:

```
READ (9, ID=1) DATA
:
:
(other processing)
:
:
WAIT (9, ID=1)
```

Using MVS Partitioned Data Sets

A partitioned data set (PDS) consists of independent groups of sequential data called *members* of the data set. Each member of a partitioned data set is identified by a member name in a directory.

Partitioned data sets are used to contain libraries of related data. For example, the results obtained from running a Fortran program might be written to a PDS in which each member contains the output data corresponding to one set of input data.

PDS members can be accessed only sequentially, and can consist of either formatted or unformatted records. Under VS FORTRAN Version 2, PDS members can be created, read from, written to, and replaced entirely, using sequential I/O statements; however, they cannot be opened with POSITION=APPEND. Following are some considerations and restrictions on the use of partitioned data sets:

- Existing members of a PDS cannot be updated, extended, or shortened, only replaced entirely; if you want to add records to a PDS member, you must completely rewrite the member.
- Each PDS member must be fully processed and closed before another member of the same PDS is opened and written to. You can, however, open and read multiple members of the same PDS at the same time. After a member is written, a CLOSE or REWIND statement must be specified for the unit representing the member before another member of the same PDS is written. If your program both reads from and writes to a PDS member, you must code a REWIND statement before switching from reading to writing, or from writing to reading.
- You cannot use the ENDFILE statement on a PDS member.
- If you do not specify the BUFNO parameter in the DD statement, the default of BUFNO=1 is used for PDS members.
- You can use the BACKSPACE statement only on a PDS member processed for input. If you use the BACKSPACE statement on a PDS member, use the default BUFNO=1 in the DD statement. Your program should not attempt to use the BACKSPACE statement on a PDS member after end-of-data is reached.
- A separate DD statement is required for each member created in the same program.
- The WRITE and OPEN statements create PDS members.
- When you are processing members for input only, you can specify the LABEL parameter and its subparameter IN in the DD statement, or ACTION='READ' on the OPEN statement.

When you are processing members for output only, you can specify the LABEL parameter and its subparameter OUT in the DD statement, or ACTION='WRITE' on the OPEN statement.

- To read multiple members of a PDS under one unit number, you must process the PDS as an unnamed file, and use the READ statement with the END= parameter. The end-of-data transfer specified by the END= statement label increases the file sequence number, so that members can be read one-by-one. You must code a separate DD statement for each file being read with a ddname of the form FTnnFmmm, where nn is the two-digit unit number, and mmm is the three-digit sequence number.

Input/Output Operations for Direct Access

This section gives an overview of the direct access method used by Fortran, and explains coding I/O statements specific to direct access.

Records in a file connected for *direct access* are arranged in the file according to their relative record numbers, which you specify when you write the records. All records are the same size and each record occupies a predefined position in the file, determined by its relative record number. You can read and write the records in any order. You cannot delete records, but you can replace them. When you replace a record, you do not affect any other records in the file, as you would with sequential access.

The types of physical files you can connect for direct access are:

- Non-VSAM disk files with fixed-length unblocked records
- Nonstriped files
- VSAM relative-record data sets.

If the file has an endfile record (any file created by a Fortran program has an endfile record), the endfile record is not considered to be part of the file when the file is connected for direct access.

The following sections discuss special processing that applies to files connected for direct access.

Connecting Files

To connect a file for direct access, you must use an OPEN statement. In the OPEN statement, you must specify RECL=*rcl*, where *rcl* is the record length. Measure the length in characters for formatted records and in bytes for unformatted records.

The OPEN statement can connect an existing file or create a new file. When a new file is created, dummy records are written throughout the space allocated for the file. The dummy records contain X'FF' in the first position and X'00' in the remaining positions. When creating new, direct, non-VSAM data sets, only the primary allocation and any existing extents of the secondary allocation of a data set are formatted.

Reading and Writing Data

On the READ and WRITE statements, you must specify `REC=rec`, where *rec* is the relative record number. For the first record, this number is 1. Following is an example of a READ statement that retrieves record number 28.

```
READ (UNIT=14, REC=28) A, B, C
```

When reading or writing formatted records, you can use the slash (/) format code, which allows data transfer to or from multiple records. In this case, data transfers to or from the records *n*, *n*+1, and so on, where *n* is the record number given in the REC specifier.

Input/Output Operations for Keyed Access

This section gives an overview of the keyed access method used by Fortran, and explains coding I/O statements specific to keyed access.

In a file connected for keyed access, records are identified by a field, called a *primary key*, that contains a unique value, such as an employee number.

In addition, the records in the file may be identified by other fields, called *alternate keys*, whose values need not be unique, but can be restricted to be unique if necessary.

Each key is in the same relative position in each record. For example, all the records might have a primary key in positions 16 through 19 and an alternate key in positions 1 through 3. The records can vary in length, but must be long enough to contain the primary key and all alternate keys that are defined for the file, regardless of whether your program refers to them.

You can directly retrieve a record anywhere in the file by referring to its primary key or one of its alternate keys.

In addition, once you retrieve a record, you can retrieve other records, based on the same key, in the sequential order of their key values.

Keyed access offers the flexibility of direct access with the additional flexibility of being able to retrieve records based on specific fields. Moreover, because you can identify records by more than one key, you don't need to store multiple copies of the same information for different applications.

Only a VSAM key-sequenced data set (KSDS) can be connected for keyed access. Before you can refer to a VSAM KSDS in your Fortran program, the data set must be defined using the Access Method Services utility program. When the data set is defined, all of its characteristics are specified. These characteristics include the position and length of the primary and any alternate keys, and whether the values for specific alternate keys must be unique. You cannot change any of these characteristics in your program.

The following sections explain special processing that applies to files connected for keyed access.

Connecting Files

To connect a file for keyed access, you must use the OPEN statement.

On the OPEN statement, you code ACCESS='KEYED' to specify that the file is to be connected for keyed access.

To indicate the kind of processing you will do with the file, you code the ACTION specifier, as shown in Figure 48 on page 184. If you omit the ACTION specifier, the default is ACTION='READ'.

Figure 48. Coding the ACTION Specifier on the OPEN Statement

If you want to:	You must specify:
Load new records into a file that doesn't exist, that is, an empty VSAM KSDS	ACTION='WRITE'
Write new records onto the end of an existing file	ACTION='WRITE' or ACTION='READWRITE' ("Loading New Records into a File" explains when to specify each of these)
Retrieve records	ACTION='READ'
Just update or both update and retrieve records	ACTION='READWRITE'

In the KEYS specifier, you give the starting and ending positions of the keys you will use during the current connection. If you are loading new records into a file (ACTION='WRITE'), you can specify only the primary key. However for retrieval and update operations (ACTION='READ' or ACTION='READWRITE'), you can specify any key or keys; you don't have to specify the primary key. You also don't have to specify the keys in any particular order. If you want to use only one key (for example, when loading new records into a file), you can omit the KEYS specifier.

In the example below, one key starts at position 16 of the record and ends at position 19. Another key starts at position 1 and ends at position 3.

```
OPEN (8, ACCESS='KEYED', ACTION='READWRITE', KEYS=(16:19, 1:3))
```

For files that are password-protected, you must specify a password with the PASSWORD specifier. If you specify ACTION='READ', the file's read password is required; otherwise, its update password is required.

For each key that you specify in the KEYS specifier of your OPEN statement, you must supply a separate file definition. Each file definition refers to a VSAM path or base cluster that represents one of the keys. (For information about VSAM paths and base clusters, see Chapter 11, "Using VSAM" on page 209.)

The ddnames for the file definitions must be of a special form. For a named file, you add the suffix 1, 2, 3, and so on to the ddname for each additional key. For example, if the name of the file is NEWDATA and you specify one key, the ddname for its file definition must be NEWDATA. If you specify two additional keys, the ddnames for their file definitions must be NEWDATA1 and NEWDATA2.

For an unnamed file, if you specify one key, the ddname of its file definition must be FTnnK01, where nn is the 2-digit unit number. If you specify additional keys, the ddnames of their file definitions must be FTnnK02, FTnnK03, and so on.

Loading New Records into a File

Loading new records means writing new records into the file in ascending collating sequence by the primary key value. If you have a file that doesn't exist, that is, an empty VSAM KSDS, you must load new records into it before you can do any other processing. (This may be only one record.) If you have an existing file, you can also load new records onto the end of it. In either case, you must specify ACTION='WRITE' on the OPEN statement. With ACTION='WRITE', loading the new records is the only operation you can perform during the current connection.

You can also write new records onto the end of an existing file by using ACTION='READWRITE'. If you specify ACTION='READWRITE', you do not have to ensure that the records are in ascending sequence. However, the processing time will be greater because VSAM must order the records.

In all of the above cases, you write the records by using a WRITE statement for each record. (If you are writing formatted records, you cannot use the slash (/) format code to advance to the next record.)

The DUPKEY specifier on the WRITE statement allows you to transfer control to another statement when a key value duplicates one that already exists in another record. If you omit the DUPKEY specifier, control passes to the statement indicated by the ERR specifier, and the integer variable or array element specified by the IOSTAT specifier, if any, is given the value 135. If you also omit the ERR specifier, an error is detected.

Below is a sample program that loads records. The records contain three fields, to which data is written from the variables DEPTN, NAM, and EMPN. The third field, which occupies positions 16 through 19, is the key given in the KEYS specifier of the OPEN statement. Therefore, in order for the loading to complete successfully, the records must be supplied in ascending order based on the value contained in EMPN. If a key value duplicates one that already exists in the file, control transfers to statement 40.

```

      INTEGER*4      EMPN
      CHARACTER*3    DEPTN
      CHARACTER*12   NAM

      OPEN (8, ACCESS='KEYED', ACTION='WRITE', KEYS=(16:19))
10  READ (1, FMT=99, END=20) NAM, EMPN, DEPTN
      WRITE (8, FMT=98, DUPKEY=40) DEPTN, NAM, EMPN
      GO TO 10
20  CLOSE (8)
      :
98  FORMAT (A3, A12, I4.4)
99  FORMAT (A12, I4, A3)
      :

```

Reading Data

To retrieve records, you use a READ statement for each record. (If you are reading formatted records, you cannot use the slash (/) format code to advance to the next record.) Before you can successfully retrieve records from a file, the file must have been loaded.

On the OPEN statement, you must specify ACTION='READ' (the default) or ACTION='READWRITE'. If you don't intend to update the file, specify ACTION='READ'.

Specifying a Key of Reference

If you specify multiple keys in the KEYS specifier of the OPEN statement, you indicate the key to be used for the retrieval by coding the KEYID specifier on the READ statement. In the KEYID specifier, you code the relative position that the key occupies in the list of keys given in the KEYS specifier. For example, if you code the following OPEN statement:

```
OPEN (8, ACCESS='KEYED', ACTION='READ', KEYS=(16:19, 1:3))
```

and want to retrieve a record based on the second key given in the KEYS specifier (that is, the key in positions 1 through 3 of the record), you specify KEYID=2 on your READ statement.

The key that you use in a particular I/O statement is called the *key of reference*. The key of reference remains the same for all subsequent I/O operations on the file until you change it by using another READ statement with the KEYID specifier.

While the file is connected for keyed access, you can retrieve a record either directly or sequentially. The following sections describe how to retrieve records directly and sequentially, and how to combine direct and sequential retrieval methods.

Retrieving Records Directly

For *direct retrieval*, specify a search argument that is compared with key values in the file's records in order to find the record. The record can be anywhere in the file. To specify the search argument, you use the KEY, KEYGE, or KEYGT specifier. You cannot use more than one of these on a single READ statement. For an example of a READ statement for direct retrieval, see Figure 49 on page 187.

The KEY Specifier: To retrieve a record with a key value that identically matches your search argument, use the KEY specifier. For example, assuming that the key of reference is three characters long, if you specify KEY='D51' the first record encountered whose key contains the value D51 will be retrieved. You can also specify a search argument that is shorter than the key value. In this case, the leading portion of the key value in the record must match the search argument. For example, assuming that the key of reference has a length greater than two, if you specify KEY='D5', the first record encountered whose key value begins with D5 will be retrieved.

The KEYGE Specifier: To retrieve the first record whose key value is equal to or greater than your search argument, use the KEYGE specifier. If the file contains a record whose key value is identical, the first such record is retrieved. If not, the first record with the next greater key value is retrieved. If your search argument is shorter than the key, the record retrieved is the first one in which the leading portion of the key value is equal to or greater than your search argument.

The KEYGT Specifier: To retrieve the first record whose key value is greater than your search argument, use the KEYGT specifier. If the key argument is shorter than the key, the record retrieved is the first one in which the leading portion of the key value is greater than your search argument.

The NOTFOUND Specifier: To handle the situation when no record in the file satisfies the search argument, you can code the NOTFOUND specifier, which allows you to branch to another statement when there is no record that satisfies your search criterion.

Note: For the purpose of comparing key values, the data is used in its internal representation (with no editing or conversion) and is interpreted as a string of characters. The data is used in this manner regardless of whether you transferred it from character or noncharacter data items. The value of that string of characters is dependent upon the internal representation of any noncharacter data. Therefore, when you use noncharacter data to form a key in a record, two key values may not have the same relationship to each other when compared as keys as they do when their numeric values are compared. For example, if the key is an integer field in the record, a value of -1 is interpreted as a greater key value than 1.

Retrieving Records Sequentially

For *sequential retrieval*, do not specify a search argument on the READ statement; the key value of the record previously read or updated is used as the starting point and the next record, in increasing sequence of the whole key value, is obtained. The key of reference from the previous I/O statement remains the key of reference for the sequential retrieval.

If the file was just connected, sequential retrieval begins with the record with the lowest key value, using as the key of reference the first of the keys indicated by the OPEN statement.

For formatted records, the order of the key values is the order defined by the EBCDIC collating sequence for the string of characters that forms the key.

For unformatted records, the order of the key values is the order defined by the EBCDIC collating sequence for the data that forms the key on the external medium.

If the key of reference is an alternate key that has duplicate values, you cannot control the order in which records with the same key value are returned.

Combining Direct and Sequential Retrieval

If you want to retrieve a group of records based on the same value in the key or in the first part of the key (for example, a group of records whose 3-position key values begin with D5), you can combine direct and sequential retrieval. First, you use direct retrieval to obtain the first record. Then, you use a series of sequential retrievals to obtain the rest of the records. On your sequential READ statements, you provide the NOTFOUND specifier, which specifies the label of a statement to which control transfers when there are no more records whose leading key values are the same. Figure 49 is an example of this scenario:

```

1 OPEN (8, FORM='FORMATTED', ACCESS='KEYED', KEYS=(16:19, 1:3))
    I=1
2 READ (8, FMT=99, KEYGT='D5', KEYID=2, NOTFOUND=100)
    DEPTNO(I), NAME(I), EMPNO(I)
30 I=I+1
3 READ (8, FMT=99, NOTFOUND=300)
    DEPTNO(I), NAME(I), EMPNO(I)
    GO TO 30
  
```

Figure 49. READ Statements for Direct and Sequential Retrieval

- 1** Opens a file with multiple keys specified
- 2** Directly retrieves the first record with a value greater than D5 in the first two positions of the key. If there is no such record, control transfers to statement 100.
- 3** Sequentially retrieves all of the remaining records whose key values begin with D6, assuming that the direct retrieval obtains a record whose key value begins with D6. Because the NOTFOUND specifier is given, control transfers to statement 300 when there are no more records whose key values begin with D6.

If you don't use the NOTFOUND specifier, the logic of your program must determine when to stop reading more records; otherwise, successive sequential retrieval operations continue to the end of the file (that is, to the record with the highest key). Control then passes to the statement indicated by the END specifier.

You cannot code both the END and NOTFOUND specifiers on the same READ statement.

Updating Files

You can update files by adding new records, replacing existing records, and deleting records.

To update a file, you must specify ACTION='READWRITE' on the OPEN statement.

Adding New Records

To add records to a file, you use the WRITE statement. You must use a single WRITE statement for each record. (If you are writing formatted records, you cannot use the slash (/) format code to advance to the next record.)

The key of reference is determined by the last direct retrieval. Or, if you have not used a direct retrieval, the first key in the KEYS specifier of the OPEN statement is used.

The record will be inserted in the file following the record with a lower key value and preceding the record with a higher key value. If the new record has a key value that doesn't have to be unique and it duplicates the key value of one or more existing records, the new record is written following the last record having the same key value.

The DUPKEY specifier on the WRITE statement allows you to transfer control to another statement when a key value that must be unique duplicates one that already exists in another record. If you omit the DUPKEY specifier, control passes to the statement indicated by the ERR specifier, and the integer variable or array element specified by the IOSTAT specifier, if any, is given the value 135. If you also omit the ERR specifier, an error is detected.

Note that even for a key that you don't specify in the OPEN statement, a duplicate key value can be detected. For example, if your program refers only to alternate keys and you write a record that causes a duplication of a primary key value, this error is detected.

An example of the WRITE statement is:

```
WRITE (8, FMT=98, DUPKEY=40) DEPTN, NAM, EMPN
```

This statement writes data from variables DEPTN, NAM, and EMPN. In order for the record to be written successfully, the value in the variable EMPN must not be duplicated as a key value in the file. If it is, control transfers to statement 40.

Replacing Records

By using the REWRITE statement, you can replace a record that you successfully retrieved by an immediately preceding sequential or direct READ statement. (You may not use any other I/O statements, such as BACKSPACE or WRITE, for the same file between the READ and REWRITE statements.) You can change any data in the record just read except for the values of the primary key and the key of reference.

The specifiers on the REWRITE statement are shown in *VS FORTRAN Version 2 Language and Library Reference*.

When coding the REWRITE statement, you must specify in the output list each item, changed or unchanged, that is to appear in the record.

The following statements demonstrate updating several records by means of a READ, REWRITE sequence.

```
      READ (8, 96, KEY='F10', KEYID=2) DEPTN, NAM, EMPN
40 REWRITE (8, 95) DEPTN, NAM, EMPN, 'MOVING TO BLDG. 10'
      READ (8, 96, NOTFOUND=120) DEPTN, NAM, EMPN
      GO TO 40
```

The direct retrieval obtains the initial record for the key F10, and this record is then replaced.

The sequential retrieval then obtains the next record and control passes to the REWRITE statement, which replaces it. This continues until no more records with the key value of F10 are found, at which point control passes to statement 120, as indicated by the NOTFOUND specifier.

Deleting Records

By using the DELETE statement, you can erase a record that was successfully retrieved by the immediately preceding direct or sequential READ operation. No other I/O operations, such as BACKSPACE or WRITE, may be issued for the same file between the READ and DELETE statements.

The specifiers on DELETE statement are shown in *VS FORTRAN Version 2 Language and Library Reference*.

Repositioning Files

To reposition a file, you can use the REWIND and BACKSPACE statements. You must code ACTION='READ' or ACTION='READWRITE' on the OPEN statement.

Using the REWIND Statement

By using the REWIND statement, you can position the the file to the record having the lowest value for the key of reference; you can then use a sequential READ statement to retrieve that record.

The specifiers on the REWIND statement are shown in *VS FORTRAN Version 2 Language and Library Reference*.

Using the BACKSPACE Statement

By using one or more BACKSPACE statements, you can reestablish the position of a file to a point prior to the current file position. You can then use a sequential READ statement to retrieve the record at which the file is positioned.

For information on BACKSPACE statement specifiers see *VS FORTRAN Version 2 Language and Library Reference*.

If the key of reference has unique key values: The first BACKSPACE statement following a READ, WRITE, or REWRITE statement positions the file to the beginning of the same record that was just read or written.

A BACKSPACE statement following a DELETE statement positions the file to the beginning of the record with the next lower key value. Subsequent BACKSPACE statements position the file to the beginning of the records with successively lower key values.

If the key of reference has nonunique key values: The first BACKSPACE statement following a READ, WRITE, or REWRITE statement positions the file to the first record with the same key value that appeared in the record that was just read or written. A BACKSPACE statement following a DELETE statement that deleted a record which was not the first record with that same key value, also positions the file to the first record with that key value.

However, if the DELETE statement deleted the first record with a given key value, then the BACKSPACE statement positions the file to the first record with the next lower key value. Each subsequent BACKSPACE statement finds successively lower key values and positions the file to the beginning of the first record with those different key values.

Therefore, when the key of reference has nonunique key values, a series of BACKSPACE statements does not position the file to all of the records that would be read with a series of sequential retrieval statements.

For example, a sequence of records in a file might be:

Record	Key Value
n	D47
n+1	D47
n+2	F10
n+3	F10
n+4	F10

Assume you have just read record $n+4$. Then, two consecutive backspaces would position the file as follows:

Record	Key Value
$n+2$	F10 (first backspace)
n	D47 (second backspace)

Because key value is not unique, backspacing causes movement through a group of records to the first record of the group having a specific nonunique key value, and not to the immediately preceding record, as it would if the key were unique.

Using BACKSPACE to Locate the Last Record: You may use BACKSPACE to locate the last record, that is, the record with the highest key value in the file. First, you must position the file beyond the last record. You can do this in one of two ways:

- By using a sequential READ statement with the END specifier after having already read the last record in the file. In this case, control will pass to the statement indicated by the END specifier.
- By using a direct READ statement with a KEYGE or KEYGT specifier which specifies a search argument so large that no record in the file satisfies the search criterion. In this case, control passes to the statement indicated by the NOTFOUND specifier.

Using BACKSPACE at the Beginning or End of a File: A BACKSPACE statement issued when the file is positioned beyond the last record repositions the file to the beginning of the record with the highest key value. (If there is more than one record with this key value, the file is positioned to the first such record.) You can then perform a sequential retrieval to read the record with the highest key value.

Issuing the BACKSPACE statement has no effect if the file is positioned at the beginning of the first record in the file (such as after an OPEN or REWIND statement has been processed). It is not permitted if the previous retrieval or update operation failed for any reason other than reaching the end of the file.

Input/Output Operations for Sequential, Direct, and Keyed Access

Specifying Your Own Format (Formatted I/O)

If you require a different format than what list-directed and NAMELIST formatting offer, you can specify your own. The records must contain series of constants, but you can control their length and format, as well as their positions in the record.

You can specify your own format for a file connected for sequential, direct, or keyed access.

Writing Data: When writing data, you can specify your own format on the PRINT, WRITE, or REWRITE statement. The most commonly used format identifiers on these statements are the statement label, character constant, and character variable. For information on coding formatted PRINT, WRITE and REWRITE statements, see *VS FORTRAN Version 2 Language and Library Reference*.

Sequential, Direct, and Keyed Access

The PRINT, WRITE, or REWRITE statement label refers to a FORMAT statement, where you specify the format in which the data is to be written. Coding rules and format codes for the FORMAT statement are discussed in the *VS FORTRAN Version 2 Language and Library Reference*.

The FORMAT statement must *always* have a label so you can refer to it from other statements. More than one PRINT, WRITE, or REWRITE statement can refer to the same FORMAT statement.

The following is an example of writing formatted data using the FORMAT and PRINT statements. If you run a program containing these statements:

```
REAL*4      R4VAR      / -12.5E+12 /  
INTEGER*2    I2ARR(2)  / 33, -44 /  
CHARACTER*4  CH4       / 'TWO' /
```

```
PRINT 5, R4VAR, I2ARR, CH4  
5 FORMAT (1X, E11.4E2, 2I4.3, A4)
```

the following record is written:

b-0.1250E+14b033-044TWO

Note that in the above FORMAT statement, the format code 1X inserts a blank as a carriage control character.

If you plan to use a particular format only once, you can code it right in the PRINT statement as a character constant. When you specify a character constant, you must delimit it by apostrophes. Within the character constant, the format codes must be within parentheses. You can use any of the format codes that are valid for the FORMAT statement. For example,

```
PRINT '(E11.4E2, 2I4.3, A4)', R4VAR, I2ARR, CH4
```

is equivalent to:

```
PRINT 5, R4VAR, I2ARR, CH4  
5 FORMAT (E11.4E2, 2I4.3, A4)
```

If you need the flexibility of being able to change the format at run time, you can specify a character variable containing the format codes. Again, you can specify any of the format codes that are valid for the FORMAT statement. You must include them within parentheses. Blank characters can precede the left parenthesis. Any data following the right parenthesis is ignored. In the following example, the character variable named FORMS is specified on the PRINT statement:

```
PRINT FORMS, R4VAR, I2ARR, CH4
```

For a file connected for keyed access, the FMT specifier and the same format identifiers are also available on the REWRITE statement.

Reading Data: When reading data, you can specify your own format on the READ statement. For coding rules, see information on formatted READ in *VS FORTRAN Version 2 Language and Library Reference*.

The format identifiers you code on the FMT specifier can be any of those mentioned above for writing data.

Most of the format codes are the same as those used for writing, except that they determine how the data will be read rather than written. A few format codes are specific to writing or reading. For example, the format code BN means ignore blanks in input.

With edit codes F, E, D, Q, Z, I, and G, you can use the comma as an input delimiter to indicate the end of data in a formatted input field. This will eliminate the need for inserting leading and trailing zeroes and blanks. For example, if you have used a format code of I10 in your program, but you specify the following input:

7,

then the input field is limited to one character, instead of the ten characters specified in the format code. In the case of the A format code, which is used for character data, a comma is a valid character; therefore, the comma will not limit the input field of an A type format code.

An example of reading formatted data using the FORMAT and READ statements is the following: If your input is:

b-12.5E12bbb33-44bTWO

running a program with these statements:

```
REAL*4      R4VAR
INTEGER*2    I2ARR(2)
CHARACTER*4  CH4

      READ (10, 5) R4VAR, I2ARR, CH4
5  FORMAT (BN, E11.5E2, 2I4.3, A4)
```

causes the variables and array elements to be set to the values shown:

```
R4VAR      -0.125E+14
I2ARR(1)    33
I2ARR(2)    -44
CH4         TWO
```

Reading and Writing Unformatted Data

The following sections explain how to read and write unformatted data.

If you don't require the records to be in character form, you can save the overhead of conversion by reading and writing unformatted records. An *unformatted record* is a sequence of data in its internal form.

Unformatted data can be read from or written to files connected for sequential, direct, or keyed access.

Note: A file cannot contain both formatted and unformatted records.

Writing Unformatted Data

To write unformatted data, you use the WRITE statement. For files connected for keyed access, you can also use the REWRITE statement. Each WRITE or REWRITE statement processes only one record, writing the data items without conversion. For coding rules for WRITE and REWRITE, see *VS FORTRAN Version 2 Language and Library Reference*.

Sequential, Direct, and Keyed Access

An example of writing unformatted data follows. If you run a program with these statements:

```
REAL*4      R4VAR      / -12.5E+12 /  
INTEGER*2    I2ARR(2)  / 33, -44 /  
CHARACTER*4  CH4       / 'TWO' /
```

```
WRITE (3) R4VAR, I2ARR, CH4
```

the following record, shown in hexadecimal representation, is written:

CBB5E6210021FFD4E3E6D640

Reading Unformatted Data

To read unformatted data, you use the READ statement. As with the WRITE and REWRITE statements, each READ statement processes only one record.

An example of reading unformatted data is the following. If your input record, shown in hexadecimal representation, is:

CBB5E6210021FFD4E3E6D640

running a program with these statements:

```
REAL*4      R4VAR  
INTEGER*2    I2ARR(2)  
CHARACTER*4  CH4  
INTEGER*4    LENGTH
```

```
READ (13, NUM=LENGTH) R4VAR, I2ARR, CH4
```

causes the variables and array elements to be set to the values shown:

```
R4VAR      -0.125E+14  
I2ARR(1)   33  
I2ARR(2)   -44  
CH4        TWO  
LENGTH     12
```

Chapter 9. Advanced I/O Topics

Dynamically Allocating a File	195
File Characteristic Defaults	197
Overriding File Characteristic Defaults	198
Calculation of Primary, Secondary, and Directory Space under MVS	200
Coding INQUIRE in Your Program	201
Sample Program	201
Gathering Information on Dynamically Allocated Files	202
Referring to Values Set by the FILEINF Routine	203

For some files, you can omit coding file definitions and VS FORTRAN will supply them to the system for you. This is called *dynamic file allocation*. Dynamic file allocation allows you to allocate, that is assign resources to, files as they are required by your program, rather than at the time the program is loaded into storage.

The types of files you can dynamically allocate are:

- DASD, named files connected by the OPEN statement with the exception of:
 - VSAM files connected for keyed access under MVS
 - All VSAM files under CMS
- Temporary files (connected with STATUS='SCRATCH'), excluding subfiles.
- Under CMS, preconnected files directed to any of the standard I/O units. The IBM-supplied defaults are units 5, 6, and 7.
- Under MVS, preconnected files directed to the error message unit (and standard output unit for WRITE and PRINT statements if different); the IBM-supplied default is unit 6.

Dynamically Allocating a File

To dynamically allocate a file, you omit the file definition for it. In addition, for named files, you must specify the CMS file identifier or MVS data set name, rather than a ddname, on the FILE specifier of the OPEN statement. For temporary files and preconnected files, you code nothing different on the I/O statements.

For preconnected files under both CMS and MVS, and for temporary files under MVS, VS FORTRAN creates a file definition with the ddname FTnnF001. The file characteristics are determined by installation defaults, discussed below under “File Characteristic Defaults” on page 197, and by the FILEINF service subroutine, discussed below under “Overriding File Characteristic Defaults” on page 198. For temporary files under CMS, the normal CMS default file definition is used.

Following are some examples of OPEN statements for named files:

Under CMS:

```
OPEN (9, FILE='/CALC DATA')
```

```
OPEN (9, FILE='/CALC MACLIB A(DATAMBR)')
```

Under MVS:

```
OPEN (9, FILE='/MILLER.MYPDS.ACCOUNT(ABC)')
```

```
OPEN (9, FILE='/MILLER.BALANCE.YEAR')
```

Warning: From a VS FORTRAN point of view, the CMS file identifier or MVS data set name specified on a file definition is logically different from the name you specify on the FILE specifier, even if the names are the same and both point to the same physical file. Thus, in the following example, the name on the second OPEN statement is not checked against the one in the first OPEN statement even though DD1 refers to the same physical file; therefore, no error is detected for connecting the same file to two different units at a time.

Under CMS:

```
FILEDEF DD1 DISK MYFILE FORT A1  
  
    OPEN (1, FILE='DD1')  
  
    ...  
  
    OPEN (2, FILE='/MYFILE FORT A1')
```

Under MVS:

```
//DD1 DD DSN=MYFILE.FORT,DISP=OLD  
  
    OPEN (1, FILE='DD1')  
  
    ...  
  
    OPEN (2, FILE='/MYFILE.FORT')
```

CMS Notes:

- The mini disk specified by the file mode must be accessed; otherwise, an error is detected.
- You may omit the file mode from the file name, in which case the default A is used.
- If you specify * for the file mode, VS FORTRAN refers to the first file name and file type that is found on any disk through the standard search from A-Z disks. If the file name and file type are not found on any of the accessed disks, the default A is used.

- For new sequential disk files defined with a record format other than undefined or fixed unblocked, the file mode number should be specified as 4; for example, A4. Otherwise, the record format will default to undefined or fixed.
- For direct access files, specify file mode number 6.

MVS Notes:

- If referring to a VSAM file, you can use only a data set that has already been defined by access method services; that is, you should specify STATUS='OLD' on the OPEN statement.
- If you specify STATUS='OLD' on the OPEN statement for an uncataloged data set and do not specify a volume serial number by means of the FILEINF service subroutine, an error is detected.
- If you specify STATUS='NEW' or STATUS='UNKNOWN' on the OPEN statement for a data set that does not exist yet, a new non-VSAM data set is created and cataloged. The IBM-supplied default for the device is SYSDA; however, it may have been changed for your site at installation time, or it may be overridden by the FILEINF service subroutine. The device is released when the file is disconnected. If processing of the OPEN statement fails, the newly created data set is physically deleted and the device is released.
- If the data set is cataloged or on the specified volume, VS FORTRAN refers to that file. The file may or may not exist to VS FORTRAN. If the data set is not physically on the volume, an error is detected.
- If you specify ACTION='WRITE', ACTION='READWRITE', or omit the ACTION specifier on the OPEN statement, the data set is made available with a disposition of OLD, which means no other programs can refer to the same data set at the same time. If you specify ACTION='READ', the data set is made available with a disposition of SHR, which means other programs can refer to the same data set at the same time.
- Specifying FILE='/NULLFILE' on the OPEN statement is equivalent to specifying DUMMY on a file definition.

File Characteristic Defaults

Most of the defaults for file characteristics (such as block size) that are used for dynamically allocated files are set up at installation time and may be modified for your site. Different defaults may be assigned to different units.

The IBM-supplied installation defaults are the same as those used when you omit file characteristics on a file definition, as shown in Chapter 12, “Considerations for Specifying RECFM, LRECL, and BLKSIZE” on page 227. Another installation default, available only for dynamically allocated files, indicates the number of records to allocate for a new DASD file. Under CMS, this default is equivalent to the XTENT option on the FILEDEF command. The IBM-supplied default value under CMS is 50. Under MVS, this default is used to calculate the primary space, as explained under “Calculation of Primary, Secondary, and Directory Space under MVS” on page 200. The IBM-supplied default value under MVS is 100.

In addition to the installation defaults, there are fixed defaults that *cannot* be modified at installation time (but can be modified by a file definition).

These are:

- The devices for files directed to the standard I/O units, which are listed in Figure 50.
- The default device for all other units on CMS, which is DISK.

Figure 50. Default I/O Devices

Operating System	Standard Input Unit	Standard Output Unit for Error Messages, WRITE, and PRINT Statements	Standard Output Unit for PUNCH Statement
CMS	TERMINAL	TERMINAL	PUNCH
MVS batch	—	A	—
MVS with TSO	—	TERMINAL	—

Overriding File Characteristic Defaults

You can override certain defaults and supply additional characteristics for a dynamically-allocated named file or temporary file by calling the FILEINF service subroutine immediately before an OPEN statement for that file. For named files, the FILE specifier on the OPEN statement must refer to an MVS data set name or CMS file identifier. (OPEN statements for preconnected files do not use information given by the FILEINF routine; nor do READ, WRITE, or PRINT statements.) This section gives an overview of the routine. For more details about the syntax, see *VS FORTRAN Version 2 Language and Library Reference*.

The syntax of the CALL statement used to call FILEINF is:

Syntax

```
CALL FILEINF [ (rcode [ ,parm-name, value, parm-name, value, ...] ) ]
```

where *rcode* is an integer variable or array element in which the return code from FILEINF is placed; and the parameters and associated values specify certain file characteristics. For example, in the statement below, FRC is the name of the integer variable for the return code, the parameter RECFM and associated value F specify the record format as fixed, and the parameter LRECL and associated value 80 specify the record length as 80.

```
CALL FILEINF (FRC, 'RECFM', 'F', 'LRECL', 80)
```

Figure 51 shows the file characteristics you can specify and the corresponding parameters on the CALL FILEINF statement. A value of 0 for an integer-type parameter or a blank value for a character-type parameter causes the parameter to be ignored.

Figure 51 (Page 1 of 2). Parameters on CALL FILEINF

File Characteristic	Parameter on CALL FILEINF
Block size	BLKSIZE
Number of records	MAXREC
<i>MVS Only:</i>	
Primary space in cylinders	CYL

Figure 51 (Page 2 of 2). Parameters on CALL FILEINF

File Characteristic	Parameter on CALL FILEINF
Primary space in tracks	TRK
Primary space in blocks	MAXBLK
Secondary space	SECOND
Number of 256-byte records in partitioned-data-set directory	DIR
Type of device (unit address such as 123, generic name such as 3380, or esoteric name such as SYSDA)	DEVICE
Maximum number of devices	DEV CNT
Maximum number of volumes an output data set requires	VOL CNT
Volume serial number	VOLSER
Multiple volume serial numbers	VOLSERS
Number of DASD I/O buffers	BUFNO

Information given on the CALL FILEINF statement is used only for the first following OPEN or INQUIRE statement that requires dynamic allocation. (The INQUIRE statement is under “Gathering Information on Dynamically Allocated Files” on page 202.) The parameters on the CALL FILEINF statement become ineffective after the next OPEN or INQUIRE statement is issued, regardless of whether the information given is used. That is, the information on the CALL FILEINF cannot be used for subsequent OPEN or INQUIRE statements.

If you code CALL FILEINF without any parameters, as follows:

```
CALL FILEINF
```

the values are reset to the installation defaults. This is necessary only if a CALL FILEINF statement with parameters has been processed (for example, in a main program that calls your routine) and you want to nullify the values it set.

If an error is detected in a CALL FILEINF statement, an error is also detected for the following OPEN or INQUIRE statement if the information on the CALL FILEINF statement applies, and the OPEN or INQUIRE statement is ignored.

Considerations for VOLSER(S): If you omit the VOLSER or VOLSERS parameter for a file, VS FORTRAN uses the catalog to locate the data set. If the data set is not cataloged and you specified STATUS='NEW' or STATUS='UNKNOWN' on the OPEN statement, a new data set is created on a public or storage volume assigned by the system and is cataloged. If the data set is not cataloged and you specified STATUS='OLD', an error is detected. If you specified STATUS='SCRATCH', a temporary data set is created on a public or storage volume assigned by the system and is not cataloged.

If you do code the VOLSER or VOLSERS parameter, VS FORTRAN attempts to locate the file on the specified volume. (If you specify multiple volumes, VS FORTRAN attempts to locate the file only on the first one; however, all the volumes are used for allocation.)

- If the file is found on the volume, VS FORTRAN refers to it and the STATUS specifier on the OPEN statement operates as usual.

- If the file is not found on the volume, the following processing occurs, *regardless of the OCSTATUS / NOOCSTATUS run-time option*:
 - If you specified STATUS='OLD', an error is detected.
 - If you specified STATUS='NEW' or STATUS='UNKNOWN' and the file is not in the catalog, a new file is created on the specified volume and *is* cataloged.
 - If you specified STATUS='NEW' or STATUS='UNKNOWN' and the file is in the catalog, a new file is created on the specified volume and is *not* cataloged.

Warning: Be aware that in this case, you will have two files with the same name, which is strongly discouraged. Later, if you intend to use the new data set but do not specify the volume, you will inadvertently access the cataloged file instead.

 - If you specified STATUS='SCRATCH', a temporary data set is created on the specified volume and is not cataloged.

Considerations for BUFNO: BUFNO applies only to DASD and tape devices under MVS. All other device types will continue to use 1 or 2 buffers. Also, BUFNO is not valid for values less than 0 or greater than 255.

INQUIRE does not provide a BUFNO variable to receive the value of the number of buffers set by either FILEINF or by JCL.

Considerations for RECFM, LRECL, and BLKSIZE: Refer to Chapter 12, “Considerations for Specifying RECFM, LRECL, and BLKSIZE” on page 227 for information on how these values are processed and the defaults that are supplied.

Considerations for MAXREC: Under CMS, MAXREC corresponds to the XTENT option of the FILEDEF command. Under MVS, MAXREC is used to calculate space, as described in the next section.

Calculation of Primary, Secondary, and Directory Space under MVS

To specify primary space for a new file, use the CYL, TRK, MAXBLK, or MAXREC parameter on CALL FILEINF.

The CYL and TRK parameters are equivalent to the CYL and TRK subparameters on a DD statement. The MAXBLK parameter is equivalent to the *blocklength* subparameter on a DD statement. In addition, if you specify MAXBLK, the value specified or defaulted for BLKSIZE becomes the block length.

If you specify the *number of records* for a file, either by coding the MAXREC parameter or by accepting the installation default for number of records, VS FORTRAN allocates primary space in blocks. Using the information from the MAXREC, RECFM, LRECL, and BLKSIZE parameters on the CALL FILEINF statement, or from the installation defaults, VS FORTRAN uses the formulas in Figure 52 on page 201 to calculate the primary space.

The formulas use the following JCL SPACE parameter options:

Option	Meaning
Block length	The average block length of the data.
Primary Quantity	The number of blocks of data that can be contained in the data set.

Figure 52. Formulas for Primary Space under MVS

RECFM	Formulas for Space
F, FA, V, or VA	Block length = record length Primary quantity = number of records
FB, FBA, VB, or VBA	Block length = block size Primary quantity = (number of records * record length) / block size
U, UA, VS, or VBS	Block length = block size Primary quantity = number of records

To specify secondary space, use the SECOND parameter of CALL FILEINF. If you do not code the SECOND parameter, no secondary space is allocated.

To specify directory space, use the DIR parameter. If you omit the DIR parameter, and the FILE specifier on the OPEN statement refers to a member of a new partitioned data set, a value of 5 is used.

Coding INQUIRE in Your Program

You can code an INQUIRE statement anywhere in your program, for instance before a file is connected, while it is connected, and after it is disconnected. However, for most of the specifiers, certain conditions must be true in order for the information to be obtained. For instance, to find out the number of the unit to which a file is connected, you can use the NUMBER specifier. However, the result will be valid *only if the file is connected to a unit*. Therefore, you should also code the OPENED specifier to determine whether the file is connected, as shown in the following example:

```
INQUIRE (FILE='MYFILE', OPENED=CONNECTED, NUMBER=UNITNO)
```

In this example, if the variable CONNECTED contains the value true, the variable UNITNO will contain the unit number to which MYFILE is connected. However, if the variable CONNECTED contains the value false, the value assigned to UNITNO will be unpredictable.

A complete list of all the required conditions for each of the specifiers is given under INQUIRE in *VS FORTRAN Version 2 Language and Library Reference*.

Sample Program

In Figure 53 on page 202 is a sample program that uses the INQUIRE statement. The purpose of the program is to combine the data from three files (SEATTLE, SANFRAN, and SANDIEGO) into a single file (WESTERN). The INQUIRE statement is used to check whether each of the three files exists before attempting to connect to and read from it.

```

      PROGRAM GATHER
      CHARACTER*8  FILE_NAME(3)
1      / 'SEATTLE', 'SANFRAN', 'SANDIEGO' /
      LOGICAL*4    FILE_EXISTS
      CHARACTER*80 RECORD
C
      OPEN (1, FILE='WESTERN', STATUS='NEW',
1      FORM='UNFORMATTED', ERR=40)
      DO 30 I = 1, 3
          INQUIRE (FILE=FILE_NAME(I),
1      EXIST=FILE_EXISTS, ERR=40)
          IF (FILE_EXISTS) THEN
              OPEN (2, FILE=FILE_NAME(I), STATUS='OLD',
1      FORM='UNFORMATTED', ERR=40)
10      READ (2, END=20, ERR=40) RECORD
              WRITE (1, ERR=40) RECORD
              GO TO 10
20      CLOSE (2, ERR=40)
          ENDIF
30      CONTINUE
      CLOSE (1, ERR=40)
      STOP
40      STOP 'ERROR HAS OCCURRED.'
      END

```

Figure 53. Sample Program—Using the INQUIRE Statement

On MVS, you may ignore any non-Fortran messages that are issued from the system.

Gathering Information on Dynamically Allocated Files

Warning: From a VS FORTRAN point of view, the CMS file identifier or MVS data set name is logically different from the name you specify on the FILE specifier of the INQUIRE statement, even if the names are the same and both point to the same physical file. For instance, in the following example, the name on the INQUIRE statement is not checked against the name in the first OPEN statement even though DD1 refers to the same physical file. Therefore, the variable OPN will contain the value false. (However, if the CMS file identifier or MVS data set name were coded on the OPEN statement, the value would be true.)

Under CMS:

```

FILEDEF DD1 DISK MYFILE FORT A1

      OPEN (1, FILE='DD1')

      ...

      INQUIRE (FILE='/MYFILE FORT A1', OPENED=OPN)

```

Under MVS:

```

//DD1 DD DSN=MYFILE.FORT,DISP=OLD

      OPEN (1, FILE='DD1')

      ...

      INQUIRE (FILE='/MYFILE.FORT', OPENED=OPN)

```

Referring to Values Set by the FILEINF Routine

The INQUIRE statement can refer to the values set by the following parameters specified on the CALL FILEINF statement:

```
RECFM
DEVICE
VOLSER
VOLSERS.
```

Figure 54 is an example of how to use INQUIRE to refer to values that are set by the FILEINF routine.

```
*
* Set record format
*
1 CALL FILEINF(IRET,'RECFM','FB')
*
* Check whether the file DEPTJ76 DATAFILE exists and whether it
* can be connected for formatted I/O
*
2 INQUIRE (FILE='/DEPTJ76 DATAFILE *', EXIST=EXT,FORMATTED=FMT)
*
* If the file exists, connect it as old
*
    IF (EXT) THEN
        OPEN (1, FILE='/DEPTJ76 DATAFILE *', STATUS='OLD')
*
* Otherwise, connect it as new:
*
* Set record format, logical record length, block size, and number of
* records
*
    ELSE
3 CALL FILEINF(IRET,'RECFM','FB','LRECL', 80,'BLKSIZE',3200,
    2          'MAXREC',500)
*
* If the file can be connected for formatted I/O, connect it that way
*
    IF (FMT .EQ. 'YES') THEN
        OPEN (1, FILE='/DEPTJ76 DATAFILE *', STATUS='NEW',
    2          FORM='FORMATTED')
*
* Otherwise, connect it for unformatted I/O
*
    ELSE
        OPEN (1, FILE='/DEPTJ76 DATAFILE *', STATUS='NEW',
    2          FORM='UNFORMATTED')
    ENDF
    ENDF
```

Figure 54. Example of Using the FILEINF Routine with INQUIRE and OPEN Statements

- 1** Sets the RECFM value to FB
- 2** Checks whether the file named DEPTJ76 DATAFILE exists and whether it can be connected for formatted I/O. VS FORTRAN uses the RECFM value to determine the appropriate value for the FORMATTED specifier. For instance, if RECFM=VS, 'NO' is returned for the FORMATTED specifier because RECFM=VS is valid only for unformatted files. (The RECFM value is used in the same way for the UNFORMATTED specifier.)
- 3** Resets RECFM to FB (because processing of the INQUIRE statement caused it to become ineffective) and also sets values for LRECL, BLKSIZE, and MAXREC.

Considerations for *VOLSER(S)*: If you omit the VOLSER or VOLSERS parameter for a file, VS FORTRAN uses the catalog to locate the data set. If the data set is cataloged, VS FORTRAN refers to it. If the data set is not cataloged, the data set is considered not to exist.

If you do code the VOLSER or VOLSERS parameter, VS FORTRAN attempts to locate the file on the specified volume. (If you specify multiple volumes, VS FORTRAN attempts to locate the file only on the first one; however, all the volumes are used for allocation.) If the file is found on the volume, VS FORTRAN refers to it. If the file is not found on that volume, the file is considered not to exist.

Chapter 10. Considerations for Double-Byte Data

Connecting A File Containing Double-Byte Data	205
Specifying Your Own Format with Double-Byte Data	206
List-Directed I/O with Double-Byte Data	206
NAMELIST I/O with Double-Byte Data	206

If your data is in a language, such as Japanese, that has double-byte characters, you can read and write both formatted and unformatted data. For formatted data, all types of external I/O are supported: You can use list-directed formatting, NAMELIST formatting, or your own format. However, you cannot use internal I/O.

Connecting A File Containing Double-Byte Data

To ensure proper processing of formatted data, use `CHAR='DBCS'` on the `OPEN` statement to connect your file. `CHAR='DBCS'` is required for the following:

- List-directed input that might have double-byte text in character constants
- NAMELIST input with double-byte names for variables or arrays
- Formatted I/O with run-time `FORMAT` statements.

For unformatted data, the `CHAR` specifier has no effect because unformatted data is transferred without conversion.

To enable double-byte character support for the error message unit, code an `OPEN` statement for the unit, as follows. (Unit 6 is the IBM-supplied default for the error message unit; if it is changed to a different number at your site, specify that number.)

```
OPEN (UNIT=6, CHAR='DBCS')
```

In order for the data to be displayable or printable, the output device must have double-byte processing capability. In addition, the double-byte portion of the data must be enclosed by the shift-out character `␣` and the shift-in character `␣`. The shift characters may be truncated by the assignment operation or the character substring operation, resulting in an invalid double-byte data string. For the assignment operation, you can ensure that the data is displayable or printable by using the `ASSIGNM` service subroutine to preserve the balanced shift characters.

Note that I/O processing does not check whether your file contains valid double-byte data. For information on valid double-byte data and the `ASSIGNM` service subroutine, see *VS FORTRAN Version 2 Language and Library Reference*.

If you need to check whether a connected file may contain double-byte characters, use the `INQUIRE` statement with the `CHAR` specifier. If `CHAR='DBCS'` was coded on the `OPEN` statement, the value `DBCS` is returned; otherwise, the value `NODBCS` is returned.

For files connected for keyed access, if a primary or alternate key contains double-byte data, the entire portion of the key that contains double-byte data, including the shift codes, must be part of the key. Be aware that the shift codes are processed as any other characters.

Following are some examples of I/O with double-byte data. The syntax notation is described under “Syntax Notation” on page x.

Specifying Your Own Format with Double-Byte Data

If you run a program with these statements:

```
CHARACTER*15 MIXED_STUFF
CHARACTER*50 FMT
FORMAT (A50)
READ (1,10) FMT      ! Read format specifier to be used for write
                      ! FMT = (2I4,1X,'WHERE IS F.R.E.D.'.S ',A15)

I = 10
J = 15
MIXED_STUFF = 'HOUSEkk'
OPEN (2,CHAR='DBCS')
WRITE (2,FMT) I,J,MIXED_STUFF
```

the following record is written:

```
10 15 WHERE IS F.R.E.D.'.S ' HOUSEkk '
```

List-Directed I/O with Double-Byte Data

If you run a program with these statements:

```
CHARACTER*12 STR
CHARACTER*6  STR1,STR2
I = 10
J = 15
STR = 'FR.E.D.'.S'
OPEN (10,CHAR='DBCS')
WRITE (10,*) I,J,STR
STR1 = 'kkkk'
STR2 = 'A.BC'
WRITE (10,*) STR1 STR2
```

The following records are written:

```
10 15 FR.E.D.'.S '
```

```
kkkk A.BC
```

NAMelist I/O with Double-Byte Data

If your input data is the following:

```
&NAM1 kk = 'A.B',B = 'AB.C'.S'&END
```

and you run a program with these statements:

```
CHARACTER*10 KK,B
NAMELIST /NAM1/ KK,B
OPEN (10, CHAR='DBCS')
READ (10,NAM1)
```

the variables are set to the values shown:

Variable	Value
KK	A.Bbb
B	ABC.'S

Chapter 11. Using VSAM

VSAM Terminology	210
Organizing Your VSAM File	210
VSAM Sequential File Organization	210
VSAM Direct File Organization	211
VSAM Keyed File Organization	211
VSAM Linear Data Set Organization	211
Defining VSAM Files	211
Defining VSAM Files—General Considerations	212
Examples of Defining a VSAM File	213
Defining a VSAM Keyed File	213
Defining a VSAM Direct File	214
Defining a VSAM Sequential File	214
Defining a VSAM Linear Data Set	215
Using VSAM Keyed Files	215
Defining Alternate Indexes	215
Alternate Index Terminology	216
Planning to Use Alternate Indexes	217
Cataloging and Loading Alternate Indexes	217
Accessing Your VSAM Keyed Files	217
Loading Your VSAM Keyed Files	219
Processing DEFINE Commands	219
Source Language Considerations—VSAM Files	220
Processing VSAM Sequential Files	221
Using OPEN Statement—VSAM Sequential Files	221
Using READ Statement—VSAM Sequential Files	221
Using WRITE Statement—VSAM Sequential Files	222
Using BACKSPACE Statement—VSAM Sequential Files	222
Using REWIND Statement—VSAM Sequential Files	222
Processing VSAM Direct Files	222
Using OPEN Statement—VSAM Direct Files	223
Using Sequential Access—VSAM Direct Files	223
Using Direct Access—VSAM Direct Files	224
Processing VSAM Keyed Files	225
Processing VSAM Linear Files	225
Obtaining the VSAM Return Code—IOSTAT Option	226

VSAM is an access method for files on direct access storage devices. Like non-VSAM access methods, VSAM can be used to load, retrieve, and update data sets.

VSAM processing has these advantages:

- Data protection against unauthorized access, including password protection for VSAM files (for installations that do not have RACF installed and active)
- Cross-system compatibility
- Device independence, because there is no need to be concerned with block size and other control information.

VSAM processing is the only way for your Fortran program to use keyed access.

VS FORTRAN Version 2 lets you use VSAM to process the following kinds of files:

- VSAM Entry Sequenced Data Sets (ESDS), which can be processed only sequentially
- VSAM Relative Record Data Sets (RRDS), which can be processed either sequentially or directly by relative record number
- VSAM Key Sequenced Data Sets (KSDS), which can be processed sequentially or directly by keys
- VSAM Linear Data Sets, which can be used by the data-in-virtual facility.

If you have complex requirements or are going to be a frequent user of VSAM, you should review the VSAM publications for your operating system.

VSAM Terminology

VSAM terminology is different from the terminology used for MVS files, as shown in Figure 55, for example.

Figure 55. VSAM Terminology

VSAM Term	Similar Non-VSAM Term
ESDS	QSAM data set
KSDS	ISAM data set
RRDS	BDAM data set
Control Interval Size (CISZ)	Block size
Buffers (BUFNI/BUFND)	BUFNO
Access Method Control Block (ACB)	Data Control Block (DCB)
Cluster (CL)	Data set
Cluster Definition	Data set allocation
AMP parameter of JCL DD statement	DCB parameter of JCL DD statement
Record size	Record length

Organizing Your VSAM File

The physical organization of VSAM data sets differs considerably from those used by other access methods. Except for relative record data sets, records need never be of a fixed length. VSAM data sets are held in control intervals and control areas; the size of these is normally determined by the access method and the way in which they are used is not visible to you. VSAM files can only exist on direct access devices.

VSAM Sequential File Organization

In a VSAM sequential file (ESDS), records are stored in the order they were entered. The order of the records is fixed. Records in sequential files can only be accessed (read or written) sequentially.

VSAM Direct File Organization

A VSAM direct file (RRDS) is a series of fixed-length slots in storage into which you place records. The relative key identifies the record—the relative key being the relative record number of the slot the record occupies.

Each record in the file occupies one slot, and you store and retrieve records according to the relative record number of that slot. When you load the file, you have the option of skipping over slots and leaving them empty.

VSAM Keyed File Organization

In a VSAM keyed file (KSDS), the records are ordered according to the collating sequence of an embedded prime key field, which you define. The prime key is one or more consecutive characters taken from the records. The prime key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records. A prime key for a record might be, for example, an employee number or an invoice number.

You can also specify one or more alternate keys to use to retrieve records. Using alternate keys, you can access the file to read records in some other sequence than the prime key sequence. For example, you could access the file through employee department rather than through employee number. Alternate keys need not be unique. More than one record can be accessed, given a department number as a key. This is permitted if alternate keys are specified to allow duplicates.

To use an alternate index, you need to define a data set called the alternate index (known by the acronym AIX). This data set contains one record for each value of a given alternate key; the records are in sequential order by alternate-key value. Each record contains corresponding primary keys of all records in the associated KSDS that contain the alternate key value. For instructions on how to define an alternate index, see the appropriate access methods services manual.

VSAM Linear Data Set Organization

A VSAM Linear Data Set stores data so that it can be accessed or updated in units of 4096 bytes. It contains only data; it does not have imbedded logical records or other control information.

Defining VSAM Files

VSAM entry-sequenced, key-sequenced, relative-record, and linear data sets can be processed by VS FORTRAN Version 2 only after you define them by means of access method services.

A VSAM **cluster** is a logical definition for a VSAM data set and has one or two components:

- The data component of a VSAM cluster contains the data records.
- The index component of a VSAM key-sequenced cluster consists of the index records.

You use the access method services DEFINE CLUSTER command to define your VSAM data clusters (sets). This process includes creating an entry in a VSAM or

Defining VSAM Files

ICF catalog without any data transfer. Specify the following information about the cluster:

- Name of the entry
- Name of the catalog to contain this definition and its password (may use default name)
- File organization—ESDS (NONINDEXED), RRDS (NUMBERED), KSDS (INDEXED), and linear (LINEAR)
- Volumes the file will occupy
- Space required for the data set
- Record size and Control Interval Size (CISZ)
- Passwords (if any) required for future access
- For KSDS, length and position of the prime key within the records, and how index records are to be stored.

For further information, see your access method services manual.

Defining VSAM Files—General Considerations

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.

The following general programming considerations apply to VSAM files:

- If you want to access data in a number of ways, and want to update or insert records at any point, use VSAM KSDSs.
- If you want to create a complete file, one in which you'll never update any records or insert new ones but to which you may add records at the end, use VSAM ESDSs.
- If you want to use work files, or files in which records must be created and later updated in place, use VSAM RRDSs.
- VSAM linear data sets are used exclusively by data-in-virtual (DIV) support on MVS/XA 2.2.0 or later.

A VSAM file may be suballocated or unique. A suballocated file shares a data space with other files; a unique file has a data space to itself.

VSAM treats all files as clusters. For key-sequenced files, a cluster consists of a data component and an index component. For entry-sequenced or relative-record files, a cluster consists of a data component only. Besides setting up a catalog entry for each component of a cluster, VSAM sets up a catalog entry for the cluster as a whole. This entry is the cluster name specified in the DEFINE command.

To define a suballocated VSAM file, first define a data space; then use the DEFINE command with the CLUSTER parameter. VSAM suballocates space for the file in the data space set up and enters information about the file in a VSAM catalog. A file can be stored in more than one data space on the same volume or on different volumes.

A unique VSAM file is defined by specifying the parameter UNIQUE and assigning space to the file with a space allocation parameter and the job control statements. The data space is acquired and assigned to the file concurrent with the file definition. However, no other file can occupy its data space(s).

Examples of 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 use the commands, you create a VSAM catalog entry for the file. The form of the entry depends upon the kind of file you'll be creating: a VSAM KSDS, a VSAM ESDS, a VSAM RRDS, a VSAM-managed sequential file, or a VSAM linear data set.

For VSAM keyed, sequential, direct, and linear 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 more information about the DEFINE commands, see the appropriate access method services manual.

Defining a VSAM Keyed File

To define a VSAM keyed file (KSDS), you can specify:

```
DEFINE CLUSTER -
  (NAME(myfile1)      -
  FILE(ddname)        -
  VOLUMES(666666)     -
  KEYS(10,5)          -
  INDEXED             -
  RECORDS(180)        -
  RECORDSIZE(80 200)) -
  DATA(NAME(myfile1.data)) -
  INDEX(NAME(myfile1.index)) -
  CATALOG(USERCAT)
```

which defines a file named myfile1 as a VSAM KSDS.

INDEXED

Specifies that the file is a VSAM keyed file (KSDS).

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.

DATA(NAME(myfile1.data))

Specifies the data component name.

INDEX(NAME(myfile1.index))

Specifies the index component name.

CATALOG(USERCAT)

Specifies the catalog in which this file is entered.

Defining a VSAM Direct File

To define a VSAM direct file (RRDS), you can specify:

```
DEFINE CLUSTER -  
    (NAME(myfile2) -  
    FILE(ddname) -  
    VOLUMES(666666) -  
    NUMBERED -  
    RECORDS(200) -  
    RECORDSIZE(80 80)) -  
    CATALOG(USERCAT)
```

which defines a file named `myfile2` as a VSAM RRDS.

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.

Defining a VSAM Sequential File

To define a VSAM sequential file (ESDS), you can specify:

```
DEFINE CLUSTER -  
    (NAME(myfile3) -  
    FILE(ddname) -  
    VOLUMES(666666) -  
    NONINDEXED -  
    RECORDS(180) -  
    RECORDSIZE(80 200)) -  
    CATALOG(USERCAT)
```

which defines a file named `myfile3` as a VSAM ESDS.

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.

Defining a VSAM Linear Data Set

To define a VSAM linear data set, you can specify:

```
DEFINE CLUSTER          -
      (NAME(DIV.EXAMPLE) -
      VOLUMES(DIVVOL)   -
      TRACKS(1,1)       -
      SHAREOPTIONS(1,3) -
      LINEAR)
```

which defines a file named DIV.EXAMPLE.

VOLUMES(DIVVOL)

Specifies that the file is contained on volume DIVVOL

TRACKS(1,1)

Specifies that one track is to be initially allocated for the file and one track is to be allocated for each extent

SHAREOPTIONS(1,3)

Specifies that one of the following file-sharing capabilities is enforced:

1. A single user can access the data set for update, *or*
2. Several users can access the data set for read only.

If SHAREOPTIONS(1,3) is not specified, data-in-virtual does not provide data set integrity when multiple programs update the data object concurrently.

Therefore, you should avoid using other options on the SHAREOPTIONS parameter when you create a linear data set.

LINEAR

Specifies that the file is a VSAM linear data set

For additional information on the creation of VSAM linear data sets and the alteration of entry-sequenced VSAM data sets, see *MVS/XA Integrated Catalog Administration: Access Method Services Reference*.

Using VSAM Keyed Files

Defining Alternate Indexes

By means of *alternate indexes*, keyed VSAM files can be arranged for access in as many different ways as desired. VS FORTRAN Version 2 can access a KSDS file through either its prime index or through any alternate index. (An ESDS file alternate index cannot be accessed by VS FORTRAN Version 2.)

When an alternate index has been built, you access the data set through an object known as an *alternate index path* that acts as a connection between an alternate index and the data set.

Two types of alternate indexes are allowed: unique key and nonunique key.

- For a unique key alternate index, each record must have a different key.
- For a nonunique key alternate index, within the defined limits of index record size, any number of records can have the same key.

For example, an employee file might be indexed by personnel number, by name, and also by department number.

In this example, the alternate index using the names could be a unique key alternate index (provided each person had a different name), and the alternate index using the department number would be a nonunique key alternate index because more than one person could be in each department. A data set accessed through a unique key alternate index path can be treated, in most respects, like a KSDS accessed through its prime index. The records may be accessed by key or sequentially, records may be updated, and new records may be added. If the data set is a KSDS, records may be deleted and the length of updated records altered. When records are added or deleted, all indexes associated with the data set are by default altered to reflect the new situation if it's an "upgrade" set (see "Alternate Index Terminology").

In data sets accessed through a nonunique key alternate index path, the record accessed is determined by the key and the sequence. The key can be used to establish positioning so that sequential access may follow. The use of the key accesses the first record with that key.

The alternate index may be password protected, as for a normal VSAM data set.

Alternate Index Terminology

An alternate index is, in practice, a VSAM data set that contains a series of pointers to the keys of a VSAM data set. When you use an alternate index to access a data path, you use an entity known as an *alternate index path*, or simply a *path*, that establishes the relationship between the index and the data set.

The data set to which the alternate index gives you access is known as the base data set, and is usually referred to in VSAM manuals as the *base cluster*.

If the indexes are defined "upgrade," alternate indexes are automatically updated. All indexes so connected are known as the *index upgrade set* of the base cluster.

Base cluster

A data component of KSDS and primary (prime) index.

Prime index

The index used in creating the data set and used when access is made through the base cluster.

Alternate indexes

Other indexes to the same base data.

Paths

Establish a path through the base data other than that implied by the prime index in a KSDS and the sequence in an ESDS. Paths connect the alternate index with the base data.

Index upgrade set

That set of indexes (always including the prime index) that will be automatically updated when the data is changed. Indexes can exist outside this set.

Planning to Use Alternate Indexes

When planning to use an alternate index, you must know:

- The type of base data set with which the index will be associated
- Whether the keys will be unique or nonunique
- Whether the index is to be password protected
- Some of the performance aspects of using alternate indexes.

The type of base cluster and the use of unique or nonunique keys determine the type of processing that you can perform with the alternate index, and so determine the Fortran statements you may use.

You use an alternate index path as if it were a separate data set.

Cataloging and Loading Alternate Indexes

If your VSAM keyed file will have one or more alternate indexes, specify a `DEFINE ALTERNATEINDEX` and `DEFINE PATH` for each one. These are VSAM commands.

`DEFINE ALTERNATEINDEX` identifies and builds a catalog entry for the alternate index. In it, you specify:

- The name of the catalog entry
- The name of the alternate index and whether it is unique or can be duplicated
- Whether or not alternate indexes are to be updated when the file is modified
- The name of the VSAM base cluster it relates to
- The name of the catalog (you may use the default name) to contain this definition and its password
- The maximum number of times you can try entering a password in response to a prompting message.

`DEFINE PATH` relates an alternate index with its base cluster.

After you have defined the alternate index and the path, and you have loaded the base cluster, you can specify a `BLDINDEX` command to load the alternate index with index records.

Accessing Your VSAM Keyed Files

The name that identifies a particular data definition statement is called a *ddname* in MVS and CMS. Opening a VSAM KSDS requires that one or more operating system data definition statements be supplied to relate the Fortran unit number to the actual file. These data definition statements are the DD statement in an MVS system and the DLBL statement in a CMS system.

This section discusses the names that are required to access a VSAM KSDS. These names depend upon the operating system, whether or not the `FILE` parameter was specified on the `OPEN` statement, and the number of `KEYS` listed in the `KEYS` parameter of the `OPEN` statement.

One Data Definition Statement: If a file has no `KEYS` parameter given on its `OPEN` statement or if only one key is listed in the `KEYS` parameter, then only a single data definition statement is required.

If there is no KEYS parameter on the OPEN statement, whichever primary or alternate index key is referred to by the data definition statement becomes the only possible key of reference for access to the file.

More than One Data Definition Statement: If the KEYS parameter lists more than one key, then the Fortran VSAM KSDS support routines actually open more than one VSAM file and a separate data definition statement (and, therefore, a different name) is required for each one. The required names are indicated in Figure 56.

Figure 56. Names Required to Open a Single Fortran Keyed File

DDNAME No.	CMS		MVS	
	FILE= <i>fn</i>	No FILE=	FILE= <i>fn</i>	No FILE=
1	<i>fn</i>	FT <i>nn</i> K01	<i>fn</i>	FT <i>nn</i> K01
<i>i</i> (<i>i</i> >1)	<i>fn</i> suffixed with <i>m</i>	FT <i>nn</i> K0 <i>i</i>	<i>fn</i> suffixed with <i>m</i>	FT <i>nn</i> K0 <i>i</i>
Note 1	Note 2			

Where:

nn Is the unit number specified in the OPEN statement.
fn Is the file name, if any, specified in the OPEN statement.
m Is *i* - 1

Notes:

1. The ddname or filename numbers do not have to correspond to the positions of the associated keys in the key list (KEYS parameter of the OPEN statement). For example, the last key listed in the KEYS parameter need not correspond to the highest numbered name.
2. In a CMS system, if the filename (*fn*) given in the FILE parameter is seven characters long, it is not possible to suffix the name as indicated above for other than the first ddname. In this case, the last character of the name is overlaid instead.

There must be a data definition statement corresponding to each key, either the primary key or an alternate index key, listed in the KEYS parameter of the OPEN statement. If the primary key is listed in the KEYS parameter, then there must be a data definition statement which refers to the base cluster. If an alternate index key is listed in the KEYS parameter, then there must be a data definition statement which refers to that alternate index path.

It is important to note that the data definition statement corresponding to an alternate index key must refer to the alternate index *path* and not to the alternate index itself. All the data definition statements which are used to open one Fortran keyed file must refer to the same base cluster.

Separate Fortran keyed files are opened with different unit numbers. If any of the files to remain open at the same time were opened with a parameter other than ACTION='READ', they must not involve the same base cluster, either through the primary key or through one of its alternate index keys. Violation of this restriction may cause unexpected or undesirable results when file updates are made.

Loading Your VSAM Keyed Files

Before a VSAM KSDS can be accessed by any retrieval or update operations, it must have been successfully defined and loaded. A file that has been defined but which has never had records loaded into it is called an *empty* file.

An empty file may be loaded in one of the following ways:

1. With an access method services command (such as REPRO)
2. By a VS FORTRAN Version 2 program which opens the file with an ACTION of WRITE, writes one or more records that are in ascending key sequence by the primary key, and then closes the file
3. For KSDS only, by an implicit load in a VS FORTRAN Version 2 program. This occurs when an empty keyed file is opened with an ACTION of READWRITE. In this case, the file is automatically opened for loading, a single dummy record is loaded into it, and the file is closed. The file is then reopened and the dummy record is deleted.
4. By a program written in some other language that has the capability of loading records into an empty VSAM file.

After a VSAM file has been defined and loaded, it is called a *nonempty* file. (In VSAM terminology, it is still called a nonempty file even if all the records loaded into it have been deleted.)

Processing DEFINE Commands

After you've created your DEFINE command, you must run it, using access method services, to create an entry in a VSAM catalog.

For MVS: You specify the following job control statements to catalog your VSAM DEFINE commands:

```
//VSAMJOB JOB
//STEP EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=A
//ddname DD VOL=SER=myvol,UNIT=SYSDA,DISP=OLD
//SYSIN DD *
```

(The DEFINE command as data)

```
/*
//
```

When you run a Fortran program to create or process a VSAM file, you define the file in a DD statement.

For example, to process the file `myfile1` in a Fortran load module called `myprog`, you specify:

```
//VSAM1 JOB
// EXEC PGM=myprog
//ddname DD DSN=myfile1,DISP=SHR
//
```

When `myprog` is run, the DD statement makes `myfile1` (and the information in its catalog entry) available to the program. In the Fortran OPEN statement, `ddname` is

the name specified in the FILE parameter. For information about job control statements, see “Coding and Processing Jobs” on page 12.

For CMS: To define a VSAM file to CMS, you specify the following commands:

- The XEDIT command (or the edit command of your choice), to create a file with a filetype of AMSERV containing the DEFINE CLUSTER command.
- The AMSERV command, to run the DEFINE CLUSTER command in the file you've created; this creates the VSAM catalog entry. For example:

```
AMSERV defname
```

This command sends the DEFINE CLUSTER command to access method services for processing.

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.

In addition, a VSAM keyed file (KSDS) has special language keywords and constructs that affect the OPEN, READ, and WRITE statements. When you are processing VSAM files, you can use all the VS FORTRAN Version 2 input/output statements, but REWRITE and DELETE can be used only with KSDS.

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.

Note: If your program contains an ENDFILE statement and processes a VSAM file, you'll get a warning message to inform you that ENDFILE has no meaning for a VSAM file and is treated as documentation.

Figure 57 summarizes the Fortran input/output statements you can use with each form of access.

Figure 57. Fortran Statements Valid with VSAM Files

Fortran I/O Statements	Access Mode			
	VSAM Sequential (ESDS)	VSAM Direct (RRDS)	VSAM Direct (RRDS)	VSAM Keyed (KSDS)
	Sequential Access	Sequential Access	Direct Access	Keyed Access
OPEN	Yes(1)	Yes(1)	Yes	Yes
WRITE	Yes(2)	Yes	Yes(3)	Yes(4)
REWRITE	No	No	No	Yes
DELETE	No	No	No	Yes
READ	Yes	No(5) Yes(6)	Yes	Yes
BACKSPACE	Yes	Yes(7) Yes(6)	No	Yes
REWIND	Yes	Yes(7) Yes(6)	No	Yes
CLOSE	Yes	Yes	Yes	Yes

Notes:

1. Sequential OPEN
2. Add a new record to the end of the file
3. Update or replace
4. Add or insert a new record
5. Empty file
6. Nonempty file
7. For a file that was empty when opened, has the effect of CLOSE

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 by VS FORTRAN Version 2 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. For general information on these statements, see *VS FORTRAN Version 2 Language and Library Reference*.

Using OPEN Statement—VSAM Sequential Files

When your program processes a VSAM sequential file, you must specify the OPEN statement. For VSAM sequential files, specify: `ACCESS='SEQUENTIAL'`.

Note: If you receive a non-Fortran error message, but your program runs, you may ignore the message.

Using 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 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.

After 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 or keyed.)

Using 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 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 run, 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. You should use direct access for RRDS when there are gaps in the relative record sequence for the file, or when you want to update records in place.

If you are using sequential access, accessing each record in turn, one after another, 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 statement as well as for sequential, direct, and keyed access, as described in the following paragraphs. For general information, see Chapter 8, "Using File Access Methods" on page 163.

Using OPEN Statement—VSAM Direct Files

When your program processes a VSAM direct file, you must specify the OPEN statement. The options you can use are:

ACCESS='DIRECT' for direct access
ACCESS='SEQUENTIAL' for sequential access.

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 is not reserved in the file. The OPEN statement option to use is ACCESS='SEQUENTIAL'.

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 cannot 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 Version 2 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 direct 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. If you use the BACKSPACE statement during file loading, the last record in the file is available for retrieval.

Using Direct Access—VSAM Direct Files

You can place records into a VSAM direct file using the WRITE statement, or retrieve records from a VSAM direct file using the READ statement. The OPEN statement option to use is ACCESS='DIRECT'

For VSAM direct files, if the relative record numbers for the file are not strictly sequential—for example, if there are gaps in the key sequence:

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. 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.

The following example illustrates the first three steps above:

```
OPEN (ACCESS='DIRECT',UNIT=10,RECL=80)
IREC = 45
WRITE (10,REC=IREC)
```

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. If you use direct access WRITE statements to load a file initially and you want to change from initial load processing to update processing, issue a direct access form of the READ statement.

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 use the BACKSPACE or REWIND statements with a directly accessed VSAM direct file; if you do, your program is terminated.

Processing VSAM Keyed Files

VSAM keyed files use VSAM key sequenced data sets (KSDS). The access mode is keyed, and record retrieval is accomplished by means of either direct or sequential READ statements, while record output is by direct WRITE or REWRITE statements.

For VS FORTRAN Version 2 users, probably the most significant property of a VSAM keyed file is the ability to process the file in more than one order within the same program. This is accomplished in VS FORTRAN Version 2 by means of the KEYS parameter in the OPEN statement, and the KEYID parameter in the READ statement.

The KEYS parameter in the OPEN statement names the established VSAM keys for the KSDS file that you will use in your program. The KEYID parameter names the key applicable to the READ statement containing it, and sets up that key as the current key of reference. Provided the key or keys you want to use is identified in the most recent OPEN statement for the file, the KEYID parameter in a later READ statement can identify any of those keys as the key of reference at any point in the program. Or, after the file is closed, another OPEN can establish another group of keys.

In working with a key, you will have to associate it with some Fortran data type when a record is retrieved, for example, if the key is put into a Fortran variable or array element by a READ. Regardless of the Fortran data types by which you may recognize or manipulate a key, VSAM considers the key to be a single string of one or more characters. VSAM always compares the keys using the EBCDIC collating sequence. If the key is seen by VS FORTRAN Version 2 as some data type other than character (as integer or real, for instance), the VSAM key comparisons may not be equivalent to the Fortran internal values. This does not mean that the key must be character data type, but it does mean that the key data type must be consistent with what VSAM expects when a record is written.

Another aspect of key processing important to VS FORTRAN Version 2 users is that a key may logically consist of more than one data item, with the same or different Fortran data types. But to VSAM, the key must form a contiguous character string in its file record. Therefore, the key used as an argument in a direct retrieval (READ with KEY=) must refer to a single data item. If you do divide the key into more than one item, an EQUIVALENCE statement can be used to define a variable that provides a single name for the composite items, and then that name can be used for the key value in the retrieval.

Processing VSAM Linear Files

After you have created a data object and defined it to the system, you must provide a DD statement to identify it, unless the data set is to be dynamically allocated. (For more information on identifying the linear data set, see *VS FORTRAN Version 2 Language and Library Reference*.)

The following JCL example will run an application program named EXAMPLE. The system will connect the actual data set name, DIV.EXAMPLE, to the program through the ddname DIVOBJ.

```
//*  
//*          RUN A DATA-IN-VIRTUAL APPLICATION  
//*  
//MYPROG      EXEC PGM=EXAMPLE  
//STEPLIB     DD DSN=MYDIV.LOAD.JOBS,DISP=SHR  
//           DD DSN=SYS1.FORTRAN.VSF2FORT,DISP=SHR  
//DIVOBJ      DD DSN=DIV.EXAMPLE,DISP=OLD  
//SYSABEND    DD SYSOUT=*  
//FT06F001    DD SYSOUT=A  
/*
```

Obtaining the VSAM Return Code—IOSTAT Option

If you specify the IOSTAT option for VSAM input/output statements, and an error occurs while VSAM is processing it, you receive the VSAM error information for the operation attempted in the IOSTAT data item.

(If the error occurs while Fortran is processing it, you receive an IOSTAT value that is the same as the VS FORTRAN Version 2 error code.)

The VSAM error information 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.

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 that contain the pair of VSAM codes. For example:

```
INTEGER*2 I2(2)  
INTEGER*4 I  
EQUIVALENCE (I2,I)  
OPEN (10,ACCESS='DIRECT',RECL=100)  
WRITE (10,REC=99,IOSTAT=I,ERR=1000)  
  
...  
  
1000 WRITE (6,*) 'VSAM ERROR: RETURN CODE=', I2(1),  
2 'REASON CODE=', I2(2)
```

The VSAM documentation for the system you're operating under gives the meaning of these return and reason codes. For more information, see the appropriate VSAM publications for your system.

Chapter 12. Considerations for Specifying RECFM, LRECL, and BLKSIZE

Priority of Processing under CMS	227
Priority of Processing under MVS	228
Default Values	228
Installation Defaults	230

The values of the record format, record length, and block size may come from several sources: file definitions (DD statements, ALLOCATE commands, or FILEDEF commands), CALL FILEINF statements for dynamically-allocated files, data set labels of existing data sets, and defaults.

The following sections discuss the priority of processing these values and list the defaults.

Priority of Processing under CMS

Under CMS, the values for RECFM, LRECL, and BLKSIZE are obtained from the file definition, or CALL FILEINF statement for dynamically-allocated files. The values, if any, will override the values from an existing file.

Because CMS uses a different file structure than MVS, not all file information is retained as part of a data set label as is done under MVS. Therefore, VS FORTRAN programs under CMS do not use information from an existing file when the file is written over. For initial READ operations, the existing file information is used if available according to the rules under "Default Values" on page 228.

Note the following considerations for coding file definitions and CALL FILEINF statements:

- If a program issues a READ statement (which may be followed by subsequent write operations) as the first I/O operation after an OPEN statement or as the first I/O operation to a preconnected file, any values specified on a file definition or CALL FILEINF statement must be the same as the values used when the file was created; otherwise, an I/O error might occur during subsequent I/O operations.
- If a program issues a WRITE statement as the first I/O operation, the record format, record length, and block size information is used from the file definition or CALL FILEINF statement. This information may be different from what was used to create a file that is being rewritten, except in the case when DISP MOD is specified on a FILEDEF command and STATUS on the OPEN statement is OLD.
- Unlike information from a FILEDEF command, the information from a CALL FILEINF statement prior to an OPEN statement is no longer available after a CLOSE statement for the same unit. When the file is subsequently reconnected, a CALL FILEINF statement may need to precede the new OPEN statement in order to provide the same information.

Priority of Processing under MVS

Under MVS, the priority of processing RECFM, LRECL, and BLKSIZE values from different sources is as follows:

1. The values are obtained from the file definition, or CALL FILEINF statement for dynamically-allocated files. The values, if any, will override the values from the existing data set if it is available.
2. The values are obtained from the existing data set if available. Values missing from the file definition or CALL FILEINF statement are used from this source.

The resulting combination of the three parameters are processed after each parameter is determined according to the priority listed above. Any missing parameters will be obtained either from installation defaults or default values assigned based on the rules specified under "Default Values."

For a file connected with an OPEN statement, the record format, record length, and block size information of an existing data set is written over as part of the data set label in the following cases:

- The ACTION on the OPEN statement is specified as WRITE
- The STATUS on the OPEN statement is specified as NEW
- The file does not exist, and STATUS is UNKNOWN
- OUT is specified on the LABEL parameter of a JCL DD statement, or OUTPUT is specified on an ALLOCATE command.

For cases other than those above, the data set label is not rewritten. The resulting combination of record format, record length, and block size should be the same as the existing data set, or I/O errors may occur.

For a preconnected file, the record format, record length, and block size information of an existing data set is written over as part of the data set label when the first I/O operation is WRITE. If the first I/O operation is a READ, I/O errors may occur if the resulting combination of the values is not the same as that of the old data set.

Default Values

This section describes the VS FORTRAN process for assigning default values for RECFM, LRECL, and BLKSIZE for programs under either CMS or MVS.

For programs under CMS, keep in mind when reading this section that information from an existing file is *not* available for WRITE operations, nor for READ operations when the file mode number is 4. For READ operations when the file mode is not 4, only partial information may be available unless the record format is F.

RECFM: If the record format is not specified and is not available from the old data set, it is obtained from the installation default (installation defaults are given under "Installation Defaults" on page 230) unless ACCESS is DIRECT, in which case the record format is always F.

LRECL and BLKSIZE: The values for LRECL and BLKSIZE depend on the RECFM value and the information available. The following describes the process for determining the LRECL and BLKSIZE values depending on which values are provided by a file definition or CALL FILEINF statement:

- **LRECL and BLKSIZE both provided**

- *RECFM* is *F*, *FA*, *U*, or *UA*: The LRECL provided is ignored and is set to equal BLKSIZE.
- *RECFM* is *V* or *VA*: The LRECL provided is ignored and is set to equal BLKSIZE minus 4.
- *RECFM* is *FB* or *FBA*: BLKSIZE is made equal to the largest exact multiple of LRECL less than or equal to the BLKSIZE provided. If LRECL is greater than the BLKSIZE, LRECL is made equal to BLKSIZE.
- *RECFM* is *VB* or *VBA*: The values provided are used unless LRECL plus 4 is greater than BLKSIZE. LRECL is set to BLKSIZE minus 4 in this case.
- *RECFM* is *VS* or *VBS*: The values provided are used.
- **BLKSIZE is provided but LRECL is not**
 - *RECFM* is *F*, *FA*, *U*, or *UA*: LRECL is made equal to BLKSIZE.
 - *RECFM* is *V* or *VA*: LRECL is set to BLKSIZE minus 4.
 - *RECFM* is *FB* or *FBA*: LRECL is obtained from the existing data set, if available. If it is not available, or is invalid, the installation default is used. If LRECL is greater than BLKSIZE, it is made equal to BLKSIZE. If BLKSIZE is not an exact multiple of LRECL, BLKSIZE is adjusted to be the largest exact multiple of LRECL less than or equal to the BLKSIZE specified.
 - *RECFM* is *VB* or *VBA*: LRECL is obtained from the existing data set, if available. If it is not available, the installation default is used. If LRECL plus 4 is greater than BLKSIZE, or is invalid, LRECL is set to BLKSIZE minus 4.
 - *RECFM* is *VS* or *VBS*: LRECL is obtained from the existing data set, if available. If it is not available, the installation default is used.
- **LRECL is provided but BLKSIZE is not**
 - *RECFM* is *F*, *FA*, *U*, or *UA*: BLKSIZE is made equal to LRECL.
 - *RECFM* is *V* or *VA*: BLKSIZE is set to LRECL plus 4.
 - *RECFM* is *FB* or *FBA*: BLKSIZE is obtained from the existing data set, if available. If it is not available, for systems that have the system determined block size feature the BLKSIZE is determined by the operating system. For operating systems that do not have this feature, the installation default is used. If LRECL is greater than BLKSIZE, BLKSIZE is made equal to LRECL. If BLKSIZE is greater than LRECL, BLKSIZE is set to be the largest exact multiple of LRECL less than or equal to the available BLKSIZE.
 - *RECFM* is *VB* or *VBA*: BLKSIZE is obtained from the existing data set, if available. If it is not available, for systems that have the system determined block size feature the BLKSIZE is determined by the operating system. For operating systems that do not have this feature, the installation default is used. If LRECL plus 4 is greater than BLKSIZE, BLKSIZE is set to LRECL plus 4.
 - *RECFM* is *VS* or *VBS*: BLKSIZE is obtained from the existing data set, if available. If it is not available, and records are blocked, and for systems that have the system determined block size feature, the BLKSIZE is

determined by the operating system. For operating systems that do not have this feature, the installation default is used.

- **LRECL and BLKSIZE both not provided**
 - *RECFM is F, FA, U, or UA:* If BLKSIZE is available, it is used; in this case, LRECL is made equal to BLKSIZE. If BLKSIZE is not available, but LRECL is, BLKSIZE is made equal to LRECL. If LRECL is not available, BLKSIZE and LRECL are both set to the installation default for BLKSIZE.
 - *RECFM is V or VA:* If BLKSIZE is available, it is used; in this case, LRECL is set to BLKSIZE minus 4. If BLKSIZE is not available, but LRECL is, BLKSIZE is set to LRECL plus 4. If neither are available, the installation default is used for BLKSIZE and LRECL is set to BLKSIZE minus 4.
 - *RECFM is FB or FBA:* If LRECL and BLKSIZE are available, they are used. If LRECL is not available, the installation default is used. If BLKSIZE is not available, for systems that have the system determined block size feature it is determined by the operating system. For operating systems that do not have this feature, the installation default is used for BLKSIZE. If LRECL is greater than BLKSIZE, LRECL is adjusted to equal BLKSIZE.
 - *RECFM is VB or VBA:* If LRECL and BLKSIZE are available, they are used. If LRECL is not available, the installation default is used. If BLKSIZE is not available, for systems that have the system determined block size feature it is determined by the operating system. For operating systems that do not have this feature, the installation default is used for BLKSIZE. If LRECL plus 4 is greater than BLKSIZE, LRECL is adjusted to equal BLKSIZE minus 4.
 - *RECFM is VS or VBS:* If LRECL and BLKSIZE are available, they are used. If records are blocked and BLKSIZE is not available, for systems that have the system determined block size feature BLKSIZE is determined by the operating system. For operating systems that do not have this feature, or records are not blocked, the installation default is used for BLKSIZE.

Installation Defaults

The IBM-supplied installation defaults for RECFM, LRECL, and BLKSIZE are given in Figure 58. These defaults may have been modified at your site.

Note: Previous to VS FORTRAN Version 2 Release 3, defaults for RECFM, LRECL, and BLKSIZE could not be modified at installation time. If you previously relied on defaults for these options and your site modified the defaults when installing Release 3, your programs may run incorrectly. For such programs, be sure to code these options on the file definition or CALL FILEINF statement in order to avoid problems.

Figure 58. IBM-Supplied Installation Defaults for File Characteristics

Characteristic	Input Unit 5	Output Unit 6	Punch Unit 7	All Other Units
RECFM				
Formatted, sequential access	F	UA	F	MVS: U CMS: F (Note 1)
Unformatted, sequential access	F	Note 2	F	VS
LRECL				
Formatted, sequential access	80	133	80	MVS: 800 CMS: 80 (Note 1)
Unformatted, sequential access	80	Note 2	80	unlimited
BLKSIZE				
Formatted, sequential access	80	133	80	MVS: 800 CMS: 80 (Note 1)
Unformatted, sequential access	80	Note 2	80	800

Notes:

1. Under CMS, if the defaults for formatted I/O were selected at installation time to have OS/VS characteristics, the MVS default applies.
2. Only files connected for sequential access and having formatted records may be directed to the error message unit.

Chapter 13. What Determines File Existence

CMS File Existence Tables	235
Basic Conditions (CMS)	236
DASD Device File Existence Table (CMS)	237
Reusable VSAM File Existence Table (CMS)	238
Nonreusable VSAM File Existence Table (CMS)	238
Library Member File Existence Table (CMS)	239
Tape File Existence Table (CMS)	239
Terminal Existence (CMS)	239
Unit Record Input Device Existence (CMS)	239
Unit Record Output Device Existence Table (CMS)	240
Files Whose File Definitions Specify DUMMY Existence Table (CMS)	240
File Existence for Files on Other Devices (CMS)	241
MVS File Existence Tables	241
Basic Conditions (MVS)	241
DASD File Existence Table (MVS)	243
Reusable VSAM File Existence Table (MVS)	244
Nonreusable VSAM File Existence Table (MVS)	244
PDS Member Existence Table (MVS)	245
Labeled Tape File Existence Table (MVS)	245
Unlabeled Tape File Existence Table (MVS)	246
Input-Stream (DD * or DD DATA) Data Set Existence (MVS)	246
System Output (Sysout) Data Set Existence Table (MVS)	247
Terminal Existence (MVS)	247
Unit Record Input Device Existence (MVS)	247
Unit Record Output Device Existence Table (MVS)	248
Files Whose File Definitions Specify DUMMY Existence Table (MVS)	248
File Existence for Files on Other Devices, Including Subsystems (MVS)	249

This chapter discusses the conditions that VS FORTRAN uses to determine whether a file exists for a Fortran program. VS FORTRAN uses the determination for:

- Determining the value returned for the EXIST specifier on the INQUIRE statement.
- Verifying whether STATUS='OLD' or STATUS='NEW' on the OPEN statement correctly reflects the status of a file's existence. (This verification is done only when OCSTATUS is in effect and only for files on certain device types.)
- Determining whether a file needs to be created when the STATUS specifier on the OPEN statement is omitted, or when STATUS='SCRATCH' or STATUS='UNKNOWN' is specified.

The INQUIRE and OPEN statements are discussed in Part 3, "Performing Input/Output Operations" on page 131. For detailed information about the syntax of these statements, see *VS FORTRAN Version 2 Language and Library Reference*.

An external file that your program can read or that your program has written is said to exist for your program. For a specific file, however, there are many factors that control whether the file exists from the Fortran point of view. Among these factors

are the presence of a file definition, the type of device the file resides on, and, for some device types, whether the file contains any data. This chapter contains a series of decision tables, one for each device type, that determine whether and when a file exists while your program runs.

For the most part, the decision tables that define file existence reflect the intuitive notion of what file existence is. For example, if there is a file definition that refers to a disk file and that disk file is present on the disk volume, then the decision table for disk files should indicate that the file exists. There are, however, a few anomalies, even for the simple case of the disk file. Some of these are discussed here. However, the detailed definition of file existence is shown in the decision tables.

Disk files on CMS minidisks: If a file is on a CMS minidisk and it is referred to by a file definition, it always exists. However, a CMS minidisk can never contain an empty file. As a result, your program will see an empty file as existing only if you have created it within that program and have not deleted it. But that same empty file will not exist when a subsequent program, that has not yet created it using a WRITE or OPEN statement, runs.

Disk files under MVS: Under MVS, a disk file that is referred to by a file definition and that has data records in it always exists. Just allocating the space for the file on a DASD volume does not mean that the file exists. This is true even though that space may have been allocated previously and your DD statement specifies DISP=OLD. The disk file doesn't exist until it has been created within a Fortran program using a WRITE or OPEN statement.

A disk file created by your program which does not contain data records will be seen as existing within that program as long as you don't delete it. However, in a program that you run later, that file will not exist until that program recreates it.

VSAM files: A VSAM file that is referred to by a file definition and that has had data written into it always exists, even if all of the records have since been deleted. Simply defining the file through Access Method Services does not cause it to exist as far as Fortran is concerned. The VSAM file doesn't exist until it has been created within a Fortran program using a WRITE or OPEN statement.

A VSAM file that was created by your program but which has never had data records written into it will be seen as existing within that program as long as you don't delete it. However, in a program you run later, that file will not exist until that program recreates it.

Striped files: The existence of a striped file is based upon the existence of the first stripe, FTnnP001. If the first stripe exists, then the striped file as a whole is seen to exist. A stripe can only be defined on disk (MVS only) or tape (CMS and MVS).

File on terminals: A terminal is an unusual I/O device in that your data is not stored permanently within the device. Also, your program can read data from or write data to the terminal regardless of any previous data transfer. As a result, the existence of a file that is on a terminal is somewhat nebulous. VS FORTRAN arbitrarily considers that files on terminals always exist.

Decision Tables: The following tables show the conditions that VS FORTRAN checks to determine file existence. The tables are divided by operating system (CMS and MVS). To use the tables, begin with Figure 59 on page 236 and Figure 60 on page 236 for CMS or Figure 68 on page 241 and Figure 69 on page 242 for MVS; those tables may then refer you to other decision tables that determine file existence by device type.

A *file*, as referred to in the decision tables, means different things. For dynamically allocated named files, a file is identified by its CMS file identifier or MVS data set name. When the FILEHIST run-time option is in effect, a file is identified only by its ddname (no dynamic file allocation). When the NOFILEHIST run-time option is in effect, a file is identified by its ddname and:

- For CMS:
 - Device type *and*
 - For files on minidisks: filename, filetype, and filemode.
- For MVS:
 - Device type, data set name, and volume serial number *or*
 - System output class for system output (sysout).

These tables assume that where dynamic file allocation is not involved, you are determining the existence of a file for which the ddname is already known.

In the tables, “—” indicates that this condition does not affect file existence in the specified instance.

CMS File Existence Tables

These types of CMS files are referenced:

- Basic conditions, page 236
- DASD files, page 237
- Reusable VSAM files, page 238
- Nonreusable VSAM files, page 238
- Library members, page 239
- Tape files, page 239
- Terminals, page 239
- Unit record input devices, page 239
- Unit record output devices, page 240
- Files whose file definitions specify DUMMY, page 240
- Files on other devices, page 241.

Basic Conditions (CMS)

Figure 59. Basic Conditions for File Existence (CMS)

File Internally Open(1)	Dynamic File Allocation	Named File	Valid Unit Number(2)	Then the File:
Yes	—	—	—	Exists
No	Yes	—	—	Note 3
No	No	Yes	—	Note 4
No	No	No	No	Doesn't exist
No	No	No	Yes	Note 5

Figure 60. Basic Conditions for Unnamed File Existence (CMS)

ddname of the form FTnnFmmm	ddname of the form FTnnPmmm	FTnnP001 Explicit FILEDEF in Effect	FTnnPmmm Explicit FILEDEF in Effect	Stripe Number mmm ≤ Total Number of Stripes(6)	Then the File:
No	No	—	—	—	Note 4
Yes	No	Yes	—	—	Doesn't exist
Yes	No	No	—	—	Note 4
No	Yes	No	—	—	Doesn't exist
No	Yes	Yes	No	—	Doesn't exist
No	Yes	Yes	Yes	No	Doesn't exist
No	Yes	Yes	Yes	Yes	Note 7

Notes to Figure 59 and Figure 60:

1. *File Internally Open* means that a file either has been connected by an OPEN statement; or, in the case of a preconnected file, a READ, WRITE, PRINT, or ENDFILE statement for that file has been successfully run. A single stripe FTnnPmmm of a striped file is considered internally open if FTnnP001 is internally open and the stripe number mmm is not greater than the total number of stripes as derived from the data set name for FTnnP001.
2. For an unnamed file, the ddname of which is of the form FTnnFmmm FTnnKkk, or FTnnPmmm, Valid Unit Number means that nn is both:
 - Less than 99
 - Less than the limit specified for unit number when the product was installed.

3. Based on the device type, refer to the CMS decision tables that follow to determine file existence.
4. If an explicit file definition has been provided, then based on the device type, refer to the CMS decision tables that follow to determine file existence. Otherwise, a default file definition is supplied of the form:

```
FILEDEF ddname DISK FILE ddname A
```

See “DASD Device File Existence Table (CMS)” on page 237 for determining file existence.
5. Refer to Figure 60.
6. The *Total Number of Stripes* is derived from the data set name for FTnnP001 in which the file type is of the form *apxy*, where *x* is 1 to 3 digits indicating the total number of stripes in the striped file.
7. This depends on the existence of FTnnP001. Based on the device type, refer to the CMS decision tables that follow to determine file existence for FTnnP001.

DASD Device File Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 61 are in effect:

Figure 61. File Existence Table for DASD Device (CMS)

Access Restricted(1)	Explicit FILEDEF in Effect	File On Minidisk	File History being Ignored(2)	File Ever Internally Opened	File Deleted by CLOSE	Then the File:
Yes	Yes	—	—	—	—	**Error detected**
Yes	No	—	—	—	—	Doesn't exist
No	—	Yes	—	—	—	Exists
No	—	No	Yes	—	—	Doesn't exist
No	—	No	No	No	—	Doesn't exist
No	—	No	No	Yes	No	Exists
No	—	No	No	Yes	Yes	Doesn't exist

Note:

1. *Access Restricted* refers to any condition that prevents VS FORTRAN from accessing the file, such as the minidisk not being accessed.
2. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Reusable VSAM File Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 62 are in effect:

Figure 62. File Existence Table for Reusable VSAM (CMS)

Subfile Sequence Number>1	File Contains Data	File History being Ignored(1)	File Ever Internally Opened	File Deleted by CLOSE	Then the File:
Yes	—	—	—	—	Doesn't exist
No	Yes	—	—	—	Exists
No	No	Yes	—	—	Doesn't exist
No	No	No	No	—	Doesn't exist
No	No	No	Yes	No	Exists
No	No	No	Yes	Yes	Doesn't exist

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Nonreusable VSAM File Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 63 are in effect:

Figure 63. File Existence Table for Nonreusable VSAM (CMS)

Subfile Sequence Number>1	File Contains Data	File History being Ignored(1)	File Ever Internally Opened	Then the File:
Yes	—	—	—	Doesn't exist
No	Yes	—	—	Exists
No	No	Yes	—	Doesn't exist
No	No	No	No	Doesn't exist
No	No	No	Yes	Exists

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Library Member File Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 64 are in effect:

Figure 64. File Existence Table for Library Member (CMS)

Access Restricted(1)	Library On Minidisk(2)	Member Exists	Then the File:
Yes	—	—	**Error detected**
No	No	—	Doesn't exist
No	Yes	Yes	Exists
No	Yes	No	Doesn't exist

Notes:

1. *Access Restricted* refers to any condition that prevents VS FORTRAN from accessing the file, such as the minidisk not being accessed.
2. A *library* is a CMS TXTLIB, MACLIB, or LOADLIB.

Tape File Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 65 are in effect:

Figure 65. File Existence Table for Tape File (CMS)

Access Restricted(1)	File On Volume	Then the File:
Yes	—	**Error detected**
No	Yes	Exists
No	No	Doesn't exist

Note:

1. *Access Restricted* refers to any condition that prevents VS FORTRAN from accessing the file, such as the tape not being attached.

Terminal Existence (CMS)

If the basic conditions listed under “Basic Conditions (CMS)” on page 236 are in effect, the file exists.

Unit Record Input Device Existence (CMS)

If the basic conditions listed under “Basic Conditions (CMS)” on page 236 are in effect, the file exists.

Unit Record Output Device Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 66 are in effect:

Figure 66. File Existence Table for Unit Record Output Devices (CMS)

File History being Ignored(1)	File Ever Internally Opened	Then the File:
Yes	—	Doesn't exist
No	No	Doesn't exist
No	Yes	Exists

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Files Whose File Definitions Specify DUMMY Existence Table (CMS)

If, in addition to the basic conditions listed under “Basic Conditions (CMS)” on page 236, the conditions listed in Figure 67 are in effect:

Figure 67. File Existence Table for Files Whose File Definitions Specify DUMMY (CMS)

File History being Ignored(1)	File Ever Internally Opened	File Deleted by CLOSE	Then the File:
Yes	—	—	Exists
No	No	—	Exists
No	Yes	No	Exists
No	Yes	Yes	Doesn't exist

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

File Existence for Files on Other Devices (CMS)

If the basic conditions listed under “Basic Conditions (CMS)” on page 236 are in effect, the file exists.

MVS File Existence Tables

These types of MVS files are referenced:

- Basic conditions, page 241
- DASD files, page 243
- Reusable VSAM files, page 244
- Nonreusable VSAM files, page 244
- PDS members, page 245
- Labeled tape files, page 245
- Unlabeled tape files, page 246
- Input-stream (DD * or DD DATA) data sets, page 246
- System output (sysout) data sets, page 247
- Terminals, page 247
- Unit record input devices, page 247
- Unit record output devices, page 248
- Files whose file definitions specify DUMMY, page 248
- Files on other devices, including subsystems, page 249.

Basic Conditions (MVS)

Figure 68. Basic Conditions for File Existence (MVS)

File Internally Open(1)	Dynamic File Allocation	File Definition in Effect	Named File	Valid Unit Number(2)	Then the File:
Yes	—	—	—	—	Exists
No	Yes	—	—	—	Note 3
No	No	No	—	—	Doesn't exist
No	No	Yes	Yes	—	Note 3
No	No	Yes	No	No	Doesn't exist
No	No	Yes	No	Yes	Note 4

Figure 69. Basic Conditions for Unnamed File Existence (MVS)

ddname of the form FTnnFmmm	ddname of the form FTnnPmmm	FTnnP001 Explicit FILEDEF in Effect	Stripe Number mmm ≤ Total Number of Stripes(5)	Then the File:
No	No	—	—	Note 3
Yes	No	Yes	—	Doesn't exist
Yes	No	No	—	Note 3
No	Yes	No	—	Doesn't exist
No	Yes	Yes	No	Doesn't exist
No	Yes	Yes	Yes	Note 6

Notes to Figure 68 on page 241 and Figure 69:

1. *File Internally Open* means that a file either has been connected by an OPEN statement; or, in the case of a preconnected file, a READ, WRITE, PRINT, or ENDFILE statement for that file has been successfully run. A single stripe FTnnPmmm of a striped file is considered internally open if FTnnP001 is internally open and the stripe number mmm is not greater than the total number of stripes as derived from the data set name for FTnnP001.
2. For an unnamed file, the ddname of which is of the form FTnnFmmm FTnnKkk, or FTnnPmmm, Valid Unit Number means that nn is both:
 - Less than 99
 - Less than the limit specified for unit number when the product was installed.
3. Based on the device type, refer to the MVS decision tables that follow to determine file existence.
4. Refer to Figure 69.
5. The *Total Number of Stripes* is derived from the data set name for FTnnP001 in which the last qualifier is of the form apxy, where x is 1 to 3 digits indicating the total number of stripes in the striped file.
6. This depends on the existence of FTnnP001. Based on the device type, refer to the CMS decision tables that follow to determine file existence for FTnnP001.

DASD File Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 70 are in effect:

Figure 70. File Existence Table for DASD (MVS)

Volume Sequence Number(1)>1	Data Set on Volume(2)	Data Set Concatenated(3)	Data Set Contains Data	Dynamic File Allocation	File History being Ignored(4)	File Ever Internally Opened	File Deleted by CLOSE	Then the File:
Yes	—	—	—	—	—	—	—	Exists
No	No	—	—	—	—	—	—	Doesn't exist
No	Yes	Yes	—	—	—	—	—	Exists
No	Yes	No	Yes	—	—	—	—	Exists
No	Yes	No	No	Yes	—	—	—	Exists
No	Yes	No	No	No	Yes	—	—	Doesn't exist
No	Yes	No	No	No	No	No	—	Doesn't exist
No	Yes	No	No	No	No	Yes	No	Exists
No	Yes	No	No	No	No	Yes	Yes	Doesn't exist

Notes:

1. For DASD, *Data Set on Volume* means that space is currently allocated for the data set on the volume.
2. The volume sequence number identifies the volume of a multivolume data set to be used to begin processing the data set. For more information, see *MVS/370 JCL Reference*.
3. *Concatenated* refers to a JCL concatenation of data sets having a single ddname, not a set of subfiles having different ddnames.
4. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Reusable VSAM File Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 71 are in effect:

Figure 71. File Existence Table for Reusable VSAM (MVS)

Subfile Sequence Number >1	Data Set Contains Data	File History being Ignored(1)	File Ever Internally Opened	File Deleted by CLOSE	Then the File:
Yes	—	—	—	—	Doesn't exist
No	Yes	—	—	—	Exists
No	No	Yes	—	—	Doesn't exist
No	No	No	No	—	Doesn't exist
No	No	No	Yes	No	Exists
No	No	No	Yes	Yes	Doesn't exist

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Nonreusable VSAM File Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 72 are in effect:

Figure 72. File Existence Table for Nonreusable VSAM (MVS)

Subfile Sequence Number >1	Data Set Contains Data	File History being Ignored(1)	File Ever Internally Opened	Then the File:
Yes	—	—	—	Doesn't exist
No	Yes	—	—	Exists
No	No	Yes	—	Doesn't exist
No	No	No	No	Doesn't exist
No	No	No	Yes	Exists

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

PDS Member Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 73 are in effect:

Figure 73. File Existence Table for PDS Member (MVS)

Access Restricted(1)	Member Exists	Then the File:
Yes	—	**Error detected**
No	No	Doesn't exist
No	Yes	Exists

Note:

1. *Access Restricted* refers to any condition that prevents VS FORTRAN from accessing the file, such as FACF protection or the inability to mount a volume.

Labeled Tape File Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 74 are in effect:

Figure 74. File Existence Table for Labeled Tape File (MVS)

Access Restricted(1)	Data Set On Volume(2)	Block and Record Lengths Are 0 in Header Label	Then the File:
Yes	—	—	**Error detected**
No	No	—	Doesn't exist
No	Yes	Yes	Doesn't exist
No	Yes	No	Exists

Notes:

1. *Access Restricted* refers to any condition that prevents VS FORTRAN from accessing the file, such as RACF protection or the inability to mount a volume.
2. For a labeled tape, *data set on volume* means that the volume can be positioned to the data set and that the data set name in the header label matches the data set name given in the DD statement or ALLOCATE command.

Unlabeled Tape File Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 75 are in effect:

Figure 75. File Existence Table for Unlabeled Tape File (MVS)

Access Restricted(1)	Data Set On Volume(2)	<i>Then the File:</i>
Yes	—	**Error detected**
No	No	Doesn't exist
No	Yes	Exists

Notes:

1. *Access Restricted* refers to any condition that prevents VS FORTRAN from accessing the file, such as RACF protection or the inability to mount a volume.
2. For an unlabeled tape, *data set on volume* means that the volume can be positioned to the data set.

Input-Stream (DD * or DD DATA) Data Set Existence (MVS)

If the basic conditions listed under “Basic Conditions (MVS)” on page 241 are in effect, the file exists.

System Output (Sysout) Data Set Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 76 are in effect:

Figure 76. File Existence Table for Sysout Data Sets (MVS)

File History being Ignored(1)	File Ever Internally Opened	<i>Then the File:</i>
Yes	—	Doesn't exist
No	No	Doesn't exist
No	Yes	Exists

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Terminal Existence (MVS)

If the basic conditions listed under “Basic Conditions (MVS)” on page 241 are in effect, the file exists.

Unit Record Input Device Existence (MVS)

If the basic conditions listed under “Basic Conditions (MVS)” on page 241 are in effect, the file exists.

Unit Record Output Device Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 77 are in effect:

Figure 77. File Existence Table for Unit Record Output Devices (MVS)

File History being Ignored(1)	File Ever Internally Opened	Then the File:
Yes	—	Doesn't exist
No	No	Doesn't exist
No	Yes	Exists

Note:

1. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

Files Whose File Definitions Specify DUMMY Existence Table (MVS)

If, in addition to the basic conditions listed under “Basic Conditions (MVS)” on page 241, the conditions listed in Figure 78 are in effect:

Figure 78. File Existence Table for Files Whose File Definitions Specify DUMMY (MVS)

File Conca- tenated(1)	File History being Ignored(2)	File Ever Internally Opened	File Deleted by CLOSE	Then the File:
Yes	—	—	—	Exists
No	Yes	—	—	Exists
No	No	No	—	Exists
No	No	Yes	No	Exists
No	No	Yes	Yes	Doesn't exist

Note:

1. *Concatenated* refers to a JCL concatenation of data sets having a single ddname, not a set of subfiles having different ddnames.
2. *File History being Ignored* means either that the NOFILEHIST run-option is in effect **or** that the subroutine IGNFHU or IGNFHDD has been invoked for this file since the preceding CLOSE statement, if any, involving this file. In either of these cases, the effect is that file existence is determined without regard for previous uses of this file in the executable program.

File Existence for Files on Other Devices, Including Subsystems (MVS)

If the basic conditions listed under “Basic Conditions (MVS)” on page 241 are in effect, the file exists.

Part 4. High Performance Features

Chapter 14. Overview of High Performance Features

Overview of the Optimization Feature	253
Optimization Level 0	253
Optimization Level 1	253
Optimization Levels 2 and 3	254
Optimizations	255
Overview of the Automatic Vector Feature	256
Overview of the Automatic Parallel Feature	257
Terminology for Automatic Vector and Parallel Processing	258
Classification of Dependences	262
Mode	262
Type	262
Direction	263
Level	264
Interchange	264
Overview of the Parallel Language Feature	266

VS FORTRAN Version 2 provides features that allow you to reduce the real time it takes to run your programs. The optimization feature generates more efficient code. The vector facility allows you to run eligible loops more quickly. The parallel feature allows you to run computationally independent parts of your programs on multiple processors. All three features are specified at compile time, and are complementary. An overview of each feature is given below.

Overview of the Optimization Feature

Optimization requires additional compile time but usually results in reduced run time. You can use the OPTIMIZE compile-time option to select one of four optimization levels, or you can choose not to optimize your program. The number of times the compiled program is to be run can help determine which optimization level to use. If a program is to be run more than a few times, use the highest workable optimization level, either OPT(2) or OPT(3).

Four optimization levels are available.

Optimization Level 0

OPTIMIZE(0) or NOOPTIMIZE is the recommended level of optimization for a program being debugged, or compiled to check syntax. While it provides the fastest compile time, it produces programs with the least efficient run time. The compiler may perform some minor optimizations unless they are suppressed with the SDUMP option.

Optimization Level 1

OPTIMIZE(1) performs register and branch optimization, a modest level of optimization for programs without nested loops. Unnecessary loads and stores are eliminated where possible. Loop structure is not considered.

Optimization Levels 2 and 3

OPTIMIZE(2) and OPTIMIZE(3) perform the most optimization. Control and data flow analysis is considered for the entire program. Optimizations such as common expression elimination, strength reduction, code motion, and global register assignment are done. Particular attention is paid to innermost loops and to subscript address calculations. This may detect errors undetectable at lower optimization levels.

Note that optimization level OPT(2) or OPT(3) is required for automatic vector and parallel processing.

Optimization Level 2: This option performs full text and register assignment. It is identical to optimization level 3, except that certain optimization procedures are suppressed to provide *interruption localizing*. The rule in interruption localizing is: do not move any code out of a loop that might cause an interruption that would not occur without optimization.

An example is division by zero within a loop. If the division were moved out of the loop by the optimizer, it is no longer checked for a zero divisor. For example, in the loop:

```
      DO 2 J=1,N
        IF (K.NE.0) M(J)=N/K
2     CONTINUE
```

code evaluating the expression N/K could be moved outside the loop, because it is invariant for each iteration of the loop. However, at OPTIMIZE(2), it will not be moved.

Invariant computations involving floating-point arithmetic or integer division (including the MOD function), or intrinsic function calls with invariant arguments, are not moved out of a loop.

Optimization Level 3: This is the highest level of optimization performed by the compiler. Invariant computations are moved outside loops wherever possible. This may result in unanticipated interruptions, but incorrect answers are not generated from a legal program. The only difference from OPTIMIZE(2) is that a different number of error conditions may be raised at run time.

If the preceding example is compiled at OPTIMIZE(3), the invariant computation N/K is moved outside the loop as follows:

```
      itemp=N/K
      DO 2 J=1,N
        IF (K.NE.0) M(J)=itemp
2     CONTINUE
```

where *itemp* is a compiler-generated temporary. If *K* is zero, an unanticipated interruption (for integer division by zero) occurs in calculating *itemp*. However, values stored in the elements of array *M* remain the same.

Optimizations

Several optimizations are done by the compiler. The more general techniques used by OPTIMIZE(2) and OPTIMIZE(3) are:

Detection of uninitialized variables: Informational messages are produced for local scalar variables that might be referenced before being defined. Variables are checked for all possible control flows. Variables in COMMON or EQUIVALENCE are not processed.

Subscript collecting: Subscript collection rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

Common expression elimination: In common expressions the same value is recalculated in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done 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.

Instruction Elimination: The compiler may eliminate code for calculations found to be unnecessary. Loads and stores are often eliminated by register optimization.

Global constant propagation and constant folding: Scalar variables (or constants) that do not change in value during processing are replaced with their numerical values. Constants used in an expression are combined and new ones generated. Some mode conversions are done and evaluation of some intrinsic functions. Constants cannot be propagated across procedures. Variables in EQUIVALENCE statements will not be optimized.

Strength reduction: Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Code motion: If variables used in a computation within a loop are not altered within the loop, it may be possible to perform the calculation outside of the loop and use the results within the loop.

Global register assignment: The variables and constants most frequently used within a loop can often be assigned to registers. The registers are initialized before entry to the loop, and if necessary, stored on exit from the loop.

Inter-loop register assignment: The compiler considers prior register assignments for sister-level loops (inner loops at the same nesting level) when assigning registers within an inner loop. This reduces moving program variables from one register to another between loops, and increases the efficiency of initializing registers for nested inner loops. It also reduces the need to spill globally optimized items to storage when moving from an inner to an outer loop. Inter-loop register assignment is most beneficial in vector loops.

Instruction Scheduling: The order of instructions is changed, if necessary, to remove conflicts in register use that cause serialization within the instruction pipeline. The order of storage updates to common blocks is preserved, so that an optimized program will operate the same way as an unoptimized program in a parallel environment.

Section oriented branching: The number of required program address registers is reduced by dividing the executable code in a very large program into sections.

Overview of the Automatic Vector Feature

The VS FORTRAN Version 2 compiler can produce programs that use the vector facility, a hardware feature that provides high-speed computation particularly suitable for scientific and engineering or financial applications.

The characteristic feature of vector instructions is that they process multiple array elements. In effect, they group and overlap the function of different iterations of a loop, and can reduce run time dramatically. Figure 79 on page 257 illustrates this reduction in run time.

```

      DO 8 K = 1, 90
        A(K)=A(K)+B(K)
8      CONTINUE

```

Traditional scalar (nonvector) processing requires each element of the array A to be computed in sequence, one after the other:

$A(1)=A(1)+B(1)$ $A(2)=A(2)+B(2)$ $A(3)=A(3)+B(3)$... $A(90)=A(90)+B(90)$
run time →

In comparison, vector processing allows the computation of multiple elements of array A to be overlapped, speeding up processing:

$A(1)=A(1)+B(1)$
 $A(2)=A(2)+B(2)$
 $A(3)=A(3)+B(3)$
 .
 .
 .
 $A(90)=A(90)+B(90)$
run time →

Figure 79. How Vector Processing Speeds Run Time

To make use of the vector facility, you must specify the VECTOR compile-time option. In addition, you must specify optimization at the OPTIMIZE(2) or OPTIMIZE(3) level. If an optimization level is not specified, OPTIMIZE(3) will be assumed.

The VECTOR compile-time option works in concert with the PARALLEL compile-time option; if you specify both, the compiler will generate what it determines is the most cost-effective code. You can use parallel and vector directives to override the compiler's decisions about whether to run loops or sections of code in vector or scalar and/or parallel or serial modes. For details about parallel and vector directives and how to use them, see "Using Parallel and Vector Directives" on page 353.

Overview of the Automatic Parallel Feature

The VS FORTRAN Version 2 parallel feature can automatically generate code that allows your serial Fortran program to run on several virtual processors of an IBM multiprocessor. By identifying multiple independent instruction streams, VS FORTRAN can allow a program to simultaneously run different iterations of a DO loop across multiple virtual processors. In general, processor time and virtual storage requirements increase for parallel processing, but the actual time needed to run a program can be reduced.

If you specify the PARALLEL(AUTOMATIC) compile-time option, the compiler will check the eligibility of your DO loops to run in parallel. If the iterations of a DO loop are eligible to run concurrently, the compiler may generate parallel code for that loop.

The PARALLEL(AUTOMATIC) compile-time option works in concert with the VECTOR compile-time option; if you specify both, the compiler will generate what it determines is the most cost-effective code. You can use parallel and vector directives to override the compiler's decisions about whether to run loops in vector or scalar and/or parallel or serial modes. For details about parallel and vector

directives and how to use them, see “Using Parallel and Vector Directives” on page 353.

Terminology for Automatic Vector and Parallel Processing

This section contains definitions of some of the terms used in discussing vector and parallel processing performed by the compiler. For a more complete glossary of terms used in the VS FORTRAN Version 2 publications, see the *VS FORTRAN Version 2 Master Index and Glossary*.

chunking for parallel processing

The number of loop iterations to be processed as a single group. For loops for which automatic parallel code is generated and for PARALLEL DO loops, you can use the PREFER CHUNK directive to specify the number of loop iterations to be processed as a single group. You can recommend a specific number of iterations, a minimum, a maximum, or a range of numbers of iterations to be processed on one virtual processor. For details about coding the PREFER CHUNK directive, see “PREFER Directive” on page 364.

A DO loop can be selected for parallel and vector code generation. In that case, the number of DO loop iterations processed by a virtual processor as a chunk is usually a multiple of the vector section size.

computational independence

A parallel thread is computationally independent of another parallel thread if the data that it references is not modified by any other parallel thread that can run simultaneously, or vice versa.

construct

A syntactic unit comprising two or more statements.

dependence

A relationship involving one or two Fortran statements. A dependence exists when a storage location is used more than once, either by successive statements or by a single statement during different iterations of a loop.

For example, a dependence exists from statement S to statement T when S stores a value later fetched by T. If there is a dependence from S to T, then T is *dependent* on S.

For information about the classifications of dependences, see “Classification of Dependences” on page 262.

induction variable

Any INTEGER*4 variable that is incremented or decremented by a fixed amount each time a loop iterates. Induction variables that are not loop variables (that is, their increments or decrements are controlled by assignment statements within a loop) are referred to as *auxiliary induction variables*.

loop distribution

The process of automatically restructuring loops. Loop distribution is possible only when the statements involved are not part of a recurrence carried by the loop being distributed.

noninductive subscript

A subscript expression in which the array elements referenced on successive iterations of a loop are not separated by a constant number of

bytes. Examples are:

```
DO 10 I = 1,N
  A(I*I) = 0.0
10 CONTINUE
```

and

```
DO 20 J = 1,N
  B(INDEX(J)) = 0.0
20 CONTINUE
```

parallel task

A complete logical environment in which parallel threads run. A parallel task has its own data storage and its own units and files associated with units.

parallel thread

A unit of work that is eligible for concurrent processing. A parallel thread can be a subroutine, iteration(s) of a loop, or a section of code.

privatization

Provides local instances of variables and arrays for use by parallel threads. In order to provide the correct values of variables to each parallel thread, certain variables are privatized. A private variable or array is one for which storage is allocated for each virtual processor and logically belongs to a single parallel thread. The private variable or array therefore exists only for the duration of the parallel thread.

All of the following are private variables:

- Variables specified in a LOCAL statement
- Temporary variables generated by the compiler which are stored within a parallel loop but not in the compiler-generated loop initialization code
- Vector temporaries created inside a parallel loop or parallel section
- Register temporaries created inside a parallel loop or parallel section
- Variables created to eliminate dependencies (similar to scalar expansion during vectorization)
- Loop induction variables
- Fortran local variables of subprograms referenced from within a parallel loop or parallel section, or called by a PARALLEL CALL.

recurrence

A group of one or more statements forming a cycle of dependences. A recurrence exists when the processing of a statement may in some way affect its processing on a later iteration. For example, statements S and T form a recurrence if there is a dependence from S to T and also a dependence from T to S. A recurrence is said to be *carried by* a loop if all dependences involved in the recurrence are caused by that loop or by some loop at a deeper level of nesting. If a recurrence exists, vectorization is not performed.

A statement that is dependent on itself forms a *single statement recurrence*. For example, consider the following DO loop.

```
DO 99 J = 1, 10
  A(J+1) = A(J) + B(J)
99 CONTINUE
```

The processing order of this loop is:

```
A(2) = A(1) + B(1)
A(3) = A(2) + B(2)
⋮
A(11) = A(10) + B(10)
```

The input on one iteration of the loop always requires the element of A, computed on the previous iteration. Therefore, the statement $A(J+1)=A(J)+B(J)$ is dependent on itself and forms a recurrence preventing vectorization.

Recurrences can also involve multiple statements as in the example below:

```
DO 100 I=1,100
80   A(I+1)=B(I-1)
90   B(I)=A(I)
100  CONTINUE
```

Statement 90 is dependent on statement 80 because of variable A. Statement 80 is dependent on statement 90 because of variable B—this forms a recurrence.

scalar expansion

The process of automatically replacing references to a scalar variable with a vector temporary. This allows the statement containing the scalar variable to be vectorized. (A vector temporary is equivalent to a one-dimensional array whose number of elements is the same as the vector section size.) For an example of scalar expansion, see page 288.

stride

The interval between elements as they are fetched and stored. It is the distance between successive data elements.

Arrays in Fortran are stored in column-major order. That means that consecutive elements are accessed in storage when the left-most subscript is varied by 1. Addressing Fortran arrays in column order is stride 1. If the array is an $i \times j$ array, the stride on the second subscript is i . Addressing Fortran arrays in row order is stride i .

Suppose you have a 4×3 array, $A(i,j)$, represented as follows:

```
A(1,1)  A(1,2)  A(1,3)
A(2,1)  A(2,2)  A(2,3)
A(3,1)  A(3,2)  A(3,3)
A(4,1)  A(4,2)  A(4,3)
```

The elements of the array are stored in ascending locations in column-major order, as shown below:

Location	Array Element
1	A(1,1)
2	A(2,1)
3	A(3,1)
4	A(4,1)
5	A(1,2)
6	A(2,2)
7	A(3,2)
8	A(4,2)
9	A(1,3)
10	A(2,3)

```

11      A(3,3)
12      A(4,3)

```

Element A(2,1) is stored immediately after A(1,1). Element A(3,1) is stored immediately after A(2,1), and so on. If you address the elements in column order, the stride is 1. If you address the elements in row order, the stride is 4.

unanalyzable loop

A loop that is ineligible for vector or parallel code generation analysis because it contains some statement or construct that prevents the compiler from gathering the information needed for vector or parallel analysis.

unsupportable loop

A loop or portion of a loop that cannot be vectorized because it contains a statement or construct that cannot be run on the vector hardware or would require a special sequence of vector instructions that the compiler does not generate.

vector

A group of array elements that are referenced in a well-defined sequence, and on which identical operations are to be performed. In a Fortran program, a vector is obtained by referring to an array inside a loop in such a way that a different element is selected on each iteration of the loop.

Examples of vectors are:

A one-dimensional array where $A(K)s$ — $A(1), A(2), \dots, A(200)$ —form a vector:

```

      REAL A(200)
      DO 10 K = 1, 200
        A(K) = 0.0
10 CONTINUE

```

A matrix where the rows and columns— $B(K,1), B(K,2), \dots, B(K,300)$ and $B(1,M), B(2,M), \dots, B(200,M)$ —form vectors:

```

      REAL B(200,300)
      DO 10 K = 1, 200
      DO 10 M = 1, 300
        B(K,M) = B(K,M) * A(K)
10 CONTINUE

```

vector length

The number of elements contained in a vector. This is equal to the number of iterations of the loop that defines the vector. For example:

```

      DO 50 I = 1,100      DO 60 J = 1,100,10
        B(I) = 0.060      A(J) = 0.0
50 CONTINUE              60 CONTINUE

```

The DO 50 loop defines a vector of length 100, while the DO 60 loop defines a vector of length 10.

vector section

A vector segment containing a fixed number of elements. This number is referred to as the *vector section size* and is sometimes represented as the letter “Z.” A vector is automatically partitioned into these sections to run on vector hardware.

Classification of Dependences

Dependences can be classified according to characteristics. These categories are:

Mode

The mode of a dependence indicates whether it results from sharing of data or because of the flow of control.

Data dependences: occur when two statements use or define identical storage locations. In the following example:

```

      DO 3 I = 1,N
1      B(I) = C(I) + 1.0
2      A(I) = B(I) + 2.0
3      CONTINUE

```

there is a data dependence from statement 1 to statement 2. The value computed by statement 1 is used as input to statement 2.

Control dependences: occur when the processing of one statement determines if another statement will be processed. In the following example:

```

      DO 3 I = 1,N
1      IF (A(I) .GT. 0.0) GO TO 3
2      A(I) = B(I) + 1.0
3      CONTINUE

```

there is a control dependence from statement 1 to statement 2: The results of the test in statement 1 determine whether statement 2 is processed. Control dependences are converted to a data dependence using a technique called IF-conversion (for more on IF-conversion, see "IF Conversion" on page 288).

Type

The type of a dependence indicates whether a variable is used or defined by each of the statements involved. The three types of dependence that are considered during vectorization are:

True dependence: If S defines a value and T references it, statement T depends upon statement S.

```

S:   X  =
T:       = X

```

S must be processed before T because S defines a value used by T. The processing of T depends upon the processing of S being completed.

Antidependence: If S references a value and T defines it, statement T depends upon statement S.

```

S:       = X
T:   X  =

```

S must be processed before T or T would store into the variable X, and S would use the wrong value. Processing of T depends upon the processing of S being completed.

Output dependence: If S stores a value also stored by T, statement T depends upon statement S.

```
S:  X  =
T:  X  =
```

S must be processed before T or the wrong value would be left behind in the variable X. The processing of T depends upon the processing of S being completed.

Direction

The direction of a dependence indicates the relative position of the statements involved in that dependence. There are two possible directions:

Forward dependence: Statement S precedes statement T, and S references a value later referenced by T.

```
      DO 3 I = 1,N
S:      A(I)  =
T:      = ... A(I) ...
      3 CONTINUE
```

Backward dependence: Statement S precedes statement T, and T references a value later referenced by S.

```
      DO 3 I = 1,N
S:      = ... A(I-1) ...
T:      A(I)  =
      3 CONTINUE
```

If a statement depends on itself, the direction of the dependence is determined by the dependence type. True and output dependences that involve only one statement are considered backward in direction, while antidependences that involve only one statement are considered forward. The reason is that a statement that depends on itself is treated as if it were written as two statements. The first assigns the value computed on the right into a temporary, and the second copies that temporary into the variable on the left. For example, in the following code:

```
      DO 3 I = 1,N
S:      A(I)  = ... A(I-1) ...
      3 CONTINUE
```

there is a true backward dependence. The loop can be rewritten exposing the dependence from statement S2 to a preceding statement S1.

```
      DO 3 I = 1,N
S1:      temp = ... A(I-1) ...
S2:      A(I)  = temp
      3 CONTINUE
```

Similarly, in the case of single statement antidependences:

```
      DO 3 I = 1,N
S:      A(I-1) = ... A(I) ...
      3 CONTINUE
```

The direction is considered to be forward since the dependence would go from statement S1 to statement S2 when the loop is rewritten:

```
      DO 3 I = 1,N
S1:    temp = ... A(I) ...
S2:    A(I-1) = temp
      3 CONTINUE
```

Level

The level of a dependence indicates the loop whose iteration causes the dependence to occur. In the following example there is a dependence at level 3 since a single element of the array is referenced on different iterations of the innermost (level 3) loop.

```
      DO 30 I = 1,N
      DO 30 J = 1,N
      DO 30 K = 1,N
        A(K,J,I) = ...
        A(K-1,J,I) = ...
      30 CONTINUE
```

In the following example, a single array element is used twice even if none of the loops are iterated. This is referred to as a *loop independent dependence*.

```
      DO 30 I = 1,N
      DO 30 J = 1,N
      DO 30 K = 1,N
        A(K,J,I) = ...
        A(K,J,I) = ...
      30 CONTINUE
```

Interchange

For an outer loop to be vectorized, it must be movable to the innermost nesting level without changing the results of the program. Dependences preventing the reordering of two loops are known as *interchange-preventing* dependences. In the following example, statement 30 has an interchange-preventing antidependence at level I.

```
      DO 40 I = 1,2
      DO 40 J = 1,2
30    A(I-1,J+1) = A(I,J)
      40 CONTINUE
```

The statements are processed in the following order:

```
I=1, J=1:    A(0,2) = A(1,1)
I=1, J=2:    A(0,3) = A(1,2)

I=2, J=1:    A(1,2) = A(2,1)
I=2, J=2:    A(1,3) = A(2,2)
```


At the end of processing, the value originally in $A(1,2)$ is stored into $A(0,3)$. If the two loops are reordered placing the I loop at the innermost level:

```
DO 40 J = 1,2
DO 40 I = 1,2
30  A(I-1,J+1) = A(I,J)
40  CONTINUE
```

the processing order of the statements becomes:

```
J=1, I=1:    A(0,2) = A(1,1)
J=1, I=2:    A(1,2) = A(2,1)

J=2, I=1:    A(0,3) = A(1,2)
J=2, I=2:    A(1,3) = A(2,2)
```

In this case, $A(2,1)$ is initially stored into $A(1,2)$ and that value is stored into $A(0,3)$. $A(0,3)$ acquires a value different than the one it had after the original loops were processed.

Sectioning for Vector Processing: Although a loop must be movable to the innermost position for it to be vectorized, you do not need to physically move the loop. The vector instructions access groups (or sections) of Z elements for a loop all at once instead of one at a time as in scalar mode. Loop controls for the loop are modified to increment by the number of elements in the groups processed by the individual instructions. Suppose the outermost loop (with index K), in the following example, is selected for vectorization. The nest:

```
DO 10 K = 1, N
DO 10 J = 1, N
DO 10 I = 1, N
    A(K,J,I) = B(K,J,I)
10  CONTINUE
```

is first conceptually rewritten by the compiler as:

```
DO 10 J = 1, N
DO 10 I = 1, N
DO 10 K = 1, N
    A(K,J,I) = B(K,J,I)
10  CONTINUE
```

to determine if the K -loop can be vectorized. Since it can, the nest is conceptually rewritten by the compiler as:

```
DO 10 K = 1, N, Z
DO 10 J = 1, N
DO 10 I = 1, N
DO 10 KK = K, K+MIN(N-K,Z-1)
    A(KK,J,I) = B(KK,J,I)
10  CONTINUE
```

The innermost loop (the loop with index KK) is not physically present; it represents processing of the vector instructions on the groups (or sections) of Z elements. In the outermost loop, the loop controls are left in place, but changed to increment by Z instead of by 1.

A DO loop can be analyzed for parallel and vector code generation, in which case the number of DO loop iterations processed by a virtual processor as a chunk is usually a multiple of the vector section size.

Overview of the Parallel Language Feature

VS FORTRAN Version 2 provides explicit parallel language constructs that allow you to code a parallel program. These constructs allow you to identify loops, sections of code, and subroutines that can be run in parallel with other loops, sections of code, and subroutines. When you explicitly code parallel language constructs in your program, you are responsible for determining that the code is valid and suitable for parallel processing. For a complete overview of the parallel feature, including the language statements and automatic parallel code generation, see Chapter 17, “Using the Parallel Feature” on page 293.

Chapter 15. Using the Optimization Feature

Increasing Optimization of Your Program	267
Optimization Recommendations	267
Programming Recommendations	268
Input/Output	268
Character Manipulations	269
Variables	269
Subroutine Arguments	271
Constant Operands	271
Arrays	272
Expressions	272
Critical Loops	273
Scalar Computations in Loops	273
Conversions	273
Arithmetic Constructions	273
IF Statements	274
Pointers	274
Debugging Optimized Programs	275

The following sections discuss programming techniques you can use to decrease the run time of your program through optimization, and considerations for debugging programs that have been optimized. For more information on the levels of optimization and descriptions of typical optimizations performed by the compiler, see "Overview of the Optimization Feature" on page 253.

Increasing Optimization of Your Program

The following section contains suggestions on how to use the optimization feature.

Optimization Recommendations

- Use OPT(0) during program development for syntax checking, testing and debugging purposes. Debugging programs at OPT(0) with the interactive debugger is straightforward, with none of the side effects of optimization.
- Use the higher optimization levels OPT(2) or OPT(3) once a program has been debugged. If the program is to be run more than once, or if the program takes more than a few CPU seconds to run, then optimization savings at run time will exceed the costs of compiling at OPT(2) or OPT(3).

Vector and parallel processing require optimization at the OPT(2) or OPT(3) level. If you are doing vector or parallel processing and you do not specify an optimization level, OPT(3) will be the default. However, the use of DEBUG or invalid Fortran statements downgrades the optimization level after it has been set. As a result, the optimization level can be downgraded to OPT(0) and vector and parallel processing will not occur.

More storage and longer compilation times are required at higher optimization levels. Depending on the complexity and number of loops in the program (opportunities for optimization), the compilation time may increase greatly. You may have to compile larger programs at OPT(0) or OPT(1) if they fail to compile at higher optimization levels.

Programming Recommendations

Programs can be either too large or too small to produce efficient code.

- Program units may be so large that register usage is affected. You can overcome this problem by coding your programs in a modular fashion.
- Be careful when designing a program in a top-down (modular) fashion. If a subroutine or function is small, the implicit cost of the call overhead may exceed the value of having the code separate from the main program. After identifying the smaller, most frequently called subroutines and functions, consider moving the code into the main program. This allows the compiler to optimize the combined code and run faster.

Input/Output

Optimization has little effect on the run time of I/O statements. It is important to write efficient I/O statements at all optimization levels. Here are some guidelines to improve I/O run time performance:

- Each of the following techniques can result in significant improvements:
 - Data striping
 - Blocking a file
 - Parallel I/O
- Unformatted I/O takes less processing time and uses less storage than formatted I/O. Unformatted I/O also maintains the precision of the data items being processed.
- When coding a block of I/O statements, place as many list items on one READ or WRITE statement as is practical. Up to 20 such items are “bundled” together to minimize calls to the I/O library.
- To save processing time, code implied DO loops in I/O statements. Certain combinations of implied-DOs are recognized and combined into a form that improves performance. Some examples of I/O statements combined for improved performance are:

```
DIMENSION A(10), B(10,20)
READ(5,10) (A(I), I=1, 10, N)
WRITE(4) ((B(I,J), I=1,10), J=1,20)
READ(3) (A(K), K=L, M, N)
WRITE(6,20) (A(J), (B(I,J),I=1, 10), J=1, 10)
```

The implied-DO level containing B is an example of improved performance, while the outer level containing A generates conventional DO loop code.

In certain cases, a simple implied-DO may be recognized as an array name and code will be generated as such. For example:

```
DIMENSION A(100)
WRITE(3) (A(I), I=1, 100)
```

writes 100 elements of array A, starting with element A(1). The following example:

```
READ(5,10) (A(J), J=N, M)
```

reads (M-N+1) elements into array A, starting with element A(N).

Character Manipulations

To generate an efficient code sequence for character move and comparison:

- Make the character length for both operands constant, less than or equal to 256, and greater than 0.
- For character moves, make the character length of the target operands less than or equal to the source operand. For character comparison, make the character length for both operands the same.

In the following example, a more efficient code sequence is generated for the first three statements (1 through 3). A less efficient code sequence is generated for the last three statements (4 through 6):

```

      CHARACTER*400 C1,C2
      CHARACTER*100 C3(5),C4,C5(10)
      :
1     C4 = C5(I)
2     C3(J) = C2
3     IF(C2(300:305).EQ.C3(J)(50:55))PRINT*, 'MATCH'

4     C1 = C2
5     C2 = C3(J)
6     IF(C2(I:I+5).NE.C3(J)(J:J+5))PRINT*, 'NOT MATCH'
```

You can also use CHARLEN to specify the actual size for compiler-generated temporary variables. However, if you use CHARLEN to specify the actual size, it will directly affect the total size of your program. Try to eliminate dependences on the value of CHARLEN in your program, because as CHARLEN changes in your program, the size of your program also changes.

If, for example, the length of a dummy character argument is specified as (*), statements involving this dummy argument can cause CHARLEN bytes of storage to be allocated for intermediate results, which would increase the size of your subprogram. In the example below, space is allocated for the value returned by CHFT, the length of which can be as large as CHARLEN:

```

SUBROUTINE SUB(CH)
CHARACTER *(*) CH, CHFT, CC*80
CH=CHFT(CC)
```

Variables

- Use logical variables of length 4 because they can be handled most efficiently.
- For more efficient code, use integer variables of length 4 for DO loop indexes.
- Use integer variables of length 8 only when the magnitude of your data requires their use. Because there are no native machine instructions for 8-byte integer arithmetic, all operations are done with software simulation. (Their use also prevents vectorization.)
- Certain variables cannot always be optimized:
 - Control variables for direct access input/output data sets cannot be optimized.
 - Variables in input/output statements and in CALL statement argument lists cannot be fully optimized in the loops that contain the statements.
 - EQUIVALENCE variables cannot be optimized.

- Variables in common blocks cannot be optimized across subroutine calls. All variables which are defined in ISPF Dialog Management Services must be specified in common blocks.
- Variables received as dummy arguments are difficult to optimize when they are used inside a loop that references other functions and routines. Assign frequently referenced scalar dummy arguments to local variables. Remember that changing a local variable does not change the argument.

Do not use DO loop indexes in any of the above ways.

- Each common block requires distinct addressing. This is the basis for the following recommendations. For a description and examples of using common blocks with parallel processing, see “Sharing Data in Common Blocks within a Parallel Task” on page 308.

1. Minimize the number of common blocks. Group concurrently referenced variables into the same common block. For example:

Three Addresses Required	One Address Required
COMMON /X/ A	COMMON /Q/ A,B,C
COMMON /Y/ B	A=B+C
COMMON /Z/ C	
A=B+C	

2. Place scalar variables before arrays in a given common block. For example:

Two Addresses Required	One Address Required
COMMON /Z/ X(5000),Y	COMMON /Z/ Y, X(5000)
X(1)=Y	X(1)=Y

3. Place small arrays before large ones. The subsequent larger arrays will probably need separate addressing. For example:

```
COMMON /A/ B(5000), C(500), D(50)
```

would be better written as:

```
COMMON /A/ D(50), C(500), B(5000)
```

4. Assign frequently referenced scalar variables in a common block to a local variable. You should be sure to assign the value back to the common variable at the end of processing. For example:

```
COMMON /Z/ X
X2=X*2
:
X3=X*3
:
X=X+X2+X3
:
```

would be better written as:

```
COMMON /Z/ X
TEMPX=X
X2=TEMPX*2
:
X3=TEMPX*3
:
TEMPX=TEMPX+X2+X3
:
X=TEMPX
```

- When you are accumulating intermediate summations, keeping the result in a scalar variable rather than in an array is usually beneficial for scalar code. For example:

```
DO I=1,10
  B(I)=0
  DO J=1,100
    IF (A(I,J) .NE. 0) THEN
      B(I)=B(I)+A(I,J)
    END IF
  END DO
END DO
```

would be better written as:

```
DO I=1,10
  TEMP=0
  DO J=1,100
    IF (A(I,J) .NE. 0) THEN
      TEMP=TEMP+A(I,J)
    END IF
  END DO
  B(I)=TEMP
END DO
```

Subroutine Arguments

Pass subroutine arguments in a common block rather than as parameters; you'll avoid the overhead of processing parameter lists. You must evaluate the effect of placing parameters into common for both the calling and the called routine.

Entry into a subprogram associates actual arguments with the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the whole subprogram become associated with actual arguments. New values will not be transmitted for arguments not listed in the ENTRY statement.

The only way to guarantee, under all optimization levels, that you'll get the current value of an argument is to have the argument listed on the ENTRY statement through which you invoke the subprogram.

Constant Operands

Define constant operands as local variables. (Operands in common or in a parameter list can change, and cannot be optimized as fully.)

Arrays

- Expand some smaller arrays to match the dimensions of the arrays they interact with. If arrays in a subprogram, block of code, loop, or nest of loops have the same shape, subscripting and indexing are more efficient.
- Subscripting of adjustable dimensioned arrays requires additional indexing computations. Using an adjustable dimensioned array as a subroutine parameter, requires an additional calculation on each entrance into the subroutine. To lessen the amount of extra processing, use the following techniques:
 - If indexing can be varied in the low-order dimensions, make the adjustable dimensions of an array the high-order dimensions. This reduces the number of computations needed for indexing the array, as shown:

Computation not Required

```
SUBROUTINE EXEC(Z,N)
REAL*8 Z(9,N)
Z(I,5)=A
```

Computation (I*N) Required

```
SUBROUTINE EXEC(Z,N)
REAL*8 Z(N,9)
Z(5,I)=A
```

- If your array boundary dimensions are correct, (that is, lower bounds never exceed corresponding upper bounds), perform all adjustable array calculations inline (rather than by a library call). To select this use the IL(DIM) compile-time option. Such inline code usually causes the program to run faster.

In the example:

```
SUBROUTINE SUB(A,I,J,K,L)
  DIMENSION A(I:J, K:L)
```

the relative values of I and J, and K and L are not checked. To request error checking, use the IL(NODIM) option.

Expressions

- If components of an expression are duplicate expressions, code them either: at the left end of the expression, or within parentheses. For example:

Duplicates Recognized

```
A=B*(X*Y*Z)
C=D*(X*Y*Z)
```

```
E=X+Y+F
G=X+Y+H
S=T+(X+Y)
```

No Duplicates Recognized

```
A=B*X*Y*Z
C=D*X*Y*Z
```

```
E=F+X+Y
G=H+X+Y
S=X+Y+T
```

will enable more recognition of duplicate expressions.

- When components of an expression in a loop are constant, code the expressions either: at the left end of the expression, or within parentheses.

If C, D, and E are constant and V, W, and X are variable, the following examples show the difference in evaluation:

Constant Expressions Recognized

```
V*W*X*(C*D*E)
C+D+E+V+W+X
```

Constant Expressions Not Recognized

```
V*W*X*C*D*E
V+W+X+C+D+E
```


Critical Loops

If your program contains a short, heavily-referenced DO loop, consider removing the loop and expanding the code inline in the program. Each loop iteration runs faster.

Scalar Computations in Loops

Factor calculations involving constant scalar operands out of loops, when possible. You can save run time by factoring, as the following example shows:

Using Factoring

```
SUM=0.0
DO 1 I=1,9
  SUM=SUM+ARR(I)
1 CONTINUE
SUM=SUM*FAC
```

Not Using Factoring

```
SUM=0.0
DO 1 I=1,9
  SUM=SUM+FAC*ARR(I)
1 CONTINUE
```

In many programs, you can factor extensively.

Conversions

- Avoid mixing integers and floating-point numbers in expressions because the required conversions use considerable run time. For example:

No Conversions Needed

```
X=1.0
DO 1 I=1,9
  A(I)=A(I)*X
  X=X+1.0
1 CONTINUE
```

Multiple Conversions Needed

```
DO 1 I=1,9
  A(I)=A(I)*I
1 CONTINUE
```

When you must use mixed-mode arithmetic, code the fixed-point and floating-point arithmetic in separate computations as much as possible.

- Conversions from single to double precision use considerable run time.

Arithmetic Constructions

In subtraction operations, if only the negative is required, change the subtraction operations into additions, as follows:

Efficient

```
Z=-2.0
DO 1 I=1,9
  A(I)=A(I)+Z*B(I)
1 CONTINUE
```

Inefficient

```
DO 1 I=1,9
  A(I)=A(I)-2.0*B(I)
1 CONTINUE
```

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.

IF Statements

Use a block or logical IF statement rather than an arithmetic IF statement. If you must use an arithmetic IF statement, try to make the next statement one of the branch destinations.

In block or logical IF statements, if your tests involve a series of AND/OR operators, try to:

- Put the simplest tested conditions in the leftmost positions. Put complex conditions (such as tests involving function references) in the rightmost positions.
- Put tests most likely to be decisive in the leftmost positions.

Pointers

The use of pointers offers great potential for hidden aliasing through association. It is your responsibility to ensure that all such aliasing is valid; no analysis is done by the compiler to recognize this aliasing during optimizations.

Pointer Association: Association exists when the same data item can be identified with different names in the same program unit, or with the same name or different names in different program units of the same executable program.

Association through pointers can occur in the following situations:

- Multiple pointees are declared with the same pointer.

```
Pointer (P,A), (P,B)    ! A and B are associated
```
- A pointer is assigned the address of a variable

```
Pointer (P,A)
P = LOC(B)              ! A and B become associated
```
- Multiple pointers are assigned the address of the same variable. This can be particularly perilous. For example, setting A to a new value sets D. But a reference to B may not cause a "reaccess" of D, and will thus not get the updated value.

```
Pointer (P,A), (Q,B)
P = LOC(D)
Q = LOC(D)              ! A, B, and D become associated
B = 3.0
. . .
A = 2.0                  ! B and A may each be in a register
. . .
= B                      ! May use 3.0 ! Or 2.0 !
```

- A pointer variable that appears as a dummy argument is assigned the address of another dummy argument or entity in a common block.

```

Pointer (P,A)
.
.
.
Call Sub(P, B)
.
.
.
Subroutine Sub(P, X)
Pointer (P,Y)
P = LOC(X)                ! Caller variables A and B become associated

```

- A pointer variable and its associated pointee variable are passed as arguments to a subprogram

```

Pointer (P1, PTee1)
P1 = loc(A)
call sub(P1, PTee1)
.
.
.
Subroutine Sub(P1, B)
Real B
Pointer (P1, PTeeA)      ! PTeeA and B are associated

```

Debugging Optimized Programs

Debugging optimized programs presents special problems. Changes made by optimization can be confusing.

Use debugging techniques that rely on examining values in storage with caution. A common expression evaluation may have been deleted or moved. A variable may be in a register, not yet stored, when storage is examined or the abend dump occurs. Variables temporarily assigned to registers may not have been saved in a storage location at the time that an abend dump occurs.

Programs that appear to work properly when compiled with OPT(0) may fail when compiled at OPT(3). This is often caused by program variables that have not been initialized. If a program that worked at OPT(0) fails when compiled at OPT(1), OPT(2), or OPT(3), it is a good idea to look at the cross-reference listing; check for variables that are fetched but never set, and for program logic that allows a variable to be used before being set.

See *VS FORTRAN Version 2 Interactive Debug Guide and Reference* for more information on debugging optimized code.

Chapter 16. Using the Vector Feature

Techniques for Improving Vectorization	277
Statements and Constructs Preventing Vectorization	277
Array References	279
Loops	280
Program Logic	281
Storage	282
Loop Structure	283
Section Size	284
Complex Data	284
Enhanced Vector Facility	285
Vectorizable Mathematical Functions	285
Examples of Vectorization	286
Compound Instructions	286
Loop Selection	287
Loop Distribution	287
Scalar Expansion	288
IF Conversion	288
Statement Reordering	288
Reduction Operations	289
Intrinsic Functions	289
EQUIVALENCE Arrays	289
Considerations and Restrictions for Vectorization	290
Vector versus Scalar Summation	290
Vectorized Exit Branches	291
Version 2 Versus Version 1 FORTRAN Math Library Routines	291
Subscript Values and Array Bounds	291
Interaction with Static Debug Statements	292

The following sections discuss and give examples of how to improve the vectorization of your program, and describe some considerations and restrictions regarding the use of vectorization. See “Terminology for Automatic Vector and Parallel Processing” on page 258 for definitions of terminology used in the following discussion.

Techniques for Improving Vectorization

Most of the programming techniques used for writing optimized Fortran programs apply to vectorization. The following suggestions can make your program run faster in both scalar and vector mode.

The use of vector directives can also improve vectorization. For more information, refer to “Using Parallel and Vector Directives” on page 353.

Statements and Constructs Preventing Vectorization

The compiler analyzes DO and certain IF loops for vectorization; only the eight innermost levels of a nest are analyzed.

Avoid the following statements or constructs when coding your program. Any loops in which one of these statements or constructs occurs will be ineligible for vectorization.

Branches

Backward branches

Exit branches in any contained loop

Data/Variables

All CHARACTER data

INTEGER*1 and INTEGER*8 data

LOGICAL*1, LOGICAL*2, and LOGICAL*8 data

UNSIGNED*1 and BYTE data

An induction variable mentioned in an EQUIVALENCE statement

A loop index or iteration control expression other than INTEGER*4

A DO variable which is a control variable of a direct access I/O statement appearing anywhere else in the program unit

Functions

All external, nonintrinsic functions

Operations

Subroutine calls

Statements

Computed or assigned GO TO statements

ASSIGN, ENTRY, or PAUSE statements

I/O statements other than READ, WRITE and PRINT statements containing array elements and arithmetic scalars. Asynchronous I/O.

The following parallel language statements:

DOAFTER	PARALLEL CALL
DOBEFORE	PARALLEL DO
DOEVERY	PARALLEL SECTIONS
END SECTIONS	SCHEDULE
EXIT	SECTION
LOCAL	TERMINATE
ORIGINATE	WAIT FOR statements

Avoid the following statements or constructs when coding your program. Any statements in which these constructs occur will not be eligible for vectorization. Also, other statements that define or use the same data that is defined or used in an ineligible statement may also become ineligible for vectorization.⁵

Branches

Exit branches in the innermost loop without specification of optimization level 3

Exit branches in the innermost loop with a condition that determines whether the branch is taken that is dependent on any variables (other than induction variables) that are modified in the loop

⁵ Some cases in which you have scalar expansion or sum reduction involving complex data might be made vectorizable by applying slight modifications to your program.

Conditions

Interchange-preventing dependences (other than innermost levels)

Unbreakable recurrences⁶

Data/Expressions/Variables

Noninductive subscripts to an INTEGER*2 array

Noninductive subscripts governed by an IF-statement

Relational expressions that need to be stored (for example, $L=A.GE.B$)

An induction variable modifying inner loop parameters or inner auxiliary induction variables

Functions

Intrinsic functions when NOINTRINSIC is specified

Intrinsic functions using REAL*16 or COMPLEX*32

Some occurrences of intrinsic in-line MIN and MAX functions using INTEGER

All intrinsic in-line functions from the following families: DIM, MOD, SIGN, NINT, ANINT, or BTEST

Operations⁷

LOGICAL*1, LOGICAL*2, and LOGICAL*8 fetches or stores

REAL*16 or COMPLEX*32 operations

INTEGER*2 fetches or stores governed by an IF-statement

INTEGER*1 and INTEGER*8 fetches or stores

UNSIGNED*1 or BYTE fetches or stores

Reduction operations involving complex variables or arrays

Statements

Simple I/O statements.

Array References

- When the same array is both fetched and stored or just stored more than once in a loop, make sure that subscripts are simple functions of loop induction variables. To vectorize programs, it is important that the compiler be able to analyze the values taken on by subscripts in an array reference. Even a trivial change, such as adding a variable to a loop index, can make the loop impossible to vectorize.
- When writing programs to be run on a vector processor, consider the effects of the following indirect subscripts (subscripted subscripts):

DO 20 N=1,10	DO 20 N=1,10
A(M(N))=A(M(N))+1	A(M(N))=B(M(N))+1
20 CONTINUE	20 CONTINUE

⁶ An unbreakable recurrence is a group of statements within a loop that cannot run in vector mode because of the way that data is defined and used. If an IGNORE RECRDEPS directive is used, the recurrence may involve dependences caused by scalar variables, IF statements, or array variables with dependences that cannot be IGNOREd. If an IGNORE RECRDEPS directive is not used, the recurrence may also involve array variables for which dependences are assumed, because of insufficient information for analysis.

⁷ An example of a fetch of A is $X = A$; an example of a store of A is $A = X$.

The program on the left cannot be vectorized because two elements of array M may have the same value. The program on the right can be vectorized because, although two elements may have the same value, none of the arrays subscripted by these elements is both fetched and stored in the loop; they are only fetched or only stored.

- Provided that subscript calculations are written consistently, it is better to write them directly as subscripts than indirectly through temporaries. The code on the left may be more efficient on a vector processor.

```
A(I+5,J-3,K*2)          I5=I+5
                        J3=J-3
                        K2=K*2
                        A(I5,J3,K2)
```

This is especially true when the variables in the subscript dimensions (I, J, and K above) are loop indexes.

- If you use variables mentioned in EQUIVALENCE statements make sure they are efficiently aligned.
- Avoid using variables that are not induction variables in subscript expressions.

Loops

- The compiler analyzes only DO loops and certain IF loops for vectorization. However, the compiler can recognize some auxiliary induction variables in addition to the loop control variable. Vectorization of a loop is prevented when it contains an induction variable:
 - that modifies inner loop parameters or auxiliary induction variables
 - that is mentioned in EQUIVALENCE statements
- Some loops containing backward branches or branches out of loops will vectorize. Loops containing branches around inner loops or branches caused by computed or assigned GOTO statements will not vectorize. Other types of conditional or unconditional forward branches may be used in the loop.
- Vectorization requires preservation of the meaning (semantics) of the original loop. If a value computed on one iteration of a loop is needed on a later iteration, vectorization may not be possible. The example below illustrates the problem of overwriting an operand on iteration J to be used on iteration J+1.

```
REAL*4 A(50)
DO 20 J = 1 , 49
    A(J+1) = A(J)
20 CONTINUE
```

In the previous example, when processed serially (in scalar hardware), element A(1) is copied through all subsequent elements of array A. If the loop were vectorized, the elements on the right, A(1) through A(49), are fetched before being stored in locations A(2) through A(50), producing a shift of the array by one storage unit. Because the semantics of the original loop are not preserved, the loop would not be vectorized.

- Unrolled loops are often seen in programs optimized for scalar processors. The code on the left, in the following example, is much more efficient on a vector processor.


```

DO 1 N=1,128,4
  A(N+0)=B(N+0)*C(N+0)+D(N+0)
  A(N+1)=B(N+1)*C(N+1)+D(N+1)
  A(N+2)=B(N+2)*C(N+2)+D(N+2)
  A(N+3)=B(N+3)*C(N+3)+D(N+3)
1 CONTINUE

```

When a short loop is heavily referenced in a program, it may be worth your effort to unroll the loop (expand the code in-line), assuming it is too short for efficient vectorization.

- Avoid using loops that have an iteration count that cannot be determined at compile time. When the compiler cannot determine the number of iterations that a loop makes, it cannot accurately judge the relative costs of vector or scalar processing.

If the compiler cannot determine the iteration count, and the ASSUME COUNT directive (discussed on page 356) is not specified, the compiler attempts to estimate the count based on the dimensions of the arrays in the loop. If the array dimensions are all unknown, a fixed default value is used. For example:

```

REAL*4 A(20,10,*)
REAL*4 B(50,500,200)

...

DO 100 I=1,N1
DO 100 J=1,N2
DO 100 K=1,N3
  A(I,J,K) = 0.0
  B(I,J*50,K*M) = 0.0
100 CONTINUE

```

<== assumed count is 20
 <== assumed count is 10
 <== assumed count is 65 (the default)

If the number of elements of an array is much larger than the number of iterations of the loop in which it is referenced, then it is best to specify the ASSUME COUNT directive to make effective use of vector cost analysis.

- A program containing a very large DO loop takes a significantly longer time to compile if you specify the VECTOR compile-time option. You can reduce the time such a program takes to compile by using the PREFER VECTOR directive, or, if appropriate, dividing the large DO loop into smaller DO loops. For information about coding the PREFER directive, see “PREFER Directive” on page 364.

Program Logic

Revisions to the original logic can be made to allow vectorization of the program. This example illustrates a simple revision:

```

DO 1 I=ILOW,IHIGH
  C(I)=A(I)+B(I)
  C(I+INC)=A(I)+B(I+INC)
1 CONTINUE

```

Because the compiler does not know the values of the variables ILOW, IHIGH, or INC, it cannot determine whether the loop involves computations overlapping the range of subscripts of the variable C.

Therefore, the loop will not be vectorized. If the value of INC is less than 1 or greater than IHIGH, you might split the loop into two loops that can be vectorized:

```
DO 1 I=ILOW,IHIGH
  C(I)=A(I)+B(I)
1 CONTINUE
DO 2 I=ILOW,IHIGH
  C(I+INC)=A(I)+B(I+INC)
2 CONTINUE
```

Storage

Use Virtual Memory: While floating-point arithmetic calculations are performed faster by vector processors, the time used by I/O operations may not change and can account for an increasing percentage of the total time required by an application.

The vector facility and its additional extended memory permit large amounts of data to be maintained in central electronic storage, rather than on paging devices. Take advantage of this feature by using large arrays and letting the operating system do the management.

Use the Smallest Stride: The sequence in which the various elements of an array are referenced within a nest of loops can have significant impact on the performance of that nest. (This sequence is sometimes referred to as the memory reference pattern.)

Optimizing the memory reference pattern for a nest of loops can be extremely complicated. In general, it is a good idea to minimize the stride of the most rapidly varying loop in a nest. For scalar code, the most rapidly varying loop is always the innermost loop. For vector code, it is the vectorized loop that varies most rapidly, regardless of its relative position.

The compiler takes the memory reference pattern into account in deciding which loop to vectorize, and will usually choose the loop that results in the greatest performance benefit. Sometimes, however, this is not possible, either because the compiler is missing some important information (such as the number of iterations of a loop), or because the layout of the data or the structure of the loops does not lend itself to optimal vectorization. For example, examine the following code:

```
REAL*8 A(10,10,1000),B(10,10,1000)
DO 5 I=1,1000
DO 5 J=1,10
DO 5 K=1,10
  A(K,J,I) = B(K,J,I)
5 CONTINUE
```

The outer loop is the best candidate for vectorization because it would result in the longest vector length. However, due to the data layout, this would result in a stride of 100 elements, thus reducing (or possibly negating) the benefits of vectorization. If you restructure the data, as in the following example, better performance is likely to result.

```

      REAL*8 A(1000,10,10),B(1000,10,10)
      DO 5 I=1,1000
      DO 5 J=1,10
      DO 5 K=1,10
        A(I,J,K) = B(I,J,K)
5      CONTINUE

```

Note that even though the declarations and the subscript expressions have been modified, the structure of the nest of loops has not been changed. Remember that after vectorization, the vectorized loop will be the loop that iterates most rapidly. If the outer loop is vectorized, then the largest possible vector length along with the shortest possible stride will result.

In some cases, nonunit stride vectorization is unavoidable. Depending on the overall memory usage pattern, this may be acceptable. However, it should be noted that certain large strides should always be avoided. Strides that are multiples of large powers of 2 usually lead to particularly poor performance. (For single precision data, strides that are a multiple of 128 or any larger power of 2 will probably result in very poor performance. For double precision data, this degradation will probably be seen for strides that are a multiple of 64.)

Loop Structure

To minimize the stride, vectorize the loop corresponding to the leftmost subscripts of the majority of the arrays referenced in the nest of loops. The vectorized loop need not be the innermost loop; in fact there are several advantages to vectorizing an *outer* loop. These include:

- Reducing the overhead of initializing the vector loop (because an outer loop is initialized less frequently than an inner loop)
- Eliminating vector storage references from an inner loop.

You do not have to do anything special to vectorize an outer loop. The compiler selects the best loop to vectorize. Some examples may help to illustrate these points. Try compiling the following programs with the VECTOR and LIST options and then look at the generated code:

- Reducing vector overhead:


```

      REAL*8 A(200,200),B(200,200)
      DO 10 I=1,200
      DO 10 J=1,200
        A(I,J)=B(I,J)
10 CONTINUE

```

In the above simple program to copy one matrix to another, the outer loop is vectorized. There are only two vector instructions, LOAD and STORE, in the inner, scalar loop. The vector instructions that control sectioning are either in the outer, sectioning loop or are not inside a loop at all. Another benefit of outer loop vectorization is that the iteration count of the vectorized loop is reduced by a factor of the section size. This means that the number of times that any nested scalar loops need to be initialized is reduced by that same factor.

- Eliminating vector storage references from an inner loop:

```
REAL*8 A(200,200),B(200)
DO 10 I=1,200
  DO 10 J=1,200
    A(I,J)=B(I) * A(I,J)
  10 CONTINUE
```

In the above program, which multiplies the columns of a matrix with a vector, the term $B(I)$ is invariant in the inner loop. It is thus computed in a vector register in the outer loop and then used, without referencing storage, in the inner loop.

Section Size

The `SIZE` suboption specifies the section size used to perform vector operations when the compiled program is processed. Section size determines the number of vector elements that the program can operate on at one time. The size is a power of two and is machine-specific.

There are three parameters for the `SIZE` suboption: `ANY`, `LOCAL`, and n . Using a specific section size—`SIZE(LOCAL)` and `SIZE(n)`—can generate more efficient object code than the variable section size specified by `SIZE(ANY)`. However, you may have to recompile the routine if you want to move it to another computer. See Chapter 2, “Compiling Your Program” on page 7 for a full discussion of the `SIZE` suboption.

A `DO` loop can be analyzed for parallel and vector code generation. In this case the number of iterations of the `DO` loop given to a virtual processor as a “chunk” to process is usually a multiple of the vector section size. Thus each virtual processor runs the optimal vector code for its iterations.

Complex Data

`COMPLEX*8` data is organized as two related single-precision real (`REAL*4`) values stored consecutively; that is, the real part is stored in the first word of the complex value, and the imaginary part is stored in the second word.

Because these words are consecutive in storage, the `CMPLXOPT` suboption is provided to instruct the vectorization process that the two words can be loaded into vector registers from storage, and placed into storage from vector registers, as if the data were `REAL*8`. Using a `REAL*8` load/store operation can increase the efficiency of the vector execution, since `REAL*8` data is accessed as stride-1 data by the hardware, whereas using 2 separate and independent accesses to the real and imaginary parts of the `COMPLEX*8` item causes each to be accessed as if it were stride-2 data. Stride-2 data accesses is significantly slower than stride-1 accesses; accessing `COMPLEX*8` as stride-1 data will improve the execution efficiency of the program.

Such data must be aligned on a double-word boundary. The `NOCMPLXOPT` suboption prohibits the optimization of accessing `COMPLEX*8` data using `REAL*8` instructions; this allows for `COMPLEX*8` not stored on the required double-word boundary to continue to be processed by VS FORTRAN.

Enhanced Vector Facility

The MODEL suboption allows you to identify the level of the ES/9000 vector facility on which the compiler program will execute. Depending on the level of the vector facility on which execution will occur, the compiler can generate inline code to use the enhanced features of that facility.

When a VS FORTRAN program is compiled for a specific level of the vector facility, the VS FORTRAN run-time vector support will ensure that execution occurs only on a vector processor capable of supporting the features of that facility.

For MODEL(VF2), the following functions are performed:

1. Generation of vector square root instructions in the compiled code if any of the SQRT family of intrinsic functions are used in vector loops.
2. Generation of scalar square root instructions in the compiled code if any of the SQRT family of intrinsic functions are used in scalar loops.

When MODEL(ANY) is in effect, the compiler will not perform any of the functions listed for MODEL(VF2). If the program contains a vector or scalar loop that uses one of the SQRT family of intrinsic functions, these functions will be performed by the appropriate run-time library routine. If the program contains short precision arithmetic calculations that are not converted to double precision values, these calculations will be performed by separate multiply and add/subtract operations.

Vectorizable Mathematical Functions

Most VS FORTRAN Version 2 intrinsic functions and mathematical operations can be vectorized. The INTRINSIC | NOINTRINSIC suboption of the VECTOR compile-time option allows you to specify whether out-of-line intrinsic functions are to be vectorized. Note that the results returned by the vector intrinsic functions are identical to those of the corresponding scalar intrinsic functions in the VS FORTRAN Version 2 library, but may be different from those obtained using VS FORTRAN Version 1. For more information, see the description of the INTRINSIC | NOINTRINSIC suboption on page 37.

Some of the intrinsic functions and mathematical operations take advantage of the vector hardware (see Figure 80 on page 286), while others are evaluated using scalar code but accept vector arguments and return vector results (see Figure 81 on page 286).

Figure 80. Intrinsic Functions and Mathematical Operations that Use Vector Hardware

Out-of-Line Intrinsic Functions:	SQRT	DLOG	DTAN	CDABS
	DSQRT	SIN	DCOTAN	ALOG10
	EXP(1)	DSIN	ATAN	DLOG10
	DEXP(1)	COS	DATAN	ATAN2
	ALOG	DCOS	CABS	DATAN2
In-Line Intrinsic Functions:	HFIX	INT	IDNINT	SNGL
	IABS	IOR	COMPLX	DBLE
	ABS	ISHFT	DCMPLX	AMAX1
	DABS	NOT	CONJG	DMAX1
	IAND	AIMAG	DCONJG	MAX1
	IBCLR	DIMAG	FLOAT	AMIN1
	IBSET	AINT	DFLOAT	DMIN1
	IDENT	DINT	DPROD	MIN1
	IEOR	ANINT	REAL	SIGN
	IFIX	DNINT	DREAL	XOR
		NINT		
Mathematical Operations:	REAL*4 raised to a REAL*4 power			
	REAL*8 raised to a REAL*8 power			
	COMPLEX*8 divide			
	COMPLEX*16 divide			

Note:

1. When the argument passed to EXP and DEXP is less than or equal to -160.0E0 and -138.0D0 respectively, the routine runs in scalar mode and no vector instructions are executed.

Figure 81. Intrinsic Functions and Mathematical Operations Evaluated Using Scalar Code

Out-of-Line Intrinsic Functions:	ACOS	ERF	DTANH	CDLOG
	DACOS	DERF	TAN	CSQRT
	ASIN	ERFC	CCOS	CDSQRT
	DASIN	DERFC	CDCOS	IBCLR
	COTAN	GAMMA	CSIN	IBSET
	COSH	DGAMMA	CDSIN	ISHFT
	DCOSH	ALGAMMA	CEXP	IBITS
	SINH	DLGAMA	CDEXP	LSHIFT
	DSINH	TANH	CLOG	RSHIFT
				ISHFTC
Mathematical Operations:	INTEGER raised to an INTEGER power			
	REAL*4 raised to an INTEGER power			
	REAL*8 raised to an INTEGER power			
	COMPLEX*8 raised to an INTEGER power			
	COMPLEX*16 raised to an INTEGER power			
	COMPLEX*8 raised to a COMPLEX*8 power			
	COMPLEX*16 raised to a COMPLEX*16 power			

Examples of Vectorization

The following examples show how the compiler vectorizes typical sequences of Fortran statements. The associated compiler reports were produced using the REPORT(LIST) suboption.

Compound Instructions

Instructions such as MULTIPLY AND ADD are important, because only by using them can the full potential of the vector facility be realized. In this program, a form of matrix multiplication, the compound instruction MULTIPLY AND ADD is the only vector instruction in the inner loop. The program achieves a floating-point operation rate approaching the limit of the hardware.

When the operands are single precision, these compound instructions produce double precision results and use double precision values for the intermediate product, the accuracy of the computed result will differ from the result produced by a separate multiply instruction that is followed by an add or subtract instruction. While the numerical accuracy of the result will be improved, and the calculation will be performed faster than would occur with two separate vector instructions, a difference will occur from the scalar single precision calculation of the same result, or from the vector calculation using separate vector single precision instructions.

The SPRECOPT suboption is provided to allow the programmer to choose that this operation should occur, realizing that the significance of the result might be different.

```

0002                                REAL*8 A(100,100),B(100,100),C(100,100)

0003 VECT +----- DO 3 I=1,100
0004 SCAL | +----- DO 2 J=1,100
0005      | | C(I,J) = 0.0
0006 SCAL | | +----- DO 1 K=1,100
0007      | | | C(I,J)=C(I,J)+B(K,J)*A(I,K)
          | | | ...
          | | |
0011      | | | END

```

Loop Selection

When an inner loop cannot be vectorized, it is possible that an outer loop can. In the following example the inner loop cannot be vectorized because the innermost subscript carries a dependence. As you can see, the I-loop is run in vector hardware.

```

0001                                DIMENSION X(100,100), Z(100,100)
0002 VECT +----- DO 110 I = 1,100
0003 SCAL | +----- DO 100 J = 1,99
0004      | | Z(I,J+1) = Z(I,J) * X(I,J)
          | | ...
0007      | | END

```

Loop Distribution

Statements occurring within a single loop in the original program may end up in separate loops after vectorization.

```

      REAL A(200), B(200)
      DO 20 I = 2, 100, 2
        A(I) = A(I) + 2.
        B(I+2) = B(I) + 2.
20    CONTINUE
      END

```

The following compiler report shows how vectorization produces the separation:

```

0001                                REAL A(200), B(200)
0002 VECT +----- DO 20 I = 2, 100, 2
0003      | A(I) = A(I) + 2.

0002 SCAL +----- DO 20 I = 2, 100, 2
0004      | B(I+2) = B(I) + 2.
0006      | END

```

Because there is no dependence between the two assignment statements, they can be placed into separate loops. The first statement can be vectorized; the second statement cannot be vectorized because of recurrence.

Scalar Expansion

The following example demonstrates scalar expansion. In the original source code, the scalar variable T is used to hold the value of an element of B and assign it to the corresponding element of array A. In the transformed code, the scalar variable T is expanded into a temporary vector (appearing only in a vector register). As you can see, the entire loop has been vectorized. Scalar expansion cannot occur if:

It appears in an EQUIVALENCE grouping
It is part of a recurrence.

In some cases, scalar expansion cannot occur if:

The scalar variable appears at different loop levels of a nest
Any definition within the loop is conditional.

```

0001                                DIMENSION A(100),B(100)
0002 VECT +-----                DO 300 I = 1,100
0003     |                               T   = B(I)
0004     |                               B(I) = A(I)
0005     |_____                       A(I) = T
                                ...

```

IF Conversion

IF-conversion converts control dependences into data dependences, allowing analysis for possible vectorization. Following is an example of a compiler report with IF-conversion:

```

0001      REAL A(128,128), B(128,128), C(128,128)
0002 SCAL  +----- DO 10 K = 1,128
0003 SCAL  | +----- DO 10 J = 1,128
0004 VECT  | | +----- DO 10 I = 1,128
0005      | |         IF(I.EQ.J) C(I,J) = 0.
0007      | |         IF(I.GT.J) C(I,J) = C(I,J) + A(I,K) * B(K,J)
      | |         _____
      | |         _____
      | |         _____
0010      | _____ END

```

Statement Reordering

The statement processing order may be changed to permit vectorization.

```

        DIMENSION A(100),B(100),C(100)
        DO 500 I = 2,100
            A(I) = B(I-1) * 3.0
            B(I) = C(I) * 3.0
500    CONTINUE
        END

```

As a result of vectorization, the following compiler report is produced:

```

0001          DIMENSION A(100),B(100),C(100)
0002 VECT +----- DO 500 I = 2,100
0004      |          B(I) = C(I) * 3.0
0003      |          A(I) = B(I-1) * 3.0
0006          END

```


Reordering is possible because all of the values of $B(I)$ are stored before they are required in the second statement of the loop. Statement reordering does not affect the results of the program.

Reduction Operations

Some statements of the form: $S = \dots + S + \dots$ show an accumulation or *reduction*. Reduction includes such common operations as:

- Sum of vector elements
- Sum of squares
- Vector inner product.

The translation of reduction operations from scalar to vector code may produce different results, as explained under “Vector versus Scalar Summation” on page 290.

An example of reduction operation recognition is given below. Vector code will be generated for the loop.

```

0001                DIMENSION A(100), B(100)
0002                SUMEL  =  0.0
0003                SUMSQ  =  0.0
0004                SUMPR  =  0.0
0005 VECT  +----- DO 500 I = 1,N
0006      |          SUMEL  =  SUMEL + A(I)
0007      |          SUMSQ  =  SUMSQ + A(I) * A(I)
0008      |          SUMPR  =  SUMPR + A(I) * B(I)
0010      |          END

```

To prevent vectorization of reduction operations, you can specify VECTOR(NOREDUCTION).

Note: Vectorization of reduction operations is not performed on complex variables or arrays.

Intrinsic Functions

The following examples show vectorizable loops containing references to the MIN and SIN intrinsic functions:

```

0001                REAL A(500),B(500),C(500)
0002 VECT  +----- DO 10 I=1,500
0003      |          B(I) = MIN(A(I),B(I),C(I))

```

```

0001                DIMENSION A(100), B(100), C(100)
0002 VECT  +----- DO 500 I = 1,N
0003      |          A(I) = B(I) * SIN(C(I))
0005      |          END

```

EQUIVALENCE Arrays

The following example shows a vectorizable loop that has a loop increment of two.

Vector Considerations

```
0001          DIMENSION A(200),B(200)
0002          EQUIVALENCE (A(1),B(2))
0003 VECT +----- DO 10 I = 1,190,2
0004      |_____   A(I) = B(I) * 2.1
0006          WRITE(6,*) A
0007          STOP
0008          END
```

The following example shows a vectorizable loop with subscript expressions. Note that a value stored into variable A is not used later by variable B.

```
0001          DIMENSION A(200),B(200)
0002          EQUIVALENCE (A(1),B(1))
0003 VECT +----- DO 10 I = 1,100,1
0004      |_____   A(I) = B(I+1) * 2.1
0005          RETURN
0006          END
```

The following example shows a vectorizable loop with a range too short to reference the storage locations where variables A and B overlap.

```
0001          DIMENSION A(100),B(1000)
0002          EQUIVALENCE (A(1),B(101))
0003 VECT +----- DO 10 I = 1,100,1
0004      |_____   A(I) = B(I) * 2.1
0005          RETURN
0006          END
```

Considerations and Restrictions for Vectorization

Results from vectorized programs may differ from those produced by programs compiled and run using VS FORTRAN (Version 1 or Version 2) with VECTOR(NOREDUCTION) or (NOVECTOR) specified.

Vector versus Scalar Summation

Summing on the scalar hardware is performed sequentially; each number is added in turn. However, summing is done differently on the vector hardware: every n -th element is added (where n is dependent on the vector hardware). These partial sums are added to form the total.

Differences can occur because floating-point addition is not associative; that is, the sum depends upon the order of addition. The floating-point numbers produce one result if added sequentially on scalar hardware. When added using vector accumulate instructions, the floating-point numbers produce a different result. The two sums, however, are algebraically equivalent even though they are not computationally equivalent.

The result of summation is also affected by the loop selected for vectorization. When the sum is driven by several loops, as in the example:

```
      DO 1 K = 1,N
      DO 1 J = 1,N
      DO 1 I = 1,N
        S = S + A(K,J,I)
1     CONTINUE
```

different sums are possible in scalar. All are algebraically equivalent.

Vectorized Exit Branches

When a loop with an exit branch, such as STOP or RETURN, is vectorized, a run-time exception may be raised that would not otherwise occur in scalar mode. For example:

```

      DO 500 I = 1,50
        IF (A(I)/B(I) .EQ. 1.0) STOP
        C(I) = A(I) * B(I)
500   CONTINUE

```

In this case, a divide exception would be raised if B(I) is ever zero. If this is true for some value of I greater than the value at which the loop termination occurs, then the exception will not occur in the scalar loop, but may occur in the vector loop. This may happen, for example, if A(10) equals B(10) and B(11) equals zero.

Version 2 Versus Version 1 FORTRAN Math Library Routines

Results generated by the VS FORTRAN Version 2 math library routines (VSF2FORT) may be different from the results generated by the VS FORTRAN Version 1 standard math routines (VSF2MATH) because the Version 2 routines have been revised to be more accurate.

For scalar out-of-line intrinsic function references in your program, you can choose which math library to use by accessing libraries in the desired order. If the intrinsic function references in your program are vectorized, however, the new VS FORTRAN Version 2 math library routines will always be used.

Therefore, if you wish to always use the old math routines for compatibility of results, you should specify the NOINTRINSIC suboption on the VECTOR option. The NOINTRINSIC suboption disables vectorization of out-of-line intrinsic functions. For more information, see the INTRINSIC | NOINTRINSIC suboption on page 37.

Subscript Values and Array Bounds

The VS FORTRAN Version 2 compiler assumes that subscripts remain inside array dimensions. LANTLRVL(77) requires that every array subscript be within its corresponding dimension declaration. LANTLRVL(66) only requires that the total subscript value be within the range of the array. In particular, addressing of the following type is permitted:

```

      REAL A(10,10)
      DO 1 I = 1,20
        A(I,2) = A(I,1)
1     CONTINUE

```

However, if vectorization is requested, the loop above will be vectorized with no check for array bounds being exceeded. Although the program may produce the correct results when run in scalar mode, it is likely that unexpected results will be obtained when the program is vectorized.

Interaction with Static Debug Statements

The use of static debug statements inhibits vectorization because static debug requires that the optimization level be 0.

Vectorization requires that the optimization level in effect be OPTIMIZE(2) or OPTIMIZE(3). If it is not, the compiler upgrades the optimization level to OPTIMIZE(3). However, certain Fortran statements, such as DEBUG or invalid Fortran statements, downgrade the optimization level to OPT(0) and no vectorization occurs.

Chapter 17. Using the Parallel Feature

Overview of Parallel Programs	294
What a Parallel Program Is	294
What a Parallel Program Does	294
Computational Independence and Controlled Access	296
Using Parallel Processing within a Parallel Task	297
Generating Parallel Code Automatically	297
Coding Parallel Constructs within a Parallel Task	298
Coding Parallel Loops	298
Coding Parallel Sections	301
Coding Private Pointers	303
Coding Private Arrays	304
Coding Dynamically Dimensioned Arrays	305
Coding Subroutines as Parallel Threads	306
Considerations for Subroutine and Function Invocations within a Parallel Thread	307
Sharing Data in Common Blocks within a Parallel Task	308
Using I/O within a Parallel Task	309
Using Parallel Processing between Parallel Tasks	310
Coding Parallel Tasks	310
Sharing Data in Common Blocks between Parallel Tasks	311
Using I/O between Parallel Tasks	315
Determining Default File Definitions for Originated Tasks	316
Using Unnamed Files in Originated Tasks	317
Using Named Files in Originated Tasks	317
Using Library Service Subroutines and Functions	317
Using Parallel Lock Services	318
Using Parallel Event Services	319
Considerations for Using Parallel Service Subroutines	322
Using the NPROCS Function	323
Using Subroutines to Control Processor Affinity	323
Considerations for Existing Service Subroutines	324
Using the Parallel Trace Facility	325
Parallel Trace and Analysis	326
Uses of Parallel Trace and Analysis	326
Interpretation of Parallel Trace and Analysis Data	327
Program Tracing Categories	328
The Trace File	330
Parallel Trace Facility Support Materials	330
Parallel Environment	331
Improving Parallel Processing	332
Using the Parallel Feature—Examples	333
Converting a Serial Program	333
Using Parallel Language Constructs	336
Improving Performance of Parallel and Vector Processing	337

This chapter describes and gives examples of how to use the parallel feature of VS FORTRAN Version 2. You can generate parallel code automatically; you can also code parallel language statements and use parallel library services to take full advantage of parallel processing.

Overview of Parallel Programs

The following sections describe the concept of parallel programs and define much of the terminology used throughout this chapter.

What a Parallel Program Is

VS FORTRAN Version 2 allows you to create parallel programs or direct the compiler to generate parallel code. A parallel program identifies threads that can run concurrently on multiple virtual processors. A *virtual processor* is a logical processor controlled by VS FORTRAN; on MVS, a virtual processor corresponds to an MVS task; on CMS, a virtual processor corresponds to a virtual CPU. Your operating system assigns virtual processors to run on hardware processors.

When you run a program in parallel on multiple virtual processors, you can reduce the elapsed time required to run the program, although the CPU time may increase. Parallel processing can be particularly beneficial for computationally intensive applications.

There are several ways for you to achieve parallel processing of a Fortran program. In this manual the term *parallel program* is used to describe a program that contains either or both of the following:

- Automatically-generated parallel code for DO loops. You use the PARALLEL (AUTOMATIC) compile-time option to request automatic parallel-code generation for DO loops. The results of such a program are computationally equivalent to that of the serial program. A *serial program* is a program in which the statements run on a single virtual processor and in the same order every time.

When you use the automatic parallel feature, VS FORTRAN is responsible for ensuring that a DO loop is valid to run in parallel.

- Explicit parallel language constructs. You use parallel language constructs in your program to exploit parallel processing for loops, subroutines, and sections of code. You can develop parallel algorithms for solving computationally intensive problems.

When you explicitly code parallel language constructs in your program, it is your responsibility to determine that the code is valid and suitable for parallel processing.

What a Parallel Program Does

A parallel program takes advantage of the multiprocessing capabilities of your operating system and hardware configuration and allows a single Fortran application to use more than one processor of that configuration simultaneously.

With the explicit parallel language constructs you can structure your program into a set of parallel tasks. A *parallel task*⁸ is a complete logical environment within which Fortran program units can run. Each parallel task has its own storage for local variables and common blocks as well as its own set of I/O unit numbers. The parallel language statements that manage originating, assigning work to, detecting

⁸ Do not confuse a parallel task with an MVS task or with a virtual processor.

completion of work in, and terminating parallel tasks are called *task management statements*. The task management statements are:

ORIGINATE statements
 TERMINATE statements
 SCHEDULE statements
 WAIT FOR statements.

A parallel program begins running in the environment of the parallel task called the *root task*. You can create additional parallel tasks with an ORIGINATE statement; these are referred to as *originated tasks*. Originated tasks can create other originated tasks, thus allowing you to create a hierarchical tasking structure, such as that shown in Figure 82.

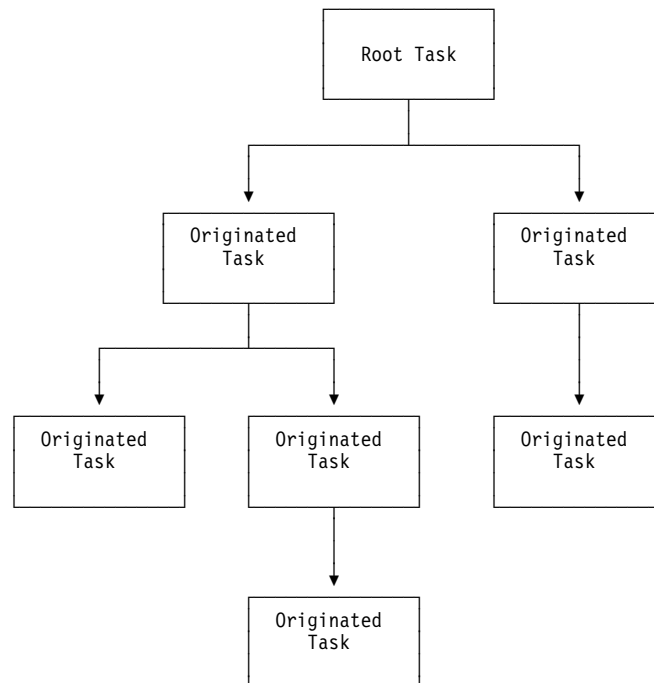


Figure 82. Example of a Hierarchical Parallel Task Structure

Within a parallel task, you assign parallel threads to that same parallel task, or assign them to a different parallel task using parallel task management statements. A *parallel thread* is a subroutine, an iteration(s) of a loop, or section(s) of code that can be run concurrently with other subroutines, loop iterations, or sections of code because it is computationally independent or accesses shared data in a controlled manner. A parallel task begins running with the primary parallel thread of that task. The *primary parallel thread* of any parallel task is the parallel thread that identifies, either directly or indirectly, all other parallel threads in that parallel task.

A parallel thread uses the logical environment of the parallel task in which it runs, and augment that environment with local variables of its own. Thus, certain variables can be shared within a parallel task while others are local to a parallel thread.

You use parallel language constructs to identify parallel threads that run within the parallel task environment in which they are identified.

The constructs you use to identify parallel threads are:

```
PARALLEL DO
PARALLEL SECTIONS
PARALLEL CALL.
```

VS FORTRAN identifies parallel threads when it automatically generates parallel code for DO loops.

When you run a parallel program, you use the PARALLEL run-time option to specify the maximum number of virtual processors to be used to run the program. If you specify PARALLEL(1) at run-time, your parallel program is run in the parallel environment on one virtual processor.

To aid in debugging your parallel program, you may want to compile your program with the PARALLEL compile-time option and then run it with the PARALLEL and DEBUG run-time options on a single virtual processor. This allows you to develop and debug your program in an environment in which the errors produced are reproducible. Once you debug your program, and it runs successfully on one virtual processor, then you may specify PARALLEL(n) and test the running of the program on multiple processors. Be aware, however, that even if your program runs with no errors on one virtual processor it may not run correctly on multiple virtual processors; some problems, such as those with timing, cannot be solved in debugging with one processor.

Note: You can only use interactive debug to debug the root task and Fortran code that is not parallel; you cannot debug parallel loops, parallel sections, subroutines called with PARALLEL CALL, originated tasks, or parallel threads with interactive debug.

Computational Independence and Controlled Access

Successful concurrent processing of parallel threads requires that they be either computationally independent or access shared data in a controlled manner. Parallel threads are *computationally independent* if they do not modify data that is being modified or referenced by another parallel thread that may be running concurrently.

At this point in your hardcopy book you will find a graphic example of some hypothetical data in an array subscripted by I, J, and K. Each of the three divisions of the box represents a section of the array that could be operated on independently of the other sections. The same parallel section could be scheduled three times, with each instance of the section processing one of the three sections of the array.

Your application may not have computational independence along the same subscript axis of K, as in this picture. The divisions might have been along one of the other subscript axes, I or J. Also, the computational independence in your application may not fall into neat, box-like divisions.

You can control access to shared data areas within a parallel program. Controlling access means that you ensure that only one parallel thread can access a shared data area at a time if that data area is being modified. Thus concurrently running parallel threads can modify a piece of data and communicate that to another parallel thread. You can use library *lock* and *event* services to control access to shared data areas.

Using Parallel Processing within a Parallel Task

Parallel threads run within parallel tasks. VS FORTRAN identifies parallel threads when it automatically generates parallel code for DO loops, and you identify parallel threads with the following parallel language constructs:

PARALLEL DO
PARALLEL SECTIONS
PARALLEL CALL.

The following sections discuss how you identify and use parallel threads within parallel tasks.

Generating Parallel Code Automatically

The easiest way to take advantage of parallel processing is to use the PARALLEL(AUTOMATIC) compile-time option. PARALLEL(AUTOMATIC) analyzes multiple levels of DO loops for loops that are eligible for parallel processing. If you have parallel constructs in your program and you specify the PARALLEL compile-time option, you should also specify the PARALLEL run-time option.

The PARALLEL and VECTOR compile-time options work together. A DO loop must meet some of the same eligibility requirements for parallel processing as for vector processing (for example, dependence testing and cost analysis). For a complete description of these eligibility requirements, see Chapter 16, “Using the Vector Feature” on page 277 and “Terminology for Automatic Vector and Parallel Processing” on page 258.

A DO loop containing any of the following is not eligible for automatic parallel code generation:

- Any loop-carried dependence
- A loop other than a DO loop
- Loops with induction variables mentioned in EQUIVALENCE statements
- An exit branch
- The following statements:

DOAFTER	PARALLEL CALL
DOBEFORE	PARALLEL DO
DOEVERY	PARALLEL
	SECTIONS
END	SCHEDULE
SECTIONS	
EXIT	SECTION
LOCAL	TERMINATE
ORIGINATE	WAIT FOR statements

- The following VS FORTRAN I/O statements:
 - Statements other than READ, WRITE, or PRINT
 - READ and WRITE statements containing END= or ERR= labels
 - Statements that specify a NAMELIST
 - Statements containing arrays without subscripts
 - Statements that specify internal files
 - Asynchronous I/O statements

Parallel Processing within a Parallel Task

- Implied-DO statements, although implied-DO statements that are eligible for vector processing are also eligible for parallel processing.

Note: A DO loop containing a call to an event service subroutine is eligible for automatic parallel code generation but causes an error at run time.

Loop iterations must be computationally independent and sufficiently computationally intensive to warrant parallel processing.

A DO loop can be analyzed for parallel and vector code generation. In this case the number of iterations of the DO loop given to a virtual processor as a chunk to process is usually a multiple of the vector section size. Thus each virtual processor runs the optimal vector code for its iterations.

When you request parallel and vector code generation together, keep the following considerations in mind:

- An outer loop induction variable affecting an inner loop induction variable inhibits vectorization, but not automatic parallel code generation.
- External references are allowed for automatic parallel code generation (with the use of the IGNORE CALLDEPS directive), but not for vectorization.
- Unit stride affects vectorization more than automatic parallel code generation; parallel processing benefits most from larger strides.
- Both vectorization and automatic parallel code generation work best for outer loops, but when both are requested, the parallel loop must be outside of the vector loop.

You can request a compiler report that shows the decisions the compiler made about parallel and vector processing in your program. You can then use the parallel and vector directives to change some of those decisions. For more information about how to tune your program using the compiler reports and parallel and vector directives, see Chapter 18, “Aids for Tuning Your Program” on page 343.

Coding Parallel Constructs within a Parallel Task

When you explicitly code task management statements in your parallel program, you separate your program into parallel tasks. Within each parallel task, you can also identify parallel threads. These parallel threads can be iterations of loops, sections of code, or subroutines.

Coding Parallel Loops

Loops that are computationally independent are eligible to be written as parallel loops. It is your responsibility to ensure that no dependence exists between different iterations of a parallel loop that you explicitly code. It is also your responsibility to use the appropriate library synchronization subroutines to control access to shared data areas.

Loops that are computationally intensive benefit most from being rewritten as parallel loops. If you have nested loops that are eligible to run in parallel, you should convert the outermost to parallel first so that you maximize the amount of work done per iteration. It may also help you to separate data areas to be accessed in parallel.

You can use the following statements to explicitly code a parallel loop:

PARALLEL DO Begins a parallel loop.

LOCAL Specifies variables and arrays that are private to each parallel thread. Local variables and arrays are not initialized when they are created. This statement is optional. Note that **PARALLEL DO** and **DO** loop index variables are automatically considered local and need not be specified as such. See “Coding Private Arrays” on page 304 for further information about private arrays.

DOBEFORE Identifies a block of statements that each virtual processor participating in the processing of the parallel loop must run once before running any iterations of the parallel loop. This block of statements is optional.

DOEVERY Identifies a block of statements that make up the body of the loop. You may omit the **DOEVERY** statement only if it is not preceded by a **DOBEFORE** block.

DOAFTER Identifies a block of statements that each virtual processor participating in the processing of the parallel loop must run once after all iterations of the parallel loop have been completed or assigned to another virtual processor. This block of statements is optional.

EXIT Stops the assigning of loop iterations to virtual processors.

Each virtual processor that participates in running the parallel loop first runs the **DOBEFORE** block, shares running the **DOEVERY** block with the other virtual processors until all iterations are run, and then runs the **DOAFTER** block. Thus, different iterations of the code included in the **DOEVERY** block of a parallel loop can run concurrently using multiple virtual processors.

Variables specified on the **LOCAL** statement of a parallel loop are *not* shared with the parallel thread which created the loop. However, a single virtual processor may run multiple iterations of a loop, and a local variable retains its value among parallel threads run by the same virtual processor. In such cases you can use local variables to accumulate information.

Arguments which are expressions or loop index variables for calls and function invocations within a parallel loop are passed by copy to each parallel thread. Any scalar argument enclosed in parentheses is an expression.

VS FORTRAN Version 2 provides a single lock for each **PARALLEL DO** loop. When you specify **LOCK** for a **DOBEFORE** or **DOAFTER** block, each parallel thread processing the block obtains the lock for the loop when it enters the block, and releases the lock when it exits the block.

You can control the access to shared data by using library lock services. For information about how to use locks, see “Using Library Service Subroutines and Functions” on page 317.

You cannot code any of the following within a parallel loop:

- **ORIGINATE** statements
- **SCHEDULE** statements
- **TERMINATE** statements

Parallel Processing within a Parallel Task

- WAIT FOR statements
- PARALLEL CALL statements
- STOP statements
- RETURN statements
- Character function references, including CHAR
- Character expressions (concatenation and parenthesized character arguments)
- Calls to event service routines (detected as an error at run time).

The following are not allowed:

- Branches to statements inside the PARALLEL DO block by statements outside the block
- Branches to statements outside the PARALLEL DO block by statements inside the block.

Note: Subroutines and functions referenced in a parallel loop may not contain any of the constructs listed above, other than the RETURN statement, character function references, and character expressions.

See “Considerations for Subroutine and Function Invocations within a Parallel Thread” on page 307 for some further considerations you should keep in mind when you code a CALL or function invocation within a parallel loop.

Figure 83 is an example of an explicitly coded parallel loop. PTOT is initialized to 0 for each virtual processor, and retains values between parallel threads run by the same virtual processor. A LOCK is specified on the DOAFTER to ensure only one virtual processor adds to the final SUM at a time.

```
@PROCESS PAR(LANG NOAUTO REP) OPT(3)
  SUBROUTINE PLOOP(SUM)
    COMMON /M/M(500,500)
  C
    SUM = 0.0
    PARALLEL DO I = 1, 500
      LOCAL PTOT
      DOBEFORE
        PTOT = 0.0
      DOEVERY
        DO J = 1, 500
          PTOT = PTOT + M(J,I)
        ENDDO
      DOAFTER LOCK
        SUM = SUM + PTOT
      ENDDO
    RETURN
  END
```

Figure 83. Example of a Parallel Loop

Figure 84 on page 301 is an example of an explicitly coded parallel loop containing an EXIT statement. The program determines if a value is exceeded in a computation, and stops calculations if it is.

```

@PROCESS PAR(LANG NOAUTO REP) OPT(3)
  PROGRAM MAIN
  PARAMETER (N=500,M=1500)
  DIMENSION X(N,M),Y(N,M)
  REAL*4 TVAL,FVAL
  FVAL=0.0
  TVAL=200000.0

  C Initialize arrays X and Y
  :
10  PARALLEL DO I = 1, N
    PARALLEL DO J=1, M
      LOCAL D1
      D1=SQRT(X(I,J)**2 + (Y(I,J)**2))
      IF (D1.GT.TVAL) THEN
        FVAL=D1
        EXIT 10
      ENDIF
    ENDDO
  ENDDO
  :
  END

```

Figure 84. Example of a Parallel Loop With an EXIT Statement

Coding Parallel Sections

You can use parallel sections to run different code segments concurrently on multiple virtual processors. It is your responsibility to determine that the code you specify in a parallel section is valid and suitable for parallel processing.

You can use the following statements to explicitly code a parallel section:

PARALLEL SECTIONS Indicates the beginning of a group of sections, where each section can be run independently and concurrently with the other sections in the specified group.

LOCAL Specifies variables and arrays that are private to each parallel thread. Local variables and arrays are not initialized when they are created. This statement is optional. Note that the PARALLEL DO and DO loop index variables for loops within sections are automatically considered local and need not be specified as such. See “Coding Private Arrays” on page 304 for further information about private arrays.

SECTION Indicates the beginning of a group of statements to be processed as a parallel thread, concurrently with other parallel threads. In addition, the SECTION statement has a WAITING clause which you can use to specify a network of sections in which some sections are not started until other sections in the group have been completed.

END SECTIONS Indicates the end of a group of sections.

Variables specified on the LOCAL statement of a parallel section are *not* shared with the parallel thread that initiated the SECTION. For more information about sharing data between parallel threads in a parallel task, see “Sharing Data in Common Blocks within a Parallel Task” on page 308.

You cannot code any of the following within a parallel section:

Parallel Processing within a Parallel Task

- ORIGINATE statements
- TERMINATE statements
- SCHEDULE statements
- WAIT FOR statements
- PARALLEL CALL statements
- STOP statements
- RETURN statements
- Character function references, including CHAR
- Character expressions (concatenation and parenthesized character arguments)
- Calls to event service routines (detected as an error at run time).

The following are not allowed:

- Branches to statements inside the SECTION block by statements outside the block
- Branches to statements outside the SECTION block by statements inside the block.

Note: Subroutines and functions referenced in a parallel section may not contain any of the constructs listed above, other than the RETURN statement, character function references, and character expressions.

See “Considerations for Subroutine and Function Invocations within a Parallel Thread” on page 307 for some further considerations you should keep in mind when you code a CALL or function invocation within a parallel section.

Figure 85 on page 303 is an example of using parallel sections to compute a simple matrix sum:

```

@PROCESS PAR(LANG AUTO REP) OPT(3)
  SUBROUTINE PSECT(STOT)
  COMMON /M/M(500,500)
C
  PARALLEL SECTIONS
  LOCAL SUM
  SECTION 1
    SUM = 0.0
    DO 100 I = 1, 125
    DO 100 J = 1, 500
      SUM = SUM + M(I,J)
100  ENDDO
    SUM1=SUM
  SECTION 2
    SUM = 0.0
    DO 200 I = 126, 250
    DO 200 J = 1, 500
      SUM = SUM + M(I,J)
200  ENDDO
    SUM2=SUM
  SECTION 3
    SUM = 0.0
    DO 300 I = 251, 375
    DO 300 J = 1, 500
      SUM = SUM + M(I,J)
300  ENDDO
    SUM3=SUM
  SECTION 4
    SUM = 0.0
    DO 400 I = 376, 500
    DO 400 J = 1, 500
      SUM = SUM + M(I,J)
400  ENDDO
    SUM4=SUM
  END SECTIONS
  STOT = SUM1 + SUM2 + SUM3 + SUM4
C
  RETURN
  END

```

Figure 85. An Example of Parallel Sections

Coding Private Pointers

When a pointee variable or array is referenced in a parallel loop or section, and its associated pointer appears in a LOCAL statement for the loop or section, the reference to the variable or array uses the private pointer variable. As with any private variable, the pointer must be initialized before it is used. This can be done by any of the valid means of initializing a pointer and is required to make the associated variable or array available.

Note that, if the local pointer is initialized with the value of a shared pointer, the storage addressed by the pointee variable or array will not be private to the loop or section.

Example

```
Pointer (gp, Gtee(100,100))
Pointer (fp, Ftee(10,1000))
Dimension Buffer(10000)
gp = Loc (Buffer)

c
  Parallel Do I = 1,1000
  Local fp
  DoBefore
    fp = gp          ! Initialize local pointer
  DoEvery
    = Ftee(2,I)      ! Ftee is accessing Buffer, which is shared
  EndDo
```

Allocating storage within parallel constructs: Any storage allocated within the parallel loop or section should be deallocated before the loop or section terminates or the storage will remain assigned but inaccessible. This is true even for shared pointers, as storage belongs to the virtual processor which allocated it, not to the global program. An affinity function (such as PFAFFS) can be used to insure storage is obtained in the "main" task if access to the storage is required after the parallel construct terminates.

Coding Private Arrays

Actual and Dummy Arrays: The appearance of an actual or a dummy array (with constant bounds) on the LOCAL statement of a Parallel Do or Parallel Sections causes a private instance of the array to be provided to each virtual processor participating in the execution of the parallel loop.

Example

```
Parameter (n=1000)
Dimension F(n), Force(n)

c
  Parallel Do I = 1,n
  Local F
  DoBefore
    do 2 k=1,n
      f(k)=0.0          ! initialize private array
    do 1 j=1,i-1
      ...
      f(i)=f(i)+function(i,j)
      f(j)=f(j)-function(i,j)
      ...
  DoAfter Lock
c    Obtain lock to update shared array with values from private
c    arrays for each virtual processor
    do 3 k=1,n
      3    force(k)=force(k)+f(k)
  EndDo
```

Pointee Arrays A pointee array is not itself private at any level, since it exists only as a template and not as a true entity. The array can be mapped to private storage by virtue of being allocated within the parallel loop or section or by being assigned the address of a private array.

Example

```

parameter (k=100, j=100)
Pointer (gp,Gtee(k,j))
Pointer (fp,ftee(k,j))
dimension x(k,j)
allocate (gtee,stat=ic)
c Let first virtual processor use shared copy of array. Have
c all others get a private copy for a work array.
icnt =0
Parallel Do I = 1,n
Local Fp,X,ic,id
DoBefore lock
  icnt=icnt +1
  id =icnt
  if (id.ne.1) then
    Allocate (ftee,stat=ic)    ! Private for each processor
  else
    fp=gp                      ! Accesses shared copy of gtee
  ...
DoEvery
  ...
  ...
DoAfter Lock
  ..
  if (id.ne.1) then
    do ik=1,k
      do ij=1,j
        gtee(i,j)=ftee(i,j) +gtee(i,j)
      enddo
    enddo
    Deallocate (ftee, stat=ic)
  endif
EndDo

```

Coding Dynamically Dimensioned Arrays

The effect of the DDIM option is the same for parallel code as for serial code: the dimensionality of the array is determined at each reference. However, when using dynamically dimensioned arrays within a parallel loop or section, the variables used to determine run-time dimensions are those which exist on entry to the program unit. Therefore, although variables private to a parallel construct may have the same names as variables used in array declarators, their values are not used to determine run-time dimensionality. The values of the variables defined in the outermost serial environment are used to determine the dimensionality at each reference.

Parallel Processing within a Parallel Task

For example:

```
@PROCESS DDIM
  Pointer (gp, Gtee(K,J))
  Read(5,*) K, J          ! Read in K=100, J=100
  Allocate (Gtee, stat=ic)

c
  icnt=0
  jtotal=j
  Parallel Do I = 1,Jtotal
  Local gp, K, lcnt,lj
  DoBefore Lock
    K = 1000
    icnt = icnt +1
    j = max(1,j/(icnt*2))
c    Gtee will be allocated with dimensions (100,j)
c    not (1000,j)
  Allocate (Gtee, stat=ic)
  DoEvery
    . . .
  DoAfter
    . . .
  Deallocate (Gtee)
EndDo
```

Coding Subroutines as Parallel Threads

You can use parallel calls to direct subroutines to run as parallel threads. Note that coding a parallel call is not the same as coding a CALL or function invocation within a parallel loop or section. A parallel call assigns a subroutine to run as a parallel thread, whereas a CALL or function invocation coded within a parallel loop or parallel section is invoked serially by each parallel thread identified for that parallel loop or section.

You can use the following statements to explicitly code a parallel call:

PARALLEL CALL Explicitly specifies a subroutine to be run as a parallel thread by a virtual processor.

WAIT FOR ALL CALLS Causes the parallel thread to wait for all subroutines it called with a PARALLEL CALL statement to finish processing before it continues.

You cannot code a PARALLEL CALL statement within a parallel loop or parallel sections construct. You cannot code any of the following in a subroutine invoked by a PARALLEL CALL statement:

- ORIGINATE statements
- TERMINATE statements
- SCHEDULE statements
- WAIT FOR statements
- Calls to event service routines (detected as an error at run time)

Note: Subroutines and functions referenced in a subroutine invoked by a PARALLEL CALL statement may not contain any of the constructs listed above.

See “Considerations for Subroutine and Function Invocations within a Parallel Thread” on page 307 for some further considerations you should keep in mind when you code parallel calls.

Figure 86 shows an example of using parallel calls to call the same subroutine twice:

```

PROGRAM MAIN
COMMON /A/ A(1000,1000)
:
:
PARALLEL CALL SUB(J1)
PARALLEL CALL SUB(J2)
:
WAIT FOR ALL CALLS
END
SUBROUTINE SUB(K)
COMMON /A/ A(1000,1000)      ! shared
COMMON /B/ B(1000,1000)      ! local to each thread
:
END

```

Figure 86. Example of a Parallel Call

Considerations for Subroutine and Function Invocations within a Parallel Thread

If you want to run a subroutine or invoke a function in parallel, all of the following conditions must be true:

- Actual arguments passed to the subroutine are not function names, subroutine names, or alternate return specifiers.
- Actual arguments passed to the subroutine can be shared with the parallel thread that is identified.
- Common blocks specified in the program unit containing the PARALLEL CALL can be shared with the parallel thread that is identified.

Subroutines can share or have their own local copies of common blocks, depending on where the common blocks are declared. If you use a parallel call to invoke the subroutine, then all common blocks declared in the program unit containing the PARALLEL CALL statement will be shared with the parallel thread that is identified. Other common blocks are local to each parallel thread identified. For a description and examples of using common blocks with parallel processing, see “Sharing Data in Common Blocks within a Parallel Task” on page 308.

- The subroutines do not depend on the initialization of variables between one run and another.
- I/O units can be shared within the parallel task that contains the PARALLEL CALL statement.

If you code a parallel call, or a CALL or function invocation within a parallel loop or parallel sections, you must understand which instance of an actual argument, private to the parallel thread or not, is associated with the corresponding dummy argument of the subprogram. If the actual argument is specified on a LOCAL statement, or is automatically considered private, such as a loop index variable, then it is private to the parallel thread. If the actual argument is not private to the parallel thread, the called subprogram accesses a shared copy.

Parallel Processing within a Parallel Task

You can ensure that the actual value of a scalar argument at the point of call is passed by enclosing it in parentheses (passed by copy). Enclosing a scalar argument in parentheses causes a private copy of the argument to be created for each parallel thread and passed to the referenced subprogram before the next parallel thread begins. Although the name of the scalar appears as the argument in the source, the actual argument is an expression whose value is that of the scalar. Hence the value accessed by the subprogram will not be affected by the actions of another parallel thread, as might be the case were the parentheses not used. You can enclose a scalar argument in parentheses to indicate that it is to be used for input only.

Sharing Data in Common Blocks within a Parallel Task

Within a single parallel thread, program units communicate by sharing data areas in the normal Fortran manner: passing arguments between program units with the CALL statement or a function invocation, or using COMMON areas referenced by more than one program unit. For information about Fortran data communication between program units, see Chapter 19, “Interprogram Communication” on page 371.

Within a parallel task, program units invoked with a PARALLEL CALL, or a CALL statement or function invocation within a parallel loop or parallel section, may share common blocks associated with that parallel task; the program unit that contains any of these constructs must contain a COMMON statement for the common blocks to be shared. When a parallel thread finishes running, any common blocks shared with the invoking parallel task retain their definition status, and common blocks that were not shared with the invoking parallel task become undefined.

Figure 87 shows an example of using parallel calls to initialize a shared common block:

```

@PROCESS PAR(LANG NOAUTO REP) OPT(3)
  PROGRAM MATSUM
C
  COMMON /M/M(500,500)
C
  PARALLEL CALL SUB1
  PARALLEL CALL SUB2
  WAIT FOR ALL CALLS
  END
C
  SUBROUTINE SUB1
  COMMON /M/M(500,500)
  DO 50 I = 1, 250
    DO 50 J = 1, 250
      M(I,J) = SQRT(FLOAT(I+J))
  50 CONTINUE
  RETURN
  END
C
C
  SUBROUTINE SUB2
  COMMON /M/M(500,500)
  DO 90 I = 251, 500
    DO 90 J = 251, 500
      M(I,J) = I*J
  90 CONTINUE
  RETURN
  END

```

Figure 87. Example of Subroutine Use of Common Blocks

You can control and synchronize access to shared data areas by using lock and event services. For information about locks and events and how you can use them, see “Using Library Service Subroutines and Functions” on page 317.

Compatibility with Previous VS FORTRAN Releases: You must recompile main programs and subprograms that share static common blocks to run them in parallel. You do not need to recompile main programs and subprograms that share dynamic common blocks.

Using I/O within a Parallel Task

Each parallel task has a unique I/O environment. You identify parallel threads with PARALLEL DO, PARALLEL SECTIONS, or PARALLEL CALL statements. A parallel thread uses the I/O environment of the parallel task in which it runs.

The VS FORTRAN library controls the access to units for all threads in a parallel task on an I/O statement basis, thus preventing simultaneous I/O operations on the same file by multiple parallel threads. It is, however, your responsibility to control the coordination of I/O statements, and end-of-file and error conditions. You can use lock and event service routines to do this.

You must also review variables and arrays on I/O statements when you are determining if a piece of code is eligible for parallel processing. For example, if multiple READ statements are running concurrently, you must ensure that each parallel thread reads data into unique data variables and that a single variable is not overwritten by simultaneous iterations of a parallel loop. You can use the LOCAL statement in either parallel loops or parallel sections to create private copies of variables for each parallel thread. Parallel threads can modify different

array elements or you can use locks to ensure that only one parallel thread modifies any variable at a time.

Use LOCAL statements or library lock services to protect variables from being skipped or overwritten by simultaneous iterations of the same parallel loop. For information about locks and events and how you can use them, see “Using Library Service Subroutines and Functions” on page 317.

Using Parallel Processing between Parallel Tasks

A parallel task is a complete logical environment within which parallel threads run. Each parallel task has its own storage for local variables and common blocks, and its own set of I/O unit numbers. The sections below describe how to code parallel tasks, share data, and perform I/O between parallel tasks.

Coding Parallel Tasks

Your parallel program begins with a root task, and thereafter, you use the ORIGINATE statement to create additional tasks called originated tasks. You can use the following statements to explicitly manage a parallel task:

- | | |
|------------------|--|
| ORIGINATE | Creates a parallel task. A parallel task is <i>owned</i> by the parallel task that creates it. Each originated task has an integer identifier, and its own storage. |
| TERMINATE | Deletes an originated task from the parallel environment when you are finished with it. |
| SCHEDULE | Assigns a subroutine to run as the primary parallel thread in an originated task. The SCHEDULE statement has keywords that you can use to share or copy data between the scheduling parallel task and the scheduled parallel task. Each SCHEDULE statement must have a corresponding WAIT FOR statement. |
| WAIT FOR | Causes a routine to wait for the previously scheduled parallel tasks. |

There are two forms of the ORIGINATE statement: ORIGINATE ANY TASK and ORIGINATE TASK. ORIGINATE ANY TASK allows for merging of different programs without concern for conflicting taskids, because each originated task gets a new taskid. However, the selection of default file definitions that are used for the originated task can vary from run to run. ORIGINATE TASK ensures the same default file definitions are used for each run of the program. However, the ORIGINATE TASK statement may cause conflicts when you merge two programs, because of the possibility of conflicting taskids. See “Determining Default File Definitions for Originated Tasks” on page 316 for information about default file definitions used with the forms of the ORIGINATE statement.

You use the SCHEDULE statement to assign subroutines to run as parallel threads in originated tasks. The subroutine you assign using the SCHEDULE statement, called a *scheduled subroutine*, becomes the primary parallel thread of the originated task specified on the SCHEDULE statement. It, in turn, can identify additional parallel threads. If a SCHEDULE statement in originated task 1 assigns a parallel thread to run in originated task 2, the parallel thread runs using the common blocks and I/O units and files of originated task 2.

Naming scheduled subroutines in the DYNAMIC compiler option for all program units in which they are called will reduce virtual storage needs.

You can schedule a subroutine to run in one or more originated tasks, but you can schedule only one subroutine to run at a time in any one originated task. A WAIT FOR statement detects when a scheduled subroutine finishes running in an originated task. When the scheduled subroutine finishes running, the same or another subroutine can be scheduled for that originated task.

Originating parallel tasks and then terminating them causes additional run-time overhead for your program. You can make your program run faster by reducing the overhead required to originate and terminate tasks if you originate tasks once, and then reuse them. Then, using SCHEDULE and WAIT FOR statements, you assign new parallel threads to run in parallel tasks as old parallel threads finish running.

Sharing Data in Common Blocks between Parallel Tasks

Each parallel task, which includes the root and all originated tasks, has its own logical environment in which parallel threads run. This environment includes private routine variables and arrays, and unique set of common blocks.

When you originate a parallel task, you can use a DATA statement to initialize variables and common blocks. Any modifications made to these initialized variables and common blocks by subroutines scheduled in the parallel task environment are retained. Figure 88 shows one configuration of parallel task environments.

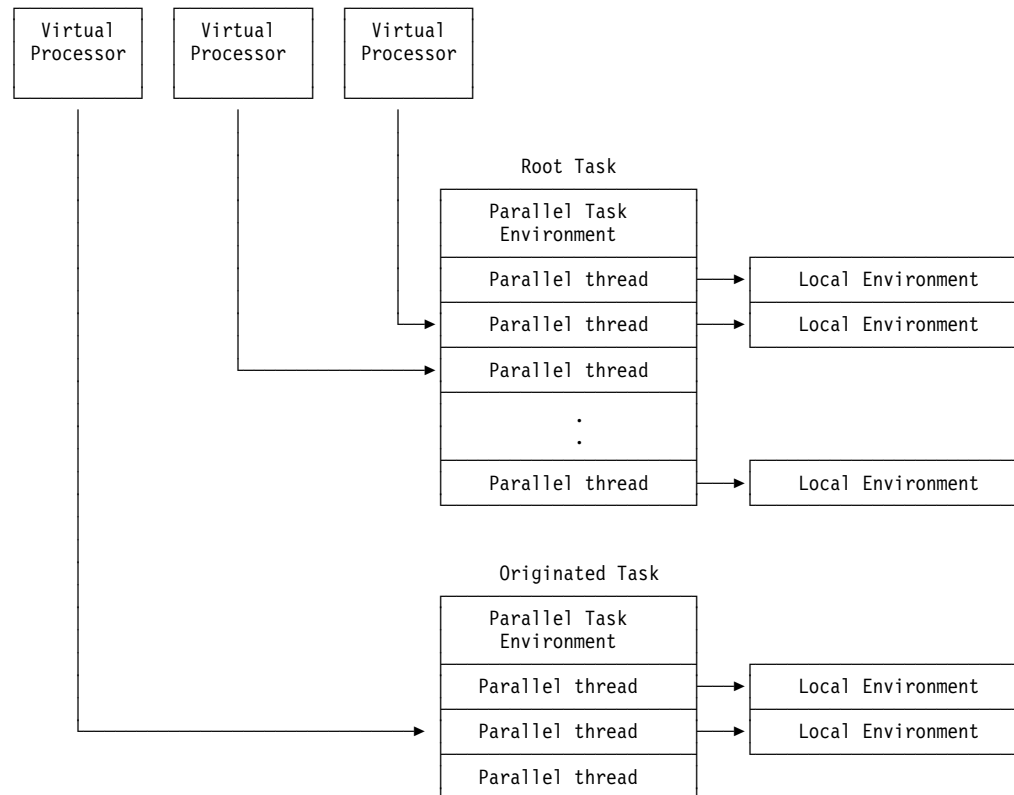


Figure 88. An Example of Parallel Task Environments

The values declared in common blocks in the primary parallel thread persist for the duration of a parallel task. The primary parallel thread of each parallel task uses the parallel task's set of common blocks. Additional parallel threads within the parallel task share the common blocks of the primary parallel thread if the parallel task to which they were assigned specifies the common blocks with a `COMMON` statement. This type of common block sharing is described in “Sharing Data in Common Blocks within a Parallel Task” on page 308.

With VS FORTRAN Version 2 you can communicate data between parallel tasks in the following ways:

- Share data areas between tasks
 - Use the `CALLING` clause of the `SCHEDULE` statement to specify arguments.
 - Use the `SHARING` clause of the `SCHEDULE` statement to share data in common blocks.
- Copy data areas between tasks: use the `COPYING`, `COPYINGI`, or `COPYINGO` clauses of the `SCHEDULE` statement to copy data in common blocks.

Common blocks used by a scheduled subroutine behave differently, and contain different information after the scheduled subroutine finishes running, depending on the clause specified on the `SCHEDULE` statement, as shown in the specific examples below.

Note: While the `SHARING` clause is “cheap” to use in terms of run-time overhead, the various `COPYING` clauses are more expensive.

Compatibility with Previous VS FORTRAN Releases: You must recompile main programs and subprograms that share static common blocks to run them in parallel. You do not need to recompile main programs and subprograms that share dynamic common blocks.

Example 1: To share a common block with another parallel task, specify the name of the common block on the `SHARING` clause of the `SCHEDULE` statement. The scheduling task's common block is then shared with the scheduled task as long as the scheduled subroutine runs. When the scheduled subroutine finishes running, the scheduling task retains the shared common block in its environment. The originated task that contained the now-completed scheduled subroutine resumes access to its own common blocks once again.

You may want to use the `SHARING` clause, for example, in the following situations:

- You want several parallel tasks to share a read-only common block that contains some initial constant data.
- You want each of several parallel tasks to update a unique section of a shared array.

Figure 89 on page 313 shows an example of a scheduled task, 1, that shares a common block with its owner.

Example 3: To copy a common block to another parallel task for input only, specify the name of the common block on the `COPYINGI` clause of the `SCHEDULE` statement. The common block is copied before the scheduled subroutine begins running, and its contents are not returned to the owner.

Use the `COPYINGI` clause, for example, if you want your root task to read sets of data into a common block and schedule different parallel tasks with different sets of data.

Figure 91 shows an example of a scheduled task, 123, that receives the contents of a common block from its owner before it runs the subroutine. 123 does not return the contents of the common block to the scheduling task. The two common blocks have the same name.

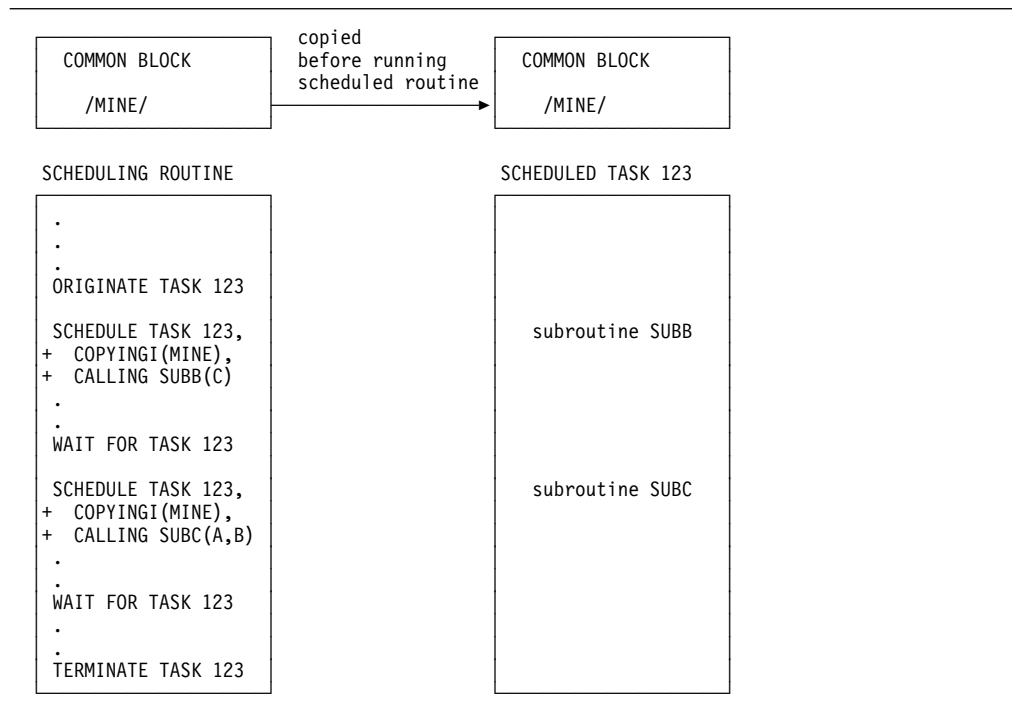


Figure 91. Example of the `COPYINGI` Clause

Example 4: To copy a common block back from another parallel task, specify the name of the common block on the `COPYINGO` clause of the `SCHEDULE` statement. The common block is copied to the scheduling task as part of processing the `WAIT FOR` statement.

Use the `COPYINGO` clause, for example, if the root task schedules several originated tasks and then waits for one to complete at a time: the root task writes the output data from the common block as each parallel task completes.

Figure 92 on page 315 shows an example of a scheduled task, 456, that copies the contents of its common block to the scheduling task when the `WAIT FOR TASK` statement detects that the scheduled subroutine has finished running. The scheduled task does not receive the common block contents of the scheduling task at any time. The two common blocks have the same name.

A READ after an end-of-file condition in a root task opens a new subfile, as it would in a serial program. In an originated task, a READ after an end-of-file condition causes an error.

Note: For sequential files, data is written and read sequentially as multiple parallel threads run. Because parallel threads run simultaneously, if data is being written to a file, it can appear in a different sequence for different runs of the program.

Determining Default File Definitions for Originated Tasks

Each originated task has an associated *error message unit*, *standard read unit*, and a *standard punch unit*. It will also have a *standard print unit* if your system programmer defined the error message and standard print units to be separate at installation time. Each subroutine running in an originated task that issues a PRINT or WRITE statement will write to the file associated with that originated task. Output produced by subsequent subroutines that are run in that originated task is added at the end of previous output.

The default file definitions for the error message unit, and the standard read, punch, and print units depend on which form of the ORIGINATE statement you use to create your originated task.

- If you use ORIGINATE ANY TASK to create your originated task, the default file definitions are:

Error Message Unit: FTSExxxx

Standard Print Unit: FTSPxxxx

Standard Punch Unit: FTSQxxxx

Standard Reader Unit: FTSRxxxx

where xxxx is a number from 1 to 9999, and is the absolute value of the *rtaskid* returned by the ORIGINATE ANY TASK statement.

The files are allocated sequentially for each originated task; for example, FTSE0001 for the first task, and FTSE0002 for the second. If an originated task created with an ORIGINATE ANY TASK statement is terminated and another originated task is created, the new task may use the file definitions of the terminated task, writing its output at the end of the previous output.

- If you use ORIGINATE TASK to create your originated task, the default file definitions are:

Error Message Unit: FTUExxxx

Standard Print Unit: FTUPxxxx

Standard Punch Unit: FTUQxxxx

Standard Reader Unit: FTURxxxx

where xxxx is a number from 1 to 9999, and is the absolute value of the *ptaskid* specified on the ORIGINATE TASK statement.

The error message unit and the standard print unit are dynamically allocated if file definitions are not provided for them.

Using Unnamed Files in Originated Tasks

You can dynamically allocate an unnamed file for use in an originated task only as a temporary file that exists for the duration of the originated task. For all other types of files you must use an OPEN statement with a FILE specifier.

Temporary I/O files for use by scheduled subroutines running within an originated task are dynamically allocated SCRATCH files. These temporary files, along with files connected to the default units are *unnamed files*. Temporary files are deleted by a CLOSE statement or when the originated task terminates.

For files connected to units other than the default units, the default for the STATUS specifier is SCRATCH. If you specify a STATUS that conflicts with SCRATCH (that is, OLD, UNKNOWN, or NEW on the OPEN statement, or KEEP on the CLOSE statement) an error will be detected. For files connected to one of the default units—standard error, standard print, standard reader, or standard punch—the default for STATUS is UNKNOWN.

Using Named Files in Originated Tasks

For named files only, all VS FORTRAN I/O statements are allowed for use by scheduled subroutines running within originated tasks. Files are closed when an originated task is terminated or when a CLOSE statement is coded. Otherwise, files remain connected when a subroutine finishes running.

On MVS only, different scheduled subroutines can read the same file. However, you must use locks or events to coordinate end-of-file or error conditions when reading the same file from different originated tasks. You can use the ACTION specifier on the OPEN statement to indicate whether a file will be updated.

Note: Each task processes a file as if it controls it completely, so scheduled subroutines should not use a file while another is updating it.

Using Library Service Subroutines and Functions

Data areas that are shared between parallel threads can be accessed independently or used for communication. If you use shared data areas for communication, you must use them in a controlled manner. VS FORTRAN Version 2 provides services for controlling access to shared data, and for synchronizing changes made when updating a shared data area. These services allow you to:

- Create *locks* to protect data that other routines might share. Locks allow you to ensure that only one parallel thread at a time modifies a data area.
- Use *events* to synchronize the processing of parallel tasks and parallel threads. Events allow you to ensure that, where required, certain calculations in parallel threads or parallel tasks are processed before others are made. For example, you can use events to signal that a parallel thread has completed an initial calculation and put the result into the shared data area for other parallel threads to use.

Using Parallel Lock Services

Using locks, you can control the concurrent running of critical operations, such as updating or referencing shared data areas of arguments or common blocks.

The following routines and functions create, obtain, release and delete locks:

- PLORIG** Creates and initializes a lock, and returns an identifier for it.
- PLTERM** Deletes a lock. Only the parallel task that created the specified lock can delete it.
- PLLOCK** Obtains a previously created lock. If the specified lock is unavailable, the parallel thread calling PLLOCK is suspended until the lock is available.
- PLCOND** Obtains a previously created lock if it is available. If the specified lock is unavailable, PLCOND does not wait for it; PLCOND always returns with an indication of whether or not it obtained the specified lock.
- PLFREE** Releases a specified lock. If other parallel threads are waiting for the specified lock, it is given to one or more of the threads before PLFREE finishes.

Before you use a lock you need to plan the following:

- Which locks you need
- What operations these locks are to protect and how the data is being accessed. Locks only work if you control all access to the same shared data areas using the same lock.
- When you need to use locks
- When locks are no longer needed. You can improve the performance of your parallel program if you minimize the amount of code between obtaining a lock and freeing a lock.

After you define what you want to do with a lock, you can use PLLOCK or PLCOND to obtain a lock. For each piece of data that you want to protect with locks, you must determine whether each parallel thread will update the data or just reference it. Choose one of the following two modes:

Exclusive mode If you want to *update* a variable in a shared data area, you can obtain a lock in exclusive mode. When a parallel thread obtains a lock in exclusive mode, it has sole access to the variable in the shared data area while it updates the variable; the thread has exclusive write-access to the variable. Until the parallel thread releases the lock, all other parallel threads that want to use the lock—in either exclusive or shared mode—must wait.

Shared mode If you only want to *reference* a variable, you can obtain the lock in shared mode; the parallel thread has read-only access to the variable. All other parallel threads that want to use the lock in shared mode can obtain the lock concurrently. Shared mode defers any requests for exclusive mode until *all* current holders release the lock. You can improve the performance of your parallel program if you use shared mode as often as possible.

Figure 93 shows what happens to a request for a lock based on the current status of that lock prior to beginning the critical operation.

Figure 93. Lock Operation Types

Request Type	Current Lock State		
	Free	Exclusive	Shared
PLLOCK Exclusive	<i>Obtain</i>	<i>Wait(1)</i>	<i>Wait(1)</i>
PLLOCK Shared	<i>Obtain</i>	<i>Wait(1)</i>	<i>Obtain(1)</i>
PLCOND Exclusive	<i>Obtain</i>	<i>Not Obtained</i>	<i>Not Obtained</i>
PLCOND Shared	<i>Obtain</i>	<i>Not Obtained</i>	<i>Obtain(1)</i>
Note: 1. An error occurs if the same parallel thread requests the lock that it already holds.			

After you finish with the critical operation, release the lock using PLFREE. When you finish all critical operations you were using the lock to protect, delete the lock using PLTERM.

Use VS FORTRAN lock services to coordinate the following in parallel programs:

- Error and end-of-file processing performed by I/O statements that use the ERR= or END= specifiers (OPEN, CLOSE, ENDFILE, BACKSPACE, READ, WRITE, REWIND, INQUIRE, or REWRITE)
- The use of a single I/O unit by different parallel threads
- Calls to DUMP, PDUMP, CDUMP, CPDUMP, and SDUMP with other I/O statements or services that use the same unit
- Calls to the UNTNOFD and UNTANY service routines. UNTNOFD and UNTANY are routines that identify available unit numbers. A problem would arise, for example, in a parallel thread that contains several subroutines that use UNTNOFD or UNTANY, because one subroutine can call UNTNOFD or UNTANY and receive an answer right before another subroutine begins to use a unit.
- Use lock services to coordinate calls to DVCCHK or OVERFL with statements that produce results that are checked by DVCCHK or OVERFL.

Using Parallel Event Services

You can use library event services to synchronize primary parallel threads of originated tasks that are running concurrently.

Events use post and wait signals sent by running parallel threads to synchronize parallel threads. When you create an event, you must specify a number of post signals, wait signals, or both post and wait signals; until the specified number of post and wait signals is reached other events will wait to continue running. The time between when an event begins—has no wait or post signals—and when it reaches its specified number of signals is called an *event cycle*. There are three types of event cycles:

Wait only
Post only

Wait and post.

During an event cycle, critical operations waiting on the event are suspended until the requisite number of post and wait signals is reached.

The following routines create, post, delete, and wait for events:

- PEORIG** Creates an event for use by parallel threads. When you call the PEORIG routine, you must request an identifier for the event and the number of post and wait signals needed to complete an event cycle.
- PETERM** Terminates an event. Only the parallel task that created the specified event can terminate it, but not during an event cycle.
- PEPOST** Signals a post for a specified event from a parallel thread. If a wait and post cycle is running for which the postcount has been reached, but the waitcount has not, a call to PEPOST suspends work until the next event cycle.
- PEWAIT** Signals a wait for a specified event from a parallel thread. The invoking parallel thread then waits until the specified event's waitcount (or wait-and postcount for wait and post cycles) is reached.

An error occurs at run time if you code a call to any of the event service routines from within any of the following:

- A PARALLEL DO construct
- A PARALLEL SECTIONS construct
- A subroutine invoked by a PARALLEL CALL statement
- Subroutines or functions given control from within a PARALLEL DO construct
- Subroutines or functions given control from within a PARALLEL SECTIONS construct
- Subroutines or functions given control from within a subroutine invoked by a PARALLEL CALL statement.

Before you set up an event cycle, you need to plan the following:

- Whether you must use events
- Where you need to post events
- Where you need to wait for events to complete.

The sample program in Figure 94 on page 321 program uses three parallel threads to calculate values X, Y, and Z. These values are added to a variable W. The three subtasks can do other work, but the main program needs to know only when all updates to W are finished.

```

@PROCESS OPT(3) PAR(AUTO LANG REP(XLIST))
C
    CALL PLORIG(ILOCK)
    CALL PEORIG(IEVENT,3,1,1)      ! Postcount = 3
C
    DO 10 I = 1,3
10  ORIGINATE TASK I
C
    W = 0
C
    SCHEDULE TASK 1, CALLING SUB01(ILOCK,IEVENT,W)
    SCHEDULE TASK 2, CALLING SUB02(ILOCK,IEVENT,W)
    SCHEDULE TASK 3, CALLING SUB03(ILOCK,IEVENT,W)
C
    CALL PEWAIT(IEVENT)
    CALL PETERM(IEVENT)
    CALL PLTERM(ILOCK)
C
C-----
C    Code using updated value of W
C-----
C
    WAIT FOR ALL TASKS
    END
@PROCESS OPT(3) PAR(AUTO LANG REP(XLIST))
C-----
C    SUBROUTINE SUB01(ILOCK, IEVENT, W)
C-----
C
C    Code to calculate X
C-----
C
    CALL PLLOCK(ILOCK,W)           ! request exclusive lock
    W = W + X
    CALL PLFREE(ILOCK,W)
    CALL PEPOST(IEVENT)           ! Inform done updating W
C
C-----
C    Possibly more SUB01 code
C-----
C
    RETURN
    END
@PROCESS OPT(3) PAR(AUTO LANG REP(XLIST))
C-----
C    SUBROUTINE SUB02(ILOCK, IEVENT, W)
C-----
C
C    Code to calculate Y
C-----
C
    CALL PLLOCK(ILOCK,W)           ! request exclusive lock
    W = W + Y
    CALL PLFREE(ILOCK,W)
    CALL PEPOST(IEVENT)           ! Inform done updating W
C
C-----
C    Possibly more SUB02 code
C-----
C
    RETURN
    END

```

Figure 94 (Part 1 of 2). Example of Library Lock and Event Services

```
@PROCESS OPT(3) PAR(AUTO LANG REP(XLIST))
C-----
C      SUBROUTINE SUB03(ILOCK, IEVENT, W)
C-----
C      Code to calculate Z
C-----
C      CALL PLLOCK(ILOCK,W)          ! request exclusive lock
C      W = W + Z
C      CALL PLFREE(ILOCK,W)
C      CALL PEPOST(IEVENT)          ! Inform done updating W
C-----
C      Possibly more SUB03 code
C-----
C      RETURN
C      END
```

Figure 94 (Part 2 of 2). Example of Library Lock and Event Services

You can use events, for example, in the following situations:

- A parallel thread completing a piece of work needs to signal to several other parallel threads when it is complete. For example, in some algorithms once $A(I,J)$ is calculated, $A(I,J)-1$ and $A(I,J)+1$ can be calculated simultaneously.
- Several parallel threads need to complete before another parallel thread continues. For example, if your program were sorting and merging, a thread that merges would need to wait for all of the sorting threads to complete before continuing.

Considerations for Using Parallel Service Subroutines

You can use locks within any parallel language construct. You cannot use event services within the following:

- PARALLEL CALL
- PARALLEL DO
- PARALLEL SECTIONS
- Any subroutine or function given control within a PARALLEL DO or PARALLEL SECTIONS construct, or a subroutine invoked by a PARALLEL CALL statement.

If you use locks or events without specifying the PARALLEL run-time option, then the parallel environment will be initialized to run on a single virtual processor—PARALLEL(1).

Lock contention occurs if two or more parallel threads try to obtain a lock at the same time. Only one parallel thread will obtain the lock and the other parallel threads will be suspended until the lock is available. Lock contention may result in a bottleneck where many threads are waiting and only one thread is active.

By planning the use of locks in a structured manner you can:

- Increase the efficiency of lock use
- Ensure your chances of obtaining a free lock upon request, thus reducing the probability of lock contention
- Avoid overhead caused when a parallel thread must wait for a specified lock.

To minimize the possibility of lock contention, do only the work that needs the protection of the lock, and then release the lock.

In general, you should define a lock in a higher-level parallel thread, and pass its identifier to the parallel threads that will use it when they are running. The parallel threads that receive the lock's identifier can use the lock directly, or pass the identifier to other parallel threads to which they assign work. Establish an order for obtaining and releasing locks, and use it in all threads that use those locks, such as the order shown in Figure 95.

```

PARALLEL SECTIONS
SECTION 1
  :
  CALL PLLOCK(IA)
  CALL PLLOCK(IB)
  :
  CALL PLFREE(IB)
  CALL PLFREE(IA)
SECTION 2
  :
  CALL PLLOCK(IA)
  CALL PLLOCK(IB)
  :
  CALL PLFREE(IB)
  CALL PLFREE(IA)
END SECTIONS

```

Figure 95. An Example of the Structured Use of Locks

By using library synchronization services carefully you can avoid not only lock contention, but a deadlock. A *deadlock* occurs when each of two parallel threads tries to obtain a set of resources, such as a lock or an event. If one parallel thread gets one resource, and another parallel thread gets another resource, they both try to obtain the resource the other is holding prior to continuing. Deadlock situations cause an error at run time.

Using the NPROCS Function

VS FORTRAN Version 2 provides the NPROCS function that your parallel program can use to determine the number of virtual processors specified at run time. You can code the NPROCS function in your program so that the work can be proportioned based on the number of virtual processors specified on the PARALLEL run-time option.

Using Subroutines to Control Processor Affinity

You can use the parallel execution controls service subroutines PFAFFS, PFAFFC, and PYIELD to control processor affinity for a particular thread. These subroutines allow you to issue system services and to avoid interlocks. See *VS FORTRAN Version 2 Language and Library Reference* for details about parallel service routines.

Example of the Use of an Affinity Service

```

@PROCESS OPT(3) SDUMP NODECK DC(FENETRE) PARALLEL(NOAUTO LANG)
  Program MOZZI
c   This example shows how MVS callable system services can be used
c   in a parallel program. Note how PFAFFS is used to set affinity
c   before the services are called and how PFAFFC is used after the
c   services are called.
c   - CSRIDAC is a callable window service
c   - CSRVIEW is a callable window service
c   - CNUDTV and CNUTFV are user routines containing parallel constructs
c
      IMPLICIT REAL*4 (A-H,O-Z)
      PARAMETER( NFEN=6, LREAL=4, NVARPAG=4096/LREAL )
      PARAMETER( NPAGFEN=6912, NVARFEN=NPAGFEN*NVARPAG )
      PARAMETER( NPAGHIP=NFEN*NPAGFEN )
      COMMON /FENETRE/ TABFEN(NVARFEN)
      CHARACTER*8 HIP
C CREATION
      CALL PFAFFS
      CALL CSRIDAC('BEGIN','TEMPSPACE','HIPER ','YES','NEW','UPDATE',
:               NPAGHIP,HIP,LHIP,IRC,ISRC)
      CALL PFAFFC
C NVAR POUR TEST UNIQUEMENT
      NVAR=NPAGHIP*NVARPAG
      CALL CNUDTV
C ECRITURE
      VMOY=0
      DO IFEN=1,NFEN
        IOFFSET=(IFEN-1)*NPAGFEN
        CALL PFAFFS
        CALL CSRVIEW('BEGIN',HIP,IOFFSET,NPAGFEN,TABFEN,'SEQ ',
:               'RETAIN ',IRC,ISRC)
        CALL PFAFFC
        DO I=1,NVARFEN
          TABFEN(I)=IOFFSET*4096 + I
          VMOY=VMOY+TABFEN(I)/NVAR
        END DO
        CALL PFAFFS
        CALL CSRVIEW('END ',HIP,IOFFSET,NPAGFEN,TABFEN,'SEQ ',
:               'RETAIN ',IRC,ISRC)
        CALL PFAFFC
      END DO
      CALL CNUTFV
      . . .
END

```

Considerations for Existing Service Subroutines

NTASKS NTASKS returns a value of zero when the PARALLEL option is in effect; AUTOTASK and PARALLEL are conflicting run-time options and if both are specified the last one specified is used.

CPUTIME CPUTIME is not allowed in a parallel program.

XUFLOW XUFLOW changes the exponent underflow mask in the parallel thread that is running. When a parallel thread is scheduled to run in an originated task, the exponent underflow mask is set according to the XUFLOW|NOXUFLOW run-time option. Parallel threads running within

the same originated task inherit the exponent underflow mask setting of the parallel thread that assigns them to run. Other parallel threads are not affected by the changes made to the exponent underflow mask, unless they are combined for processing as a group with the parallel thread that changes the exponent underflow mask.

For example, a group of iterations of a parallel loop are running as a parallel thread. If the XUFLOW service routine is called within an iteration of the loop, the exponent underflow mask is only set for that group of iterations. A nested parallel loop would inherit the exponent underflow mask setting of the iteration or group of iterations that assigned it to run.

SYSRCX and EXIT

SYSRCX and EXIT can be called only in the primary parallel thread of the root task.

SYSRCS and SYSRCT

The return codes from SYSRCS and SYSRCT have meaning only in the root task.

ERRMON, ERRSAV, ERRSET, ERRSTR, ERRTRA

To use these service subroutines correctly, you must remember that each parallel task has its own error option table. A parallel thread uses the error option table associated with the parallel task it runs in.

Locks must be used to set and retrieve entries in the error option table if they are accessed by multiple parallel threads.

Data-in-virtual service subroutines

These routines are allowed only in the root task of parallel programs.

Multitasking facility (MTF) subroutines

These routines are not allowed in parallel programs.

Using the Parallel Trace Facility

You can record events occurring during parallel execution by the use of the parallel trace facility, triggered by use of the PTRACE run-time option or the trace service subroutines.

The Parallel Trace Facility can generate trace records to identify entry to and exit from every subprogram, entry to the main program, termination of the main program, and many trace events occurring during the execution of a parallel program.

By using the data obtained via the Parallel Trace Facility, you can analyze the behavior of parallel programs and perform necessary tuning and debugging functions.

The Parallel Trace Facility records each significant action as a record in an output file, known as the **Trace File**. Separate programs, collectively referred to as *trace tools*, are used to analyze these records; trace tool programs must be used as post-processors against the Trace File after execution of the VS FORTRAN Version 2 parallel program has completed.

You can specify the PTRACE run-time option when executing a VS FORTRAN Version 2 program even if the PARALLEL(TRACE) option was not used to compile it. (See 111 for more information.) However, without the compile option certain trace events, such as subroutine entry and exit, will not appear in the resulting trace file. (See 34 for more information.) Program compilation and execution times increase slightly with the use of these options.

Parallel Trace and Analysis

Uses of Parallel Trace and Analysis

To improve the performance of a parallel application. By tracing the execution of a parallel program you can identify serial sections of execution, possible bottleneck areas, unequal distribution of parallel work, etc. The trace facility collects information at run time about the allocation of work among virtual processors. Identifying when virtual processors are idle, allows you to analyze the code that is executing to determine if there is a way to improve its serial execution or to make use of parallelism within it. Bottleneck areas may be associated with the obtaining and freeing of locks. Trace execution analysis indicates when locks are obtained, when locks are waited for, and when locks are freed. If excessive time is spent waiting to obtain locks, you can then analyze their algorithms and use of locks to minimize the delays associated with locks.

Similarly parallel programs using events may encounter bottlenecks. For example, an algorithm could use events to process data in a pipelined fashion: task 1 processes the first piece of data, then passes that data onto task 2 for more processing while it processes the next piece of data. In such an algorithm, the work done by each task must be approximately equal or a bottleneck may occur.

The trace facility allows you to determine if one task is slower than the other tasks and if it is affecting the potential speedup of the application. For PARALLEL DOs and PARALLEL SECTIONS, trade-offs may be made between allocation of small numbers of large chunks of work and allocation of large numbers of small chunks of work. If the work is equally allocated among the chunks then it is best to minimize overhead by having a small number of pieces of work defined. If the work varies in size then this can cause processor idle time at synchronization points. In this case it may be better to allocate more chunks of work that can be self load balancing. The trace facility aids you in identifying these conditions.

To verify parallel execution paths of a program. The problems associated with verifying a serial program's execution exist, plus new problems introduced by the possibility of concurrent and/or different orders of execution of pieces of code. The trace facility cannot be used to ensure that a parallel program is correct. However it can be used to review the execution characteristics of a parallel program and to determine what code paths have been exercised. For example, did a parallel do loop execute concurrently or did it execute serially? Or, did a parallel thread for task 1 always obtain the lock prior to a parallel thread of task 2? By executing and tracing a parallel program multiple times and varying the number of virtual processors the execution paths may vary thru the parallel constructs. The analyzed trace files may be compared to determine what code paths have been exercised. In some cases, a finer analysis of code path execution may be required. An interface for putting user-specified information into the trace execution file is provided.

To aid in debugging of parallel programs. The trace facility is one tool that may help the difficult job of debugging a parallel program. Parallel programs are susceptible to timing related problems - sometimes the answer is right and sometimes it is not. Comparing the execution-time traces of such a parallel program's executions may allow you to identify where different execution occurs. The different execution paths can then be reviewed to see if they are the cause of the problem. A simple example might be when the correct answer is determined, a parallel thread for task 1 always obtains the lock prior to a parallel thread from task 2. When the incorrect answer is determined task 2 obtained the lock prior to task 1. You may then review the code executed under lock control in both tasks to determine if there is a possible problem in this area. A correct parallel program will have many valid, but different, execution-time traces, thus analysis of this type may not be easy. However it may help to narrow in on the problem code. The user interface for putting information into the trace file may be useful in a debugging mode also.

Interpretation of Parallel Trace and Analysis Data

The trace facility can be used when parallel programs are executed in both dedicated and non-dedicated environments. The initial development and testing of a parallel program will likely be done in non-dedicated environments. The final version of the parallel program may be planned to be executed in a dedicated mode at all times, or it may be executed in a dedicated mode at critical times and non-dedicated mode at other times. The final performance tuning should be done in the environment that the application will be executed in.

You should be aware of the following characteristics of the trace facility when you interpret the results of a parallel program's execution:

A probe effect can occur. Sometimes a situation you are trying to analyze disappears or changes due to the analysis process. This is known as a probe effect. The collection of trace records at execution-time can alter a parallel program's timing and therefore its execution. This can cause a probe effect. The generation of the trace records, will introduce some additional overhead at points where parallel threads are created and where they are synchronized. Thus when analyzing very fine grain parallel applications the activation of trace may distort the information you are trying to determine. The generation of trace records may also alter the concurrency of execution paths. Thus, a timing related problem in your program may appear or disappear when trace is activated.

Trace data is recorded for virtual processors specified, not real processors. You specify the number of virtual processors to be used for a program's execution with the PARALLEL execution-time option. This defines the maximum degree of parallelism that can be achieved for that parallel program's execution. However the operating system maps the virtual processors onto the real processors available during program execution. If the number of real processors is less than the number of virtual processors specified or if there are other programs executing at the same time as the parallel program, then the real parallelism achieved may be less than the number of virtual processors specified. In these situations the operating system may temporarily suspend virtual processors from execution. These suspensions cannot be seen by the trace facility and may affect the interpreting of the trace facility's data. To minimize the ambiguity of the trace data, parallel programs may be executed in dedicated mode or with high priority.

Real Time: Real time is the consistent clock across all virtual processors. Virtual time is per virtual processor. The trace facility uses the real, or wall-clock, time as time-stamps in the trace records created. If the parallel program is executed in a dedicated mode or with high priority, this allows some performance calculations to be done. However when the program is executed on loaded systems, the times recorded may be longer than the actual times used by the parallel program. For example, in a dedicated environment a scheduled task may start at time x and complete at time $x+100$, for an elapsed time of 100. In a non-dedicated environment the same task may start at time y and complete at time $y+1000$, for an elapsed time of 1000. The task may not have used any more cpu time in the non-dedicated environment, but the operating system may have suspended the virtual processor executing the scheduled task and performed some other work. To minimize such interference parallel programs may be executed in dedicated mode or with high priority.

The trace facility provides an option to obtain a virtual cpu time stamp on each trace record. The cpu time stamp will indicate how much cpu time has been spent on a particular virtual processor, thus the elapsed cpu times of executed work may be determined. However, since each processor has a different cpu clock "ticking" it may be difficult to determine the cpu time of work that is waiting for execution.

Program Tracing Categories

The significant actions for which tracing occurs in VS FORTRAN Version 2 parallel programs fall into the following categories:

1. Task-related activities

Actions in this category include:

- a. Origination of a task (via the ORIGINATE statements).
- b. Scheduling of a task (via the SCHEDULE statements).
- c. Waiting for task activity (via the WAIT FOR statement).
- d. Terminating a task (via the TERMINATE statement).

2. Event-related activities

Actions in this category include:

- a. Origination of an event (via the PEORIG service routine).
- b. Posting of an event (via the PEPOST service routine).
- c. Waiting for an event (via the PEWAIT service routine).
- d. Removal of an event (via the PETERM service routine).

3. Lock-related activities

Actions in this category include:

- a. Origination of a lock (via the PLORIG service routine).
- b. Unconditionally obtaining a lock (via the PLLOCK service routine).
- c. Conditionally obtaining a lock (via the PLCOND service routine), or failure to obtain the lock.
- d. Freeing of a lock (via the PLFREE service routine).
- e. Removal of a lock (via the PLTERM service routine).

4. Thread-related activities

Actions in this category relate to the internal processing of execution threads by the VS FORTRAN Version 2 run-time environment and include:

- a. Origination of a thread.
- b. Scheduling a thread for execution.
- c. Beginning execution of a thread.
- d. Completing execution of a thread.
- e. Deletion of a thread.

5. Parallel call-related activities

Actions in this category relate to the processing of the PARALLEL CALL and WAIT FOR ALL CALLS statements by the VS FORTRAN Version 2 run-time environment and include:

- a. Scheduling a parallel called routine for execution.
- b. Waiting for completion of all parallel called routines.

6. Parallel loop and section related activities

- a. Initial scheduling of a Parallel DO loop for execution.
- b. Termination of the Parallel DO loop after completing execution of all iterations for the current thread.
- c. Initial scheduling of a Parallel Section for execution.
- d. Termination of the Parallel Section after completing execution of all sections.

7. Common-block management

Actions in this category relate to the processing of common blocks by the VS FORTRAN Version 2 run-time environment and include:

- a. Creation of a common block (first-time use).
- b. Sharing of a common block between tasks.
- c. Copying of a common block between tasks.

8. Trace Environment management

Actions in this category relate to the internal management of the Parallel Trace Facility and include:

- a. Activation of the Parallel Trace Facility and initial Trace File control record.
- b. Modification of trace control values (via the PTPARM service routine).
- c. Termination of the Parallel Trace Facility and final Trace File control record.

Additional significant actions for which tracing occurs in VS FORTRAN Version 2 parallel programs, which occur only when a parallel program has been compiled with the PARALLEL(TRACE) compiler option, fall into the following categories:

1. Parallel DO-related activities

Actions in this category relate to the processing of a Parallel DO loop by the VS FORTRAN Version 2 run-time environment and include:

- a. Entry to the DO BEFORE clause of a Parallel DO loop.

Parallel Trace Facility

b. Entry to the DO AFTER clause of a Parallel DO loop.

c. Beginning execution of each chunk of a Parallel DO loop.

2. Parallel Section-related activities

Actions in this category relate to the processing of a Parallel Section by the VS FORTRAN Version 2 run-time environment and include:

a. Beginning execution of each new chunk of Parallel Sections.

3. Parallel Program-related activities

Actions in this category relate to the processing of a Parallel program unit (main, subroutine or user function) by the VS FORTRAN Version 2 run-time environment and include:

a. Initial entry to the program (main unit).

b. Entry to a subroutine or function program.

c. Exit from a subroutine or function program.

In addition to these categories, additional trace records are generated in response to user requests made by calling the PTWRIT service routine.

The Parallel Trace Facility provides user controls over which trace actions are to cause records to be generated in the Trace File.

The Parallel Trace Facility allows the user to include or exclude which of the above categories should be handled by the Parallel Trace Facility, on an overall program basis via the PTRACE run-time option, and selectively during program execution via the PTPARM service routine.

The Trace File

The Trace File triggered by the PTRACE run-time option or trace service subroutines contains information such as time stamp, virtual processor id, source code location, event type, and thread/task id. It is in machine readable form; not in text or character form. In addition, records in this file might not be in sequential time sequence order because they are written by different physical processors in a multiprocessing system. Consequently, a post-processor program, suitable for formatting the various records and placing them into proper time sequence order, is required.

The ddname to which the file is associated is AFBTRACE.

Parallel Trace Facility Support Materials

A sample Fortran program, AFBTRAC, is provided in source format as part of VS FORTRAN Version 2 Release 6 to format the contents of the Trace File into readable format and in proper record sequence. The AFBTRAC program is to serve as a basis for you to use to create your own program.

This routine is included in the optional source materials available with VS FORTRAN Version 2 Release 6. It is also provided on the product tape and is installed on CMS as AFBTRAC FORTRAN on the product disk; it is installed on MVS as the member AFBTRAC of dataset SYS1.VSF2.SUPPORT.

This source contains complete descriptions of each record in the Trace File, as well as sample processing for this file.

In addition to this source, the product tape contains the format of each record and the reason it was created by VS FORTRAN Version 2. This is installed on CMS as PARTRACE INFO on the product disk; it is installed on MVS as the member PARTRACE of dataset SYS1.VSF2.SUPPORT on the product disk.

Note: Your MVS product installer might have chosen a name other than SYS1.VSF2.SUPPORT. Check with your local product support for the name used by your installation.

Parallel Environment

Three conditions are required for a program to run in parallel:

1. Multiple parallel threads are ready to be run.
2. Multiple virtual processors are associated with your program.
3. Multiple real processors are available to your program.

You control the first two of these conditions by directing the compiler to automatically generate parallel code for DO loops, using parallel language constructs, and by using the PARALLEL run-time option to specify the number of virtual processors on which your parallel program can run. The third condition is dependent on the scheduling policies established by your system administrators and your operating system; as processors become free, parallel threads can be run simultaneously.

The number of virtual processors you can specify at run time is dependent on your system:

- On VM/XA or VM/ESA, if you specify more than one virtual processor, each virtual processor is assigned to a separate virtual CPU. An additional virtual processor is assigned to CMS for processing CMS service requests. You can use the execution controls subroutines to assign a particular thread to this processor, which enables you to request system services from that thread. The maximum number of virtual processors allowable in the VM/XA or VM/ESA environment depends on the limit specified in the VM/XA or VM/ESA directory.
- On MVS, if you specify more than one virtual processor, each virtual processor is assigned to a separate MVS task. An additional MVS task is assigned to process the opening and closing of files. You can use the execution controls subroutines to assign a particular thread to this task, which enables you to request system services from that thread.

Check with your system programmer for limits on available virtual processors at your site.

With the VS FORTRAN Version 2 parallel feature, you can identify more parallel threads than there are processors. While threads remain to be processed, when a processor finishes one parallel thread it will select another and continue to run.

VS FORTRAN Version 2 parallel programs can run in a dedicated or nondedicated run-time environment; a *dedicated run-time environment* is one in which your program does not have to contend with other users' programs for the use of processors. A parallel program runs more quickly in a dedicated run-time

environment. When you are coding your parallel program you should have in mind whether it will run in a dedicated or nondedicated run-time environment.

If your parallel program will run in a dedicated run-time environment, consider assigning each virtual processor a fixed and equal amount of work to run concurrently, because each virtual processor corresponds directly to a real processor.

If your parallel program will run in a nondedicated run-time environment, code your program to perform dynamic load balancing on virtual processors because your program will be competing with other programs for use of the real processors. If your program uses dynamic load balancing, it can make the best use of real processors as they become available. For examples of how you can code your program to perform dynamic load balancing, see “Improving Performance of Parallel and Vector Processing” on page 337.

Improving Parallel Processing

Following are some guidelines for coding efficient parallel programs. In general, you should design or restructure your program to:

- Optimize serial code
- Reduce the amount of code that must be run serially
- Gain maximum parallel use of multiple vector facilities (for specific information about requesting automatic generation of parallel and vector code together, see “Generating Parallel Code Automatically” on page 297)
- Leave wide spaces between data areas accessed by different virtual processors
- Reduce virtual storage requirements (and load time) by the use of the DYNAMIC compiler option for scheduled subroutines, subroutines named in PARALLEL calls, and subprograms referenced within parallel constructs. Note that all referencing program units, whether parallel or serial, must use the DYNAMIC option in order to benefit.
- Reduce the overhead required to run parallel constructs. Some methods of achieving this are:
 - Originate tasks once and reuse them to reduce the overhead required for originating and terminating tasks. See “Coding Parallel Tasks” on page 310 for more information.
 - Maximize the work each parallel task can do in order to avoid additional overhead associated with task management.

You can think of parallel processing in terms of granularity. *Granularity* is an expression of the ratio between computation and communication in a parallel program. Parallel threads that contain a minimal amount of work have a fine granularity; running single iterations of loops as threads on several virtual processors may, for example, give your program a very fine granularity. Fine-grained programs spend relatively more time communicating than coarse-grained ones.

- Maximize the work subroutines do when they are called in a parallel loop, parallel section, or specified on the PARALLEL CALL statement.

- Maximize the work that function subprograms do when they are referenced in a parallel loop or parallel section.
- Do not code parallel constructs or request automatic parallel code generation for loops with a PREFER PARALLEL directive if the work is not computationally intensive. There is some overhead involved in distributing work among virtual processors. If the amount of work being distributed is small, the overhead can outweigh the performance benefit.
- Where you can, use library lock and event services to coordinate processing between tasks.
- Avoid unnecessary copying of common blocks.
- Balance the work loads assigned to virtual processors. Some methods of achieving this are:
 - Use loops for which parallel code is generated automatically and loops created with the PARALLEL DO statement to provide automatic work load balancing.
 - Use the following statements to balance work loads during processing as parallel threads are completed:


```
SCHEDULE
WAIT FOR ANY TASK
WAIT FOR TASK.
```
 - Use library lock and event services to program algorithms that dynamically balance your program.
 - Review execution using the Parallel Trace Facility.

Using the Parallel Feature—Examples

The following sections consist of examples that further illustrate how to use and take advantage of parallel processing. The sections show:

Converting a serial program to a parallel program
 Using the parallel language constructs
 Improving performance of parallel processing.

Converting a Serial Program

The subprogram in Figure 96 on page 334 computes the average test score of students in one of several classes. Each student's highest and lowest scores are excluded, the remaining scores are averaged, and the result is placed in a two dimensional array of students and classes.

```

SUBROUTINE GRADES(IDCLASS)
COMMON /STAVE/ SAVG

:

DIMENSION SCORES(NUMTEST,NUMSTU), SAVG(NUMSTU, NUMCLS),
+ SMIN(NUMSTU), SMAX(NUMSTU)
READ (10+IDCLASS,99) SCORES
99  FORMAT .....
DO 40 I=1, NUMSTU
    SAVG(I,IDCLASS) = 0.0
    SMIN(I) = 100.0
    SMAX(I) = 0.0

C Find the sum of the test scores for student I in the class
C called IDCLASS

    DO 10 J=1, NUMTEST
        SAVG(I, IDCLASS) = SAVG(I, IDCLASS) + SCORES(J, I)
10  ENDDO

C Find the minimum grade for student I

    DO 20 J=1, NUMTEST
        SMIN(I) = MIN(SMIN(I), SCORES(J,I))
20  ENDDO

C Find the maximum grade for student I

    DO 30 J=1, NUMTEST
        SMAX(I) = MAX(SMAX(I), SCORES(J,I))
30  ENDDO

C Calculate the average test score for student I

    SAVG(I, IDCLASS) = (SAVG(I, IDCLASS) - SMAX(I) - SMIN(I))
+    / (NUMTEST - 2)
40  CONTINUE
END

```

Figure 96. A Serial Program to Rewrite with Parallel Constructs

The code in Figure 96 can be rewritten to use parallel constructs, as shown in Figure 97 on page 335. Because the DO 40 loop iterations in Figure 96 do not refer to previous iterations (each student's grades are independent of the other student's grades) the DO can be replaced with a PARALLEL DO. Replacing the DO with a PARALLEL DO, as shown in Figure 97 on page 335, allows different virtual processors to calculate averages for different students simultaneously.

With the exception of the DO variable J, the inner loops, DO 10, DO 20, and DO 30 are also computationally independent. Using the PARALLEL SECTIONS construct in Figure 97 on page 335 shows that each of these sections is provided with a private DO variable. Each section can then safely be run in parallel with the other sections.

```

@PROCESS PAR(LANG) OPT(3)
  SUBROUTINE GRADES(IDCLASS)

    :

    DIMENSION SCORES(NUMTEST,NUMSTU), SAVG(NUMSTU, NUMCLS),
+   SMIN(NUMSTU), SMAX(NUMSTU)
    READ (10+IDCLASS,99) SCORES
99   FORMAT .....
    PARALLEL DO 40 I=1, NUMSTU
      SAVG(I,IDCLASS) = 0.0
      SMIN(I) = 100.0
      SMAX(I) = 0.0

    PARALLEL SECTIONS

    C Find the sum of the test scores for student I
    C in the class called IDCLASS

      SECTION
      DO 10 J=1, NUMTEST
        SAVG(I, IDCLASS) = SAVG(I, IDCLASS) + SCORES(J, I)
10      ENDDO

    C Find the minimum grade for student I
      SECTION
      DO 20 J=1, NUMTEST
        SMIN(I) = MIN(SMIN(I), SCORES(J,I))
20      ENDDO

    C Find the maximum grade for student I
      SECTION
      DO 30 J=1, NUMTEST
        SMAX(I) = MAX(SMAX(I), SCORES(J,I))
30      ENDDO

    END SECTIONS

    C Calculate the average test score for student I
      SAVG(I, IDCLASS) = (SAVG(I, IDCLASS) - SMAX(I) - SMIN(I))
+      / (NUMTEST - 2)
40    CONTINUE
    END

```

Figure 97. Nested Parallel Constructs

Using Parallel Language Constructs

The program in Figure 98 consists of program samples used throughout this chapter. It computes simple matrix sums using parallel sections and a parallel loop. The matrix is initialized using parallel calls.

```

@PROCESS PAR(LANG NOAUTO REP) OPT(3)
C      PCALL--->SUB1(INIT)
C      L__>SUB2(INIT)
C
C      TASKS--->PLOOP(SUM)
C      L__>PSECT(SUM)
C
C      PROGRAM MATSUM
C
C      INTEGER*4 ITASK1, ITASK2
C      COMMON /M/M(500,500)
C
C      PARALLEL CALL SUB1
C      PARALLEL CALL SUB2
C      WAIT FOR ALL CALLS
C
C      ORIGINATE ANY TASK ITASK1
C      ORIGINATE ANY TASK ITASK2
C      SCHEDULE TASK ITASK1, SHARING(M), CALLING PLOOP(PTOT)
C      SCHEDULE TASK ITASK2, SHARING(M), CALLING PSECT(STOT)
C      WAIT FOR ALL TASKS
C      TERMINATE TASK ITASK1
C      TERMINATE TASK ITASK2
C
C      PRINT*, 'MATRIX SUM FINISHED'
C
C      END
@PROCESS PAR(LANG NOAUTO REP) OPT(3)
C      SUBROUTINE PLOOP(SUM)
C      COMMON /M/M(500,500)
C
C      SUM = 0.0
C      PARALLEL DO I = 1, 500
C      LOCAL PTOT
C      DOBEFORE
C      PTOT = 0.0
C      DOEVERY
C      DO J = 1, 500
C      PTOT = PTOT + M(J,I)
C      ENDDO
C      DOAFTER LOCK
C      SUM = SUM + PTOT
C      ENDDO
C      RETURN
C      END

```

Figure 98 (Part 1 of 2). Example of Coded Parallel Language Constructs

```

@PROCESS PAR(LANG AUTO REP) OPT(3)
  SUBROUTINE PSECT(STOT)
  COMMON /M/M(500,500)
C
  PARALLEL SECTIONS
  LOCAL SUM
  SECTION 1
    SUM = 0.0
    DO 100 I = 1, 125
    DO 100 J = 1, 500
      SUM = SUM + M(I,J)
100  ENDDO
    SUM1=SUM
  SECTION 2
    SUM = 0.0
    DO 200 I = 126, 250
    DO 200 J = 1, 500
      SUM = SUM + M(I,J)
200  ENDDO
    SUM2=SUM
  SECTION 3
    SUM = 0.0
    DO 300 I = 251, 375
    DO 300 J = 1, 500
      SUM = SUM + M(I,J)
300  ENDDO
    SUM3=SUM
  SECTION 4
    SUM = 0.0
    DO 400 I = 376, 500
    DO 400 J = 1, 500
      SUM = SUM + M(I,J)
400  ENDDO
    SUM4=SUM
  END SECTIONS
  STOT = SUM1 + SUM2 + SUM3 + SUM4
C
  RETURN
END

@PROCESS PAR(LANG NOAUTO REP) OPT(3)
  SUBROUTINE SUB1
  COMMON /M/M(500,500)
  DO 50 I = 1, 250
  DO 50 J = 1, 250
    M(I,J) = SQRT(FLOAT(I+J))
50  CONTINUE
  RETURN
  END

@PROCESS PAR(LANG NOAUTO REP) OPT(3)
  SUBROUTINE SUB2
  COMMON /M/M(500,500)
  DO 90 I = 251, 500
  DO 90 J = 251, 500
    M(I,J) = I+J
90  CONTINUE
  RETURN
  END

```

Figure 98 (Part 2 of 2). Example of Coded Parallel Language Constructs

Improving Performance of Parallel and Vector Processing

The following set of examples show various ways to perform a matrix multiply. For information about compiler reports and a description of their various parts, see “Producing Compiler Reports” on page 344.

Figure 99 shows the matrix multiply in its simplest form. Figure 100 is the compiler report showing how the code in Figure 99 is vectorized.

```

@PROCESS DC(AC) VEC(REP(LIST)) OPT(3)
PROGRAM MATMUL
PARAMETER (N1=500, N2=1500, N3=200)
REAL*8 A,B,C
COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
.
C INITIALIZE VARIABLES
.
DO 20 I = 1, N1
DO 20 K = 1, N3
C(I,K) = 0.D0
DO 20 J = 1, N2
20 C(I,K) = C(I,K)+(B(J,K)*A(I,J))
.
.
.
END

```

Figure 99. Simple Matrix Multiply

```

VECT +----- DO 20 I = 1, N1
SCAL +----- DO 20 K = 1, N3
      |          C(I,K) = 0.D0
SCAL | +----- DO 20 J = 1, N2
      | | 20      C(I,K) = C(I,K)+(B(J,K)*A(I,J))
      | |
      | |
      | |
      | |

```

Figure 100. Compiler Report for Simple Matrix Multiply

Now you want to generate parallel code for the matrix multiply without degrading the vector performance of the enhanced version of the code. Figure 100 shows that the algorithm is vectorized over the I loop and that vector registers are obtained for reuse by storing C(I,K) only outside of the J loop. The K loop is thus the prime candidate for parallel code generation, and Figure 101 on page 339 shows how you can perform dynamic load balancing by using automatically generated parallel DO loops. If you reverse the ordering of the I and K loops of the matrix multiply, the K loop becomes available for automatic parallel code generation while the I and J loops maintain their relationship to each other for generating efficient vector code. Figure 102 on page 339 shows the compiler report for the code in Figure 101 on page 339. The compiler will automatically determine the iteration chunk by the number of processors and the computational intensity of the loop.

```

@PROCESS DC(AC) PAR(AUTO REPORT(LIST)) VEC
PROGRAM MATMUL
PARAMETER (N1=500, N2=1500, N3=200)
REAL*8 A,B,C
COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
.
C INITIALIZE VARIABLES
.
.
DO 20 K = 1, N3
DO 20 I = 1, N1
C(I,K) = 0.D0
DO 20 J = 1, N2
20 C(I,K) = C(I,K)+(B(J,K)*A(I,J))
.
.
.
END

```

Figure 101. Matrix Multiply With Parallel DO Loop

```

PARA +----- DO 20 K = 1, N3
VECT | +----- DO 20 I = 1, N1
    | |          C(I,K) = 0.D0
SCAL | +----- DO 20 J = 1, N2
    | |          C(I,K) = C(I,K)+(B(J,K)*A(I,J))
    | |          20
    | |
    | |
    | |
    | |

```

Figure 102. Compiler Report for Matrix Multiply With Parallel DO Loop

Figure 103 on page 340 shows how you can perform dynamic load balancing with the PARALLEL CALL statement. For this technique, the K loop of the matrix multiply is removed from the MLT subroutine and is used as the limit for the loop containing the PARALLEL CALL statement. K is passed as an argument to MLT. Each invocation of the MLT subroutine runs one iteration of the K loop, giving a fine granularity with N3 parallel calls.

Note: In this example, you maintain computational independence because different parts of the same common block are being accessed and all common blocks can be shared.

Figure 104 on page 340 is the compiler report for the code in Figure 103 on page 340 which shows that you have maintained the vectorization of the algorithm.

```

@PROCESS DC(AC) VEC OPT(3) PAR
  PROGRAM MATMUL
  PARAMETER ( N1=500, N2=1500, N3=200)
  REAL*8 A,B,C
  INTEGER ITASK(N3), KNI(N3), IDENT(N3)
  COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
  .
C  INITIALIZE VARIABLES
  .
  .
  DO 20 K = 1, N3
20    PARALLEL CALL MLT(K)
    WAIT FOR ALL CALLS
    .
    .
  END
@PROCESS DC(AC) VEC(REP(LIST)) OPT(3) PAR
  SUBROUTINE MLT(K)
  PARAMETER (N1=500, N2=1500, N3=200)
  REAL*8 A,B,C
  COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
  DO 20 I = 1, N1
    C(I,K) = 0.D0
    DO 20 J = 1, N2
20      C(I,K) = C(I,K)+(B(J,K)*A(I,J))
    RETURN
  END

```

Figure 103. Matrix Multiply With PARALLEL CALL

```

VECT +----- DO 20 I = 1, N1
      |          C(I,K) = 0.D0
SCAL +----- DO 20 J = 1, N2
      |          C(I,K) = C(I,K)+(B(J,K)*A(I,J))
      |          20
      |

```

Figure 104. Compiler Report for Matrix Multiply With PARALLEL CALL

Although multiple parallel tasks are not required to do the matrix multiply problem, for the purpose of illustration we are now going to show how to use parallel tasks to do matrix multiply. Figure 105 shows how you can achieve a coarser granularity using the SCHEDULE and WAIT FOR statements. You place the actual enhanced matrix multiply into a subroutine named MLT, which you schedule to run in parallel with the SCHEDULE statement. The main program sets the number of tasks equal to the number of processors⁹ specified. The arguments passed to subroutine MLT show the subroutine's position in the scheduling loop and the number of instances of MLT being scheduled; each instance of MLT uses this information to determine which iterations of the K loop it should process. Figure 106 on page 341 is the compiler report which shows that you have maintained the vectorization of the algorithm.

The parallelization technique in Figure 105 on page 341 works well in a dedicated run-time environment, where each parallel thread scheduled is assured of being assigned to a real processor without interference. However, if the program is running in an environment where there is contention for the real processors, the

⁹ Figure 105 on page 341 is designed to run on a number of processors that divides evenly into N3.

effectiveness of parallel processing is determined by the parallel thread which receives the lowest level of service.

```

@PROCESS DC(AC) VEC OPT(3) PAR
PROGRAM MATMUL
PARAMETER (N1=500, N2=1500, N3=200)
REAL*8 A,B,C
INTEGER ITASK(N1), KNI(N3)
COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
NTASK = NPROCS()
DO 10 I = 1, NTASK
    10    ORIGINATE ANY TASK ITASK(I)
    .
C INITIALIZE VARIABLES
    .
    .
    DO 20 K = 1, NTASK
    20    SCHEDULE TASK ITASK(K),
    +    SHARING(AC),
    +    CALLING MLT(K, NTASK)
    WAIT FOR ALL TASKS
    .
    .
    .
    DO 40 I = 1, NTASK
    40    TERMINATE TASK ITASK(I)
    END
@PROCESS DC(AC) VEC(REP(LIST)) OPT(3) PAR
SUBROUTINE MLT(KN, KT)
PARAMETER (N1=500, N2=1500, N3=200)
REAL*8 A,B,C
COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
KUB = N3*KN/KT
KLB = 1+N3*(KN-1)/KT
DO 20 I = 1, N1
    DO 20 K = KLB, KUB
        C(I,K) = 0.D0
        DO 20 J = 1, N2
            20    C(I,K) = C(I,K)+(B(J,K)*A(I,J))
        RETURN
    END

```

Figure 105. Matrix Multiply With SCHEDULE

```

VECT +----- DO 20 I = 1, N1
SCAL +----- DO 20 K = KLB, KUB
      |         C(I,K) = 0.0D0
SCAL |         DO 20 J = 1, N2
      |         C(I,K) = C(I,K)+(B(J,K)*A(I,J))
      |         _____ 20
      |         _____
      |         _____

```

Figure 106. Compiler Report for Matrix Multiply With SCHEDULE

Figure 107 on page 342 shows how you can perform dynamic load balancing by scheduling a limited number of parallel tasks and using a parallel lock for obtaining and updating the K loop induction variable which is shared between the tasks.

Figure 108 on page 342 is the compiler report for the code in Figure 107 on page 342 which again shows that you have maintained the vectorization of the algorithm.

```

@PROCESS DC(AC) VEC OPT(3) PAR
  PROGRAM MATMUL
  PARAMETER (N1=500, N2=1500, N3=200)
  REAL*8 A,B,C
  INTEGER ITASK(N1)
  COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
  NTASK = NPROCS()
  DO 10 I=1, NTASK
10    ORIGINATE ANY TASK ITASK(I)
    CALL PLORIG(ILOCK)
    .
C  INITIALIZE VARIABLES
    .
    .
    KN=1
    DO 20 I=1, NTASK
20    SCHEDULE TASK ITASK(I),
      x  SHARING(AC),
      x  CALLING MLT(ILOCK, KN)
    WAIT FOR ALL TASKS
    .
    .
    .
    CALL PLTERM(ILOCK)
    DO 40 I=1, NTASK
40    TERMINATE TASK ITASK(I)
    END
@PROCESS DC(AC) VEC OPT(3) PAR(REP(LIST))
  SUBROUTINE MLT(ILOCK, KN)
  PARAMETER (N1=500, N2=1500, N3=200)
  REAL*8 A,B,C
  COMMON /AC/ A(N1,N2), B(N2,N3), C(N1,N3)
  GOTO 40
10  DO 20 I = 1, N1
    C(I,K) = 0.D0
    DO 20 J = 1, N2
20    C(I,K) = C(I,K)+(B(J,K)*A(I,J))
40  CALL PLLOCK(ILOCK,0,KN)
    K = KN
    KN = KN+1
    CALL PLFREE(ILOCK, KN)
    IF(K .LE. N3) GOTO 10
  RETURN
  END

```

Figure 107. Matrix Multiply With SCHEDULE and Locks

```

VECT +-----10    DO 20 I = 1, N1
    |               C(I,K) = 0.D0
SCAL | +-----    DO 20 J = 1,N2
    | |_____20    C(I,K) = C(I,K)+(B(J,K)*A(I,J))
    |
    |

```

Figure 108. Compiler Report for Matrix Multiply With SCHEDULE and Locks

Chapter 18. Aids for Tuning Your Program

Producing Compiler Reports	344
Displaying a Report on a Terminal	344
Printing Compiler Reports	345
Using Parallel and Vector Directives	353
Interactions between Parallel and Vector Directives	354
Specifying Parallel and Vector Directives	355
Local and Global Parallel and Vector Directives	355
Rules for Specifying Multiple Parallel and Vector Directives	356
ASSUME COUNT Directive	356
Examples of ASSUME COUNT	357
IGNORE Directive	358
Examples of IGNORE	360
PREFER Directive	364
Examples of PREFER	365
Verifying Correct Application of Directives	367
Gathering Run-Time Statistics	368

VS FORTRAN Version 2 provides compiler reports and parallel and vector directives you can use to tune your source program for vector and/or parallel processing. Compiler reports allow you to study the type of code generated (vector, scalar, parallel, or serial). Depending on what suboptions you specify, a report will be generated that:

- Lists the results of the compiler's source code analysis, and shows which loops were chosen to run in parallel, vector or both
- Explains why parallel or vector code was or was not generated
- Lists compile-time statistics for parallel statements in a table
- Lists compile-time statistics on the length and stride of each DO loop in the source program.

Parallel and vector directives allow you to override certain compiler decisions regarding the automatic generation of vector and parallel code.

For vector code only, run-time statistics can also be generated by interactive debug.

To aid in debugging your parallel program, you can compile your program with the PARALLEL compile-time option and then run it on a single virtual processor. This allows you to develop and debug your program in an environment where the errors produced are reproducible. Note that running your parallel program on a single virtual processor is not the same as running a program serially, because of the existence of parallel language statements and synchronization subroutines in your program.

Producing Compiler Reports

This section describes the main features of the reports produced using the REPORT suboption on the PARALLEL or VECTOR compile-time option. The PARALLEL compile-time option is discussed on page 31. The VECTOR compile-time option is discussed on page 36.

Reports can contain several different sections and can be displayed on a terminal or placed in a print file, depending on what you specify on the PARALLEL or VECTOR option. The reports show the compiler's analysis of the source program and may not exactly indicate how the program will run.

If you specify the REPORT(LIST) or REPORT(XLIST) suboption of the PARALLEL or VECTOR option, the following flags are printed to the left of affected statements to indicate the type of code generated:

ELIG	The loop is eligible for parallel or vector processing (if VECTOR is specified), but was chosen to run in serial and scalar modes. A message indicates if the loop is eligible for parallel processing, vector processing, or both. (XLIST only)
PARA	The loop was chosen to run in parallel and scalar modes.
PAVE	The loop was selected for both vector and parallel processing. In this case, the number of iterations of the DO loop given to a virtual processor as a chunk is usually a multiple of the vector section size.
RECR	The loop was not chosen to run in parallel or vector modes because it carries a recurrence. (XLIST only)
SCAL	The loop was eligible for analysis and chosen to run in scalar mode. If both the PARALLEL and VECTOR compile-time options were specified, this flag also indicates that the loop was chosen to run in serial mode. (LIST only)
SERI	The loop was analyzed and chosen to run in serial mode, or a PARALLEL DO or PARALLEL SECTIONS construct was serialized. (LIST only)
VECT	The loop was chosen to run in vector and serial modes.
UNAN	The loop could not be analyzed.
UNSP	The loop could not be run in vector or parallel mode because it contains operations not supported by the compiler, the vector hardware, or the parallel feature.

Displaying a Report on a Terminal

To display a report on your terminal, specify the REPORT(TERM) suboption on the PARALLEL or VECTOR option. The TRMFLG option must also be in effect.

The terminal report, an example of which is shown in Figure 109 on page 345, displays only selected portions of the parallel or vector program. These portions include all DO statements and all statements contained within analyzable loops. The statements are displayed in the same order as that of the generated object code.

[illegible]

THE DO-LOOPS HAVE BEEN PROCESSED AS INDICATED.

Figure 109. Sample Compiler Report as Displayed on a Terminal

To the far left of the DO statements are flags indicating the following:

PARA Loop was selected for parallel processing.

PAVE Loop was selected for both vector and parallel processing.

VECT The loop was chosen to run in vector and serial modes.

SCAL The loop was eligible for analysis and chosen to run in scalar mode. If both the **PARALLEL** and **VECTOR** compile-time options were specified, this flag also indicates that the loop was chosen to run in serial mode.

UNAN The loop could not be analyzed.

Also to the left of the statements are nested brackets that mark the beginning and end of each loop in every block of loops that was analyzable for vectorization. The brackets indicate how the statements were grouped after the vectorization selection (see “Statements and Constructs Preventing Vectorization” on page 277). These brackets are the only way to accurately determine the relative nesting of the statements in the compiler report.

Printing Compiler Reports

Figure 110 on page 346 shows a printed listing with a compiler report produced by specifying the following suboptions on the VECTOR and PARALLEL compile-time options. In addition, the IGNORE directive, discussed on page 358, was specified.

```
VECTOR(REPORT(LIST XLIST SLIST STAT))
PARALLEL(REPORT(LIST XLIST SLIST STAT))
```

The sections of the report will always be printed in the order shown in the figure. For information on `VECTOR(REPORT)` and `PARALLEL(REPORT)` compile-time suboptions, see pages 31 and 36 in Chapter 3, “Using the Compile-Time Options.”

Compiler Reports

1
LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10,1993 09:42:18 PAGE: 1
REQUESTED OPTIONS (EXECUTE): VECTOR,OPT(3),PAR(AUTO LANG)
REQUESTED OPTIONS (PROCESS): VEC(REP(TE LI XL SL ST)) OPT(3) DIR('DIR') NOSOURCE
OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK NOSOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT SDUMP(ISN)
NOSXM VECTOR IL(DIM) NOTEST NODC NOICA DIRECTIVE NOBCS NOSAA PARALLEL NOSAVE NOTABS
OPT(3) LONGLVL(77) NOFIPS FLAG(1) AUTODBL(NONE) LINECOUNT(60) CHARLEN(500)
VECTOR: NOIVA INTRINSIC REDUCTION SIZE(ANY) REPORT(LIST TERM XLIST SLIST STAT)
PARALLEL: AUTOMATIC LANGUAGE NOREDUCTION REPORT(LIST TERM XLIST SLIST STAT)
ISN 8 SUM=0.0
ERROR 1976(I) AN IGNORE DIRECTIVE HAS BEEN SPECIFIED. IF THE INFORMATION PROVIDED BY THIS DIRECTIVE IS INCORRECT,
INVALID VECTORIZATION MAY OCCUR AND WRONG RESULTS MAY BE PRODUCED AT EXECUTION TIME.

2
LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 NAME:EXP01 PAGE: 2
REPORT(LIST) VECTORIZATION/PARALLELIZATION ANALYSIS
ISN FLAG NESTING *.....1.....2.....3.....4.....5.....6.....7.*.....8
0001 SUBROUTINE EXP01(B,IMAX,JMAX,N)
0002 REAL*8 A(100,100)
0003 REAL*8 B(-IMAX:IMAX,-JMAX:JMAX)
0004 REAL*8 C(100,100)
0005 REAL*8 DIFSUM(100),SUM
0006 INTEGER*4 IMAX,JMAX,N

0007 SCAL +----- C
| DO 200 J=1,100
| *DIR IGNORE RECRDEPS(B)
0009 VECT +----- DO 100 I=1,100,2
| | A(I,J) = B(I+N,J+N) + B(I+N,J-N)
0010 | | C(I,J) = B(I-N,J+N) + B(I-N,J-N)
0011 | | B(I,J) = A(I,J) + C(I,J)
0012 | |

0007 PARA +----- DO 200 J=1,100
0008 | SUM = 0.0
| *DIR IGNORE RECRDEPS(B)
0009 VECT +----- DO 100 I=1,100,2
| | 100 SUM = SUM + ABS(A(I,J)-C(I,J))
0013 | | 200 DIFSUM(J) = SUM
0014 | C

0015 SCAL +----- DO 300 J=1,100
0016 | 300 IF(DIFSUM(J).GT.1.0) PRINT*,J,DIFSUM(J)
0018 | RETURN
0019 | END

Figure 110 (Part 1 of 3). Printed Listing Including Compiler Report

3		LEVEL 2.6.0 (NOV 1993)	VS FORTRAN	NOV 10, 1993 09:42:18 NAME:EXP01	PAGE: 3
		REPORT(XLIST) VECTORIZATION/PARALLELIZATION ANALYSIS			
ISN	FLAG	NESTING	*.....1.....2.....3.....4.....5.....6.....7.....8	MESSAGES	
0001			SUBROUTINE EXP01(B,IMAX,JMAX,N)		
0002			REAL*8 A(100,100)		
0003			REAL*8 B(-IMAX:IMAX,-JMAX:JMAX)		
0004			REAL*8 C(100,100)		
0005			REAL*8 DIFSUM(100),SUM		
0006			INTEGER*4 IMAX,JMAX,N		
		C			
0007	PARA	+-----	DO 200 J=1,100		
0008			SUM = 0.0		
		*DIR	IGNORE RECRDEPS(B)		
0009	VECT	+-----	DO 100 I=1,100,2	"IGNORE RECRDEPS" USED	572
0013			SUM = SUM + ABS(A(I,J)-C(I,J))	VECTOR SUM REDUCTION	-50
0014				UNSUPPORTABLE DEPENDENCE	335
			200 DIFSUM(J) = SUM		
0007	RECR	+-----	DO 200 J=1,100	POSSIBLE RECURRENCE	520
		*DIR	IGNORE RECRDEPS(B)		
0009	VECT	+-----	DO 100 I=1,100,2	"IGNORE RECRDEPS" USED	572
0010			A(I,J) = B(I+N,J+N) + B(I+N,J-N)	OFFSET UNKNOWN	518
				INTERCHANGE PREVENTING DEP	523
				POTENTIAL RECRDEP ELIMINATE	577
0011			C(I,J) = B(I-N,J+N) + B(I-N,J-N)	OFFSET UNKNOWN	518
				INTERCHANGE PREVENTING DEP	523
				POTENTIAL RECRDEP ELIMINATE	577
0012			B(I,J) = A(I,J) + C(I,J)	OFFSET UNKNOWN	518
				INTERCHANGE PREVENTING DEP	523
				POTENTIAL RECRDEP ELIMINATE	577
		C			
0015	UNSP	+-----	DO 300 J=1,100		
0016			IF(DIFSUM(J).GT.1.0) PRINT*,J,DIFSUM(J)	I/O OPERATIONS	541
				CONDITIONAL SCALAR CODE	558
0018			RETURN		
0019			END		

4		LEVEL 2.6.0 (NOV 1993)	VS FORTRAN	NOV 10, 1993 09:42:18 NAME:EXP01	PAGE: 4
		REPORT(SLIST) VECTORIZATION/PARALLELIZATION ANALYSIS			
IF DO	ISN	*.....1.....2.....3.....4.....5.....6.....7.....8	MESSAGES		
	1		SUBROUTINE EXP01(B,IMAX,JMAX,N)		
	2		REAL*8 A(100,100)		
	3		REAL*8 B(-IMAX:IMAX,-JMAX:JMAX)		
	4		REAL*8 C(100,100)		
	5		REAL*8 DIFSUM(100),SUM		
	6		INTEGER*4 IMAX,JMAX,N		
		C			
P1	7		DO 200 J=1,100	POSSIBLE RECURRENCE	520
				SCALAR FASTER THAN VECTOR	-48
				POSSIBLE SYNCHRONIZATION	400
p1	1	8	SUM = 0.0		
		*DIR	IGNORE RECRDEPS(B)		
V2	1	9	DO 100 I=1,100,2	"IGNORE RECRDEPS" USED	572
				SERIAL FASTER THAN PARALLEL	348
v2	2	10	A(I,J) = B(I+N,J+N) + B(I+N,J-N)	OFFSET UNKNOWN	518
				INTERCHANGE PREVENTING DEP	523
				POTENTIAL RECRDEP ELIMINATE	577
v2	2	11	C(I,J) = B(I-N,J+N) + B(I-N,J-N)	OFFSET UNKNOWN	518
				INTERCHANGE PREVENTING DEP	523
				POTENTIAL RECRDEP ELIMINATE	577
v2	2	12	B(I,J) = A(I,J) + C(I,J)	OFFSET UNKNOWN	518
				INTERCHANGE PREVENTING DEP	523
				POTENTIAL RECRDEP ELIMINATE	577
p1v2	2	13	100 SUM = SUM + ABS(A(I,J)-C(I,J))	VECTOR SUM REDUCTION	-50
				UNSUPPORTABLE DEPENDENCE	335
p1	1	14	200 DIFSUM(J) = SUM		
		C			
S1	15		DO 300 J=1,100		
s	1	16	300 IF(DIFSUM(J).GT.1.0) PRINT*,J,DIFSUM(J)	I/O OPERATIONS	541
				CONDITIONAL SCALAR CODE	558
		18	RETURN		
		19	END		

Figure 110 (Part 2 of 3). Printed Listing Including Compiler Report

Compiler Reports

5
LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 NAME:EXP01 PAGE: 5
NUMBER ISN FLAG VS FORTRAN VECTOR/PARALLEL REPORT MESSAGES

ILX0520I 0007 RECR THIS LOOP COULD NOT BE PARALLELIZED OR VECTORIZED BECAUSE OF A POSSIBLE RECURRENCE INVOLVING THE ARRAY(S) "B". THE INFORMATION NEEDED TO DETERMINE WHETHER OR NOT THE RECURRENCE EXISTS WAS NOT AVAILABLE TO THE COMPILER. IF NO RECURRENCE EXISTS, PARALLELIZATION AND VECTORIZATION CAN BE ACHIEVED WITH AN IGNORE DIRECTIVE.

ILX0148I 0007 ELIG CODE THAT WAS ELIGIBLE TO EXECUTE IN VECTOR MODE WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SCALAR.

ILX0400I 0007 RECR THIS LOOP COULD NOT BE PARALLELIZED BECAUSE SYNCHRONIZATION IS POSSIBLY REQUIRED FOR THE LOOP CARRIED DEPENDENCE(S) INVOLVING THE VARIABLE(S) "B". THE INFORMATION NEEDED TO DETERMINE WHETHER OR NOT SYNCHRONIZATION IS REQUIRED WAS NOT AVAILABLE TO THE COMPILER. IF SYNCHRONIZATION IS NOT REQUIRED, PARALLELIZATION CAN BE ACHIEVED WITH AN IGNORE DIRECTIVE.

ILX0572I 0009 PVDR AN "IGNORE RECRDEPS" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP FOR PARALLELIZATION AND VECTORIZATION ANALYSIS.

ILX0348I 0009 ELIG CODE THAT WAS ELIGIBLE TO EXECUTE IN PARALLEL MODE WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SERIAL.

ILX0518I 0010-0012 RECR THE OFFSET NEEDED TO ADDRESS THE ARRAY(S) "B" COULD NOT BE ANALYZED FOR PARALLELIZATION OR VECTORIZATION. THERE MAY BE AN UNKNOWN TERM IN A SUBSCRIPT OR IN A LOOP LOWER BOUND, OR THE ARRAY(S) MAY HAVE ADJUSTABLE DIMENSIONS. THE COMPILER ASSUMES DEPENDENCES FOR THE ARRAYS AT NESTING LEVEL(S) "1".

ILX0523I 0010-0012 RECR THE ARRAY(S) "B" CARRY FORWARD DEPENDENCES AT NESTING LEVEL(S) "1" THAT MAY BE INTERCHANGE PREVENTING. THESE LOOP(S) CAN NOT BE VECTORIZED.

ILX0577W 0010-0012 PVDR POTENTIAL BACKWARD DEPENDENCE(S) INVOLVING THE ARRAY(S) "B" HAVE BEEN IGNORED FOR PARALLELIZATION AND VECTORIZATION BECAUSE OF AN "IGNORE RECRDEPS" DIRECTIVE APPLIED TO THE LOOP(S) AT NESTING LEVEL(S) "2".

ILX0150W 0013 VECT VECTORIZATION WAS DONE USING SUM OR PRODUCT REDUCTION ON THE VARIABLE(S) "SUM". RESULTS MAY DIFFER FROM SCALAR CODE.

ILX0335I 0013 UNSP THIS CODE IS CONSIDERED UNSUPPORTABLE FOR PARALLELIZATION BECAUSE IT IS LINKED TO SOME UNSUPPORTABLE STATEMENT(S) THROUGH MUTUAL DEPENDENCES.

ILX0541I 0016 UNSP I/O OPERATIONS ARE NOT SUPPORTED FOR PARALLELIZATION OR VECTORIZATION.

ILX0558I 0016 SCAL THIS CODE IS CONDITIONALLY EXECUTED. THE PARALLEL/VECTOR REPORT LISTING MAY FAIL TO INDICATE THE BRANCH STATEMENT(S) THAT AFFECT THE EXECUTION OF THIS REGION.

6
LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 NAME:EXP01 PAGE: 6
VECTOR/PARALLEL STATISTICS TABLE

<u>ISN</u>	<u>ARRAY/INDUCTION</u>	<u>LEVEL 1</u>	<u>LEVEL 2</u>
7	J	COUNT=100	
9	I		COUNT=50
10	B	65?	2V
	B	65?	2V
	A	100	2V
11	B	65?	2V
	B	65?	2V
	C	100	2V
12	B	65?	2V
	A	100	2V
	C	100	2V
13	A	100P	2V
	C	100P	2V
14	DIFSUM	1P	
15	J	COUNT=100	
16	DIFSUM	1	
	DIFSUM	1	

7
LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 NAME:EXP01 PAGE: 7
TABLE OF DEPENDENCES ELIMINATED BY IGNORE DIRECTIVES

<u>DIRECTIVE</u>	<u>ARRAY NAME</u>	<u>FROM</u>	<u>TO</u>	<u>DEP</u>	<u>AFFECTED</u>	<u>LOOP</u>
<u>TYPE</u>	<u>(EQUIV NAME)</u>	<u>ISN</u>	<u>ISN</u>	<u>TYPE</u>	<u>DIMENSIONS</u>	<u>VARIABLES</u>
BACKDEP	B	12	10	TRUE	1	I
BACKDEP	B	12	11	TRUE	1	I

8
LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 NAME:EXP01 PAGE: 8
NUMBER MODULE LEVEL ISN VS FORTRAN ERROR MESSAGES

ILX1976I ASRT 0(I) 8 AN IGNORE DIRECTIVE HAS BEEN SPECIFIED. IF THE INFORMATION PROVIDED BY THIS DIRECTIVE IS INCORRECT, INVALID VECTORIZATION MAY OCCUR AND WRONG RESULTS MAY BE PRODUCED AT EXECUTION TIME.

STATISTICS SOURCE STATEMENTS: 18, PROGRAM SIZE: 163460 BYTES, PROGRAM NAME: EXP01 PAGE: 1.
STATISTICS 1 DIAGNOSTIC GENERATED. SEVERITY CODE IS 0.
EXP01 END OF COMPILATION 1 *****

TIME STAMP: 90.24909.42.18

Figure 110 (Part 3 of 3). Printed Listing Including Compiler Report

The listing shown in Figure 110 on page 346 has the following sections:

- 1** The options requested and the options in effect.
- 2** A vector and parallel processing analysis report produced by REPORT(LIST) on the VECTOR and PARALLEL compile-time options.

This report offers an overview of the transformations performed. From this report you can find out how the statements and loops in a program were restructured and which loops were chosen for vector and parallel processing.

Note that as a result of optimizations which alter part of the program, when the compiler report is constructed, some of the information does not appear on the report or may be in another location. This condition results, for example, when GO TO statements occur in a loop. It can also happen with statements lacking array references.

The report contains the following information:

- Internal statement numbers. These are helpful for comparing statements in the compiler report to statements in the source program listing or pseudo-assembler listing.
- Flags next to DO statements, indicating the following:

PARA	The loop was chosen to run in parallel and scalar modes.
PAVE	The loop was selected for both vector and parallel processing.
SERI	The loop was analyzed and chosen to run in serial mode, or a PARALLEL DO or PARALLEL SECTIONS construct was serialized.
VECT	The loop was chosen to run in vector and serial modes.
SCAL	The loop was eligible for analysis and chosen to run in scalar mode. If both the PARALLEL and VECTOR compile-time options were specified, this flag also indicates that the loop was chosen to run in serial mode.
UNAN	The loop could not be analyzed.
- Nested brackets marking the beginning and end of each loop in every block of loops that was analyzable for vector or parallel processing. The brackets indicate how the statements were grouped after vector and parallel processing selection (see “Statements and Constructs Preventing Vectorization” on page 277). These brackets are the only way to accurately determine the relative nesting of the statements in the compiler report.
- Source statements.

- 3** An extended vector and parallel analysis report produced by REPORT(XLIST) on the VECTOR and PARALLEL compile-time options.

This report is similar to that produced by REPORT(LIST) but gives you diagnostic messages as well as more detailed information about why loops were not processed in vector or parallel modes. It also differs from the REPORT(LIST) report in that statements and loops may be structured differently. For REPORT(LIST), the loop structure corresponds to the structure of the generated code, whereas for REPORT(XLIST), each loop that appears in the listing corresponds to a strongly connected region of statements.

For example, it is possible that two statements will appear in different loops in the REPORT(XLIST) output, but will be in the same loop in the REPORT(LIST) output. This means that those two statements were independent of one

another from the point of view of vector and parallel analysis, but were grouped together into a single loop for run-time processing.

The report contains the following information:

- Internal statement numbers. These are helpful for comparing statements in the compiler report to statements in the source program listing.
- Flags next to DO statements, indicating the following:
 - PARA** The loop was chosen to run in parallel and scalar modes.
 - PAVE** The loop was selected for both vector and parallel processing.
 - ELIG** The loop is eligible for parallel or vector processing (if VECTOR is specified), but was chosen to run in serial and scalar modes. A message indicates if the loop is eligible for parallel processing, vector processing, or both.
 - VECT** The loop was chosen to run in vector and serial modes.
 - UNSP** The loop could not be run in vector or parallel mode because it contains operations not supported by either the compiler or the vector or parallel features.
 - UNAN** The loop could not be analyzed.
- Nested brackets marking the beginning and end of each loop in every block of loops that was analyzable for vector or parallel processing. The brackets indicate how the statements were grouped after the recurrence detection (see “Statements and Constructs Preventing Vectorization” on page 277). These brackets are the only way to accurately determine the relative nesting of the statements in the compiler report.
- Source statements.
- Short forms of diagnostic messages. The last 3 columns of each message contain a hyphen (-) followed by the last two digits of the message number. REPORT(XLIST) also produces the long forms of diagnostic messages; these appear later in the listing.

- 4** A compiler analysis report showing the entire source program. This is produced by REPORT(SLIST) on the VECTOR and PARALLEL compile-time options. Note that syntax and semantic messages that appear in the source program listing are not displayed in the REPORT(SLIST) report.

This report contains the following information:

- Flags indicating whether parallel or vector code was generated for the statements. In the first column are characters, indicating the following:
 - P** Parallel code was generated for the marked parallel language construct.
 - P** Parallel code was generated for the marked DO loop.
 - V** Loop was partially or completely vectorized.
 - PV** Parallel and vector code was generated for the marked DO loop.
 - S** Serial and scalar code was generated for the marked parallel language construct.
 - U** Loop was unanalyzable.

- p** Parallel code was generated for the marked non-DO statement.
- p_v** Parallel and vector code was generated for the marked non-DO statement.
- v** Statement was vectorized.
- s** Statement was run in serial and scalar mode.

In the second column are:

- Single digit numbers, indicating the vector or parallel analysis depth of the loop (or for statements, of the loop chosen as the vector sectioning loop for the statement).
- Strings of between 1 and 7 digits indicating the loops containing the statements for which the parallel code was generated.

Note: For the **p_v** flag, numbers are placed within the flag in the following manner: *pnnnm*.

The analysis depth is 1 for each outermost analyzable loop and increases by 1 for each nested loop up to a maximum depth of 7 for parallel analysis and 8 for vector analysis.

- Nesting levels of IF and DO statements.
- Internal statement numbers.
- Plus signs (+) next to lines that were included in the source program by means of the INCLUDE directive.
- Source statements.
- Short forms of diagnostic messages. The last 3 columns of each message contain a hyphen (-) followed by the last two digits of the message number. REPORT(SLIST) also produces the long forms of diagnostic messages; these appear later in the listing. See below for information about these messages.

- 5** Long forms of the compiler report messages. These are produced by either REPORT(XLIST) or REPORT(SLIST) on the PARALLEL or VECTOR compile-time options.

The long form of the message consists of a message number (with the prefix ILX), one or two internal statement numbers to identify the statement or range of statements to which the message applies, a status flag, and the complete message text.

Most of the messages explain why a statement caused a loop not to be processed in vector or parallel modes. However, some messages, issued for loops that were processed in vector or parallel modes, highlight situations where vector or parallel processing can change the result of a computation. Other messages clarify certain ambiguities in the compiler report listing. Messages also appear when parallel and vector directives are specified; each message identifies the loops and statements affected by the directive.

It is possible for seemingly contradictory messages to be associated with a single statement. For example, a statement may be nested in two loops, where the outer loop carries a recurrence and the inner loop is vectorizable. A message explaining why the statement failed to vectorize, and another indicating vectorization occurred might both be produced. The former message

indicates how the statement may have caused the recurrence in the outer loop to exist.

See Appendix E, “Compiler Report Diagnostic Messages” on page 491 for a complete list of the short and long forms of the messages, possible responses, and additional information. Note that under the possible responses, certain code transformations are discussed that might help increase the amount of vector or parallel processing. Be careful when applying these transformations because it is possible that they may degrade scalar performance without increasing vectorization. It is also possible that applying a transformation may change the results of a program.

Explanations of the messages are also available online under interactive debug. For more information about the online messages, see *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

- 6** A vector and parallel statistics table produced by REPORT(STAT) on the the VECTOR and PARALLEL compile-time options. The table displays the iteration count for each analyzable loop along with the strides for each array reference in the loop. The table contains the following information:
- Internal statement numbers. These are helpful for mapping statements in the compiler report to statements in the source program listing.
 - Array name or loop induction variable.
 - Up to eight columns, one for each level of loop nesting. In these columns, *one* of the following is given, depending on whether the reference under the heading ARRAY/INDUCTION is to an array name or loop induction variable:
 - For a reference to an array name, the stride is given at each possible level of vectorization. For loops that are actually chosen for vectorization, the stride is followed by “**V.**” For loops chosen for automatic parallel code generation, the stride is followed by “**P.**” Strides that cannot be fully determined at compile time are followed by “**?**.”
- or:**
- For a reference to a loop induction variable, the iteration count of the loop is given under the column corresponding to the nesting level of that loop. Values that are estimated because they cannot be determined at compile time are followed by “**?**.”

- 7** A table that identifies dependences that have been eliminated or modified as a result of the IGNORE directive. (For a discussion of the IGNORE directive, see “IGNORE Directive” on page 358.) This table can help you understand how the IGNORE directive was applied and whether it is used correctly. It is produced by specifying an IGNORE directive and REPORT(XLIST) or REPORT(SLIST) on the PARALLEL or VECTOR compile-time options.

The table contains the following information:

- Flags indicating the type of IGNORE directive used and the action taken:
 - BACKDEP** A potential backward dependence that has been eliminated because of an IGNORE RECRDEPS directive.
 - PREVDEP** A forward dependence that was assumed not to be interchange preventing because of an IGNORE RECRDEPS directive.

- The names of arrays carrying dependences.
- The names of EQUIVALENCE arrays involved in dependences that have been eliminated by an IGNORE RECRDEPS directive.
- Internal statement numbers of the statements where the dependences originated and ended.
- The dependence types (TRUE, ANTI, or OUTPUT).
- The subscript positions (relative to the leftmost position) that are varied by the loops carrying dependences. A subscript position is varied by a loop when either the loop variable or an auxiliary induction variable of that loop is used in that position. Usually, this involves only a single dimension. However, if the loop varies more than one subscript position, the table will contain a list of positions.
- The induction variables of the loops carrying dependences. For information on dependence types, see “Classification of Dependences” on page 262.

8 General diagnostic and informative messages.

Using Parallel and Vector Directives

Occasionally the VS FORTRAN compiler does not make the desired vector or parallel processing decisions. You can use parallel and vector directives to override or influence the compiler's decisions. However, while parallel and vector directives can improve performance you must use them with caution. If the vector or parallel processing performed by the compiler is sufficient, or you can improve performance by recoding, do not use parallel and/or vector directives.

The parallel and vector directives are:

ASSUME COUNT	Specifies a value that is to be used for vector or parallel cost analysis when a loop iteration count cannot be determined at compile time.
IGNORE	Instructs the compiler to ignore specified dependences in a loop.
PREFER	Specifies that a loop be processed in vector or scalar mode and/or parallel or serial mode, regardless of decisions made by vector or parallel cost analysis. Also specifies a recommended number of loop iterations to be processed as a group for PARALLEL DO loops or loops for which parallel code was automatically generated.

Problem areas that might benefit from the use of parallel and vector directives are:

1. When the iteration count of a loop cannot be determined at compile time, vector or parallel cost analysis algorithms might make an incorrect estimate of the loop count. Use the ASSUME COUNT directive to specify a value.
2. In determining how to process a loop, vector or parallel cost analysis might make an undesirable decision. Use the PREFER directive to request that particular loops be run in the desired mode.

It may be necessary to study the timing of several different runs to know the best way to apply the ASSUME COUNT and PREFER directives.

3. The compiler may lack sufficient information to perform dependence analysis. In such cases it must assume that dependences exist. In the example:

```
DO 10 I = 0,100
  A(I+IBASE) = A(I)
10 CONTINUE
```

the subscript offset is unknown. A recurrence may exist, depending on the value of IBASE. (When IBASE is negative or greater than or equal to 100, no recurrence exists.) Use the IGNORE directive to indicate that certain dependences do not exist.

Using the IGNORE directive requires an understanding of the dependences that are ignored, and of the run-time conditions that could make those dependences exist.

4. Balancing the work loads assigned to virtual processors for parallel processing helps improve the run-time of your parallel program. Use the PREFER CHUNK directive to recommend numbers of loop iterations to be processed as a group on a virtual processor.

Interactions between Parallel and Vector Directives

The effects of combinations of directives are discussed below.

ASSUME COUNT and PREFER affect the application of cost analysis. ASSUME COUNT assists analysis by providing additional information while PREFER completely overrides the process. If both of these directives are applied to a single nest of loops, PREFER overrides any effects of ASSUME COUNT.

ASSUME COUNT and PREFER CHUNK operate independently of each other.

IGNORE and PREFER VECTOR operate independently of one another. Using IGNORE makes a loop more likely to be eligible for vectorization. If the compiler still decides not to vectorize the loop because of economic considerations, PREFER VECTOR can be used to override the compiler.

IGNORE and PREFER PARALLEL operate independently of one another. Using IGNORE makes a loop more likely to be eligible for parallel processing. If the compiler still decides not to run the loop in parallel mode because of economic considerations, PREFER PARALLEL can be used to override the compiler.

IGNORE and PREFER SCALAR using IGNORE makes a loop more likely to be eligible for vectorization, but PREFER SCALAR prevents the loop from being chosen. If both directives are applied to a single nest of loops, PREFER SCALAR overrides any effects that IGNORE may have.

IGNORE and PREFER SERIAL using IGNORE makes a loop more likely to be eligible for parallel processing, but PREFER SERIAL prevents the loop from being chosen. If both directives are applied to a single nest of loops, PREFER SERIAL overrides any effects that IGNORE may have.

Specifying Parallel and Vector Directives

To specify parallel and vector directives, you code them in your source program, preceding the loops to which they apply. In addition, you must specify the DIRECTIVE compile-time option using an @PROCESS statement. The DIRECTIVE compile-time option is discussed on page 26 in Chapter 2, "Compiling Your Program."

To code a parallel and/or vector directive:

1. Begin with a comment symbol (C or * for FIXED format input, or " for FREE format input).
2. Immediately following the comment symbol, code a character string that matches the *trigger-constant* specified on the DIRECTIVE compile-time option. To use a single quote (') in the character string, specify two consecutive single quotes in the *trigger-constant*.
3. Code at least one blank, followed by the parallel and/or vector directive. The syntax for each directive is given in the following sections.

The directive must be coded within the first 72 columns. If it refers to any array names, it must appear after the statements declaring those names to be arrays.

The following example shows a DIRECTIVE compile-time option, in which *VDIR: is the trigger-constant, and an ASSUME COUNT directive:

```
@PROCESS DIRECTIVE('*VDIR:')
:
C*VDIR: ASSUME COUNT(3)
```

Local and Global Parallel and Vector Directives

A parallel directive that affects only the first loop that follows it is called a *local parallel directive*; a vector directive that affects only the first loop that follows it is called a *local vector directive*. A parallel directive that affects multiple loops is called a *global parallel directive*; a vector directive that affects multiple loops is called a *global vector directive*. Only the ASSUME COUNT, PREFER SERIAL and PREFER SCALAR directives can be used as global directives.

Global directives are specified with the additional keywords ON and OFF. For example:

```
C*VDIR: PREFER SCALAR ON
:
C*VDIR: PREFER SCALAR OFF
```

If you specify the ASSUME COUNT, PREFER SERIAL, or PREFER SCALAR directive with the ON keyword, it will apply to all loops until either the end of the program unit or a matching directive with the OFF keyword is reached.

On the ASSUME COUNT global directive, you can specify multiple values, where each value is associated with particular loop induction variable. For example:

```
C*VDIR: ASSUME COUNT(I=4, J=200) ON
```

Rules for Specifying Multiple Parallel and Vector Directives

The following rules apply when you have multiple parallel and vector directives in your program:

- Local directives take precedence over global directives. If a local directive is specified for a loop where a global directive is already in effect and the two directives conflict, the local directive will be used for that loop, but the global directive will continue to be in effect for subsequent loops.
- Multiple ASSUME COUNT and PREFER local directives are not allowed for a single loop. If two ASSUME COUNT local directives or two PREFER local directives are specified for the same loop, the first one is used and the second one is ignored. Multiple IGNORE directives are allowed for a single loop.
- If two ASSUME COUNT global directives are used where one specifies a list of variables associated with values and the other specifies a single value, the latter is used for all loops whose induction variables are not on the list of the former. The rule applies regardless of the order in which the two directives appear.
- If there is a conflict between two ASSUME COUNT global directives, where a variable is associated with one value on the first directive but with another value on the second directive, the first value is used until the second one is encountered, at which point the second one is used from there on.
- ASSUME COUNT OFF cancels all preceding ASSUME COUNT ON directives.

For an example of values used when multiple ASSUME COUNT directives are specified, see page 358.

ASSUME COUNT Directive

ASSUME COUNT specifies a value that is to be used for vector or parallel cost analysis when loop iteration counts cannot be determined at compile time.

If you specify ASSUME COUNT for a loop with a known iteration count, the directive is ignored and a warning message is issued.

Using the ASSUME COUNT directive has no effect on program results. If you specify an incorrect count, the results are identical to those produced by scalar or serial code. At worst, an incorrectly specified ASSUME COUNT directive results in increased run time.

ASSUME COUNT only affects vector or parallel cost analysis. The information is not used to determine the existence of dependences or recurrences.

SyntaxLocal Directive:**ASSUME COUNT (*val*)**Global Directive:**ASSUME COUNT ({*val* | *var*=*val* [,*var*=*val*] ...}) ON****ASSUME COUNT OFF****ASSUME COUNT (*val*)**

Begins an ASSUME COUNT local directive.

val Is an integer constant, or a named constant with an integer value, indicating the iteration count.

If a named constant is used, the PARAMETER statement that defines that constant must precede the directive.

ASSUME COUNT ({*val* | *var*=*val* [,*var*=*val*] ...}) ON

Begins an ASSUME COUNT global directive.

val Is an integer constant, or a named constant with an integer value, indicating the iteration count.

If a named constant is used, the PARAMETER statement that defines that constant must precede the directive.

var

Is the name of a four-byte integer variable that is used as the enumeration variable for one or more loops in the range of the directive.

ASSUME COUNT OFF

Cancels all preceding ASSUME COUNT ON statements.

Examples of ASSUME COUNT**Example 1:** In this example, knowledge of a short loop helps the compiler avoid uneconomical vectorization.

```

C*VDIR: ASSUME COUNT(3)
        DO 95 J = 1, N
            A(J) = B(J)/C(J)
95      CONTINUE

```

Example 2: In this example, information about the relative size of each loop may affect the compiler's choice of a loop to vectorize.

```
C*VDIR: ASSUME COUNT(10)
      DO 95 J = 1, N
        A(J) = 0.0
C*VDIR: ASSUME COUNT(450)
      DO 96 K = 1, M
        A(J) = A(J) + B(K,J) * C(J,K)
96      CONTINUE
95      CONTINUE
```

Example 3: In this example, a named constant with an integer value is used for the assumed iteration count.

```
      PARAMETER (NLIM=300,MLIM=5)
      .
      .
C*VDIR: ASSUME COUNT(NLIM)
      DO 100 I=1,N                      <== assumed count is 300
C*VDIR: ASSUME COUNT(MLIM)
      DO 100 J=1,M                      <== assumed count is 5
```

Example 4: In this example, two global directives and one local directive are specified.

```
C*VDIR ASSUME COUNT(50) ON
C*VDIR ASSUME COUNT(I=4, J=200) ON
      DO 100 I=1,N1                      <== assumed count is 4
      DO 100 J=1,N2                      <== assumed count is 200
      DO 100 K=1,N3                      <== assumed count is 50
      .
      .
C*VDIR ASSUME COUNT(500)
      DO 200 I=1,N4                      <== assumed count is 500
      DO 200 J=1,N5                      <== assumed count is 200
      DO 200 L=1,N6                      <== assumed count is 50
      .
      .
C*VDIR ASSUME COUNT OFF
      DO 300 I=1,N7                      <== assumed count is 65 (by default)
      DO 300 J=1,N8                      <== assumed count is 65 (by default)
```

IGNORE Directive

IGNORE instructs the compiler to ignore specified dependences in a loop. You must specify at least one option on the IGNORE directive; multiple options may be specified in any order. The compiler accepts the IGNORE directive only when the compiler lacks enough information to determine the existence of dependences. After the compiler accepts the IGNORE directive, the loop may still not run in vector or parallel mode, for the following reasons:

- The existence of dependences that cannot be overridden
- The existence of unanalyzable and unsupported constructs
- Outer loops affecting the iteration parameters of inner loops
- Cost analysis (although cost analysis can be influenced by use of the PREFER and ASSUME COUNT directives).

The IGNORE directive (CALLDEPS or RECRDEPS) causes the compiler to ignore backward dependences and interchange-preventing dependences, but not forward loop-carried dependences. Note that loop-carried dependences inhibit the automatic generation of parallel code. If your code contains forward loop-carried

dependences, use the PREFER PARALLEL and IGNORE directives together to cause the loop to run in parallel mode.

Use IGNORE with *extra caution*. Incorrectly specifying IGNORE can produce erroneous program results.

An installation option determines whether the IGNORE directive is enabled or disabled. See your system programmer to check whether the IGNORE directive is enabled for your installation.

By specifying REPORT(XLIST) on the PARALLEL or VECTOR compile-time option, you can produce a table of ignored dependences in the compiler report. This table can help you understand how the IGNORE directive was applied and whether you used it correctly. For more information about this table, see page 352.

Syntax

```
IGNORE [RECRDEPS [(array-list)] [CALLDEPS [(name [,name...])]]]
```

RECRDEPS

Requests that, in cases where the compiler is not certain whether a dependence that might be part of a recurrence is present, the compiler should assume that the dependence does not occur. RECRDEPS acts on every applicable array in the loop unless qualified by an array-list.

Backward and interchange-preventing dependences can be ignored. For more information on dependences, see “Classification of Dependences” on page 262.

Loops identified by the flag “RECR” in the XLIST compiler report are candidates for using this directive for increased parallel or vector processing. However, use IGNORE RECRDEPS *only* if the specified dependences are perceived by the compiler but do not in fact occur when your program runs.

array-list

Is an optional list of array variable names separated by commas. The names, which can include EQUIVALENCE array variables, must fit on one line and must not extend beyond column 72. If you cannot fit all the names on one line, you can continue coding them on an additional IGNORE directive for the same loop.

An array-list restricts the directive to the specified arrays and to arrays that are equivalenced to specified arrays. (Thus, to cause a dependence to be ignored for EQUIVALENCE arrays, you need to specify only one of the arrays involved in the dependence.) If you omit the array-list, the directive applies to all applicable arrays within the loop.

CALLDEPS [(name [,name...])]

Requests that if the compiler is uncertain whether a dependence is caused by:

- Arguments of the specified subroutine(s)
- Arguments of the specified function subprogram(s)
- Any variables in common blocks,

it should assume that the dependence does not occur.

Backward and interchange-preventing dependences can be ignored. For more information about dependences, see “Classification of Dependences” on

page 262. For a description and examples of using common blocks with parallel processing, see “Sharing Data in Common Blocks within a Parallel Task” on page 308.

You are responsible for ensuring that there are no dependencies on arguments or common blocks shared with a parallel thread.

name

Specifies the name of a subroutine or function subprogram. If you do not specify any names, a list of all subprogram names appearing in the loop(s) is assumed. Names must fit on one line and must not extend beyond column 72. If you cannot fit all the names on one line, you can continue coding them on an additional IGNORE directive for the same loop.

EQUDEPS

IGNORE EQUDEPS remains a valid directive, but it has no effect on dependence analysis. If you specify IGNORE EQUDEPS, an informational message will be generated to denote the change.

Examples of IGNORE

The examples below include code to check for correct application of the directive at run time. Such code is useful when testing and debugging loops using vector and parallel directives.

You must check run-time relationships to determine whether dependences exist. While this requires extensive analysis of the application, it is necessary because incorrect usage of the IGNORE directive can cause unexpected results.

In the examples, a routine (DIRERR) is invoked to report instances where a directive application is incorrect. This subroutine is defined in “Verifying Correct Application of Directives” on page 367.

Example 1. Unknown loop index upper bound: Correct directive application may depend on vector length.

```
      IF (.NOT.(N .LE. 77))
+      CALL DIRERR( 20, 1, '(N .LE. 77)')

C*VDIR: IGNORE RECRDEPS(A)
      DO 20 K = 1, N
          A(K+77) = A(K) * B(K)
20      CONTINUE
```

Example 2. Unknown auxiliary induction variable increment: Correct directive application depends on the direction of the increment.

```
      IF (.NOT.(M .GT. 0))
+      CALL DIRERR( 40, 41, '(M .GT. 0)')

C*VDIR: IGNORE RECRDEPS(A)
      J = 200
      DO 40 I = 1, 1000
          A(I) = A(J) * B(I)
          J = J + M
40      CONTINUE
```


It would also be safe to vectorize this loop when M is less than or equal to -200. However, a simplified test reduces the amount of computation involved in verification. The test chosen insures that incorrect vectorization is detected.

Example 3. Unknown subscript offsets (simple case): Correct directive application prohibits subscript overlap.

```

        IF (.NOT.(M1 .LE. 0 .OR. M1 .GE. 1000))
+       CALL DIRERR( 50, 71, '(0.LE.M1 .OR. M1.GT.1000)')

C*VDIR: IGNORE RECRDEPS(A)
        DO 50 I = 1, 1000
            A(I+M1) = A(I) * B(I)
50      CONTINUE

```

Example 4. Unknown subscript offsets (full generality)

```

        IF (.NOT.(M2.GE.M1 .OR. ABS(M1-M2).GE.1000))
+       CALL DIRERR(60,72,
+               '(M2.GE.M1 .OR. ABS(M1-M2).GT.1000)')

C*VDIR: IGNORE RECRDEPS(A)
        DO 60 I = 1, 1000
            A(I+M1) = A(I+M2) * B(I)
60      CONTINUE

```

Example 5a. Indirect Addressing: When indirect addressing is used, it is difficult to verify correct directive application. In the example, a logical function (DUPIND) checks the indexing array for duplicate values and returns .TRUE. if any are found. (DUPIND is defined in Figure 112 on page 368.) Verification is possible in this case because the same indexing array is used on both sides of the equation.

```

        IF (DUPIND(J, 1000, 0))
+       CALL DIRERR(70,72,'Independent Indirect Indexes')

C*VDIR: IGNORE RECRDEPS(A)
        DO 70 I = 1, 1000
            A(J(I)) = A(J(I)) * B(I)
70      CONTINUE

```

Example 5b. Indirect Addressing: In the following example, the left side reference to A uses indirect addressing, while the right side reference does not. You must check that no duplicates exist in the indexing array, and that a value stored on one iteration is not referenced on a later iteration. An added loop checks that no value of $J(I)$ matches a value of I on a later iteration.

```

        IF (DUPIND(J, 1000, 0))
+       CALL DIRERR(80,72,'Independent Indirect Indexes')
        DO 79 I = 1, 1000
            IF (J(I).GT.I .AND. J(I).LE.1000)
+       CALL DIRERR(80,72,'Independent Indirect Indexes')
79      CONTINUE

C*VDIR: IGNORE RECRDEPS(A)
        DO 80 I = 1, 1000
            A(J(I)) = A(I) * B(I)
80      CONTINUE

```

Example 5c. Indirect Addressing: In the following example, two different indexing arrays are used.

```
C*VDIR:  IGNORE RECRDEPS(A)
          DO 90 I = 1, 1000
            A(JLEFT(I)) = A(JRIGHT(I)) * B(I)
          90  CONTINUE
```

The IGNORE directive is valid only when no entry in the JLEFT array matches a later entry in the JRIGHT array. For example,

```
JLEFT = (1,3,5, ... 2*k-1)
JRIGHT = (2,4,6, ... 2*k)
```

can be vectorized, while the following example can not.

```
JLEFT = (1,2,3, ... ,n)
JRIGHT = (n,n-1,n-2, ... 1)
```

It is possible to write a subroutine similar to DUPIND that checks whether there is a dependence in this case.

Example 6. Interchange-preventing dependence: In the following example, the compiler assumes that there are two interchange-preventing dependences carried by the outer loop:

- A true, forward dependence from statement 10 to statement 20 carried by array A.
- An anti, backward dependence from statement 20 to statement 10, also carried by array A.

Using IGNORE RECRDEPS permits vectorization by modifying the forward dependence so that it is no longer considered to be preventing.

```
C*VDIR:  IGNORE RECRDEPS
          DO 100 I = 1,N
            DO 100 J = 1,M
              10  A(I+L,J) = B(I,J)
              20  C(I,J) = A(I,J)
            100  CONTINUE
```

Example 7. Nested loops: When IGNORE RECRDEPS is used on one loop within a nest of loops, it only applies to the specified loop. In the following example, using IGNORE RECRDEPS makes the outer loop eligible for vectorization, but does not change the eligibility of the inner loop.

```
          IF (.NOT.(L2 .GE. 0))
+          CALL DIRERR(95, 71, '(L2.GE.0)')

C*VDIR:  IGNORE RECRDEPS(B)
          DO 95 J = 1, N
            DO 97 K = 1, M
              B(K,J) = B(K+L1,J) + B(K,J+L2)
            97  CONTINUE
          95  CONTINUE
```

Example 8. Statement reordering: Backward dependences causing statement reordering are ignored if IGNORE RECRDEPS is used. No reordering occurs if the loop is subsequently vectorized. The following example contains a loop (where N

is known to be positive) requiring reordering to vectorize properly. Reorder statements 10 and 20 before using IGNORE RECRDEPS on this loop.

```

      DO 100 I = 1, 100
10      A(I) = B(I)
20      B(I+N) = C(I)
100    CONTINUE

```

Example 9. Multiple IGNORE directives: In the following example, the array-list is too long to fit on one directive. The list is continued on an additional directive.

```

C*VDIR: IGNORE RECRDEPS (ADDRESS,BYTE,CODE,EXTENT,FIELD,GRAPH,HM,IMAGE)
C*VDIR: IGNORE RECRDEPS (JOB,KB,LOG,MESSAGE)
      DO 100 I=N,M
        ADDRESS(I) = ADDRESS(I+K1)
        ...
100    CONTINUE

```

Example 10a. EQUIVALENCE Arrays: The possibility of recurrences involving EQUIVALENCE arrays are ignored if IGNORE RECRDEPS is used. In the following example, values stored into variable A will never be used later by variable B if the loop increment is greater than one.

```

      DIMENSION A(200),B(200)
      EQUIVALENCE (A(1),B(2))

      IF (.NOT.(N .GT. 1))
+      CALL DIRERR(120, 74, 'N .GT. 1')

C*VDIR: IGNORE RECRDEPS(A)
      DO 120 I = 1,190,N
        A(I) = B(I) * 2.1
120    CONTINUE

```

Example 10b. EQUIVALENCE Arrays: The possibility of recurrences involving EQUIVALENCE arrays are ignored if IGNORE RECRDEPS is used. In the following example, a dependence exists if the upper bound of the loop is large enough for variable B to refer to a storage location assigned to it on a previous iteration through variable A.

```

      DIMENSION A(200),B(200)
      EQUIVALENCE (A(1),B(101))

      IF (.NOT.(N .LE. 100))
+      CALL DIRERR(130, 75, 'N .LE. 100')

C*VDIR: IGNORE RECRDEPS(A)
      DO 130 I = 1,N,1
        A(I) = B(I) * 2.1
130    CONTINUE

```

Example 11. IGNORE CALLDEPS: The possibility of dependences involving arguments of subroutines or function subprograms, or variables in common blocks is ignored if IGNORE CALLDEPS is used. In the following example, the possibility that SUBA will modify elements of arrays A and B exists.

```
COMMON/COM1/B(100)
REAL*8 A(10000)
C@PARA PREFER CHUNK(5000) PARALLEL
C@PARA IGNORE CALLEPS(SUBA)
DO I = 1,10000
    A(I) = SQRT(FLOAT(I))
    CALL SUBA(A,I)
ENDDO
END
```

PREFER Directive

PREFER specifies that particular loops be run in vector or scalar and/or parallel or serial modes, regardless of decisions made during cost analysis. PREFER may also be used with PARALLEL(AUTO) or PARALLEL DO loops to recommend the number of loop iterations to be processed as a group. You must specify at least one option on the PREFER directive; multiple options may be specified in any order. If you specify any of the PREFER options more than once, or if you specify a PREFER option that conflicts with a previously specified value all but the first are ignored. If you code more than one consecutive PREFER directive, all but the first are ignored.

Use PREFER only after carefully studying run times of a loop in different modes. Do not use PREFER if the same result can be achieved using the ASSUME COUNT directive because PREFER can make a program hardware dependent.

Syntax

Local PREFER:

```
PREFER {SCALAR | VECTOR |
        SERIAL | PARALLEL |
        CHUNK ({n|n.:m|n:m})}
```

Global PREFER:

```
PREFER {SCALAR | SERIAL} ON | OFF
```

SCALAR

Specifies that the following loop be run in scalar mode. This request will *always* be honored regardless of any other considerations.

VECTOR

Specifies that the following loop be run in vector mode. This request will be honored *only* if the loop is eligible for vectorization. If a loop is not eligible for vectorization, the directive will be ignored and normal vectorization processing will be applied to the remaining loops within the nest.

If you specify PREFER VECTOR for more than one loop in a nest, it will be honored only for the most deeply nested loop that is eligible for vectorization.

Misuse of PREFER VECTOR may cause inefficient code to be generated, but loop results remain the same.

SERIAL

Specifies that the following loop be run in serial mode. This request will *always* be honored regardless of any other considerations.

PARALLEL

Specifies that the following loop be run in parallel mode. This request will be honored *only* if the loop is eligible to be processed in parallel mode and the PARALLEL(AUTOMATIC) compile-time option is in effect. If you specify PREFER PARALLEL for a loop that is ineligible to be processed in parallel mode, the directive is ignored.

If you specify PREFER PARALLEL for more than one loop in a nest, it will be honored for all loops that are eligible to be processed in parallel mode.

CHUNK ({*n*|*n*:*m*|*n*:*m*})

Specifies a recommended number, or range of numbers, of loop iterations to be processed as a group on a virtual processor. The number, or range of numbers, should divide evenly into the total number of loop iterations.

Both *n* and *m* must be nonnegative integer constants, or named constants with an integer value.

n Indicates the recommended number of iterations to be processed as a single group

n: Indicates the minimum recommended number of iterations to be processed as a single group

:*m* Indicates the maximum recommended number of iterations to be processed as a single group

n:*m* Indicates the recommended range of minimum to maximum number of iterations to be processed as a single group *n* must be less than *m*.

CHUNK is only valid for parallel or parallel vector loops. If you specify PREFER SERIAL, or the compiler does not generate parallel code for the specified loop, and you also specify PREFER CHUNK, PREFER CHUNK is ignored. If you specify PREFER PARALLEL VECTOR, or parallel and vector are chosen by the compiler, the chunk size is a multiple of the vector section size.

ON

Specifies that the SCALAR and/or SERIAL modes apply to all loops until one of the following occurs:

- The program END is encountered
- A matching directive with the OFF keyword is reached
- A local directive is specified.

OFF

Cancels the preceding global directive that set the specified mode ON.

Examples of PREFER

Example 1: In this example, PREFER VECTOR is used to vectorize a loop that the compiler might run in scalar mode.

Parallel and Vector Directives

```
REAL X(INCX,N), Y(INCY,N), A
...
C*VDIR: PREFER VECTOR
DO 10 I=1,100000
    Y(1,I) = Y(1,I) + A*X(1,I)
10    CONTINUE
```

Example 2: In this example, PREFER SCALAR is used to prevent vectorization of a loop.

```
C*VDIR: PREFER SCALAR
DO 100 I = 1, N
    IF(B(I).NE.0) THEN A(I) = B(I)*C(I)* ...
100    CONTINUE
```

Example 3: In this example, both a global and a local directive are specified. The global directive, PREFER SCALAR ON, applies to the first two loops. The local directive, PREFER VECTOR, takes precedence over the global directive so it is used for the third loop. PREFER SCALAR ON resumes effect for the fourth loop. PREFER SCALAR OFF cancels the global directive and the last two loops are processed as normal.

```
C*VDIR  PREFER SCALAR ON
DO 100 I=1,100    <== will not be vectorized
DO 100 J=1,100    <== will not be vectorized
...
C*VDIR  PREFER VECTOR
DO 200 I=1,N3     <== will be vectorized if eligible
DO 200 J=1,N4     <== will not be vectorized
...
c*VDIR  PREFER SCALAR OFF
DO 300 I=1,N4     <== normal vectorization processing will be applied
DO 300 J=1,N5     <== normal vectorization processing will be applied
```

Example 4: In this example, a local directive is used for a DO loop for which parallel code is generated automatically:

```
@PROCESS PAR(AUTO) VEC DIR('@PARA') OPT(3)
...
REAL*8 A(1000,64),B(64)
...
C@PARA PREFER PARALLEL
DO 10 J = 1,64
    IF B(J).LT.0 GOTO 10
C@PARA PREFER VECTOR
DO 20 I = 1, 1000
    A(I,J) = I*10000 + J*100
20    CONTINUE
10    CONTINUE
...
END
```

Example 5: In this example, a local directive is used for a PARALLEL DO construct. The user recommends that the loop be broken up into two groups of 5000 iterations each for processing.

```

@PROCESS PAR DIR('@PARA') OPT(3)
    REAL*8 A(10000)
C@PARA PREFER CHUNK(5000)
    PARALLEL DO I = 1,10000
        A(I) = SQRT(FLOAT(I))
        CALL SUBA
    END DO
END

```

Example 6: In this example, PREFER is used to specify loops to be run in scalar or serial mode, and then in scalar mode only:

```

C*VDIR PREFER SCALAR SERIAL ON
    (loops preferred in scalar and serial mode)
C*VDIR PREFER SERIAL OFF
    (loops preferred in scalar mode)
C*VDIR PREFER SCALAR OFF

```

Verifying Correct Application of Directives

To verify that you supplied an ASSUME COUNT or IGNORE directive correctly in your vectorized program or the serial parts of the root task in your parallel program, you can write a subroutine similar to the one in Figure 111. The subroutine generates a message and invokes one of the return code subroutines (SYSRCX) listed in *VS FORTRAN Version 2 Language and Library Reference*. The entry point shown here causes processing to terminate with the specified return code. The routine could be rewritten to allow processing to continue by replacing the call to SYSRCX with a call to SYSRCS.

```

        SUBROUTINE DIRERR (ISTMT, ICODE, TEXT)
C-----Print text and ISTMT, terminate with ICODE
        CHARACTER *(*) TEXT
C
        WRITE (*,1) TEXT, ISTMT, ICODE
1      FORMAT('---- Loop Vectorization Assumption "',A,'" /
X          ' Failed at Statement ',I6,
X          ' with Error Code',I12)
        CALL SYSRCX(ICODE)
        END

```

Figure 111. Sample Routine to Report an Invalid Directive

The example in Figure 112 on page 368 is of a logical function, DUPIND. It checks indexing arrays for duplicate values and returns .TRUE. if any are found.

```
      LOGICAL FUNCTION DUPIND(J, N)
C
C Routine to check for independent indirect-index values.
C Returns FALSE if no duplicates, TRUE if duplicates.
C
C J is the INTEGER*4 array of indirect subscripts.
C N is the number of such values.
C
      INTEGER J(N)
C
      DUPIND = .FALSE.
C
C First sort the list, this step takes N log N time
C
      CALL SORT (J, N)
C
C Now, test each value of J(*) in turn
C
      DO I = 2, N
        IF (J(I).EQ.J(I-1)) THEN
          DUPIND = .TRUE.
          WRITE (*,9) I
          FORMAT(' - *** Indirect index No.',I6,' is repeated')
          RETURN
        END
      ENDDO
      END
```

Figure 112. Sample Routine to Check Indexing Arrays for Duplicate Values

Gathering Run-Time Statistics

You can measure the elapsed run time for program fragments by calling the `CLOCK` or `CLOCKX` service subroutine before and after the fragments. For more information about `CLOCK` and `CLOCKX`, see the *VS FORTRAN Version 2 Language and Library Reference*.

You can collect run-time statistics on the vector length and stride of each loop by using the interactive vectorization aid function of interactive debug. The timing and sampling facilities provide information on the relative efficiency of each loop.

To make use of the interactive vectorization aid, you must specify the `IVA` suboption on the `VECTOR` compile-time option, described on page 38, and the `SDUMP` compile-time option, described on page 35. You may also specify the `VECTOR(REPORT(SLIST))` option. For more information, see vector tuning information in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

The `VECTOR(IVA)` compile-time option conflicts with the `PARALLEL` compile-time option; if you specify `VECTOR(IVA)` and `PARALLEL`, `NOVECTOR` and `NOPARALLEL` are assumed.

Part 5. Advanced Coding Topics

Chapter 19. Interprogram Communication

In Fortran, there are two ways to share data: by passing arguments between the programs and by using common data areas (areas that can be shared by more than one program).

- Passing arguments. You can pass data values between a calling program and a called program 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.
- Using common storage. You can use the COMMON statement to specify shared data storage areas for two or more program units, and to name the variables and arrays occupying the shared area.

This chapter discusses both of these ways to share data. It also discusses the intercompilation analysis feature (ICA), which VS FORTRAN provides to help you verify that passed arguments have been specified correctly.

Passing Arguments to Subprograms

You can use actual and dummy arguments to pass data between a calling program unit and a subprogram. For example, in the following CALL statement:

```
CALL MAXNUM(PMAX,P1,P2)
```

PMAX, P1, and P2 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(XMAX,X1,X2)
```

Dummy arguments of subroutine subprogram MAXNUM are XMAX, X1, and X2.

When the CALL statement is run, the actual arguments are associated with the matching dummy arguments:

PMAX	is associated with	XMAX
P1	is associated with	X1
P2	is associated with	X2.

When control returns to the calling program, the current values in XMAX, X1, and X2 are also the current values of PMAX, P1, and P2 in the calling program.

Note: Arguments that are passed by copy do not return their value to the calling routine. A parallel call, scheduled call, or a call or function reference within a parallel loop or parallel section may have arguments passed by copy—loop index variables and expressions (including parenthesized primaries).

The following are general rules for coding arguments:

- You must define dummy arguments to correspond in number, order, and type with the actual arguments.

- Actual arguments are passed by reference, if you alter the value of an argument in the subprogram, you're altering the value in the calling program as well. Some arguments within a parallel loop or parallel sections construct may be passed by copy so that the value of the argument in the calling program is not altered.
- 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 that follows and includes the actual array element you specify.
- 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. The intercompilation analysis feature will detect this type of error.
 - Your subprograms should not assign new values to dummy arguments that are associated with either other dummy arguments in the subprogram or variables in the common area. For example, if you include the following elements in the calling program:

```
COMMON B
  ⋮
CALL DERIV (A, B, A)
```

and you define the subprogram DERIV as:

```
SUBROUTINE DERIV (X,Y,Z)
COMMON W
```

the DERIV subprogram should not assign new values to:
 - X and Z because they are both associated with the same argument, A.
 - Y because it is associated with argument B, which is in the common area.
 - W because it is also associated with B.
 - If you do assign new values, you may get unexpected results; but, in the case of dummy arguments associated with other dummy arguments, the intercompilation analysis feature will give you a warning message.

Using Common Areas

The following discussion applies in general to the use of common areas in VS FORTRAN Version 2. For specific information about how to use common areas in parallel programs, see “Sharing Data in Common Blocks within a Parallel Task” on page 308 and “Sharing Data in Common Blocks between Parallel Tasks” on page 311.

Items shared in a common area are subject to the same rules as arguments passed in a subprogram argument list (see “Passing Arguments to Subprograms” on page 371).

For example, 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 each 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 113. Each column of variables starts at the beginning of the common area. Variables on the same line share the same storage locations.

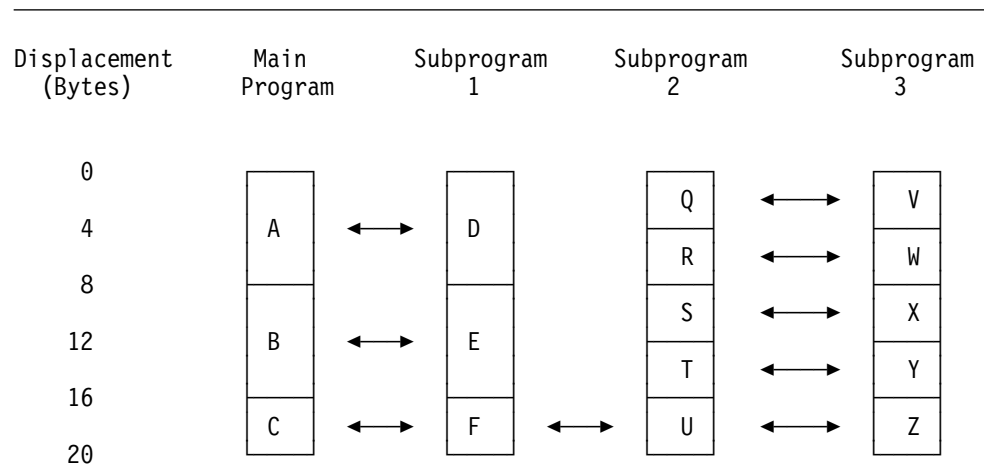


Figure 113. Transmitting Assignment Values between Common Areas

The main program can safely 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 should not, 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 these common variables differ.

In the same way, subprogram 2 and subprogram 3 should not 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).

Referencing Shared Data in Common

In general, shared data in the common area should be referenced with the same descriptions in different sharing program units. While the name of the common area must be the same, the names of corresponding variables and arrays in the common area may be different.

The examples shown previously for passing arguments in common also illustrate sharing data in the common area.

Shared data in the common area can be referenced with different descriptions, provided the different descriptions are not contradictory. Describing the data differently for different uses may be advantageous in the application you are programming. But you must be careful to maintain the identity of the data itself within the differing descriptions.

Character-type data, for instance, can be referenced as strings of differing lengths. For example, in subprogram 1, you could write

```
COMMON CHV2  
CHARACTER CHV2 * 20
```

and in subprogram 2, you could write

```
COMMON CHA2  
CHARACTER CHA2 * 5(4)
```

Subprogram 1 references the 20 bytes of character data as a single character variable, CHV2. Subprogram 2 references the same 20 bytes as a character array, CHA2, having four elements of five bytes each.

You can ascertain whether different descriptions of the same data are contradictory by considering the format of the data itself, as represented in the running program. For example, a complex number is represented as two adjacent real numbers. Thus, you can correctly write in subprogram 1:

```
COMMON CV  
COMPLEX*8 CV
```

and in subprogram 2:

```
COMMON RV1,RV2
```

This allows subprogram 2 to reference the real and imaginary parts of the complex variable CV as two separate real numbers, RV1 and RV2.

For detailed information on the formats of the various types of data in the running program, see "Internal Representation of VS FORTRAN Version 2 Data" on page 469.

Efficient Arrangement of Variables in Common

Your programs lose some run-time efficiency unless you ensure that all of the common variables have proper boundary alignment. You need not align complex, integer, logical, or real variables to have your program run successfully. However, if an array is not on an appropriate boundary, a vectorized program will not run and a scalar program will run inefficiently.

You can ensure proper alignment either by arranging the noncharacter type variables in a fixed descending order according to length, or by defining the block so that dummy variables force proper alignment.

Using a Fixed Order of Variables: If you use the fixed order, noncharacter type variables must appear in the following order:

Length	Type
32	COMPLEX
16	COMPLEX, REAL, or DOUBLE COMPLEX
8	REAL, INTEGER, or LOGICAL
8	COMPLEX or DOUBLE PRECISION
4	REAL, INTEGER, or LOGICAL
2	INTEGER or LOGICAL
1	LOGICAL, INTEGER, UNSIGNED, or BYTE

Using Dummy Variables: 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 number of bytes in the boundary alignment requirement 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 boundary alignment requirement for each type of variable is as follows:

Type Specification	Length Specification	Boundary Alignment Requirement
LOGICAL	4	Word
INTEGER	4	Word
REAL	4	Word
COMPLEX	8	Word ¹
LOGICAL	1	Byte
INTEGER	1	Byte
UNSIGNED	1	Byte
BYTE	1	Byte
INTEGER	2	Halfword
LOGICAL	2	Halfword
INTEGER	8	Doubleword
LOGICAL	8	Doubleword
REAL	8	Doubleword
REAL	16	Doubleword
DOUBLE PRECISION	8	Doubleword
COMPLEX	16	Doubleword
DOUBLE COMPLEX	16	Doubleword
COMPLEX	32	Doubleword

Note: ¹A doubleword boundary is required if VECTOR(CMPLXOPT) is used and the data accessed as a vector.

Common Storage

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 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 the Common Area
A	0
K	4
CMPLX	8
	16

The displacements of K and CMPLX are evenly divisible by the number of bytes in their boundary alignment requirements.

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 4). 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 the Common Area
A	0
DV	2
K	4
CMPLX	8
	16

EQUIVALENCE Considerations

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)
LOGICAL*1 L1A
LOGICAL L4A
```

which are defined in the common area 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)
L1A	20	Word
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 both R4B and R8B are now also inefficiently aligned.

2. Second Example (illegal):

```
EQUIVALENCE (R8A, R4N(7))
```

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. This is illegal since it causes the common area to be extended to the left.

3. Third Example (valid but inefficient):

```
EQUIVALENCE (R8A, R4N(2))
EQUIVALENCE (R4M, R4N(5))
```

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))

These results are 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 byte boundaries.

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))

These results are 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.

Using Blank and Named Common Blocks

You can define both blank (unnamed) and named common blocks. Blocks given the same name occupy the same storage space.

- A blank common is an unnamed common storage area (common block) that you have not named.
- A named common is a common storage area that you can name using the COMMON statement.
- You can define only one blank common block in a program (although you can specify many COMMON statements defining items in blank common). You can define many individually named common blocks in a program.
- You can define a blank common as having different lengths in different program units, but you must define a given named common block with the same length in every program unit that uses it.
- You can't initialize values in variables or array elements in blank common using DATA statements.
- In a named common, you can initialize values in variables and array elements, through a block data subprogram that contains DATA statements or explicit specification statements.

Initializing Named Common Blocks: 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 assigns the type of and initializes array B in ELJ and array Z in DAL.
- The INTEGER type statement initializes array JC in ELJ.
- Because they're not included in either type statement or in a DATA statement, item A in ELJ and item Y in DAL are assigned default types and are not initialized.

Using Static, Dynamic, and Extended Common Blocks

In VS FORTRAN Version 2 Release 5, three types of common blocks can be defined: static common, dynamic common, and extended common.

Static Common: Common blocks that are compiled in a VS FORTRAN program and reside within the running program are known as static common blocks. Static common blocks are limited in size because the combined sizes of all the blocks and the program code can not exceed 16MB of storage. Use the SC compile-time option to indicate static common as the default type or to identify individual common blocks that are to be static.

Dynamic Common: While the program runs, dynamic common blocks are allocated from within the storage assigned to the running program. The storage space available for all dynamic and static common blocks and the compiled program code is limited to 2 gigabytes in size. Only named common blocks can be allocated as dynamic common. Use the DC compile-time option to indicate dynamic common as the default type or to identify individual common blocks that are to be dynamic.

Dynamic common blocks are useful in the MVS/XA, VM/XA, MVS/ESA, and VM/ESA environments for the expanded address capability. Also, the size of a load module is reduced when dynamic common is used because no space is allocated for the dynamic common in the object modules that make up the load module.

For information on using dynamic common with MTF, see Chapter 20, "The Multitasking Facility (MTF)" on page 397.

Extended Common: Extended common blocks are allocated at run time. Extended common allows you to define multiple extended common blocks, each as large as 2 gigabytes in size. The number of extended common blocks and their combined size is not limited by VS FORTRAN. Whereas dynamic common blocks are allocated storage from within the program space, extended common blocks are allocated storage from within an ESA/390 data space.

To identify these common blocks as allocatable at run time, and to specify that they are extended common blocks, use the EC compile-time option. Only named common blocks can be allocated as extended common.

Extended common blocks, once allocated, exist for the duration of your program. Use the EMODE compile-time option so that subroutines or function subprograms that are passed data residing in extended common blocks can correctly address these items.

Use of extended common blocks at lower levels of optimization requires greater CPU utilization. Therefore, optimization levels below OPT(2) are not recommended.

Coding Common Options: Three options are provided for classifying your common blocks:

SC	for static common
DC	for dynamic common
EC	for extended common

One of these options may have the * form; SC(*) is assumed if neither DC(*) nor EC(*) is specified. The * form gives the default type of common block; all common blocks not named in another option will be of the default type.

If you want most common blocks to be of the same type, use the * form of the corresponding option and specify the exceptions in the option(s) for the other type(s). For example, if you want common block C1 to be a static common block, C2 to be an extended common block, and all other common blocks to be dynamic, use the options DC(*), SC(C1), EC(C2) (in any order).

The listing for your program may show NODC, NOEC, and/or NOSC as the options in effect. Although the options cannot be specified in this form, it is used to indicate the combined effect of the three options for the compilation.

Restrictions on the Use of Common Blocks

- The extended common feature cannot be used with the multitasking facility.
- The extended common feature cannot be used with the static debug or the interactive debug.
- The virtual storage use within a program's address space will be smaller when using extended common blocks than for programs using dynamic or static common blocks.
- Data residing in an extended common block cannot be passed to a program that is written in another language. Assembler language programs that have been modified to use the required addressing operations can accept data passed from an extended common block. For more information on using extended common with assembler language programs, see Appendix B, "Assembler Language Considerations" on page 451.

Intercompilation Analysis

Intercompilation analysis provides a way to identify incompatibilities between program units—particularly in the specification of parameters passed to external procedures.

Note: Under MVS, intercompilation analysis files must be physical sequential data sets.

An executable Fortran program may consist of multiple program units. Each program unit may compile successfully; however, when you combine several program units and attempt to run them together, they may fail to run successfully because of inconsistencies in the way actual and dummy arguments are specified. Often these inconsistencies show up only as incorrect program results. It is very

difficult to detect such inconsistencies without some way of analyzing the program units as a group.

The VS FORTRAN intercompilation analysis feature detects these inconsistencies at compile time so that you can correct them before running the program, thus saving time in debugging large, complex programs.

Some common problems are:

- Use of conflicting actual and dummy arguments in subroutine calls and function references
- Specification of conflicting lengths for named common blocks
- Use of conflicting external names.

A list of intercompilation messages (see Figure 118 on page 395), identifying the discrepancies detected during compilation is produced, as well as an external cross reference list containing the names of subprograms and common blocks referenced by each program unit. (See Figure 117 on page 394.)

Types of Errors Detected by Intercompilation Analysis

Specifying the ICA option for a group of program units causes the actual arguments specified in external references to be checked against the dummy arguments specified for these subprograms. The lengths of named common blocks and the usage of external names are checked for consistency throughout the group of program units.

The following sections discuss the following types of errors:

- Conflicting argument usage
- Conflicting function type
- Conflicting external name usage
- Conflicting common block lengths
- Conflicting common block storage assignment
- Conflicting argument and common block association.

In addition to errors described below, intercompilation analysis also detects the presence of recursion when you specify the RCHECK suboption of the ICA compile-time option. ICA (RCHECK) issues a warning message if it detects recursion in your programs.

Note: The CVAR and CLLEN suboptions of the ICA compile-time option must be specified to analyze information for named common blocks.

Conflicting Argument Usage

One of the most common interprocedural errors is the incorrect use of arguments in subroutine calls or function references. The intercompilation analysis feature detects violations of the following conditions:

Conflicting Type and Length: The length and data type of the actual arguments in the calling program unit must agree with the length and data type of the dummy arguments in the called subprogram.

This example illustrates conflicting data types:

Calling Program Unit:

```
REAL*4 A, B  
PARALLEL CALL SUB (A, B)
```

Called Subprogram:

```
SUBROUTINE SUB (I, J)  
INTEGER*4 I, J
```

Conflicting Array Specifications: Array specifications should conform to the following conditions:

- The number of dimensions in an array passed by the calling program unit should be the same as the number of dimensions specified in the called subprogram.

In this example, the number of dimensions in the calling program unit is different from the number specified in the called subprogram.

Calling Program Unit:

```
REAL*4 A(2, 7)  
CALL SUB (A)
```

Called Subprogram:

```
SUBROUTINE SUB (X)  
REAL*4 X(14)
```

- The shape of an array passed by the calling program unit should agree with that of the array as it is specified in the called subprogram. In other words, the number of elements in each dimension of an array of the calling program unit must agree with the number of elements in the corresponding dimension of the array as specified in the called subprogram.

The following example illustrates inconsistencies in specifying the shapes of arrays in the calling program unit and called subprogram.

Calling Program Unit:

```
REAL*4 A(4, 7)  
SCHEDULE TASK 1, CALLING SUB(A)
```

Called Subprogram:

```
SUBROUTINE SUB (X)  
REAL*4 X(7, 4)
```

- The size of an array passed by the calling program unit should agree with the size of the array as specified in the called subprogram.

Conflicting Character Variable Lengths: The length of a character variable passed by the calling program unit must agree with that specified in the called subprogram.

The following example illustrates conflicting character variable lengths:

Calling Program Unit:

```
CHARACTER*8 B  
CALL SUB (B)
```

Called Subprogram:

```
SUBROUTINE SUB (Y)
  CHARACTER*35 Y
```

Array Misalignment: A REAL*8 array passed by the calling program unit must be aligned on a double-word boundary when VECTOR is specified in the called subprogram.

The following example illustrates incorrect array alignment. If array A is aligned on a double-word boundary, array B will be aligned on a single-word boundary.

Calling Program Unit:

```
REAL*4 A(7)
REAL*8 B(3)
EQUIVALENCE (A(2), B(1))
CALL SUB (B)
```

Called Subprogram:

```
SUBROUTINE SUB (Y)
  REAL*8 Y(3)
```

Conflicting Number of Arguments: The number of arguments specified in the calling program unit should match that specified in the called subprogram.

In the following example, the number of arguments specified in the calling program unit does not match the number of arguments specified in the called subprogram.

Calling Program Unit:

```
CALL SUB (A, B, C)
```

Called Subprogram:

```
SUBROUTINE SUB(X, Y)
```

Conflicting Number of Alternate Returns: The number of alternate returns in the calling program unit should match that specified in the called subprogram.

The following example illustrates conflicting numbers of alternate returns:

Calling Program Unit:

```
CALL SUB (A, B, *10)
```

Called Subprogram:

```
SUBROUTINE SUB (X, Y, *, *)
```

Conflicting Argument Class: The argument class—that is, whether the argument is an array or scalar variable—specified in the calling program unit should match that specified in the called subprogram; for example, an actual scalar argument should not be passed to a dummy array name.

The following example illustrates conflicting argument classes:

Calling Program Unit:

```
REAL*4 A
CALL SUB (A)
```

Called Subprogram:

```
SUBROUTINE SUB (X)
REAL*4 X(7)
```

Certain ambiguous situations cannot be diagnosed; for example, if the calling program unit passes XYZ(1), it cannot be determined whether XYZ(1) is a scalar or the beginning of an array.

Modification of Constants and Expressions: Constants or expressions must not be modified in the called subprogram. A constant or expression is followed down the CALL chains until it is determined whether the constant has been modified.

If a recursive call is detected while following a constant down a CALL chain, the search is terminated. The following example illustrates invalid modification of constants and expressions:

Calling Program Unit:

```
CALL SUB (1.7, B+2.6)
```

Called Subprogram:

```
SUBROUTINE SUB (X,Y)
X = 32.8*Y + 17.3
Y = SQRT(X)
```

The following example illustrates a recursive call and an undefined subroutine call while following a constant down a CALL chain:

Calling Program Unit:

```
Y = 55.
CALL SUB1 (1.3, Y)
```

Called Subprogram 1:

```
SUBROUTINE SUB1 (X1,Y1)
CALL SUB2 (X1,Y1)
```

Called Subprogram 2:

```
SUBROUTINE SUB2 (X2,Y2)
CALL SUB3 (X2,Y2)
```

Called Subprogram 3:

```
SUBROUTINE SUB3 (X3,Y3)
CALL SUB1 (X3,Y3)
CALL SUBB (X3,Y3)
```

Invalid Modification of Arguments: If the calling program unit causes a dummy argument to share the same storage as another dummy argument, neither dummy argument can be defined in the called subprogram.

In the following example, the EQUIVALENCE statement in the calling program unit causes the dummy arguments, X and Y, in the called subprogram to share the same storage. Therefore, the modification of the dummy argument Y is invalid.

A check is made only to ensure that the called subprogram does not reference an argument that was not defined in the caller. If the argument was passed through the caller from another program unit, and was not defined in the caller, a message is produced. No attempt is made to trace back before the immediate caller.

Calling Program Unit:

```
EQUIVALENCE (Q,R)
CALL SUB(Q,R)
```

Called Subprogram:

```
SUBROUTINE SUB (X,Y)
Y = X + C
```

Undefined Arguments: Arguments referenced in a called subprogram must be defined in the calling program unit. No attempt is made to trace back before the immediate caller.

In the following example, the actual arguments, Q and R, are not defined. Therefore, references to the corresponding dummy arguments, X and Y, are invalid.

Calling Program Unit:

```
CALL SUB(Q,R)
END
```

Called Subprogram:

```
SUBROUTINE SUB (X,Y)
C = X + Y
```

Conflicting Function Type

The type specified for a function in a program unit that references it should be the same as the type specified in the function subprogram. A function referenced as REAL when the function type is specified as INTEGER is listed as an error.

An example of conflicting function typing is:

Calling Program Unit:

```
REAL TRANSLATE
XPOSN = TRANSLATE(X1,X2)
```

Called Subprogram:

```
FUNCTION TRANSLATE(I1,I2)
INTEGER TRANSLATE
```

Conflicting External Name Usage

External names in the calling program unit and called subprograms are checked to ensure that the names are used consistently across compilations. For example, a name used as a function in one subprogram should not be defined as a subroutine in another. If this condition occurs within a single compilation unit, it is diagnosed when that program unit is compiled.

The following examples illustrate inconsistent external name usage:

Calling Program Unit:

```
CALL ABCD(3.4)
```

Called Subprogram:

```
SUBROUTINE SUB(X)
COMMON /ABCD/A,B,C,D
```

Calling Program Unit:

```
CALL WXYZ(3.2)
```

Called Subprogram:

```
FUNCTION WXYZ(X)
```

Calling Program Unit:

```
X=WXYZ(3.2)
```

Called Subprogram:

```
SUBROUTINE WXYZ(X)
```

Conflicting Common Block Lengths

The lengths specified for named common blocks should agree in all program units within an executable program. Violation of this rule may not always produce errors at run time.

The following example illustrates conflicting common lengths:

Calling Program Unit:

```
COMMON /COM1/ A, B, C  
REAL*4 A, B, C
```

Called Subprogram:

```
COMMON /COM1/ A, B, C  
REAL*4 A, B  
REAL*8 C
```

Conflicting Common Block Storage Assignment

To ensure correct program references to common blocks, a named common block must not be used in one program unit as one type of common block and used in another program unit as a different type of common block.

If a named common block is declared as a dynamic common block, all program units sharing that common block must declare it as dynamic.

In the following example, the called subprogram declares the common block ABC as dynamic but the calling program does not.

Calling Program Unit:

```
@PROCESS  
COMMON /ABC/ A, B, C  
REAL*4 Q, R  
CALL SUB(Q,R)
```

Called Subprogram:

```
@PROCESS DC(ABC)  
SUBROUTINE SUB (X,Y)  
COMMON /ABC/ A, B, C
```

Similarly, if a named common block is used in one program unit as an extended common block, it cannot be used in a different program unit as a dynamic or static common block. The intercompilation analyzer detects such conflicts at compile time.

Any attempt to define a named common block as extended, dynamic, or static while another common block of the same name already exists in a different context, will cause the running program to be terminated and an appropriate error message to be issued.

Conflicting Argument and Common Block Association

If a variable named in a COMMON statement is also passed as a dummy argument to a subprogram, and if the subprogram receives the variable both in a COMMON statement and as a dummy argument, then the variable must not become defined in the called subprogram. It must also not become defined in any subprogram called by the called subprogram. If so, either the dummy argument or the associated common member may be invalidly modified in the subprogram.

In the following example, the calling program passes the variable in the common block and as an argument to the called subprogram. The called subprogram receives the variable as two different variables—one as a common block entity, and the other as an argument passed by the subprogram:

Calling Program Unit:

```
COMMON B
CALL XYZ (B)
```

Called Subprogram:

```
SUBROUTINE XYZ (A)
COMMON C
A = 3.3
```

When to Use Intercompilation Analysis

This section suggests several instances when you might want to use the intercompilation analysis feature.

Suppose your installation is developing a large application containing program units written by many programmers. When a clean compilation of a program unit is produced, that program unit is added to the data base containing the rest of the coded program units.

In spite of the fact that a single program unit compiled successfully, there may be incompatibilities between the actual argument lists passed to other subprograms and the dummy arguments specified for those subprograms. There may even be incompatibilities in the way a subprogram is referenced and the way it is specified; for instance, it may be called as a subroutine, but it may have been defined as a function. In addition, even though all programmers use the same named common definitions, inconsistencies may exist in the types specified for some of the variables in a common. An example of the kind of error that would cause different common lengths is typing a REAL variable as length 8 instead of length 4.

In order to detect such errors before combining all program units to be run, you can use the intercompilation analysis feature to check them before adding them to the data base. First make sure, however, that none of the program units have compilation errors of level 8 or higher; such errors prevent the program units from being analyzed.

Once all the errors detected by the intercompilation analysis feature are corrected, the data defining the interfaces between this program unit and other program units

in the application can be incorporated into an intercompilation analysis file for other programmers to use as they finish coding their program units.

Time spent in the testing cycle can be reduced significantly by eliminating many of these types of errors—errors that can be very difficult and time-consuming to identify.

Managing Large Programs with Intercompilation Analysis

A group of program units that make up a common library—a collection of matrix handling subroutines, for example—can be compiled and the entries describing their interfaces added to a single intercompilation analysis file using the UPDATE suboption. There can be separate intercompilation analysis files for each such library.

Those entries for program units which are specific to a single application program can be saved in another intercompilation analysis file. Several programmers may share this file, or create their own private copies of it, as they update old program units or create new ones. When final changes are applied to the production version of the program, the shared intercompilation analysis file(s) can then be updated.

Managing Small Programs with Intercompilation Analysis

Small programs, consisting of only a few program units, can be easily maintained using a single intercompilation analysis file. Whenever a program unit is updated and recompiled, the intercompilation analysis file can be updated by specifying the name of the file in the UPDATE suboption. A module cross-reference can be generated for documentation by using the MXREF suboption.

Using Intercompilation Analysis

To use intercompilation analysis, specify the ICA compile-time option, as well as any suboptions you want, when you invoke the compiler. If you do not want to analyze certain program units, specify NOICA on an @PROCESS statement for each of those program units.

Using the USE, UPDATE, and DEF Suboptions

The USE and DEF suboptions specify the names of intercompilation analysis files containing entries from previous compilations. For both the USE and DEF suboptions, the number, type, and usage of the program unit's arguments will be checked for consistency against the arguments in the called subprograms listed in the named ICA files. For the USE suboption only, all of the arguments in the subprograms listed in the ICA files will then be checked for their consistency across calls. The file name in the UPDATE suboption may be the name of a new file, or it may be the name of a file containing information from previous compilations.

Your installation may have a number of files containing information derived from compilations using the intercompilation analysis feature. One file may contain entries describing the defined interfaces for a set of general-purpose program units used by many projects; another may have entries for program units used by a single application.

Before attempting to run new program units for the application, you can analyze the program units to detect discrepancies in the use of arguments, in the use of external names, or in the lengths specified for named common blocks. At the same

time, you can update the intercompilation analysis file containing entries for program units associated with that application.

For example, assume that your installation has a collection of subprograms that are used by all the developers. Before these subprograms were made available for general use, they were analyzed to ensure that there were no inconsistencies in argument specification and usage that would prevent them from running together successfully.

An intercompilation analysis file, containing information describing interfaces to these subroutines and functions, was created by compiling all the subprograms with the UPDATE suboption as follows:

```
ICA (UPD (GENERAL))
```

Until the compilation was run, the intercompilation analysis file GENERAL did not exist; the file was created at the end of the compilation.

To compile the subprograms, issue the command:

```
FORTVS2 GENSUB (ICA (UPD (GENERAL))
```

GENSUB FORTRAN contains:

```
SUBROUTINE GENSUB1(A, B, C)
:
END
SUBROUTINE GENSUB2(A, B)
:
END
FUNCTION GENFUN1(A, B, C, D)
:
END
FUNCTION GENFUN2(A)
:
```

These compilations create the file, GENERAL. As new subprograms are completed, entries describing their interfaces can be added to the file by compiling them with the same option:

```
ICA (UPD (GENERAL))
```

Records describing the interface for a specific subprogram can be replaced by recompiling that subprogram.

As programmers develop and code additional program units, they can use the intercompilation analysis feature to check their coded interfaces against the interfaces in the intercompilation analysis file GENERAL by compiling the program units with the option:

```
ICA (USE (GENERAL))
```

To save entries that describe subprograms unique to specific projects, they compile those subprograms with:

```
ICA (USE (GENERAL) UPD (MINE))
```

With this option, the interfaces between the new subprograms and the subprograms whose interfaces are described in GENERAL are checked. A new intercompilation

analysis file named MINE with entries describing the interfaces of the new subprograms for a specific project is created.

To create the intercompilation analysis file, MINE, the new subprograms are compiled with:

```
FORTVS2 MYSUB (ICA (USE (GENERAL) UPD (MINE))
```

MYSUB FORTRAN might contain:

```
SUBROUTINE MINE(A)
  :
  :
END
SUBROUTINE MINE2(A,B)
  :
  :
END
FUNCTION MINEF1(A,B)
  :
  :
END
  :
```

In this instance, the intercompilation analysis file GENERAL is used to compare the interfaces for the new subprograms with interfaces for existing subprograms. In addition, entries for new interface specifications are added to the intercompilation analysis file MINE.

Search Order for Intercompilation Analysis Files: The sequence in which the file names appear in the USE, UPDATE and DEF suboptions determines the search order used during the search for duplicate external name definitions—that is, program unit names, entry names, and common names. You will probably want to specify the UPDATE suboption first; however, if you want another search order, you may place the UPDATE suboption anywhere in the sequence. You can, for instance, place the UPDATE suboption between two USE suboptions.

```
ICA (USE (FILE1) UPD (FILE2) USE (FILE3,FILE4))
```

The first occurrence of the external name definition is considered to be the valid one; however, names in newly-compiled programs are given priority. The search order for external name definitions is:

1. New compilations
2. Names specified in the USE, UPDATE and DEF suboptions.

Use care in the way you specify USE, UPDATE and DEF to be sure that the external name definition you want to be considered the valid name is either a name in one of the newly-compiled programs, or is the first name encountered in the files specified in the USE, UPDATE and DEF suboptions.

Considerations for Intercompilation Analysis Files in CMS: The intercompilation analysis file name is the file name specified in the USE, UPDATE or DEF suboption; the file type is ICAFILE; and the file mode is determined as follows:

- If the file is a new file, the UPDATE file is written to the same disk as the source.
- If the file is an existing file, it is written to the disk containing the file to be updated.

If the source program or a file specified in the UPDATE suboption resides on a read-only disk, the compiler looks for a disk that can be written to and writes the new or updated intercompilation analysis file to that disk.

Considerations for Intercompilation Analysis Files in MVS: An intercompilation analysis file name used in an MVS environment is a ddname. At compilation time, there must be a valid DD statement for the ddname. Do not use DISP='MOD' in the DD statement for the intercompilation analysis file. The data set organization must be physical sequential.

Allocating Space for an Intercompilation Analysis File: When you are creating a new intercompilation analysis file, you must be sure to allocate enough space for the file. The following information should help you determine how much space you will need.

The intercompilation analysis file contains one record for each definition, reference, or COMMON definition. The amount of space each record requires depends on the number and class of the arguments and the lengths of the argument names as follows:

- Each definition, reference, or COMMON block requires 60 bytes.
- Each argument requires 13 bytes, plus the length of the argument name. If the argument represents an array, you will need an additional 4 bytes for each dimension.
- If you use the CVAR suboption you will need an additional 10 bytes, plus the length of the name, for each CVAR entry.

Figure 3 on page 14 shows the device type and device class for intercompilation analysis files; Figure 4 on page 16 shows the default values for data set characteristics for intercompilation analysis files.

Using the MSGON and MSGOFF Suboptions to Suppress Messages

When using intercompilation analysis for a program in which you both expect and allow certain error conditions to occur, you may want to suppress the related messages. You can do so by using either the MSGON or MSGOFF suboption. With the MSGON suboption, you can specify that only certain messages be issued; or, conversely, with the MSGOFF suboption, you can specify that certain messages not be issued. You cannot specify both MSGON and MSGOFF.

Figure 114 lists the message numbers, the corresponding message text, and an explanation of each message.

Figure 114 (Page 1 of 2). Intercompilation Analysis Messages

Number	Text	Explanation
61	CONFLICTING NAME USAGE	Same name referred to as a subprogram and common
62	INCORRECT NO. OF ARGS.	Number of arguments do not agree
63	CONFLICTING ARGUMENT CLASS (SECONDARY ENTRY)	For example, scalar vs. array
64	INCORRECT FUNCTION TYPE	For example, real vs. integer
65	CONFLICTING ARGUMENT CLASS (MAIN ENTRY)	For example, scalar vs. array

Figure 114 (Page 2 of 2). Intercompilation Analysis Messages

Number	Text	Explanation
67	CONFLICTING NAME USAGE	Subroutine vs. Function
68	CONFLICTING NAME USAGE	Subroutine vs. Function entry
71	CONFLICTING COMMON LENGTH	Named common lengths differ
72	MODIFIED CONSTANT ARGUMENT	Subprogram modifies a constant passed as an argument
73	ARGUMENT SIZE (STRING)	Length of character string or Hollerith constant doesn't match dummy argument
74	ARRAY DIMENSIONS/SHAPE	Number of dimensions or shape don't agree
75	MISALIGNED ARRAY	Not on required boundary
76	IGNORED NAME	Ignored because of conflicts
77	REMOVED NAME	Entries replaced in the intercompilation analysis file
78	CONFLICTING NAME USAGE	Name defined as both a common and subprogram
80	REMOVED NAME	See 77 (intercompilation analysis file not saved)
81	REMOVED NAME	Replaced in analysis by prior intercompilation analysis file
83	CONFLICTING COMMON USAGE	Static vs. dynamic vs. extended
84	UNDEFINED ARGUMENT	Referenced in called; undefined in caller
86	INVALID ARGUMENT MODIFICATION	Modification of arguments sharing storage
90	INCORRECT NO. OF ALT. RETURNS	Number of alternate entries do not agree
91	CONFLICTING TYPE (ENTRY)	For example, real vs. integer
92	CONFLICTING TYPE (MAIN ENTRY)	For example, real vs. integer
93	ARGUMENT SIZE (ARRAY)	Sizes of arrays do not agree
190	INVALID ARGUMENT/COMMON ASSOCIATION	Either the dummy argument or the associated common member may be invalidly modified in called subprogram
192(1)	INCOMPLETE CONSTANT ARGUMENT CHECK	A circular call is detected when checking for a modified constant
194(1)	INCOMPLETE CONSTANT ARGUMENT CHECK	An undefined subroutine call is detected when checking for constant modification

Note:

1. These messages cannot be suppressed.

Using Intercompilation Analysis with Non-Fortran Program Units

Program units in your application that are written in languages other than VS FORTRAN—in assembler, for instance—can still be included in the analysis.

First, create a Fortran subroutine or function with the same name and dummy arguments as those of the actual assembler program unit, thus creating a Fortran interface describing the assembler program unit. Then, compile this subroutine/function and add it to the intercompilation analysis file, thus making it available for intercompilation analysis.

Suppose, for example, you want to use an assembler program unit called SCNR2L to scan a string, from right to left, for a certain character.

Invoke SCNR2L from a Fortran subprogram as follows:

```
CALL SCNR2L (STRING,CHAR,POSITION)
```

A Fortran subroutine which describes the assembler procedure might be:

```

      SUBROUTINE SCNR2L (STRING,CHAR,POSITION)
      *
      *   Scan a character STRING from right to left for the
      *   character CHAR. Return the POSITION of the character.
      *
      CHARACTER*(*)  STRING
      CHARACTER*1     CHAR
      INTEGER*4       POSITION
      POSITION = 1
      RETURN
      END

```

Sample Programs Compiled with Intercompilation Analysis

Figure 115 shows the source language coding for a group of program units to be compiled together, with ICA specified. Figure 116 on page 394 shows the options specified for ICATEST.

IF DO	ISN	*.....1.....2.....3.....4.....5.....6.....7.*.....8
		C
		C PROGRAM TO ILLUSTRATE THE USE OF THE ICA OPTION
		C
	1	PROGRAM ICATEST
	2	REAL*8 R8S, R8A(8,3)
	3	COMPLEX*8 CX8
	4	CALL ICASUB1 (0,R8S)
	5	CALL ICASUB2 (CX8,4,R8A)
	6	CALL ICASUB3 (R8S,4)
	7	END
		.
		.
	1	SUBROUTINE ICASUB1(X,Y,Z)
	2	COMMON /ICACOMM/ ARRAY(3,8), C16
	3	COMPLEX*16 C16
	4	REAL*8 Y
	5	IF (Z .EQ.0) THEN
1	6	X = SIN(Y)
1	7	ELSE
1	8	X = SIN(Y/X)
1	9	ENDIF
	10	Y = ICASUB2(C16,3,ARRAY) + X
	11	RETURN
	12	END
		.
		.
	1	REAL FUNCTION ICASUB2(X,IA,Z)
	2	COMMON /ICACOMM/ ARRAY(3,8), C16
	3	REAL*8 ARRAY
	4	COMPLEX*8 X
	5	REAL*4 Z(8,3)
	6	DO 10 I = 1,IA
1	7	10 Z(I,2) = REAL(X) + ARRAY(IA,3)
	8	ICASUB2 = Z(8,3)
	9	RETURN
	10	END

Figure 115. ICATEST Output Listing

Intercompilation Analysis

LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 PAGE: 1

REQUESTED OPTIONS (EXECUTE): ICA NOTERM

OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK SOURCE NOTERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT SDUMP(ISN)
NOSXM NOVECTOR IL(DIM) NOTEST NODC ICA NODIRECTIVE NODBCS NOSAA NOPARALLEL NOSAVE NOTABS
OPT(0) LANGLVL(77) NOFIPS FLAG(I) AUTODBL(NONE) LINECOUNT(60) CHARLEN(500)
ICA: MXREF(L) CLEN MSG(NEW)

Figure 116. Options Specified for ICATEST

As you can see, ICA was specified with no suboptions; therefore, no intercompilation analysis files were used in the analysis. The group was analyzed only for consistency of the external references within the group.

Output from the Sample Program

As no suboptions were specified, the default suboption MXREF was in effect and produced the External Cross Reference listing shown in Figure 117.

LEVEL 2.6.0 (NOV 1993) VS FORTRAN NOV 10, 1993 09:42:18 PAGE: 4

EXTERNAL CROSS REFERENCE

ARGUMENT USAGE: (ONE CHAR/ARGUMENT) A-ARGUMENT PASSED D-FETCHED & PASSED AS ARGUMENT S-SET
 B-SET & FETCHED E-SET & FETCHED & PASSED AS ARGUMENT U-UNREFERENCED
 C-SET & PASSED AS ARGUMENT F-FETCHED

NOTES: NO. - NNN: COMPILATION NUMBER FN: ICA FILE NUMBER
REFERENCE - TRAILING ASTERISK: THIS NAME IGNORED IN ERROR ANALYSIS AND NOT WRITTEN TO AN ICA FILE
 - (NNN): MULTIPLE REFERENCES

1 NAME	2 NO.	3 TYPE	4 ARGUMENT USAGE	5 REFERENCES
ICASUB1	2	SUBROUTINE	BBF	ICASUB2
ICASUB2	3	FUNCTION(R*4)	FFB	
ICATEST	1	MAIN PROGRAM		ICASUB1 ICASUB2 ICASUB3

6 NAME	7 TYPE	8 IS REFERENCED BY
ICACOMM	COMMON	ICASUB1(B) ICASUB2(F)
ICASUB1	SUBROUTINE	ICATEST
ICASUB2	FUNCTION(R*4)	ICASUB1 ICATEST
ICASUB3	SUBROUTINE	ICATEST

Figure 117. ICATEST Output Listing - External Cross Reference

The external cross reference consists of two tables. The columns of the first table show:

- 1** Truncated names of all analyzed program units
- 2** Number indicating the sequence of the module in the compilation. If the number is preceded by F, it identifies the intercompilation analysis file containing the entries for the name in the NAME column.

- 3** Program unit type
- 4** Argument usage (see the top of the listing for a description of the symbols).
- 5** Truncated names of the program units referenced by each module

The columns of the second table show:

- 6** Names of subroutines, functions, entry points, block datas, and commons
- 7** Type of program unit. If the referenced name is an entry in a subprogram, it is so indicated.
- 8** Truncated names of the program units that reference the program unit specified in NAME.

The messages in Figure 118 were generated during the analysis and indicate problems which can produce incorrect results when the application is run.

```

LEVEL 2.6.0 (NOV 1993)      VS FORTRAN      NOV 10, 1993  09:42:18      PAGE: 5

*** INTERCOMPILATION MESSAGES. ***
  NUMBER  MODULE  LEVEL  ISN  VS FORTRAN ERROR MESSAGES
ILX0071I  ICIA    4(W)    CONFLICTING COMMON LENGTH -- THE LENGTHS OF COMMON BLOCK ICACOMM IN ICASUB1(COMPI-
ILX0062I  ICIA    4(W)    INCORRECT NO. OF ARGUMENTS -- THE NUMBER OF ARGUMENTS FOR ICASUB1 IN ICATEST(COMPI-
ILX0072I  ICIA    4(W)    MODIFIED CONSTANT ARGUMENT -- ARGUMENT NO. 1 TO ICASUB1 AT ISN 4 IN ICATEST(COMPI-
ILX0092I  ICIA    4(W)    CONFLICTING TYPE -- AT THE MAIN ENTRY TO SUBROUTINE, ICASUB1(COMPI-
ILX0068I  ICIA    4(W)    CONFLICTING NAME USAGE -- THE NAME, ICASUB2, HAS BEEN REFERENCED AS A SUBROUTINE IN
ILX0084I  ICIA    4(W)    UNDEFINED ARGUMENT -- ARGUMENT NO. 1, "X", TO ICASUB2 IS REFERENCED BUT THE ARGUMENT PASSED AS
ILX0092I  ICIA    4(W)    CONFLICTING TYPE -- AT THE MAIN ENTRY TO FUNCTION, ICASUB2(COMPI-
ILX0093I  ICIA    4(W)    ARGUMENT SIZE -- THE SIZE OF THE ARRAY IN ARGUMENT NO. 3, "R8A", TO ICASUB2 AT ISN 5 IN
ILX0064I  ICIA    4(W)    FUNCTION TYPE -- FUNCTION, ICASUB2, REFERENCED IN ICASUB1 (COMPI-
ILX0092I  ICIA    4(W)    CONFLICTING TYPE -- AT THE MAIN ENTRY TO FUNCTION, ICASUB2(COMPI-
ILX0074I  ICIA    4(W)    ARRAY DIMENSIONS -- THE SHAPE OF THE ARRAY IN ARGUMENT NO. 3, "ARRAY", TO ICASUB2 AT ISN 10 IN
***** INTRACOMPILATION STATISTICS ***** 0 DIAGNOSTICS GENERATED. HIGHEST SEVERITY CODE IS 0.
***** INTERCOMPILATION STATISTICS *(ICA)* 11 DIAGNOSTICS GENERATED.

```

Figure 118. ICATEST Output Listing - Compilation Messages

Chapter 20. The Multitasking Facility (MTF)

Introducing MTF	398
What MTF Is	398
What MTF Does	398
Initialize	399
Schedule	399
Synchronize	399
The Concept of Computational Independence	400
Running a VS FORTRAN Version 2 Program Without MTF	400
Running a VS FORTRAN Version 2 Program With MTF	401
Running with One Parallel Subroutine	402
Processor Use	402
Sample Program	403
Running with Two Different Parallel Subroutines	403
Processor Use	404
Sample Program	404
Running with Multiple Instances of the Same Parallel Subroutine	405
Processor Use	405
Sample Program	406
Designing and Coding Applications for MTF	406
Step 1: Identify Computationally-Independent Code	407
Step 2: Create Parallel Subroutines	407
Step 3: Insert Calls to Parallel Subroutines	408
Examples of Changing Applications to Use MTF	409
Example 1	409
Example 2	412
Coding Rules	414
Compiling and Linking Programs That Use MTF	414
Creating the Main Task Program Load Module	414
Creating the Parallel Subroutines Load Module	415
Link-Editing Considerations	415
Running Programs That Use MTF	416
AUTOTASK Keyword in the EXEC Statement	416
AUTOTASK DD Statement	416
DD Statements for Unnamed Files	417
Example of JCL	417
Debugging Programs That Use MTF	418
Using MTF with Load Mode	418
What to Avoid When Using MTF	418
Converting MTF Programs to Parallel Programs	418

Introducing MTF

What MTF Is

MTF can be used by computationally-intensive application programs to improve turnaround time on tightly-coupled System/370* or System/390* multiprocessor (MP) and attached-processor (AP) configurations (for example, the 3090*-200 or ES/9000 500). When a program uses MTF on such a system, the elapsed time required to run the program can be reduced.

MTF is easy to use and requires very little knowledge of the MVS multitasking capabilities upon which it depends. From the programmer's perspective, MTF is simply four VS FORTRAN Version 2-supplied subroutines, and a subparameter on the EXEC statement. Because of this simplicity, it is easy to introduce MTF to existing applications and code new MTF applications to gain the benefits of multitasking.

Note: Programs that use MTF are not the same as parallel programs that use the VS FORTRAN Version 2 parallel language constructs or contain parallel code generated automatically by VS FORTRAN Version 2. **You cannot use MTF and the VS FORTRAN Version 2 parallel feature together. You cannot use MTF and VS FORTRAN Version 2 extended common together.**

What MTF Does

MTF takes advantage of the multitasking capabilities of the MVS operating systems to allow a single application program to use more than one processor of a multiprocessing configuration simultaneously. (MTF provides multitasking only on the MVS operating systems.) MVS operating systems organize all work into units called tasks. These tasks are used by the operating system to dispatch work to the processors of the multiprocessor configuration.

MTF's facilities allow a single application to be organized so it can be run in a "main task" and in one or more "subtasks." As a result of this organization, the system can schedule these individual tasks to run simultaneously. This can significantly reduce the elapsed time needed to run the program.

When a Fortran program is organized in this manner, the main task runs the part of the program that controls the overall processing. This part is referred to as the *main task program* throughout this manual.

The subtasks run the portions of the program that can run independently of the main task program and of each other. These portions of the program are referred to as parallel subroutines. The functions provided by MTF allow the main task program to schedule and all the parallel subroutines to run independently.

The parallel subroutines are coded the same as normal Fortran subroutines, with the exception of a few rules discussed under "Designing and Coding Applications for MTF" on page 406. They can perform I/O and can share large amounts of data with the main task program by means of dynamic common blocks.

* System/370, System/390, and 3090 are trademarks of the International Business Machines Corporation.

MTF can be thought of as three functions that do the following:

- Initialize the MTF environment
- Schedule parallel subroutines to run
- Synchronize their completion.

Initialize

The Library creates the MTF environment required to run the separate parts of a program simultaneously. This initialization occurs when the keyword AUTOTASK is specified in the PARM parameter of the EXEC statement used to run the program. The AUTOTASK keyword has two subparameters associated with it. The first subparameter is the name of a load module that contains the program's parallel subroutines. The second subparameter is the number of subtasks that should be created for the program. The AUTOTASK keyword is specified as:

```
AUTOTASK(loadmodname,n)
```

Schedule

The main task program schedules a parallel subroutine to run by calling the MTF subroutine DSPTCH.

```
CALL DSPTCH( subname [,arg1[,arg2]...])
```

subname

Is a character variable or literal that specifies the name of the parallel subroutine to be scheduled. The parallel subroutine is dispatched to a subtask and is run simultaneously with the main task program.

subname can be 8 characters long. If *subname* is a Fortran subprogram or has more than 8 characters, you must provide the 7-character external name created during compilation. (The external name is formed by concatenating the first three characters and last four characters.) This name appears on the External Symbol Summary produced by the compiler. See *VS FORTRAN Version 2 Language and Library Reference* for examples.

arg1, arg2, ...

Are arguments passed to the parallel subroutine.

The main task program can schedule multiple instances of a parallel subroutine for simultaneous processing by repeating the call to DSPTCH using the same parallel subroutine name, but passing different arguments with each call. Alternatively, it can schedule several different parallel subroutines.

A program can determine the number of subtasks specified with the AUTOTASK keyword by calling the MTF subroutine NTASKS.

```
CALL NTASKS(n)
```

n is an integer*4 variable in which the number of subtasks is returned.

Synchronize

The main task program synchronizes its own processing with the completion of the parallel subroutines by calling the MTF subroutine SYNCRO.

```
CALL SYNCRO
```

SYNCRO causes the main task program to wait until all of the currently-scheduled parallel subroutines finish running.

The Concept of Computational Independence

To successfully use multitasking, the parallel subroutines must have computational independence. This means that no data modified by either the main task program or a parallel subroutine is examined or modified by a parallel subroutine that might be running simultaneously.

At this spot in your hardcopy book you will find a graphic example of some hypothetical data in an array subscripted by I, J, and K. Each of the three divisions of the box represents a section of the array that could be operated on independently of the other sections. The same parallel subroutine could be scheduled three times, with each instance of the subroutine processing one of the three sections of the array.

Your application may not have computational independence along the same subscript axis of K, as in this picture. The divisions might have been along one of the other subscript axes, I or J. Also, the computational independence in your application may not fall into neat, box-like divisions.

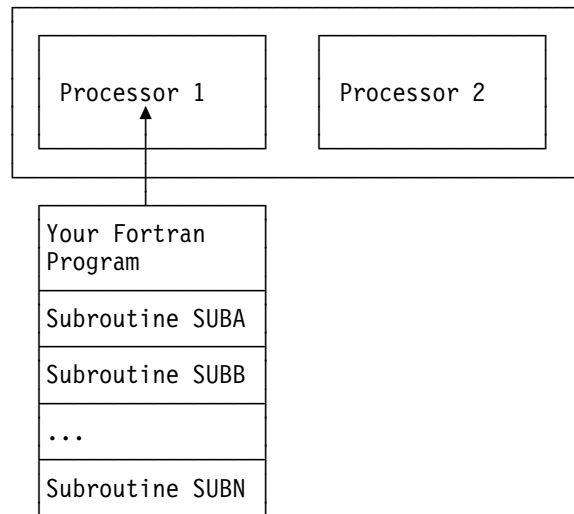
It is also possible to have computational independence that is not based on sections of the same array, but rather on separate arrays (perhaps with completely different types of data), the values of which do not depend on each other. In this case, separate parallel subroutines could be scheduled, with each subroutine processing its own unique data.

Computational independence also applies to input/output files. One parallel subroutine should not use a file while another is updating it. However, different subroutines can successfully read the same file.

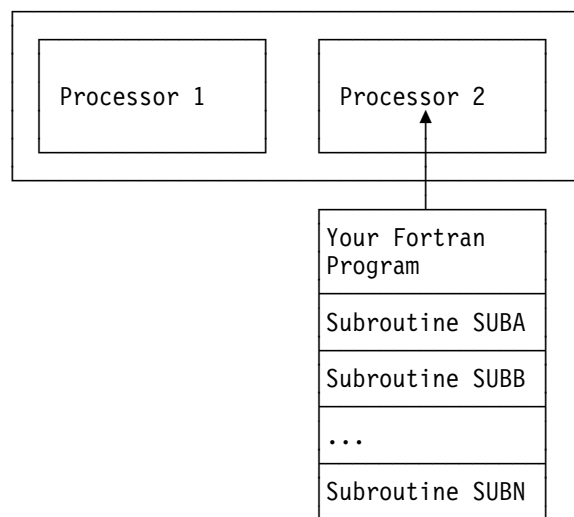
Running a VS FORTRAN Version 2 Program Without MTF

The following diagrams illustrate the way a Fortran program runs without multitasking. The program and its subroutines must run in a strictly sequential manner, routine following routine, using one processor at a time. Consequently, your program takes more elapsed time to complete than it would if it could use several processors at the same time.

In the following example, without multitasking, your Fortran program and all its subroutines can only use one processor, Processor 1...



...or the other processor, Processor 2.



While running, your program may be switched back and forth between the processors, but it can only run on one processor at a time.

Running a VS FORTRAN Version 2 Program With MTF

To illustrate the concept of multitasking, this section shows three examples of running a Fortran program with MTF. These examples show programs using:

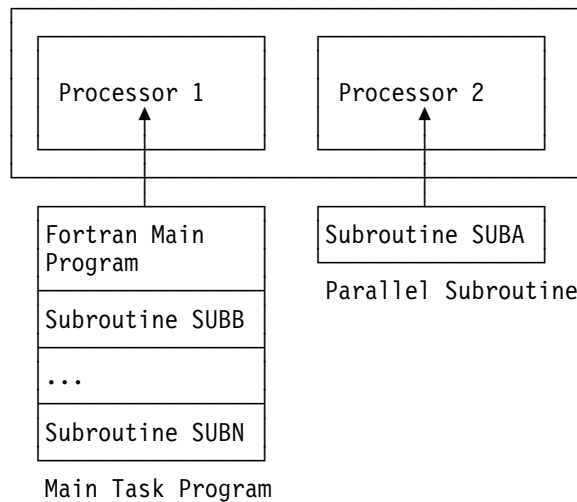
- One parallel subroutine
- Two different subroutines
- Two or more instances of the same subroutine.

Each example provides illustrations of how the processors are used and how the program is organized to accomplish the particular use of the processors.

Running with One Parallel Subroutine

If your Fortran program uses MTF, the main task program and a computationally-independent parallel subroutine can run simultaneously.

Processor Use

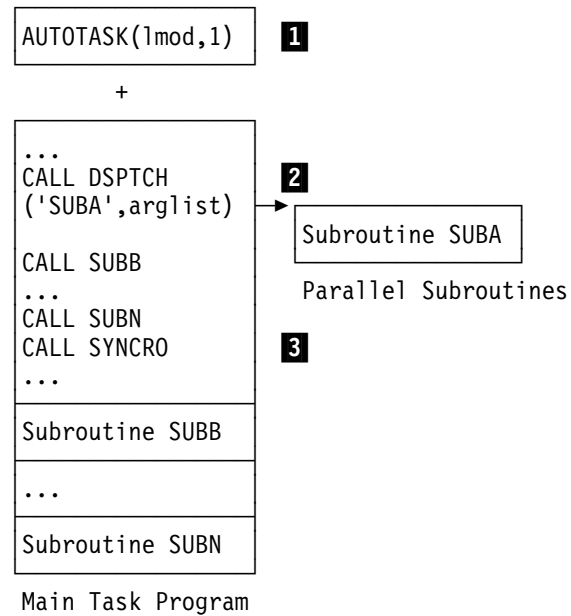


In the example to the left, only subroutine SUBA has computations that can be done independently of the main task program, which includes the Fortran main program plus its subroutines.

With the appropriate MTF request, the parallel subroutine, SUBA, is scheduled to run in a subtask.

The arrows to Processor 1 and Processor 2 are for illustration only. The main task program could have run on Processor 2 and the parallel subroutine, SUBA, on Processor 1; in fact, while they run, they may be switched among the processors.

Sample Program



What the MTF functions do:

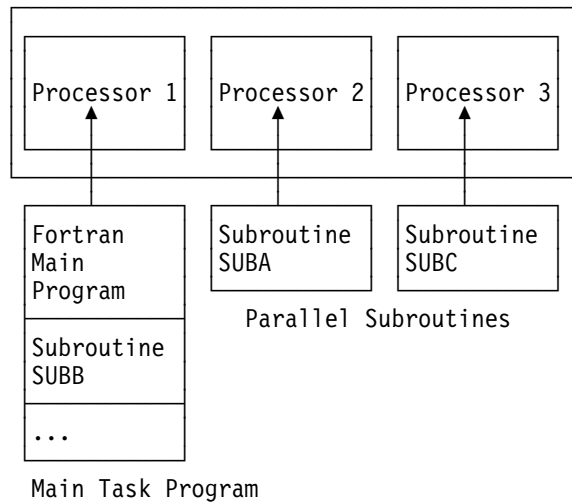
- 1** The `AUTOTASK` keyword, in the `PARM` parameter of the `EXEC` statement which runs the job, specifies one subtask.
- 2** `DSPTCH` schedules the parallel subroutine, `SUBA`, to run. `SUBA` is computationally independent of the main task.
- 3** `SYNCRO` makes the main task program wait until `SUBA` finishes before the main task program continues.

A few lines of JCL and two calls to MTF subroutines accomplish this.

Running with Two Different Parallel Subroutines

If your Fortran program uses MTF, the main task program and several different computationally-independent parallel subroutines can run simultaneously.

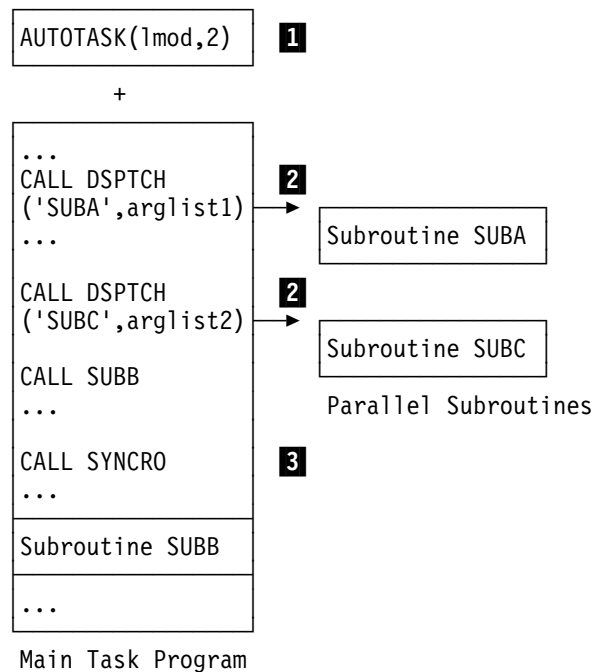
Processor Use



In the example to the left, subroutines SUBA and SUBC are independent of each other and of the main task program.

As with “Running with One Parallel Subroutine” on page 402, the arrows to Processors 1, 2, and 3 are for illustration only. The main task program and the parallel subroutines could run on any of the processors.

Sample Program



What the MTF functions do:

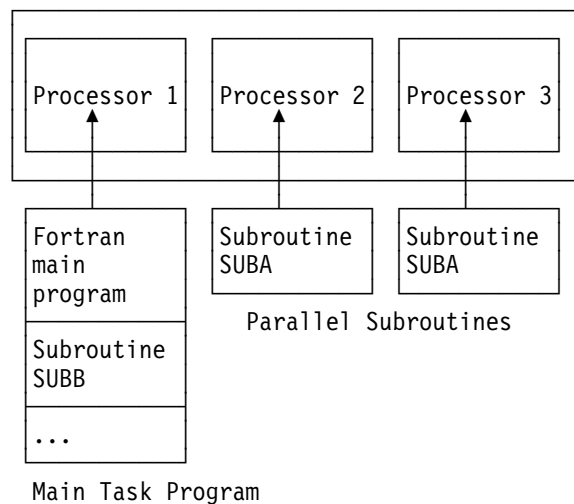
The logic is similar to that for only one parallel subroutine and can be extended to as many parallel subroutines as necessary to complete the logic of the program.

- 1** The AUTOTASK keyword in the PARM parameter of the EXEC statement which runs the job specifies two subtasks.
- 2** Each call to DSPTCH schedules one of the parallel subroutines, passing different data to each for processing. SUBA and SUBC are computationally-independent parallel subroutines.
- 3** SYNCRO makes the main task program wait until both SUBA and SUBC finish before the main task program continues its processing.

Running with Multiple Instances of the Same Parallel Subroutine

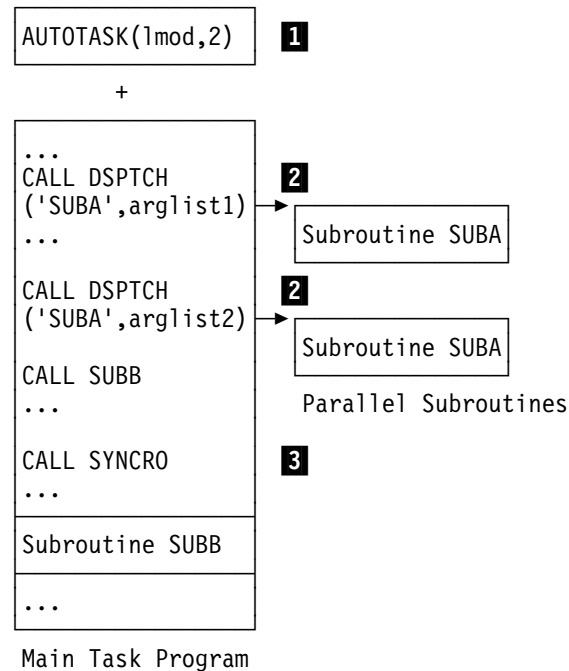
If your Fortran program uses MTF, the main task program and multiple instances of the same parallel subroutine can run simultaneously.

Processor Use



In the example to the left, parallel subroutine SUBA has data you can divide, so two instances of SUBA run independently of the main task program and of each other.

Sample Program



What the MTF functions do:

- 1** The AUTOTASK keyword in the PARM parameter of the EXEC statement which runs the job specifies two subtasks.
- 2** Each call to DSPTCH schedules one instance of the parallel subroutine to run and supplies separate data to be processed by that instance of SUBA. The data to be processed by each instance of the parallel subroutine could be two different sections of the same array. Both instances of SUBA in the loop are computationally independent of the main task program and each other, since each instance of SUBA processes different data.
- 3** SYNCRO makes the main task program wait until all instances of SUBA finish before the main task program continues.

Designing and Coding Applications for MTF

You can use the following steps when preparing a Fortran application to work with MTF:

1. Identify computationally-independent code
2. Create parallel subroutines
3. Insert calls to parallel subroutines.

You can design new programs to use MTF and reconstruct existing programs.

Step 1: Identify Computationally-Independent Code

The first step in adapting an application program for MTF is to identify groups of computations that can be performed simultaneously. In order to produce correct results, the computations that are done simultaneously must be computationally independent. Computational independence is explained under “The Concept of Computational Independence” on page 400.

Step 2: Create Parallel Subroutines

After the segments of code that are computationally independent are identified, they are separated from the main task program and placed in parallel subroutines. A parallel subroutine is coded as a normal Fortran subroutine that follows several rules required for proper operation with MTF. In addition to data independence, the rules are:

Calling Other Subroutines

- A parallel subroutine can be coded as a series of subroutines that call one another. All of these subroutines operate in the parallel subroutine’s subtask environment and must follow the rules of a parallel subroutine. However, a subroutine that is called *within* this environment can use alternate return specifiers.
- A parallel subroutine cannot call the MTF subroutines NTASKS, DSPTCH, SYNCRO, and SHRCOM. Such calls can only be used in the main task program.
- When a parallel subroutine receives control, the program mask, including the exponent underflow mask, is set to the value that was in effect in the main task program at the time DSPTCH was called to schedule the parallel subroutine. A parallel subroutine can change this setting by calling the subroutine XUFLOW; however, the change is effective only for the current instance of the parallel subroutine.

Passing Data

- A parallel subroutine is always invoked in its last-used state. If, for example, a parallel subroutine has initialized a variable with a DATA statement, then the variable has that value the first time that copy of the parallel subroutine is used. If the value is modified, the modification is available the next time that copy of the parallel subroutine is run. You cannot, however, control which copy of a parallel subroutine is used when the parallel subroutine is scheduled. Therefore, your parallel subroutine must not depend on residual values from previous uses of a copy of itself.
- Data can be passed between the main task program and parallel subroutines, and between parallel subroutines, only by means of shared dynamic common blocks or argument lists, or common files on disk.
- The dummy argument list of a parallel subroutine cannot contain an alternate return specifier (asterisk).
- If a parallel subroutine is to use a shared copy of a dynamic common block, the main task program must designate that common block as shareable before the subroutine is scheduled. The main task program designates a dynamic common block as shareable by calling the MTF subroutine SHRCOM. For information on the SHRCOM subroutine, see *VS FORTRAN Version 2 Language and Library Reference*.

- A dynamic common block that is shared among the main task program and the parallel subroutines may be the virtual storage window that corresponds to part of a data object. However, all of the data-in-virtual calls are restricted to the main task program.
- If a parallel subroutine refers to a dynamic common block that has not been designated shareable or to a static common block, a copy of the common block is acquired for the exclusive use of the subroutine and is made available to all program units within the subroutine. The common block cannot be shared with the main task program or other parallel subroutines.
- If the main task program designates as shareable a dynamic common block that has already been acquired for the exclusive use of a parallel subroutine, an error is detected.

Input/Output

- For unnamed files, the only VS FORTRAN I/O statements allowed in a parallel subroutine are:
 - INQUIRE
 - PRINT and WRITE directed to the error message unit (or directed to the standard output unit for WRITE and PRINT statements if it is different from the error message unit at your site). The IBM-supplied default for this unit is 6.
- For named files, all VS FORTRAN I/O statements are allowed. Each parallel subroutine processes the file as if it had complete control over the file; therefore, one subroutine should not use a file while another is updating it. However, different subroutines can successfully read the same file. The ACTION specifier on the OPEN statement can be used to indicate whether a file is to be updated.

A named file must be connected within each MTF subtask that uses it. The connection does not remain after the subtask finishes running—when a subtask finishes running, any named files that remain connected are automatically disconnected.
- All forms of the INQUIRE statement are allowed in parallel subroutines. The INQUIRE statement provides information about the unit or file *only as it is seen within the main task program or parallel subroutine in which the INQUIRE statement is processed*. For example, if a file is connected in subroutine A but not in subroutine B, an INQUIRE statement in subroutine B will report that the file is not connected.
- Asynchronous I/O is not allowed in parallel subroutines.

Step 3: Insert Calls to Parallel Subroutines

In the original program, replace each segment of code that was identified for simultaneous computation with a call to DSPTCH, which schedules the corresponding parallel subroutine. If concurrent operation is to be achieved by scheduling the same subroutine multiple times with different data, the CALL statement can be placed within a DO loop.

The following items must not be used as the actual arguments supplied to the parallel subroutine using the CALL DSPTCH statement:

- Expressions requiring evaluation; for example, $A+2*B**3$

- Function names
- Subroutine names
- Alternate return specifiers; that is, the form *n, where *n* is a statement label
- A DO variable if its value might be incremented before you call the SYNCRO subroutine.

After inserting calls to the parallel subroutines, insert a call to SYNCRO wherever the program requires that all previously-scheduled parallel subroutines have finished running.

The next sections show examples of how to change existing Fortran programs to use MTF following the steps just outlined.

Examples of Changing Applications to Use MTF

Example 1

Identify Computationally-Independent Code: Figure 119 shows a computation that performs a number of operations on three dimensional arrays of data. The processing within the loop structure may be separated in the K dimension. This is because each iteration of the K loop can be performed without requiring the results computed in any other iteration of the K loop. The iterations are therefore computationally independent of each other.

```

:
DO 10 K=1,KM
  VS(1,1,K) = 0.0
  DO 10 J=2,JM
    DO 10 I=2,IM
      VS(I,J,K)=VX(I,J,K)**2+VT(I,J,K)**2+VR(I,J,K)**2
      P(I,J,K) =0.5*RHO(I,J,K)*VS(I-1,J-1,K)
10 CONTINUE
:

```

Figure 119. Sample Code to Be Changed to Use MTF

Create Parallel Subroutines: The segments of the program that have been identified to run as parallel subroutines are then recoded as new Fortran subroutines. In this case, there will be one parallel subroutine, multiple instances of which will be scheduled. The parallel subroutine corresponding to the code in Figure 119 now looks like Figure 120.

```

SUBROUTINE SUB (KLIM1,KLIM2,VX,VT,VS,VR,RHO,P,IM,JM)
REAL VX(50,100,50), VT(50,100,50), VR(50,100,50)
REAL RHO(50,100,50), P(50,100,50), VS(50,100,50)

DO 10 K=KLIM1,KLIM2
  VS(1,1,K)=0.0
  DO 10 J=2,JM
    DO 10 I=2,IM
      VS(I,J,K)=VX(I,J,K)**2+VT(I,J,K)**2+VR(I,J,K)**2
      P(I,J,K) =0.5*RHO(I,J,K)*VS(I-1,J-1,K)
10 CONTINUE
RETURN
END

```

Figure 120. The Sample Code as a Parallel Subroutine

The dummy arguments KLIM1 and KLIM2 are used to specify the loop limits on the K loop so that the subroutine SUB operates over any specified range of K index values.

Insert Calls to Parallel Subroutines: The segments of the program that have been removed to form parallel subroutines are replaced by calls to them. For the sample code in Figure 119, an instance of subroutine SUB is scheduled for each subtask that will be used at run time. In order to do this, the computations controlled by the K index must be divided so that each instance of the subroutine SUB operates on a different part of the original range of the K DO variable. See Figure 121 for an example of how two instances of a parallel subroutine can be scheduled.

```

:
C SCHEDULE 2 INSTANCES OF PARALLEL SUBROUTINE SUB
  KH=KM/2
  KHS=KH+1
  CALL DSPTCH('SUB',1,KH,VX,VT,VS,VR,RHO,P,IM,JM)
  CALL DSPTCH('SUB',KHS,KM,VX,VT,VS,VR,RHO,P,IM,JM)

C WAIT FOR BOTH INSTANCES OF SUB TO FINISH RUNNING
  CALL SYNCRO
:

```

Figure 121. Scheduling Two Instances of a Parallel Subroutine

In the JCL that runs this program, code an AUTOTASK keyword (in the PARM parameter of the EXEC statement) that specifies at least two subtasks.

If you want to make your program sensitive to the actual number of subtasks that are available when the program runs, then you may call the subroutine NTASKS to determine the number of subtasks and use that value to calculate the lower and upper bounds of the K loop for each instance of the subroutine. To do this, see Figure 122. (This figure shows only a portion of the program's use of MTF.)

```

:
  INTEGER*4 KLB(10),KUB(10),NT
C DETERMINE NUMBER OF SUBTASKS AVAILABLE
  CALL NTASKS(NT)
  NT = MIN(NT,10)
  IF(NT .LT. 1) THEN
    PRINT *, ' MTF NOT INITIALIZED.'
    PRINT *, ' SPECIFY AUTOTASK KEYWORD.'
    STOP 20
  ENDIF
C COMPUTE BOUNDS FOR EACH INSTANCE OF THE SUBROUTINE
  KLB(1)=1
  KUB(NT)=KM
  DO 25 I=1,NT-1
    KUB(I)=KM*I/NT
25  KLB(I+1)=KUB(I)+1
:
  END

```

Figure 122. Calculating Lower and Upper Bounds for Each Instance of the Subroutine

As an example, assume there are three subtasks available; that is, NT is set to 3 by the NTASKS subroutine. Also assume the dimension of the K loop (that is, KM) is 10000. The upper and lower bounds are then computed as follows in Figure 123 on page 411.

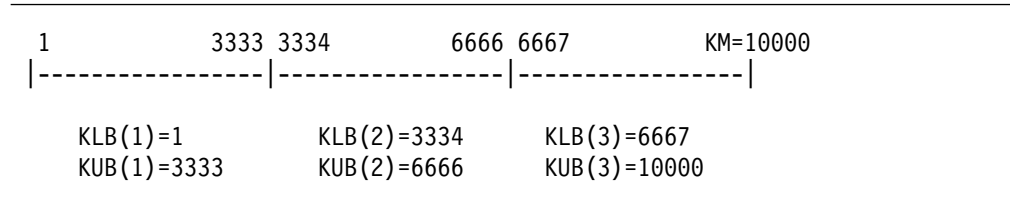


Figure 123. Lower and Upper Bounds for the K Loop

Based on the computation of the bounds in Figure 122 on page 410, the code that schedules the variable number of instances, NT, of the parallel subroutine SUB, and waits for them to finish running, would look like Figure 124.

```

:
C RUN NT INSTANCES OF PARALLEL SUBROUTINE SUB
DO 10 I=1,NT
  CALL DSPTCH('SUB',KLB(I),KUB(I),VX,VT,VS,VR,RHO,P,IM,JM)
10  CONTINUE

C WAIT FOR ALL INSTANCES OF SUB TO FINISH RUNNING
  CALL SYNCRO
:

```

Figure 124. Scheduling a Variable Number of Instances of a Parallel Subroutine

Computational independence must be maintained between the parallel subroutines and the main task program, as well as among all of the parallel subroutines. The requirements for computational independence between the main task and the parallel subroutines would be violated if variables that are used as parallel subroutine arguments were redispached, either within the scheduling DO loop or before SYNCRO is called by the main task program.

This violation of computational independence would have occurred in the above example if, instead of placing the lower and upper bounds in array elements, each bound were stored in a single variable and the calls to DSPTCH were included in the same loop that computed the bounds. This would result in each instance of the parallel subroutine using the same variables even though the values were intended to be different for each instance of the parallel subroutine.

For the same reason, if the DO variable of the scheduling loop is needed in the subroutine, then it must be placed in a separate storage location for each instance of the subroutine. That value can be passed as an argument to the subroutine in an array element, as shown in Figure 125.

```

:
C RUN NT INSTANCES OF PARALLEL SUBROUTINE XYZ
DO 10 I=1,NT
  IX(I)=I
  CALL DSPTCH('XYZ',IX(I),...)
10  CONTINUE
:

```

Figure 125. Passing the Value of the DO Variable to a Parallel Subroutine

Example 2

Not all application programs contain parallelism within the iterations of a DO loop structure. The following example illustrates parallel computations that appear as different segments of code in the original program. Also illustrated is the use of shared dynamic common areas for passing data, and I/O operations to named files in parallel subroutines.

Identify Computationally-Independent Code: Figure 126 shows two nested loops that perform operations on two-dimensional arrays of data. The maximum loop iteration values for both loops are read from a file, and a record is written after each of the two loops is processed to different files. The computation of each iteration of the loops requires the results computed in the previous iteration. Therefore, they cannot be separated and scheduled as multiple instances of the same parallel subroutine. However, the entire first nested loop is computationally independent of the entire second nested loop. The two loops can be run simultaneously in two different parallel subroutines.

```

      DIMENSION VX(100,100),VY(100,100),VA(100,100),VB(100,100)
      DIMENSION A(100,100),B(100,100),C(100,100),D(100,100)
      OPEN (1,FILE='/MAXVAL.INPUT',ACCESS='DIRECT',RECL=4)
      OPEN (2,FILE='/VA.OUTPUT')
      OPEN (3,FILE='/VB.OUTPUT')
      READ (1,REC=1) JM
      READ (1,REC=2) IM
C The following three lines represents code to initialize array values
      .
      .
      .
      DO 10 J=2,JM
        DO 10 I=3,IM
          VX(I,J) = A(I,J) + B(I,J)
          VA(I,J) = VA(I-2,J-1) + VX(I,J)**3
10    CONTINUE
      WRITE(2,FMT=*) VA(IM,JM)
      READ (1,REC=3) LM
      DO 20 L=3,LM
        DO 20 I=2,IM
          VY(I,L) = C(I,L) + D(I,L)
          VB(I,L) = VB(I-1,L-2) + VY(I,L) * 1.3
20    CONTINUE
      WRITE (3,FMT=*) VB(IM,LM)
      WRITE (*,*) 'Program has completed'
      STOP
      END

```

Figure 126. Sample Code to be Changed to Use MTF

Create Parallel Subroutines: The two loops identified as parallel computations are recoded as new Fortran subroutines. Data is passed from the main routine to the parallel subroutines by means of dynamic common areas. The parallel subroutines corresponding to the code in Figure 126 are shown in Figure 127 on page 413.

```

@PROCESS DC(DYNCOMA)
  SUBROUTINE SUBA
    COMMON /DYNCOMA/ VX(100,100),VA(100,100),
    C A(100,100),B(100,100)
    OPEN (1,FILE='/MAXVAL.INPUT',ACCESS='DIRECT',RECL=4,
    C ACTION='READ')
    OPEN (2,FILE='/VA.OUTPUT')
    READ (1,REC=1) JM
    READ (1,REC=2) IM
    DO 10 J=2,JM
      DO 10 I=3,IM
        VX(I,J) = A(I,J) + B(I,J)
        VA(I,J) = VA(I-2,J-1) + VX(I,J)**3
10  CONTINUE
    WRITE(2,FMT=*) VA(IM,JM)
    RETURN
  END

@PROCESS DC(DYNCOMB)
  SUBROUTINE SUBB
    COMMON /DYNCOMB/ VY(100,100),VB(100,100),
    C C(100,100),D(100,100)
    OPEN (1,FILE='/MAXVAL.INPUT',ACCESS='DIRECT',RECL=4,
    C ACTION='READ')
    OPEN (2,FILE='/VB.OUTPUT')
    READ (1,REC=3) LM
    READ (1,REC=2) IM
    DO 20 L=3,LM
      DO 20 I=2,IM
        VY(I,L) = C(I,L) + D(I,L)
        VB(I,L) = VB(I-1,L-2) + VY(I,L) * 1.3
20  CONTINUE
    WRITE(2,FMT=*) VB(IM,LM)
    RETURN
  END

```

Figure 127. The Sample Code as Two Subroutines

In this example there is no need to use new dummy arguments to specify the loop limits because multiple instances of each parallel subroutine are not used. Note that each subroutine must issue an OPEN statement for the shared input file MAXVAL.INPUT. The ACTION='READ' parameter is added to the OPEN statement for this file to indicate that the file is not to be updated. Because each subroutine has different output files, the default READWRITE action may be used for these files.

Insert Calls to Parallel Subroutines: The segments of the program that have been removed to form parallel subroutines are replaced by calls to those parallel subroutines. For the sample code in Figure 126 on page 412 and in Figure 127, the code that schedules the two parallel subroutines SUBA and SUBB and waits for them to finish running is shown in Figure 128 on page 414.

```
@PROCESS DC(DYNCOMA,DYNCOMB)
      COMMON /DYNCOMA/ VX(100,100),VA(100,100),
      C A(100,100),B(100,100)
      COMMON /DYNCOMB/ VY(100,100),VB(100,100),
      C C(100,100),D(100,100)
C The following three lines represents code to initialize array values
:
C ALLOW DYNAMIC COMMONS TO BE SHARED
      CALL SHRCOM('DYNCOMA')
      CALL SHRCOM('DYNCOMB')
C RUN PARALLEL SUBROUTINES SUBA AND SUBB
      CALL DSPTCH('SUBA')
      CALL DSPTCH('SUBB')
C WAIT FOR THE TWO PARALLEL SUBROUTINES TO COMPLETE
      CALL SYNCRO
      WRITE (*,*) 'Program has completed'
      STOP
      END
```

Figure 128. Scheduling Two Different Parallel Subroutines

Coding Rules

For complete information on the rules for coding calls to the MTF subroutines, consult *VS FORTRAN Version 2 Language and Library Reference*.

Compiling and Linking Programs That Use MTF

Programs that use MTF run using two MVS load modules: a load module that contains the main task program and a load module that contains the parallel subroutines. The standard VS FORTRAN Version 2 cataloged procedure VSF2CL can be used to do the compilations and link-edits needed to produce each of these two load modules.

Creating the Main Task Program Load Module

The main task program load module is the load module that first receives control when MVS starts running your program. It is the load module named in the PGM keyword of the EXEC statement. This load module contains your application's Fortran main program and all subprograms which are to run as part of the main task program.

The procedures that you normally use to compile and link-edit a program can be used to create the main task program load module. For example, the JCL sequence in Figure 129 uses the standard VS FORTRAN Version 2 cataloged procedure VSF2CL to compile the Fortran source for the main task program (stored in data set USERPGM.FORTRAN(MTASKPGM)) and create a main task program load module named MTASKPGM in data set USERPGM.LOAD.

```
//MTASKPGM EXEC VSF2CL,FVPOPT=3
//FORT.SYSIN DD DSN=USERPGM.FORTRAN(MTASKPGM),DISP=SHR
//LKED.SYSLMOD DD DSN=USERPGM.LOAD(MTASKPGM),DISP=OLD
```

Figure 129. Sample JCL to Compile and Link Main Task Program

Creating the Parallel Subroutines Load Module

The parallel subroutines load module is the load module named in the MTF AUTOTASK keyword. This single load module contains all of your main task program's parallel subroutines. It must not contain any Fortran main programs. The entry point of this load module must be a VS FORTRAN Version 2-supplied entry point named VFEIS#, which controls processing of the parallel subroutines when you schedule them by calling the DSPTCH subroutine.

The procedures that you normally use to compile and link-edit a VS FORTRAN Version 2 program must be modified to cause library module VFEIS# to be the entry point of the parallel subroutines load module. When link-editing this load module, the following linkage editor control statements will cause the module VFEIS# to be included:

```
INCLUDE SYSLIB(VFEIS#)
ENTRY    VFEIS#
```

For example, the JCL sequence in Figure 130 uses the standard VS FORTRAN Version 2 cataloged procedure VSF2CL to compile the Fortran source for the parallel subroutines (stored in data set USERPGM.FORTRAN(SUBTASK)) and create a parallel subroutines load module named SUBTASK in data set USERPGM.LOAD. This load module contains the module VFEIS#, and has VFEIS# as the load module's entry point.

```
//SUBTASK EXEC VSF2CL,FVPOPT=3
//FORT.SYSIN DD DSN=USERPGM.FORTRAN(SUBTASK),DISP=SHR
//LKED.SYSLMOD DD DSN=USERPGM.LOAD(SUBTASK),DISP=OLD
//LKED.SYSIN DD *
INCLUDE SYSLIB(VFEIS#)
ENTRY    VFEIS#
/*
```

Figure 130. Sample JCL to Compile and Link Parallel Subroutines

Under MVS/XA and MVS/ESA, the AMODE attribute of the parallel subroutine load module determines the addressing mode for a parallel subroutine. If the parallel subroutine load module attribute is AMODE(31) or AMODE(ANY), the parallel subroutine will receive control in 31-bit addressing mode. If the attribute is AMODE(24), the parallel subroutine will receive control in 24-bit addressing mode.

Link-Editing Considerations

1. Both the main task program load module and the parallel subroutines load module must be link-edited to operate in the same mode: either load or link mode. If you link-edit the main task program load module to operate in link mode (by concatenating SYS1.VSF2LINK ahead of SYS1.VSF2FORT), be sure to link-edit the parallel subroutines load module to operate in link mode as well.
2. If you are using link mode, both the main task program load module and the parallel subroutines load module must be link-edited using the same release of the VS FORTRAN Version 2 library. In this case, if later releases of the library become available and you relink either load module, then you must relink the other load module as well.
3. Do not specify the NE linkage editor option when link-editing the parallel subroutines load module. MTF cannot schedule parallel subroutines that are contained in a load module link-edited with the NE option.

Running Programs That Use MTF

To run your program, you use the usual MVS JCL for Fortran programs, plus a few additional JCL statements that are required for MTF to run. This additional JCL is shown in Figure 131. This figure shows only the additional JCL required for MTF. You must also supply any other JCL required to run your program.

```
//GO      EXEC    ...,PARM='...AUTOTASK(loadmodname,subtasks)'  
//AUTOTASK DD      DSN=USERPGM.LOAD,DISP=SHR  
//FTERR001 DD      ...  
//FTERR002 DD      ...  
//FTERR0.. DD      ...  
//FTERR0nn DD      ...
```

Figure 131. Run-Time JCL for MTF

AUTOTASK Keyword in the EXEC Statement

The AUTOTASK keyword in the EXEC statement PARM parameter causes the VS FORTRAN Version 2 library to create the environment that MTF uses to run a program simultaneously. If you do not use the AUTOTASK keyword, calls to the MTF subroutine DSPTCH will fail.

```
//GO      EXEC    ...,PARM='...AUTOTASK(loadmodname,subtasks)'
```

AUTOTASK

Indicates that you are using MTF. The AUTOTASK keyword has two subparameters:

loadmodname

Is the name of the load module that contains the main task program's parallel subroutines. This load module must be a member of the load module library specified in the AUTOTASK DD statement.

subtasks

Is the number of subtasks to create.

Range: 1 through 99.

Note: Generally, the number of subtasks should be close to the number of processors available on the configuration where the program runs. MVS can simultaneously run only as many tasks as there are processors. Scheduling more parallel subroutines than processors can increase system overhead.

AUTOTASK DD Statement

The AUTOTASK DD statement specifies the load module library that contains the load module with the parallel subroutines.

```
//AUTOTASK DD      DSN=user.dsn,DISP=SHR
```

user.dsn

Is the name of the load module library that contains the parallel subroutines load module.

The parallel subroutines load module *loadmodname* named in your AUTOTASK keyword in your EXEC statement must be contained in this data set.

DD Statements for Unnamed Files

For unnamed files, MTF assigns a unique object-time output file to each parallel subroutine. These output files contain diagnostic messages that the library may issue while the parallel subroutines are running. They also contain output that is the result of any DEBUG packets, PRINT statements, or WRITE statements.

If the installation defaults at your site have been changed such that output from PRINT and WRITE statements is directed to a unit other than the one for diagnostic messages, MTF assigns an additional output file for each subtask containing PRINT or WRITE statements.

Because these files are automatically allocated while the program runs, you need not supply DD statements for them unless you wish to override the default device type or other file characteristics. The default device type is a terminal in TSO or SYSOUT=A in batch.

If you do supply DD statements, use the following ddnames:

- FTERRsss for files containing diagnostic messages and possibly output from PRINT or WRITE statements
- FTPRTsss for files containing output from PRINT or WRITE statements (if PRINT and WRITE statements are directed to a different output unit than diagnostic messages)

where sss is the subtask number; that is, 001, 002, 003, and so on. Thus, for example, if your site used the same unit for diagnostic messages and WRITE or PRINT statements, and your program had four subtasks and the first two used PRINT statements, you would use the ddnames FTERR001, FTERR002, FTERR003, FTERR004. If your site used a different unit for PRINT or WRITE statements, you would use the ddnames FTERR001, FTERR002, FTERR003, FTERR004, FTPRT001, and FTPRT002.

Example of JCL

An example of the run-time JCL to run a program that uses MTF is shown in Figure 132. This figure shows the JCL that is unique to running MTF, as well as the other JCL the program would typically require. (Some programs might require additional DD statements.)

```
//GO      EXEC  PGM=MTASKPGM,PARM='AUTOTASK(SUBTASK,4) '
//STEPLIB DD    DSN=USERPGM.LOAD,DISP=SHR
//        DD    DSN=SYS1.VSF2FORT,DISP=SHR
//AUTOTASK DD    DSN=USERPGM.LOAD,DISP=SHR
//FTERR001 DD    SYSOUT=A,DCB=(RECFM=F)
//FTERR002 DD    SYSOUT=A,DCB=(RECFM=F)
//FTERR003 DD    SYSOUT=A,DCB=(RECFM=F)
//FTERR004 DD    SYSOUT=A,DCB=(RECFM=F)
//FT05F001 DD    DSN=USERPGM.INPUT,DISP=SHR
//FT06F001 DD    SYSOUT=A,DCB=(RECFM=F)
```

Figure 132. Example Run-Time JCL

MTASKPGM is the name of the main task program load module, and is the load module that gets control when MVS first starts running the program. In this example, this load module is contained in data set USERPGM.LOAD, which is referred to by the STEPLIB DD statement.

Converting MTF Programs to Parallel Programs

SUBTASK is the name of the load module that contains all of the main task program's parallel subroutines. This load module is contained in data set USERPGM.LOAD, which is referred to by the AUTOTASK DD statement.

This program has four subtasks.

The FTERR001 through FTERR004 DD statements specify that the run-time error messages and other printed output from all four subtasks are to be written to SYSOUT class A and that the record format is to be fixed-length. These DD statements are necessary only if you do not want to accept the defaults.

The FT05F001 DD statement specifies the data set that contains the program's input data.

The FT06F001 DD statement specifies that the main task program's run-time error messages and other printed output are to be written to SYSOUT class A and the record format is to be fixed-length. This DD statement is necessary only if you do not want to accept the defaults.

Debugging Programs That Use MTF

Interactive debug can be used to debug your main task program. It cannot, however, be used to debug your parallel subroutines.

Using MTF with Load Mode

When using the multitasking facility with load mode, the VSF2LOAD library must be concatenated in STEPLIB or JOBLIB instead of with SYS1.LINKLIB in the system link list. For more information, see "Specifying Load Mode" on page 76.

What to Avoid When Using MTF

To prevent undesirable results, be aware of the following concerns:

- Do not update a file with one task if the other tasks read the same file. Files may be destroyed if this is attempted.
- Synchronization between tasks by the operating system may cause the data in a shared dynamic common to be corrupted or to be presented in a different order.
- If using data-in-virtual subroutines, do not terminate or view a data object in the main task prior to synchronizing the tasks to completion. If a data object is terminated or viewed in this way, data-in-virtual spaces may be lost, and this loss may not be noticeable.
- Parallel language constructs, automatically generated parallel code, and parallel lock and event services.

Converting MTF Programs to Parallel Programs

You must convert programs that use the multitasking facility (MTF) in order to use parallel language extensions, automatic parallelization of DO loops, or library lock and event services in them.

If you want to convert MTF programs into parallel programs consider the following:

- Parallel language constructs or automatically parallelized loops cannot run simultaneously with MTF service subroutines.
- You must convert MTF programs that use link mode to load mode, because parallel programs can only be run in load mode.
- Virtual storage requirements can increase (although you can use the RENT compile-time option and the separation tool to reduce these requirements).
- Assembler routines cannot use MVS supervisor services macros that are sensitive to the task in which they are run (for example, OPEN and CLOSE).

You must perform the conversions shown in Figure 133 to use parallel language constructs, automatic parallelization of DO loops, or the lock and event services.

Figure 133. MTF Conversion to Parallel Language

MTF Calls to...	Parallel Conversion
DSPTCH(1)	SCHEDULE statements
SHRCOM	SHARING clauses on the SCHEDULE statement
SYNCRO	WAIT FOR ALL TASKS statements
NTASKS	NPROCS function
Add ORIGINATE statements for the number of subtasks specified on the AUTOTASK run-time option.(2)	

Notes:

1. You can use the PARALLEL CALL statement to replace a call to DSPTCH only if the following are true:
 - All common blocks in the main task program and subprograms can be shared, with the program processing remaining computationally independent,
 - Common blocks specified in the program unit containing the PARALLEL CALL may be shared with the subroutine specified for asynchronous processing and all common blocks that are required to be shared are specified in the program unit containing the PARALLEL CALL statement, *and*
 - All units including the error message and print units can be shared.
2. If your program schedules more parallel subroutines than the number of subtasks specified on the AUTOTASK run-time option, you may need to add extra statements to control the scheduling of the parallel subroutines.

Converting MTF Programs to Parallel Programs

Figure 134 is example of an MTF program that specifies the same number of subtasks on the AUTOTASK run-time option as the number of parallel subroutines that are scheduled:

```
@PROCESS DC(C1,C2)
COMMON /C1/,/C2/
.
.
.
CALL SHRCOM(C1)
CALL SHRCOM(C2)
CALL NTASKS(NT)
DO 10, I=1,NT
    CALL DSPTCH('PSUB1',VAR1,VAR2)
10  CONTINUE
    CALL SYNCRO
    STOP
    END
```

Figure 134. An MTF Main Task Program Before Conversion

Figure 135 shows the MTF program in Figure 134 converted to use parallel language constructs:

```
@PROCESS DC(C1,C2) PAR OPT(3)
COMMON /C1/,/C2/
.
.
.
DO 10, I=1,NPROCS()
    ORIGINATE TASK I
    SCHEDULE ANY TASK ITASK,
+     SHARING(C1,C2),
+     CALLING PSUB1(VAR1,VAR2)
    CONTINUE
    WAIT FOR ALL TASKS
    STOP
    END
```

Figure 135. The MTF Main Task Program After Conversion

Programs that scheduled more parallel subroutines than the number of subtasks specified on the AUTOTASK run-time option may need to add statements for controlling the scheduling of parallel tasks and parallel threads. Figure 136 is an example of an MTF program that schedules a greater number of parallel subroutines than the number of subtasks available:

```
@PROCESS DC(C1,C2)
COMMON /C1/,/C2/
.
.
.
CALL SHRCOM(C1)
CALL SHRCOM(C2)
DO 10, I=1,5000
    CALL DSPTCH('PSUB1',VAR1,VAR2)
10  CONTINE
    CALL SYNCRO
    STOP
    END
```

Figure 136. An MTF Program Before Conversion

Figure 137 shows the MTF program in Figure 136 on page 420 converted to use parallel language constructs. It shows how you can use two SCHEDULE statements to control the dispatching of subroutines:

```

@PROCESS DC(C1,C2) PAR OPT(3)
COMMON /C1/,/C2/
.
.
.
DO 5, I=1,NPROCS(J)
  ORIGINATE TASK I
  SCHEDULE ANY TASK ITASK, SHARING(C1,C2),
+    CALLING PSUB1(VAR1,VAR2)
5  CONTINUE
DO 10, I=1,5000-J
  WAIT FOR ANY TASK ITASK
  SCHEDULE ANY TASK ITASK, SHARING(C1,C2),
+    CALLING PSUB1(VAR1,VAR2)
10 CONTINUE
  WAIT FOR ALL TASKS
  STOP
  END
  
```

Figure 137. The MTF Program After Conversion Using SCHEDULE

Figure 138 shows the MTF program in Figure 136 on page 420 converted to use the PARALLEL CALL and WAIT FOR ALL CALLS statements (see the notes to Figure 133 on page 419 for conditions that restrict the use of PARALLEL CALL to replace a call to DSPTCH):

```

@PROCESS DC(C1,C2) PAR OPT(3)
COMMON /C1/,/C2/
.
.
.
DO 10, I=1,5000
  PARALLEL CALL PSUB1(VAR1,VAR2)
10 CONTINUE
  WAIT FOR ALL CALLS
  STOP
  END
  
```

Figure 138. The program after conversion using PARALLEL CALL

Compiling and Link-Editing: To compile and link-edit your parallel program for the first time after conversion, you must produce a single load module containing your converted program. To supply routines that do not require conversion for your program, you can include the existing MTF parallel subroutine load module, normally specified on the AUTOTASK statement in an MTF program, in the load library for your program.

The sample JCL in Figure 139 on page 422 shows how to compile and link-edit your parallel program after conversion.

Converting MTF Programs to Parallel Programs

```
//CONVERT EXEC VSF2CL,PARM.FORT='OPT(3),PARALLEL'  
//FORT.SYSIN DD DSN=userpgm.fortran,DISP=SHR  
//LKED.SYSLMOD DD DSN=userpgm.load(newpgm),DISP=OLD  
//MTFSUBS DD DSN=userpgm.load,DISP=SHR  
//SAMPLIB DD DSN=SYSQ.SAMPLIB,DISP=SHR  
//LKED.SYSIN DD *  
    INCLUDE SAMPLIB(AFBVLKED)  
    INCLUDE MTFSUBS(loadmod)  
/*
```

Figure 139. Sample JCL to Compile and Link-Edit a Converted MTF Program

Follow the standard procedures for subsequent compilations and link-edits of your program.

JCL Conversion: After recompiling and linking your program, you must convert your JCL for running it. Replace the AUTOTASK run-time option with PARALLEL.

You must convert the following data definition statements:

Change MTF	For use with ORIGINATE ANY TASK	For use with ORIGINATE TASK
FTERR0nn	FTSExxxx	FTUExxxx
FTPRT0nn	FTSPxxxx	FTUPxxxx

Note:

If you are using PARALLEL CALL you can delete the FTERR0nn and FTPRT0nn data definition statements instead of replacing them.

Chapter 21. Creating Reentrant Programs

Comparing Reentrant and Nonreentrant Programs	423
Sharing a Reentrant Program	425
Limitations and Disadvantages of Reentrancy	426
Preparing to Use a Reentrant Program	426
Creating and Using a Reentrant Program under CMS	430
Step 1: Design and Code	430
Step 2: Compile	430
Step 3: Separate the Two Parts	430
3a. Choosing the Assigned or Default Name Form	430
3b. Invoking the Separation Tool	431
Using CMS EXEC Files to Run the Separation Tool	432
Step 4: Prepare an Executable Program from the Nonshareable Parts	433
Step 5: Link-Edit the Shareable Parts	434
Special Considerations for VM/XA	435
Step 6: Install the Shareable Parts in a DCSS	435
6a. Gathering the Necessary Information	436
6b. Defining the DCSS to VM/SP	437
6c. Storing the Parts in the DCSS	438
Step 7: Running the Program	438
Creating and Using a Reentrant Program under MVS	439
Step 1: Design and Code	439
Step 2: Compile	439
Step 3: Separate the Two Parts	439
3a. Choosing the Assigned or Default Name Form	439
3b. Invoking the Separation Tool	440
MVS Cataloged Procedures for the Separation Tool	441
Step 4: Prepare an Executable Program from the Nonshareable Parts	442
Step 5: Link-Edit the Shareable Parts	442
Special Considerations for MVS/XA and MVS/ESA	443
Step 6: Install the Shareable Parts in an LPA	443
Step 7: Run the Program	444
Link-Editing and Running a Reentrant Program under MVS with TSO	444

This chapter explains the concept of reentrant programs and why you might want to use them. It also discusses the advantages and limitations of reentrant programs, and gives you a brief overview of the process involved in creating such programs.

For detailed information about creating reentrant programs see:

“Creating and Using a Reentrant Program under CMS” on page 430

“Creating and Using a Reentrant Program under MVS” on page 439.

Comparing Reentrant and Nonreentrant Programs

It is possible that several users may want to run a particular program at the same time. Usually, in such a case, each user is given a separate, private copy of it. A nonreentrant program can be used only in this way. Thus, if there are three concurrent users, there will be three copies of the program in main storage; if there are twenty concurrent users, there will be twenty copies. Figure 140 on page 424 shows this.

Reentrant Programs

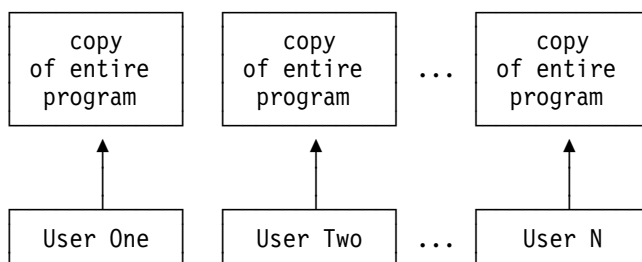


Figure 140. Nonreentrant Program Requires Multiple Copies for Concurrent Use

Private copies are necessary when the program is nonreentrant because concurrent users of a single copy would interfere with the values of each other's variables. Sharing a single copy of a nonreentrant program would result in erroneous processing and output.

To understand this, suppose that several users are allowed to share a single copy of a program containing a variable A. In this program, A is initially zero, and gets set to other values as processing proceeds. User One starts running the single copy of the program, reads a data item from a file, and adds that value (say 8.3) to variable A. A is now 8.3. At this moment, User Two starts running the program. User Two reads a data item (with a value of say 6.6) and adds it to A. But instead of A being 6.6, as expected, A is now actually 14.9. User Two's subsequent actions or output based on A will now be incorrect. And any other concurrent users trying to share the program would have similar problems.

Reentrant programs in VS FORTRAN Version 2 are programs that are structured so they can overcome this difficulty. Several users are allowed to share a single copy of the code, but each user has a private copy of the nonshareable data. By specifying the RENT compile-time option, you request a program structured this way. Figure 141 on page 425 shows the concept of sharing.

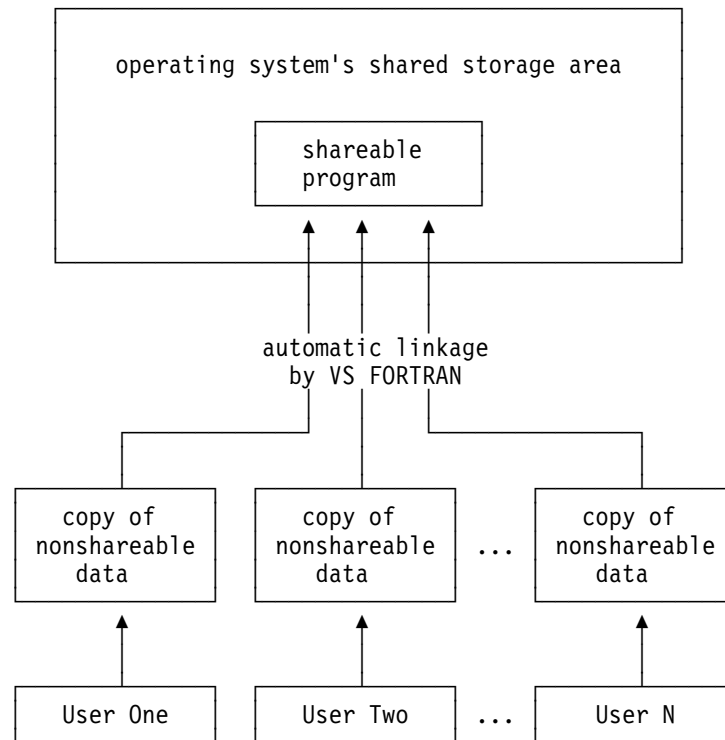


Figure 141. Reentrant Program Saves Space for Concurrent Users

Sharing a Reentrant Program

Sharing is made possible by dividing the program into two parts:

1. A nonshareable part—variables and other information whose values can be altered during processing
2. A shareable part—information and instructions that are not modified during processing

Each concurrent user is given a private copy of the nonshareable part of the program. Thus, altering these values does not affect other users. The shareable part of the program contains the program's instructions. This shareable part can be placed in a special area of storage that allows sharing and protects against modification. Communication between the two parts of the program is established automatically by VS FORTRAN. From your perspective as a user, running a reentrant program is no different than running a regular, nonreentrant one, and run-time results are the same.

Advantages of Sharing Reentrant Programs: Because it allows sharing, reentrancy has the following advantages:

- Less main storage usage (the more users sharing the program concurrently, the greater the savings)
- Performance improvement (less paging to auxiliary storage, and higher priority paging).

Dividing a program unit compiled with the RENT option into its nonshareable and shareable parts has another potential advantage that has nothing to do with sharing. It can provide a type of dynamic loading capability.

If a subprogram is not always called during processing, it can be compiled as reentrant and separated into its nonshareable and shareable parts. The nonshareable part would be part of your executable program (and would always be in main storage when the program was running, whether called or not). However, the shareable part could be kept in a library on auxiliary storage, and would only be loaded and run if the subprogram were called. When the program is run but the subprogram is not called, the shareable part is not loaded, and the main storage requirement is reduced.

No special coding or assembler language interface, other than the normal Fortran CALL to the subprogram, is necessary.

Limitations and Disadvantages of Reentrancy

Reentrancy is not valuable or practical in all cases.

First, separating a program into its two parts and installing the shareable part in the system's shareable area is most advantageous if the program will have multiple concurrent users. (However, it is possible that there will be a minor improvement in performance even if the program will have only one user.) Furthermore, only programs with a large amount of executable code lend themselves to sharing, since it is only the executable code that is shared.

Even if a program has a large amount of shareable executable code and will have multiple concurrent users, you should weigh the advantages of sharing against the extra preparation work involved: separating the shareable and nonshareable parts, preparing the shareable area of the operating system, and re-IPLing the operating system.

Another limitation is that dynamic loading of the shareable part is practical only if the program logic does not always require its loading, and if the shareable part is relatively large.

Preparing to Use a Reentrant Program

Before looking at the detailed steps involved in creating and running a reentrant program on your system, it may help to look at the process in general and to examine one of the most important steps: separating the nonshareable and shareable parts of a program.

To create a reentrant program unit, you code it as usual and then compile it with the RENT compile-time option. The object module produced by the compiler must then be separated into its nonshareable and shareable parts. To do this, VS FORTRAN Version 2 supplies you with a program called the separation tool. Figure 142 on page 427 shows the input to and output from the separation tool program in the case of a single object module.

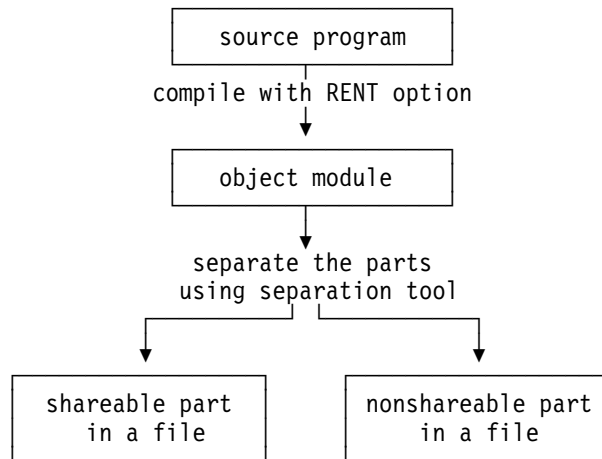


Figure 142. Using the Separation Tool on a Single Program

Figure 142 shows a very simple case, but your applications will probably involve more than one source program unit, and may also involve nonreentrant program units. For such cases, the separation tool can accept multiple object files.

Figure 143 shows a more complex situation, with greater detail about input to and output from the separation tool.

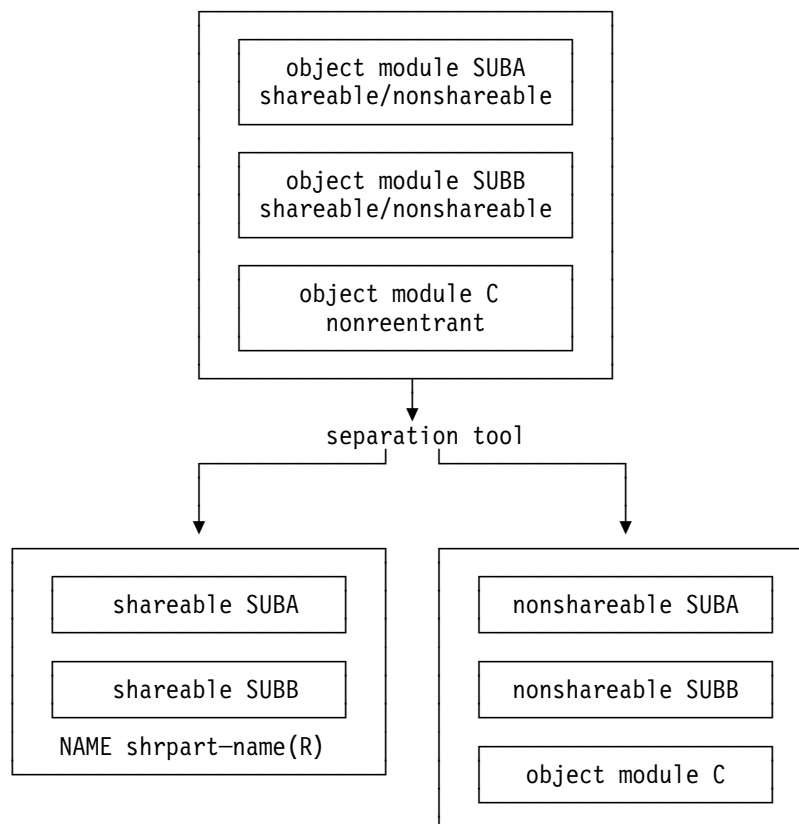


Figure 143. Using the Separation Tool on Multiple Programs with the Assigned Name Form

Notice that, in this example, the input to the separation tool includes two reentrant program units with nonshareable and shareable parts (SUBA and SUBB, compiled with the RENT option), and one nonreentrant program unit (C). The shareable and nonshareable parts are again divided into two output files. The separation tool adds a linkage editor NAME statement; in Figure 143 on page 427, the following is added to the shareable output file:

```
NAME shrpart-name(R)
```

For this discussion, *nonreentrant* program units are either Fortran program units compiled without the RENT option, or non-Fortran program units. If your executable program needs to contain nonreentrant program units, these can either be supplied as input to the separation tool in the same run as the reentrant program units, or they can be merged with the nonshareable parts before preparing the program to be run.

The separation tool automatically places any nonreentrant program units together with the nonshareable parts of the reentrant program units in one file (or data set) as output. Later, you may want to break this file into separate pieces so you can link-edit them separately, as needed.

You can choose the form in which the shareable parts of the output will be produced. The output will take one of two forms:

Assigned Name Form

The output file containing the shareable parts will contain one linkage editor NAME statement. You supply the name for the linkage editor NAME statement when you invoke the separation tool. Later, the linkage editor will create one member, using the assigned name, containing all the shareable parts in this file.

Figure 143 on page 427 shows the output from the separation tool in the assigned name form.

Default Name Form

In the output file from the separation tool, each shareable part will be followed by a linkage editor NAME statement. The name on each statement is the name of the original program unit preceded by the character @. Later, the linkage editor will create an individual member for each of the shareable parts. The name of each member is the default name (program unit name preceded by @).

Figure 144 on page 429 shows the output from the separation tool in the default name form. In our example, the program unit names would be used to build the default names @SUBA and @SUBB, respectively.

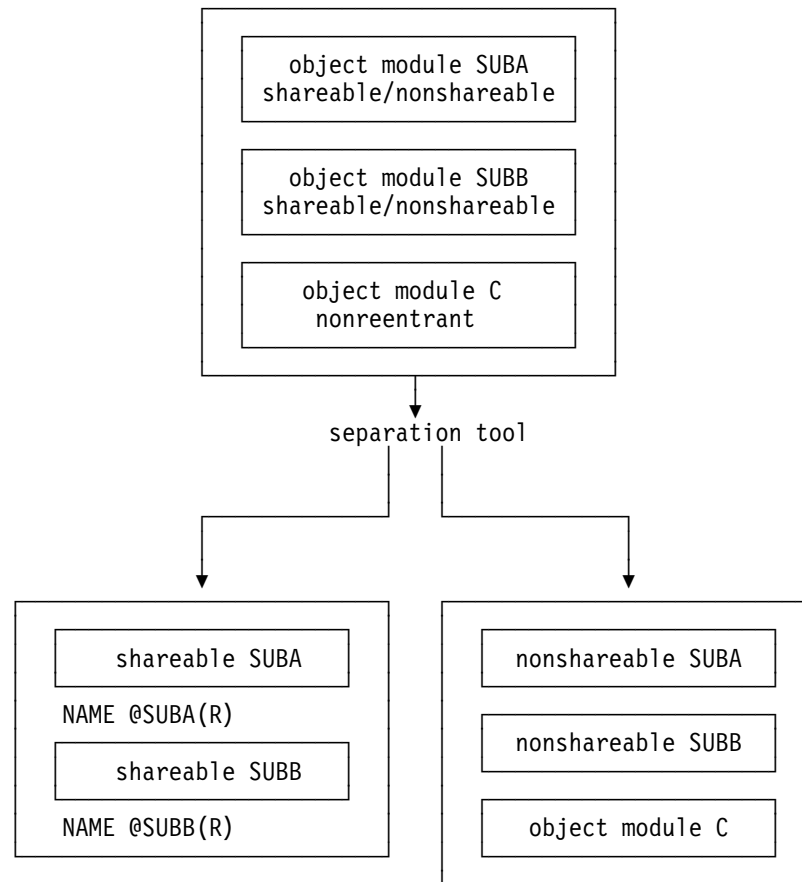


Figure 144. Using the Separation Tool with the Default Name Form

It is important to understand the entire process of creating a reentrant program before beginning. As you plan how you will create your reentrant program, keep in mind that all the program units you need do not have to be sent through the separation tool at once, as shown above. For example, if you know you will be installing the shareable parts in a DCSS and plan to use the default name form, you may want to consider compiling and separating each program unit individually. This will save you time later on.

After the separation tool has produced its output, you can complete the process of creating a reentrant program:

- Link-edit the shareable parts and prepare the nonshareable parts to be run.
- To share the shareable parts, install them in a discontinuous shared segment (DCSS) for CMS, or link pack area (LPA) for MVS.
- To run the program, invoke the nonshareable part just as you would invoke any nonreentrant program. The corresponding shareable part will be automatically located and used.

Note that the nonshareable and shareable parts of a program unit compiled with RENT are synchronized to work only with each other. If you have to change your source program for any reason, you must rebuild both parts by recompiling the program unit and running the separation tool again. Otherwise, the program will not work correctly.

Creating and Using a Reentrant Program under CMS

This section explains the steps you must complete to create and use a reentrant program.

Step 1: Design and Code

Design and code your program as you would normally. Nothing different need be done for a program that will be reentrant.

Step 2: Compile

When you compile your program unit, specify the RENT compile-time option. The compiler will produce an object module composed of two parts: the nonshareable code and the shareable code.

Using the RENT option does not alter the result of running your program.

Before proceeding to the next step, debug your program in the usual way to be sure that it is error-free. If you wait until later to do this, you will have to repeat the following steps. Furthermore, once you install the program in a shareable area, you will not be able to use the interactive debug to debug it unless you have compiled it with the TEST option. However, a program compiled with the TEST option generally has poor performance because the code is not optimized and because there are many additional calls to run-time service subroutines.

Step 3: Separate the Two Parts

The separation tool is supplied as part of the VS FORTRAN Version 2 product. You will use the separation tool to separate your compiler-produced object modules into their shareable parts and nonshareable parts.

This step consists of two parts:

- “3a. Choosing the Assigned or Default Name Form”
- “3b. Invoking the Separation Tool” on page 431.

3a. Choosing the Assigned or Default Name Form

Before you invoke the separation tool, you must make a decision about whether or not to override the default names for the linkage editor NAME statements for the shareable parts.

If you choose the *assigned name form*, the file containing the shareable parts will contain one linkage editor NAME statement. You supply the name for the linkage editor NAME statement when you invoke the separation tool. The linkage editor, invoked with the LKED command, will later create one LOADLIB member with this name. The LOADLIB will contain all the shareable parts from the file produced by the separation tool. The assigned name will also be the name of the DCSS if you share the parts.

If you choose the *default name form*, each shareable part in the file will be followed by a linkage editor NAME statement. The name on each statement is the name of the respective program unit, preceded by the character @. The names on the linkage editor NAME statements will later become the names of the members of a LOADLIB. If you decide to share the shareable parts, these must later be the names of the DCSSs.

If you are supplying multiple reentrant programs as input to the separation tool, your decision about the assigned or default name form has some important consequences.

Assigned Name Form

If you choose the assigned name form, there will be only one LOADLIB member or DCSS containing the shareable parts of all the program units processed by the separation tool. If you plan to share the shareable parts, this option creates less work for the system programmer who must define the DCSS because there will be only one DCSS to build. This option is also more efficient if your executable program consists of many program units that will always be used together. In addition, performance may be better with this option, especially if the modules will be loaded from a LOADLIB, because you will be loading fewer modules.

If the shareable parts of multiple reentrant program units will later be installed in a DCSS for sharing, you must divide the program units into groups of fewer than 255 and run each group through the separation tool. Each group will be installed in its own DCSS. You cannot use the assigned name form for 255 or more routines in a single run of the separation tool if you plan to place them into a DCSS.

Choosing the assigned name form also means that it will be more difficult for you to change any program unit in the group. To change one program unit, you will have to run the object modules for all the program units in the group through the separation tool again in a single run. (For this reason, it is a good idea to keep a copy of the object modules produced by the compiler.)

Default Name Form

If you specify the default name form, there will be multiple LOADLIB members, one for each shareable part. If you plan to install the shareable parts of multiple program units in DCSSs for sharing, you must first edit the shareable parts file to break it into separate files, with one shareable object module per file. You will have to build a separate DCSS for each shareable part to be shared.

This option is preferable if you will be using shareable parts in many varying combinations with the dynamic loading capability. This allows you to be more flexible, loading only the parts you need. Also, if you have many program units, it will be easier to change them later because you can rerun each object module to be changed through the separation tool individually, without rerunning the others.

3b. Invoking the Separation Tool

When you invoke the separation tool, you must provide FILEDEF statements with the following ddnames:

- SYSIN** Input to the separation tool. These are the object modules produced by the compiler.
- SYSPRINT** A file containing messages from the separation tool. You normally direct this either to your terminal or to your disk.
- SYSUT1** A file produced by the separation tool containing the nonshareable parts of the reentrant program units, and any nonreentrant program units.
- SYSUT2** A file produced by the separation tool containing the shareable parts of the reentrant program units.

SYSUT3 A work file used internally by the separation tool.

After you have provided the necessary FILEDEF statements, invoke the separation tool with the AFBVRSEP command. If you want the shareable parts file to take the assigned name form, include a name on the AFBVRSEP command as follows:

```
AFBVRSEP shrpart-name
```

If you do not include a name on the AFBVRSEP command, you have chosen the default name form for the shareable parts file.

For example, your command sequence to invoke the separation tool might be similar to this:

```
FILEDEF SYSIN      DISK MYPROG      TEXT      A
FILEDEF SYSPRINT   DISK MYPROG      SEPLIST   A
FILEDEF SYSUT1     DISK MYPROG      TEXTNSHR  A
FILEDEF SYSUT2     DISK shrpart-text TEXT      A
FILEDEF SYSUT3     DISK MYPROG      TEMP      A
AFBVRSEP shrpart-name
ERASE  MYPROG TEMP      A
ERASE  MYPROG TEXTORIG A
RENAME MYPROG TEXT      A  MYPROG TEXTORIG A
RENAME MYPROG TEXTNSHR A  MYPROG TEXT      A
```

Notice that, in this example, the output file MYPROG TEXT A contains only the nonshareable part of the program unit after the separation tool has been run. If you need to rerun the separation tool, remember that MYPROG TEXTORIG A contains the original object modules used as input to the separation tool.

The following describes two EXECs that can be used to perform this separation step. The input may include object modules from nonreentrant program units.

Using CMS EXEC Files to Run the Separation Tool

VS FORTRAN Version 2 provides two EXEC files to help compile reentrant program units and separate them into their shareable and nonshareable parts. These EXEC files are:

VSF2RCS Compiles a reentrant program unit and then invokes the separation tool to separate the nonshareable and shareable parts. To run this EXEC, issue the following command:

```
VSF2RCS name shrpart (options ...
```

where name, shrpart and options are described below.

VSF2RSEP Invokes the separation tool to separate the nonshareable and shareable parts of an existing TEXT file. To run the EXEC, issue the following command.

```
VSF2RSEP name shrpart (option
```

where name, shrpart and option are described below.

name

For running VSF2RCS, this is the filename of the Fortran source program. This file must have a filetype of FORTRAN.

For running VSF2RSEP, this is the name of the input TEXT file on your A-disk. It is the contents of this file that are to be separated. This file is

updated to contain the nonshareable parts of the reentrant VS FORTRAN programs, and all of the nonreentrant programs that are found in the input.

shrpart

Is the filename of the TEXT file into which the shareable parts are to be placed. This name must be different from *name* above. If you use the ASGNNAME option, this name will become the name of the LOADLIB member, and of the DCSS if you share the shareable parts.

option(s)

Either of the following options may be coded to specify the structure of the file containing the shareable parts. This controls the names of the LOADLIB members or of the DCSSs that contain the shareable parts.

For running VSF2RCS, any of the VS FORTRAN Version 2 compile-time options may be specified in addition to one of the following options. RENT need not be given because the EXEC already provides it.

For running VSF2RSEP, only one of the following options may be specified.

ASGNNAME

The output file produced by the separation tool will take the assigned name form. You will later combine the shareable parts of all the reentrant programs to create one LOADLIB member with the LKED command, using the name you assign as *shrpart*, or one DCSS with the name *shrpart*. The ASGNNAME option is the default.

DEFNAME

The output file produced by the separation tool will take the default name form. A separate LOADLIB member or DCSS will be created for the shareable parts of each reentrant program that is compiled, using the program name preceded by @.

Following separation of the nonshareable and shareable parts, the shareable parts file must be used to build one or more LOADLIB members and, optionally, to build one or more DCSSs.

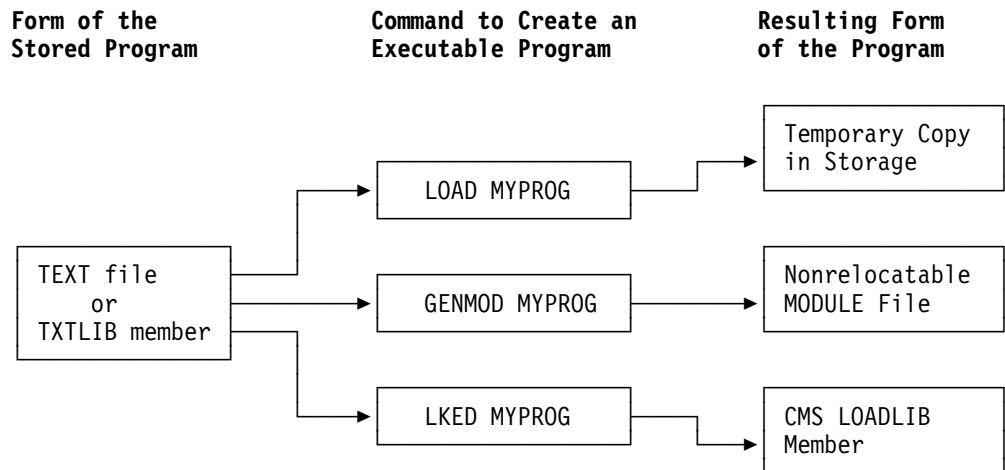
Step 4: Prepare an Executable Program from the Nonshareable Parts

The separation tool groups all the nonshareable parts and any nonreentrant program units in a single TEXT file. This file can be used to create an executable program which, when invoked, will automatically locate and use the corresponding shareable parts.

If the TEXT file contains multiple program units that you want to make available individually to other users, see “Dividing the File into Individual Members” on page 434.

Reentrant Programs under CMS

As with any program, you have several options of preparing the nonshareable parts of the program to be run:



MYPROG is the name of the file containing the nonshareable parts and any nonreentrant program units. For more information about each of these methods, see “Creating an Executable Program” on page 63.

Note: Your choice of link mode or load mode is not related to the fact that you are using reentrant program units. You can load the shareable parts of program units from a DCSS or from a CMS LOADLIB regardless of whether your program is run in link mode or load mode.

Dividing the File into Individual Members: If the TEXT file with the nonshareable parts contains multiple program units that you want to make available to other users, you might want to create individual members in a TXTLIB for each of the program units. To do this, first edit the file to add a linkage editor NAME statement after the object module for each program unit. The statement must be preceded by at least one blank and must have the following form:

```
NAME sub-name(R)
```

Then use this file as input to the TXTLIB ADD command. For example, the following command produces a TXTLIB member for each program unit:

```
TXTLIB ADD LIBNAME MYPROG
```

where LIBNAME is the file name of the TXTLIB, and MYPROG is the name of the TEXT file containing the nonshareable parts produced by the separation tool.

Step 5: Link-Edit the Shareable Parts

If you are using reentrant program units for the dynamic loading capability, this step allows you to later load the shareable parts as you need them.

If you plan to share the shareable parts in a DCSS, this step is not required, but is recommended. The output of the LKED command will be used later to determine the size required for your DCSS. In addition, after completing this step, the LOADLIB members will always be available, even if the copies installed in the DCSSs are not.

To link-edit the shareable parts of your program into a CMS LOADLIB, use the LKED command. Only the shareable parts produced by the separation tool can be link-edited in this step.

If you chose the default name form, the LKED command will produce one CMS LOADLIB member for each program unit. The member names will always be the program unit names preceded by @. (Note that these default names cannot be changed except by renaming the source program unit, recompiling, and rerunning the separation tool.)

If you chose the assigned name form, the LKED command will produce a single member. Its name will be the name you specified when you invoked the separation tool. The member name will always be the name originally produced by the separation tool on the linkage editor NAME statement. (Note that the only way to change the assigned name is to rerun the separation tool, specifying a new assigned name.)

For example, you might issue this command:

```
LKED shrpart-text (RENT MAP NCAL LIBE libname
```

where shrpart-text is the name of the TEXT file produced by the separation tool that contains the shareable parts of your program units. libname is the file name of the CMS LOADLIB that will hold the shareable parts of your program units.

Special Considerations for VM/XA

Shareable load modules of reentrant programs run in the same addressing mode as their corresponding nonshareable parts.

If you place the shareable load modules in a CMS LOADLIB using the LKED command, they will be assigned the default value ANY for RMODE, allowing them to reside above the 16-megabyte line.

If, for any reason, the nonshareable parts of your program must run in 24-bit addressing mode, you must override the default on the LKED command so that the shareable load modules reside below the 16-megabyte line.

Discontiguous shared segments (DCSSs) in which you store shareable load modules can reside above or below the 16-megabyte line. However, if the nonshareable parts of your program must run in 24-bit addressing mode, the segments must reside below the line. You need not worry if the segments overlap your virtual machine, but they must not overlap storage that CMS has already allocated for other use (such as loaded modules or acquired virtual storage).

Step 6: Install the Shareable Parts in a DCSS

This step consists of three parts:

- “6a. Gathering the Necessary Information” on page 436
- “6b. Defining the DCSS to VM/SP” on page 437
- “6c. Storing the Parts in the DCSS” on page 438.

For the shareable parts of a program to be shared, they must be in the operating system's shareable area, called a discontiguous shared segment (DCSS). If you only want to take advantage of the dynamic loading capability of reentrant

programs, you can skip this step and go on to the next one, “Step 7: Running the Program” on page 438.

If you change one or more programs after completing this step, you need to reinstall the updated module in the same DCSS by repeating “6c. Storing the Parts in the DCSS.”

Some of the procedures in this step are the responsibility of your VM/SP system programmer. Others require that you be a Class E privileged user.

6a. Gathering the Necessary Information

Before your VM/SP system programmer can prepare the DCSS for you, you will need to work together to determine the following information about the DCSS:

Name of the DCSS

If you chose the assigned name form for your shareable file, you must build one DCSS whose name is the name you specified when you invoked the separation tool.

If you chose the default name form of the shareable file, you must build one DCSS for every reentrant program that you want to share. Each DCSS name must be the program name prefixed by @. If there are multiple shareable parts in the file produced by the separation tool, you will have to edit the file to create multiple files with only one shareable part in each file. From each file, you will build a separate DCSS whose name is the program name prefixed by @.

Size of the DCSS

You can determine the size needed for your DCSS by looking at the link-edit map produced by the LKED command in step 5. The link-edit map is in a file whose file name is the same as that of the shareable TEXT file produced by the separation tool, and whose file type is LKEDIT.

The map provides the length of the shareable load module. Your DCSS must have at least the same length. However, defining a DCSS larger than is actually needed will allow you to expand your program (within limits) without redefining the DCSS.

Starting address of the DCSS

With the assistance of your VM/SP system programmer, choose this address using the following guidelines:

- The address should be at least as large as the virtual machine size of any of the users you expect to use this DCSS.
- The address should not be unnecessarily high; if it is, storage is wasted for unreferenced CP segment table entries.
- The addresses should not allow any DCSS to overlap any other saved segment that may be used at the same time. For example, if you run the program from an ISPF panel, the DCSS should not overlap any saved segments that contain parts of ISPF. Your VM/SP system programmer can help you determine a potential overlap.

6b. Defining the DCSS to VM/SP

Your VM/SP system programmer must complete the following steps before the shareable parts can be installed in the DCSS:

1. Allocate permanent space on a CP-owned DASD volume to contain the DCSS. Determine the number of pages required by dividing the size of the shareable load module (found in the link-edit map, above) by 4K, or X'1000', and rounding upward to the next page. (Refer to *VM/SP Planning Guide and Reference*, for information on the amount of disk space needed.)
2. Define the DCSS by adding a NAMESYS macro instruction to your installation's DMKSNT ASSEMBLE module. (See *VM/SP Planning Guide and Reference*, and *VM/SP System Programmer's Guide*.) If more than one DCSS is to be built to hold copies of the shareable parts of various programs, there must be a NAMESYS macro instruction defining each DCSS. The following example of the NAMESYS macro instruction defines a DCSS named SHRPART. The sample numbers given illustrate a possible set of numbers and are not intended as the only location or size for a DCSS.

```
NAMESYS SYSNAME=SHRPART,           x
        SYSSIZE=128K,               x
        SYSHRSG=(48,49),            x
        SYSPGNM=(768-799),          x
        VSYADDR=IGNORE,             x
        SYSVOL=VMSRES,              x
        SYSSTRT=(072,1)
```

In the example:

- The SYSNAME parameter specifies the name of the DCSS (SHRPART in this example).
 - The SYSHRSG parameter provides a list of consecutive segment numbers. (Specifying these numbers allows the segments to be shared by all users.) Determine the number of segments required by dividing the size of the shareable load module by 64K, or X'10000', and rounding upward to the next segment. Compute the first segment number by dividing the starting address of the DCSS by 64K. In this example, the starting address is X'300000', or 3072K. Dividing this by 64K gives a starting segment number of 48.
 - The SYSPGNM parameter specifies the range of page numbers that comprise the DCSS. Compute the first page number by dividing the starting address of the DCSS by 4K. In this example, dividing X'300000', or 3072K, by 4K gives a starting page number of 768. A range of 32 pages is specified here to correspond to the 2 segments.
 - The SYSVOL parameter gives the volume serial number of the CP-owned volume which will hold the DCSS.
 - The SYSSTRT parameter gives the starting cylinder and page address (on the volume specified by the SYSVOL parameter) which will hold the DCSS.
3. Assemble the new system name table (DMKSNT) and regenerate the CP nucleus by using the GENERATE EXEC procedure as described in *VM/SP Installation Guide*.
 4. Re-IPL the VM/SP system.

6c. Storing the Parts in the DCSS

You must have Class E privileges to complete the installation of the shareable parts into a DCSS.

1. Bring the shareable part of the program into your virtual machine using the LOAD command. Remember that the size of your virtual machine (if you are issuing this command) must be large enough to contain the shareable part at the desired address.

```
LOAD shrpart-text (CLEAR ORIGIN dcss-address
```

where shrpart-text is the file name of the TEXT file containing the shareable parts, and dcss-address is the starting address of the DCSS.

For example, if the shareable parts of your program are in a TEXT file named SHRPART and the DCSS begins at the address 300000, you would issue this command:

```
LOAD SHRPART (CLEAR ORIGIN 300000
```

2. Save the shareable parts in the DCSS by issuing the command:

```
SAVESYS dcss-name
```

where dcss-name is the name of the DCSS.

For example, you might issue this:

```
SAVESYS SHRPART
```

Note: If the @ character is defined as the “character delete” symbol for your terminal, you will have to use the escape character to enter the default names of the shareable parts, or issue the command:

```
TERMINAL CHARDEL OFF
```

before issuing the SAVESYS command.

Step 7: Running the Program

To run the program, invoke the nonshareable part just as you would any regular (nonreentrant) program. The shareable part of the program will be automatically located and used by VS FORTRAN Version 2.

You should always include a GLOBAL LOADLIB statement that refers to the CMS LOADLIB containing the shareable parts, even if they are also installed in a DCSS. If the shareable parts are in a DCSS and if the starting address of the DCSS does not overlap your virtual machine, the DCSS copy will be used. Otherwise, the GLOBAL LOADLIB command will ensure that you can still access the copy in the CMS LOADLIB.

To run a program in link mode from TEXT files on your disk, you could use one of the following sets of commands:

- If VSF2LINK and VSF2FORT are separate libraries at your site, use:

```
GLOBAL LOADLIB libname
GLOBAL TXTLIB VSF2LINK VSF2FORT CMSLIB
LOAD MYPROG
START
```

- If VS FORTRAN Version 2 has been installed at your site with the combined LINK library, you do not need to specify VSF2FORT in your GLOBAL TXTLIB command.

You can use the following coding:

```
GLOBAL LOADLIB libname
GLOBAL TXTLIB VSF2LINK CMSLIB
LOAD MYPROG
START
```

You must also provide whatever FILEDEF commands the program requires.

Creating and Using a Reentrant Program under MVS

This section explains the steps you must complete to create and use a reentrant program. Before beginning the procedures listed here, be sure you have read Chapter 21, “Creating Reentrant Programs” on page 423. The procedures below assume you have read and understood the information explained there.

Step 1: Design and Code

Design and code your program in the usual ways. Nothing different need be done for a program that will be reentrant.

Step 2: Compile

When you compile your program unit, specify the RENT compile-time option. The compiler will produce an object module composed of two parts: the shareable code and the nonshareable code.

Using the RENT option does not alter the result of processing your program.

Before proceeding to the next step, debug your program in the usual way to be sure that it is error-free. If you wait until later to do this, you will have to repeat the following steps. Furthermore, once you install the program in a shareable area, you will not be able to use the interactive debug to debug it unless you have compiled it with the TEST option. However, a program compiled with the TEST option generally has poor performance because the code is not optimized and because there are many additional calls to run-time service subroutines.

Step 3: Separate the Two Parts

The separation tool is supplied as part of the VS FORTRAN Version 2 product. You will use the separation tool to separate your compiler-produced object modules into their shareable parts and nonshareable parts.

This step consists of two parts:

- “3a. Choosing the Assigned or Default Name Form”
- “3b. Invoking the Separation Tool” on page 440.

3a. Choosing the Assigned or Default Name Form

Before you invoke the tool, you must make a decision about whether or not to override the default names for the linkage editor NAME statements for the shareable parts.

If you choose the assigned name form, the data set containing the shareable parts will contain one linkage editor NAME statement. You supply the name for the linkage editor NAME statement when you invoke the separation tool. The linkage

editor will later create one member containing all the shareable parts in this data set.

If you choose the default name form, each shareable part in the data set will be followed by a linkage editor NAME statement. The name on each statement is the name of the respective program unit, preceded by the character @.

The names on the linkage editor NAME statements will later become the names of the shareable load modules.

If you are supplying multiple reentrant programs as input to the separation tool, your decision about the assigned or default name form has some important consequences.

Assigned Name Form

If you choose the assigned name form, there will be only one load module containing the shareable parts of all the program units processed by the separation tool. This option is more practical if your executable program consists of many program units that will always be used together. In addition, performance may be better with this option, especially if the shareable load modules will not be in the LPA, because you will be loading fewer modules.

Choosing the assigned name form also means that it will be more difficult for you to change any program unit in the shareable load module. To change one program unit, you will have to run the object modules for all the program units through the separation tool again in a single run. (For this reason, it is a good idea to keep the object modules produced by the compiler.)

Default Name Form

If you specify the default name form, there will be multiple shareable load modules, one for each shareable part.

Use this option if you will be using shareable parts in many varying combinations. This allows you to be more flexible, loading only the parts you need. Also, if you have many program units, it will be easier to later change them because you can rerun each object module through the separation tool individually.

3b. Invoking the Separation Tool

When you invoke the separation tool, you must provide DD statements with the following ddnames:

SYSIN Input to the separation tool. These are the object modules produced by the compiler.

SYSPRINT A data set containing messages from the separation tool.

SYSUT1 A data set produced by the separation tool containing the nonshareable parts of the reentrant program units, and any nonreentrant program units.

SYSUT2 A data set produced by the separation tool containing the shareable parts of the reentrant program units.

SYSUT3 A data set used internally by the separation tool.

Invoke the separation tool by running the program AFBVSFST, which is in SYS1.VSF2LOAD. If you want the data set containing the shareable part to take the assigned name form, include a name on the PARM parameter of the EXEC

statement. If you omit the PARM parameter, you have chosen the default name form for the data set.

For example, to invoke the separation tool in batch mode, you might code the JCL for the separation tool as follows:

```
//SEP      EXEC  PGM=AFBVSFST,PARM='shrmod-name'  
//STEPLIB DD   DSN=SYS1.VSF2LOAD,DISP=SHR  
//SYSPRINT DD  SYSOUT=A  
//SYSUT1   DD   DSN=&&NONSHR,DISP=(NEW,PASS),UNIT=SYSDA,  
//          SPACE=(3200,(25,1)),DCB=BLKSIZE=3200  
//SYSUT2   DD   DSN=&&SHRPART,DISP=(NEW,PASS),UNIT=SYSDA,  
//          SPACE=(3200,(25,1)),DCB=BLKSIZE=3200  
//SYSUT3   DD   DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,  
//          SPACE=(3200,(25,1)),DCB=BLKSIZE=3200  
//SYSIN    DD   DSN=myprog.OBJ,DISP=SHR
```

This JCL separates the input data set identified by ddname SYSIN. It will put your nonshareable parts in the temporary data set &&NONSHR, and your shareable parts in the temporary data set &&SHRPART.

You may wish to combine this separation step with other steps in the preparation process. Examples of several cataloged procedures are provided to help you do this. They are listed below in “MVS Cataloged Procedures for the Separation Tool.” VFT2RCL compiles a program unit with the RENT option, separates the nonshareable and shareable parts, and link-edits your modules. VFT2RCLG compiles a program unit with the RENT option, separates the parts, link-edits the modules, and runs them. VFT2RLG separates the nonshareable and shareable parts from a previous compilation, link-edits the modules, and runs them.

MVS Cataloged Procedures for the Separation Tool

Figure 145 shows cataloged procedures that you can use to compile, separate, link-edit, and run your VS FORTRAN Version 2 programs. The procedures create programs that run in *load mode*. These procedures should be located in your appropriate system procedure library.

Figure 145. IBM-Supplied Cataloged Procedures for the Separation Tool

Action	Procedure
Compile, separate, and link	VFT2RCL
Compile, separate, link, and go	VFT2RCLG
Separate, link, and go	VFT2RLG

If you want the data set containing your shareable parts to take the assigned name form, specify a name on the FVNAME parameter when you run the procedure. (In the procedures, SHRMOD is used as the assigned name.) If you want the data set containing your shareable parts to take the default name form, you must nullify the FVNAME parameter when you run the procedure. For example:

```
//RCL      EXEC  VFT2RCL,FVNAME=
```

Supply your own DD statements, which refer to the input required by the cataloged procedures:

Procedure	ddname	Contents of Data Set
VFT2RCL	FORT.SYSIN	Fortran source program.
	LKEDNSHR.SYSIN	Linkage editor control statements.
VFT2RCLG	FORT.SYSIN	Fortran source program.
	LKEDNSHR.SYSIN	Linkage editor control statements.
VF2RLG	SEP.SYSIN	Object modules resulting from a compilation with the RENT option. This data set may also contain object modules for nonreentrant programs.
	LKEDNSHR.SYSIN	Linkage editor control statements for link-editing the nonshareable parts (optional).

Note: In the procedures that contain a processing step (GO), the shareable load module is made available in a private library, not in the link pack area. You can use these procedures to test your separated programs, but if you want to share them, you will have to put them in the LPA, as described in “Step 6: Install the Shareable Parts in an LPA” on page 443.

Step 4: Prepare an Executable Program from the Nonshareable Parts

The separation tool groups all the nonshareable parts and any nonreentrant program units in a single data set. You can use this data set to create an executable load module by link-editing it just as you would for any nonreentrant program. When invoked, this executable load module will then load and use the corresponding shareable parts.

For example, to create a load module that runs in load mode from the nonshareable parts of the program, use the following JCL:

```
//LKEDNSHR EXEC PGM=IEWL,PARM='XREF,LIST'
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,1))
//SYSLIN DD DSN=&&NONSHR,DISP=(OLD,DELETE)
//SYSLIB DD DSN=SYS1.VSF2FORT,DISP=SHR
//SYSLMOD DD DSN=MYLIB(MYPROG),DISP=OLD
```

Note: Your choice of link mode or load mode is not related to the fact that you are using reentrant program units. You can load the shareable parts of program units regardless of whether your program is run in link mode or load mode. The terms *link mode* and *load mode* refer to whether the VS FORTRAN Version 2 service subroutines are link-edited with the compiler-generated code, or whether these routines are loaded as needed when running the program.

Step 5: Link-Edit the Shareable Parts

You must link-edit the shareable parts of your program into a library. Only the shareable parts file produced by the separation tool can be link-edited in this step.

If you chose the default name form, the link-edit step will produce one shareable load module for each program unit. The member names in the library will be the program unit names preceded by @. (Note that these default names cannot be changed except by renaming the source program unit, recompiling, and rerunning the separation tool.)

If you chose the assigned name form, this step will produce one shareable load module. Its member name will be the name you specified when you invoked the separation tool. The member name will always be the name originally produced by the separation tool on the linkage editor NAME statement. (Note that the only way to change the assigned name is to rerun the separation tool, specifying a new assigned name.)

For example, you might use the following JCL:

```
//LKEDSHR EXEC PGM=IEWL,PARM='MAP,LIST,RENT,NCAL'
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,1))
//SYSLIN DD DSN=88SHRPART,DISP=(OLD,DELETE)
//SYSLMOD DD DSN=SYS1.TESTLIB,DISP=OLD
```

If you plan to put your shareable load modules in the link pack area (LPA) for sharing, be sure to put these load modules in a library that will not be referred to in a JOBLIB or STEPLIB when you run the program. (However, if you will not be placing the shareable load modules in the LPA, it may be more practical to use the same library for both the nonshareable and shareable load modules.)

Special Considerations for MVS/XA and MVS/ESA

If you will be running your reentrant program on an MVS/XA or MVS/ESA system, you must pay special attention to the residence mode of the shareable load module.

Unless you override the default, the residence mode assigned to the shareable load module by the linkage editor will be ANY, and the load module will be loaded above the 16-megabyte line. This means that the corresponding nonshareable parts of the program must run in 31-bit addressing mode to reach the shareable parts.

However, if the nonshareable parts must run in 24-bit addressing mode for any reason, you must be sure that the shareable load module is assigned a residence mode of 24. This causes the load module to be loaded below the 16-megabyte line, so it can be reached by the nonshareable part.

To assign a residence mode of 24 for the shareable load module, add the following to the PARM parameter in the JCL for this step, above.

```
RMODE=24
```

You need not worry about the addressing mode for the shareable load module, because it will be invoked in the same addressing mode as its corresponding nonshareable part.

Step 6: Install the Shareable Parts in an LPA

If you only want to take advantage of the dynamic loading capability of reentrant programs, you can skip this step and go on to the next one, “Step 7: Run the Program” on page 444.

To enable the shareable parts of your program to be shared, your MVS system programmer must install each shareable load module, created in the previous step, into the operating system's shareable area, called the link pack area (LPA).

For example, to put the shareable parts in a modified link pack area, your MVS system programmer would complete the following steps:

1. Add entries to a SYS1.PARMLIB member (for example, IEALPA09) to indicate the library containing the shareable load module, and the name of the shareable load module to be loaded when the link pack area is created. For example:

```
SYS1.TESTLIB SHRMOD
```

2. Re-IPL MVS to make the shareable part of the program available in the link pack area. In response to the IEA101A message, specify that the link pack area be created and that the SYS1.PARMLIB member (for example, IEALPA09) be used. For example:

```
R 00,MLPA=09
```

Step 7: Run the Program

To run the program, invoke the nonshareable part just as you would any regular (nonreentrant) program. The shareable part of the program will be automatically loaded and used by VS FORTRAN Version 2.

For example, you can invoke the executable program, created in “Step 4: Prepare an Executable Program from the Nonshareable Parts” on page 442, with the following JCL:

```
//GO      EXEC  PGM=MYPROG
//STEPLIB DD    DSN=MYLIB,DISP=SHR
//        DD    DSN=SYS1.VSF2LOAD,DISP=SHR
//FT06F001 DD    SYSOUT=A
```

This example assumes that either:

- Both the nonshareable and shareable load modules are in the same library (MYLIB), or
- The shareable load module is in the LPA.

Link-Editing and Running a Reentrant Program under MVS with TSO

You can link-edit a reentrant program in TSO using one of the following methods:

- Link-edit the object file as if it were a nonreentrant program.
- Use the MVS separation tool to divide the reentrant object file into its shareable and nonshareable parts. Only the nonshareable part requires the VS FORTRAN Version 2 library.
- Perform the separation using the CLIST in Figure 146 on page 445. This CLIST creates a single load module; to create multiple load modules, eliminate the parameter list &RENTPART in the CALL statement.

```

PROC 2 INPUT RENTPART
FREE F(SYSIN SYSPRINT SYSUT1 SYSUT2 SYSUT3)
FREE ATTR(DCBPARMS)
ATTR DCBPARMS BLKSIZE(3120) LRECL(80) RECFM(F,B)
RENAME &INPUT..OBJ &INPUT..OBJ2
ALLOC F(SYSIN) DA(&INPUT..OBJ2) SHR
ALLOC F(SYSPRINT) DA(+)
ALLOC F(SYSUT1) DA(&INPUT..OBJ) NEW SP(10,2) +
    TRACK USING(DCBPARMS)
ALLOC F(SYSUT2) DA(&RENTPART..OBJ) NEW SP(10,2) +
    TRACK REUSE USING(DCBPARMS)
ALLOC F(SYSUT3) SP(10,2) TRACK NEW USING(DCBPARMS)
CALL 'SYS1.VSF2LOAD(AFBVSFST)' '&RENTPART.'
FREE F(SYSIN SYSPRINT SYSUT1 SYSUT2 SYSUT3)
FREE ATTR(DCBPARMS)

```

Figure 146. CLIST to Invoke the Separation Tool

Note: The CLIST doesn't link either the shareable or nonshareable parts, and assumes that the input object file does not have an associated OBJ2 file.

- Provide access to the shareable load modules by placing them in a shareable library and using one of the access procedures described in “Selecting Link Mode or Load Mode” on page 92.

For example, the following coding link-edits the contents of the object file represented by shrpart with the library SHRLIB.

```
LINK shrpart LOAD(SHRLIB) PRINT(M1) XREF LET LIST RENT
```

The actual names of the modules placed into SHRLIB are generated by the separation tool on NAME control records.

- Insert the shrpart modules into the link pack area (LPA). The procedure for loading modules into the LPA varies among installations and can only be performed by someone with the proper authorization. Ask your system administrator for the procedure.

Once you have link-edited a reentrant program, you can run it with a CALL statement that specifies the name of the link-edited program, in this case noshrpmg, for example:

```
CALL mylib(noshrpmg)
```

Part 6. Appendixes

Appendix A. Internal Limits in VS FORTRAN Version 2

This appendix describes the specifications for the maximum sizes and lengths of various VS FORTRAN statements and constructs.

The internal limits of VS FORTRAN Version 2 that are unique to the compiler are given in Figure 147.

Figure 147. Compiler Unique Internal Limits in VS FORTRAN Version 2

Language Item	Limit
Levels of nested DO loops, implied DO loops, parallel loops, and parallel sections	25
Expression evaluation	The maximum depth of the push-down stack for expression evaluation is 660. 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 660. For example, if an expression starts with 660 left parentheses before any right parentheses, this expression exceeds the push-down stack limit.
Levels of nested statement function references	50
Statement function arguments in a nested reference	50
Arguments in a statement function definition	20
Levels of nested INCLUDE directives	16
Levels of nested block IF statements; that is, the number of IF... THEN, ELSE, and ELSEIF... THEN statements occurring before the occurrence of an ENDIF.	125
Length of character constants in a FORMAT statement	255 characters (255 bytes)
Length of character constants in a PAUSE or STOP statement	72 characters (72 bytes)
Length of Hollerith constants	255 characters (255 bytes)
Number of ICA file names	40
Referenced variables in a program unit	32767
Nested parentheses groups in a format	51
Statement labels	The limit is 2000 user source labels and compiler-generated labels. However, if table overflow occurs at optimization level 2 or 3, you may be able to alleviate the problem by removing all unreferenced user labels.
DISPLAY statements in a program unit	Unlimited; however, only 100 unique namelist names will be available. After the 100th name (NM.L99), the compiler will recycle names starting with NM.L00, but will display the desired items in the display list.
Number of times a format code can be repeated	255

The internal limits of VS FORTRAN Version 2 that are unique to run time are given in Figure 148.

Figure 148. Run-Time Unique Internal Limits in VS FORTRAN Version 2

Language Item	Limit
Number of programs or subroutines processed by the traceback routine	12
Number of available units(1)	Up to 2001 (IBM default: 00 to 99)
Number of unnamed units(1)	100 (00 to 99)
Number of named units(1)	Installation default minus 3 (IBM default is 97, or 100 minus 3)
Number of originated tasks created with the ORIGINATE TASK statement	999
Number of originated tasks created with the ORIGINATE ANY TASK statement	999

Note:

1. For parallel programs, the number of available units applies to the number of available units per parallel task.

Appendix B. Assembler Language Considerations

This appendix is intended to help you call Fortran subprograms and invoke Fortran main programs from your assembler language program. It also describes calling assembler subprograms from Fortran, requesting compilation of a Fortran program from an assembler program, and retrieving arguments in an assembler program.

You can use both Fortran and assembler language in the same application, either as a Fortran main program with assembler subprograms or vice versa.

In assembler programs, you can invoke the Fortran subprogram by initializing the run-time environment (if it has not previously been initialized) and then calling the subroutine. In Fortran programs, you can invoke the assembler subprogram in either of two ways: through CALL statements or through function references in arithmetic expressions. Be aware that the use of the pound sign (#) in the name of an assembler subprogram may cause a conflict with a VS FORTRAN routine of the same name; the order in which libraries are specified determines which subprogram or routine is selected.

Using Fortran Data in Assembler Subprograms

Your assembler language subprograms can use data defined in Fortran subprograms that is either contained in common areas or passed to your subprogram as arguments.

Likewise, you can pass your data to Fortran subprograms as arguments.

Arrays used in Fortran programs should be efficiently aligned. For a description of alignment, see "EQUIVALENCE Considerations" on page 377. If an inefficiently aligned array is passed to a subroutine, each time the array is referenced in a vectorized statement a vector boundary misalignment error message is issued and the library performs corrective action. The time consumed in fielding the interrupt and performing the corrective action seriously degrades performance.

Floating-point data should be normalized. If unnormalized data is used by the vector hardware, an unnormalized operand exception can occur, resulting in abnormal termination of the program.

Using Common Data in Assembler Subprograms

Assembler language subprograms can access data in both named and blank common areas. Accessing data stored in extended common blocks requires additional considerations.

Parallel tasks have their own copies of common areas; an assembler routine must reference the data for its parallel task. For a description and examples of using common blocks with parallel processing, see "Sharing Data in Common Blocks within a Parallel Task" on page 308.

Using Named Common Data in Assembler Programs

To refer to a named common area, your assembler program should use an external A-type address constant. For example:

```
        EXTRN name-of-common-area
comaddr DC    A(name-of-common-area)
```

Note: You can use this method in Fortran only if the named common block is a static common block. Assembler language subprograms can access data that is within a dynamic or extended common block only when that data is passed as an argument to the assembler language subprogram.

Using Blank Common Data in Assembler Programs

To refer to a blank common area, your assembler program must also define a blank common area, using the COM assembler instruction. Only one blank common area is generated; 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:

```
        L      11,=A(name)
        USING  name,11
        .
        .
        COM
name     DS      0F
```

Using Extended Common Data in Assembler Programs

VS FORTRAN Version 2 maintains extended common blocks within ESA/390 data spaces. In order for an assembler language subprogram to access data in an extended common block, the subprogram must be capable of accessing data in these data spaces. This requires that the assembler language subprogram be written to run in Access Register mode. (For more information on ESA/390, including data spaces, Access Register mode, and the instructions necessary to access data, see the *Enterprise Systems Architecture/390 Principles of Operation* manual.)

Accessing data in a data space requires the program to use 8-byte (64-bit) extended addresses. The first 4 bytes of the 64-bit address identify the data space to be referenced, and the second 4 bytes identify the storage location within this data space.

Because data in an extended common block, which is maintained in a data space, is accessed using 8-byte addresses, new linkage conventions are required when programs are passed arguments that are within extended common blocks. See "Linkage Conventions for VS FORTRAN Programs" on page 453 for more information on the linkage conventions used by VS FORTRAN.

In VS FORTRAN, a program that runs in Access Register mode is said to run in *Extended mode* (i.e. EMODE). A program that does not run in Extended mode because there is no need to access data in a data space is said to run in *Standard mode*. Any program compiled before the availability of extended common support or not explicitly compiled with the EMODE compile-time option also runs in standard mode.

Two types of VS FORTRAN programs must run in Extended mode:

- Programs compiled with the EC compile-time option that contain explicit references to extended common blocks.
- Subprograms or function programs that, at some time, pass data in an extended common block as parameters.

Linkage Conventions for VS FORTRAN Programs

VS FORTRAN uses two different linkage conventions to pass data as arguments to subprograms. A linkage convention, called the *Standard Linkage Convention*, is used to pass arguments that reside within the same address space as the executing program. A second linkage convention, called the *Extended Linkage Convention*, is used to pass arguments when *any* of the arguments resides within a data space.

Standard Linkage Convention

The Standard Linkage Convention consists of the following requirements:

- Register 13 must contain the address of an 18-word save area. The second word of the save area should contain the address of the previous caller's save area until the previous save area is reached.
- Register 14 must contain the address to which control will return when the subroutine is completed.
- Register 15 must contain the address of the subroutine entry point.
- Register 1 must contain the address of the argument address list. If there are no character arguments, register 1 points to a list of consecutive words, each containing the address of an argument to be passed. The last word in the list should have a 1 in the high-order (sign) bit.
- The call to the subprogram always occurs in Primary execution mode.

Assuming that register 13 already points to a save area, a call to a VS FORTRAN subprogram using the Standard Linkage Convention would look similar to that in Figure 149.

```

LA      1,parmlist
L       15,=V(fortran-subprogram)
BALR   14,15
      .
      .
parmlist DC    A(argument1)
      .
      .
      DC      A(argumentn+X'80000000')
```

Figure 149. Calling a Fortran Subprogram Using Standard Linkage Convention

Passing Character Arguments Using the Standard Linkage Convention

The linkage convention for passing character arguments between subprograms is different from the linkage convention for passing non-character arguments.

FORTRAN 77 standards specify two attributes for each character argument:

- address*, required of all arguments
- length*, required of character arguments.

The convention for supplying the *address* argument is the same for character and non-character arguments. In the calling subprogram, a sequence of addresses is entered in the order of the called subprogram's argument list; one word containing an address for each argument in the list. The high-order bit is set to 1 in the last address to signify the end of the address list.

To supply the *length* argument, a sequence of one-word addresses pointing to the length attributes of each argument in the list is used. There is a one-to-one correspondence of addresses and lengths. The high-order bit is set to 1 in the last address to signify the end of the length list.

Note: If both character and non-character arguments are passed, address and length attributes must be supplied for each argument.

Address and length lists are arranged contiguously in storage. Two words precede these lists. The first, X'C2E90000', identifies this list as one with character arguments. The second word contains the length, in bytes, of the argument address list. The value is used as an offset from each entry in the address list to point to its corresponding entry in the length list.

Figure 150 illustrates the linkage convention of a call to a subprogram with three character arguments. It assumes that the VS FORTRAN environment has already been initialized and that register 13 is already pointing to an 18-word save area.

	LA	1,PL
	L	15,=V(subroutine)
	BALR	14,15
	.	.
	DS	0F
	DC	X'C2E90000'
	DC	A(PLL-PL)
PL	DC	A(A)
	DC	A(B)
	DC	A(C+X'80000000')
PLL	DC	A(LA)
	DC	A(LB)
	DC	A(LC+X'80000000')
	.	.
A	DS	CL5
B	DS	CL11
C	DS	CL11
LA	DC	A(L'A)
LB	DC	A(L'B)
LC	DC	A(L'C)

Figure 150. Calling a Fortran Subprogram with Character Arguments Using Standard Linkage Convention

Extended Linkage Convention

The Extended Linkage Convention consists of the following requirements:

- Register 13 must contain the address of an 18-word save area.
- Register 14 must contain the address to which control will return when the subroutine is completed.
- Register 15 must contain the address of the subroutine entry point.
- Register 1 must contain the address of the argument address list. If there are no character arguments, register 1 points to a list of consecutive words, each containing the address of the *address* of the argument to be passed. Each word in the argument address list points to an **extended address**. These extended addresses are 8 bytes in size. The first word of the extended address identifies the address space in which the argument resides (0 if the space is the program's address space), and the second word is the address within that space at which the argument is located. The last word in the list should have a 1 in the high-order (sign) bit.
- The call to the subprogram always occurs in Access Register mode (*Extended mode*).

Assuming that register 13 already points to a save area, a call to a VS FORTRAN subprogram using the Extended Linkage Convention would look similar to that in Figure 151.

```

      LA      1,parmlist
      L       15,=V(fortran-subprogram)
      BALR   14,15
      .
      .
parmlist DC    A(argument1_address)
      .
      .
      DC     A(argumentn_address+X'80000000')
      .
      .
argument1_address DC  A(id_of_space,argument1)
      .
argumentn_address DC  A(id_of_space,argumentn)

```

Figure 151. Calling a Fortran Subprogram Using Extended Linkage Convention

Considerations When Using the Extended Linkage Convention

When VS FORTRAN calls a subprogram using the Extended Linkage Convention, it attempts to ensure that the called subprogram will fail if the subprogram does not expect an argument list in the Extended linkage format. It does this by setting Access Registers 1 and 15 to invalid values. Therefore, a subprogram that is called using the Extended Linkage Convention must use the LAE or CPYA instructions to set Access Registers 1 and 15 to valid values before using these registers to access any storage, including the parameter list.

For example,

LAE 1,0(1,0) and LAE 15,0(15,0)

set Access Registers 1 and 15 to the correct values.

Alternatively,

CPYA 1,13 and CPYA 15,13

also set Access Registers 1 and 15 to the correct values.

Not setting Access Registers 1 and 15 to the correct values at entry to a subprogram called using the Extended Linkage Convention can prevent the successful execution of the subprogram.

Passing Character Arguments Using the Extended Linkage Convention

The Extended Linkage Convention, like the Standard Linkage Convention, has different linkage conventions for passing character and non-character arguments.

The convention for supplying the *address* argument is the same for both character and non-character arguments.

To supply the *length* argument, a sequence of one-word addresses pointing to the *actual* length attribute of each argument in the list is used. There is a one-to-one correspondence of addresses and lengths. The high-order bit is set to 1 in the last address to signify the end of the length list.

Note: If both character and non-character arguments are passed, address and length attributes must be supplied for each argument.

Address and length lists are arranged contiguously in storage. Two words precede these lists. The first, X'C2E90000', identifies this list as one with character arguments. The second word contains the length, in bytes, of the argument address list. The value is used as an offset from each entry in the address list to point to its corresponding entry in the length list.

Figure 152 on page 457 illustrates the linkage convention of a call to a subprogram with three character arguments. It assumes that the VS FORTRAN environment has already been initialized and that register 13 is already pointing to an 18-word save area.

```

        LA      1,PL
        L       15,=V(subroutine)
        BALR    14,15
        .
        .
        DS      0F
        DC      X'C2E90000'
        DC      A(PLL-PL)
PL      DC      A(address_A)
        DC      A(address_B)
        DC      A(address_C+X'80000000')
PLL     DC      A(LA)
        DC      A(LB)
        DC      A(LC+X'80000000')
        .
        .
address_A DC      A(id_space,A)
address_B DC      A(id_space,B)
address_C DC      A(id_space,C)
        .
        .
A        DS      CL5
B        DS      CL11
C        DS      CL11
LA       DC      A(L'A)
LB       DC      A(L'B)
LC       DC      A(L'C)

```

Figure 152. Calling a Fortran Subprogram with Character Arguments Using Extended Linkage Convention

Calling Fortran Subprograms from Assembler Programs

Initializing the Run-Time Environment

If your main program is not written in Fortran and it calls VS FORTRAN Version 2 service subroutines or other Fortran routines, the calling program must initialize the run-time environment.

No program can call a Fortran main program or issue an unconditional request to initialize the VS FORTRAN run-time environment if there is a run-time environment already active. You must terminate the first run-time environment before initializing another.

VFEIN# and VFEIL# Entry Points

You can use the standard entry point VFEIN# when you know that initialization is necessary. You can use the entry point VFEIL# when you do not know if the module has been initialized.

VFEIN# or VFEIL#, to which the initialization instructions branch, initializes return coding and prepares routines to handle interruptions. If this initialization is omitted, an interruption or error might cause abnormal termination. After initialization, the routines return to the instruction following the call.

The load module that contains the call for initialization must remain in virtual storage during the entire time that the VS FORTRAN Version 2 run-time library remains active. In other words, you must not delete the module. For VFEIL#, this requirement applies only if initialization occurs because of the call and not if a previous initialization occurred.

VFEIL# allows for initialization within a dynamically loaded module. A dynamically loaded module is a module in which the Fortran program is in a different load module than the one from which the VS FORTRAN run-time environment was initialized.

Unlike a call to VFEIN#, a call to VFEIL# does not result in job termination, even if the run-time environment was previously established. However, a level of initialization still occurs in a dynamically loaded module to allow the vector-valued mathematical functions to operate regardless of whether a previous initialization has occurred. Therefore, the use of VFEIL# allows you to perform initialization without having to determine whether it has already been done.

VFEIN# and VFEIL# both accept a parameter string containing run-time options. Register 1 points to a word that has the high-order bit on and that has a pointer to the parameter area. The parameter area consists of a halfword that is the length of the parameter string. Following the length is the actual parameter string. The parameter string contains the run-time options in the same format in which they are coded in the compiler invocation. That is, the options are separated by commas and the parameter string contains no embedded blanks.

The assembler language calls for VFEIN# and VFEIL# with run-time options are:

```
LA      1,parameter-list
L       15,=V(VFEIN# or VFEIL#)
BALR    14,15

parameter-list DC    A(parameter-area+X'80000000')
parameter-area DC    Y(L'parameter-string)
parameter-string DC  C'option,...'
```

The assembler language calls for VFEIN# and VFEIL# with no run-time options are:

```
SR      1,1
L       15,=V(VFEIN# or VFEIL#)
BALR    14,15
```

When running under MVS/ESA or VM/ESA, always call VFEIN# and VFEIL# in Standard mode. Do not call these routines in access register mode.

When to Terminate the Run-time Environment

To ensure that any partially-filled output buffers get written, include instructions in your program to terminate the run-time environment if the program runs any I/O statements, or produces any error messages.

The run-time environment can be terminated by a:

- STOP statement from a Fortran program
- END statement from a Fortran main program
- CALL EXIT statement
- CALL SYSRCx statement.

When the run-time environment is terminated, control returns to the routine that invoked the routine that initialized the run-time environment. This may be the operating system, or it may be another subprogram. Regardless of the reason the run-time environment was terminated, do not attempt to call another Fortran subroutine.

Considerations for MVS Subtasks: If you have two MVS subtasks running nonparallel programs that require the VS FORTRAN run-time library, you must terminate the run-time environment of one subtask before you can run the second. You cannot have two such subtasks active in the same region at the same time.

Vector Interrupt Support

If you are running in a vector environment and have initialized the run-time environment, you may still want interrupt support. If your load module contains no vector mathematical routines and no vectorized Fortran programs, you must provide a strong external reference for the label VFVIX#. For example:

```
DC    V(VFVIX#)
```

This informs the library that you are using vector instructions, and that you require vector interrupt support. If the strong external reference is missing, then any vector interrupt you encounter will be interpreted as a terminating error.

Extended Common Support

If you are using an ESA/390 data space to contain data that will be used by a VS FORTRAN program or library routine, you must provide a strong external reference for the label VFECW#. For example:

```
DC    V(VFECW#)
```

This informs the library that you are using a data space, and require support for the ESA/390 environment. If the strong external reference is missing, you might encounter run-time errors if a library routine is passed data in the data space.

Determining the Linkage Convention for Calling VS FORTRAN Subprograms

When you call a Fortran subprogram, you must follow certain linkage conventions.

Use the *Standard Linkage Convention* if none of the arguments that you are passing reside in an ESA/390 data space, or if you are not passing arguments.

You *must* use the Standard Linkage Convention if:

1. the Fortran subprogram was compiled prior to VS FORTRAN Version 2 Release 5, **or**
2. the Fortran subprogram was not compiled with the EMODE or EC compile-time options.

Use the *Extended Linkage Convention* if *any* of the arguments that you are passing reside in an ESA/390 data space. Use this linkage convention only if:

1. the Fortran subprogram was compiled after VS FORTRAN Version 2 Release 4, **and**
2. the Fortran subprogram was compiled with the EMODE or EC options.

Calling a Fortran Main Program from an Assembler Subprogram

There may be times when you wish to call a Fortran main program. In this case, you should not attempt to initialize the Fortran environment; the Fortran main program will do that. The instructions for calling a Fortran main program are shown in Figure 153.

```

        LA      1,parmlist
        L       15,=V(fortran-program)
        BALR    14,15
        .
        .
parmlist DC      A(parmdata+X'80000000')
parmdata DC      Y(L'options)
options  DC      C'run-time-options'
```

Figure 153. Calling a Fortran Main Program

This calling sequence is similar to the one used for initializing the Fortran environment. Register 1 points to a word that has a 1 in the high-order bit and points to the parameter area. The parameter area consists of a halfword that contains the length of the parameter string. The parameter string immediately follows the length halfword and contains the run-time options separated by commas. Do not embed blanks in the parameter string. When calling a main program in MVS/ESA or VM/ESA, always call in Standard mode.

Multiple Copies of Load Modules: You cannot reuse a virtual storage copy of any executable program or load module that interacts with the VS FORTRAN run-time library after the library environment has been terminated. To run the program again, delete the old copy and load a new copy into virtual storage.

Calling Assembler Subprograms from Fortran Programs

This section contains information on calling assembler language subprograms from VS FORTRAN programs. Use the Standard Linkage Convention as previously described in “Standard Linkage Convention” on page 453 if the data that is passed does not reside in extended common blocks.

If any of the data resides in extended common blocks, use the Extended Linkage Convention.

Note: Assembler subprograms that run in the parallel environment cannot use MVS task sensitive supervisor macros unless the PFAFFS service subroutine is first called to establish virtual processor affinity. MVS task sensitive supervisor macros include macros such as:

- OPEN and CLOSE
- LOAD and DELETE
- ATTACH and DETACH
- GETMAIN and FREEMAIN
- STAE and ESTAE
- SPIE and ESPIE
- STAX
- STIMER, STIMERM, and TTIMER for timer services.

Assembler subprograms that run in the parallel environment cannot use VM/XA or VM/ESA system services without first calling the PRAFFS service subroutine.

The instructions at the subprogram entry point must save the registers in the given save area, establish addressability, and establish a new save area if this subprogram calls other subprograms. The form of these instructions depends on whether the assembler language subprogram is called using the Standard Linkage Convention or the Extended Linkage Convention.

When called using the Standard Linkage Convention, the assembler language program should use an entry code instruction sequence similar to the one shown in Figure 154.

routine	SAVE	(14,12),,*
	DROP	15
	LR	12,15
	USING	routine,12
	LA	15,save-area
	ST	15,8(0,13)
	ST	13,4(0,15)
	LR	13,15

Figure 154. Subprogram Entry Point Instructions (Standard Linkage Convention)

When called using the Extended Linkage Convention, the assembler language program should use an entry code instruction sequence similar to the one shown in Figure 155. This instruction sequence uses an extended form for the save area in which the caller's register are saved. See "Extended Save Area Format for Programs Called Using the Extended Linkage Convention" on page 462 for a description of this extended save area format.

routine	SAVE	(14,12),,*
	USING	routine,15
	LA	14,save-area
	ST	14,8(0,13)
	ST	13,72(14,0)
	LR	13,14
	IAC	14
	ST	14,76(0,13)
	STAM	14,12,84(13)
	LAE	12,0(15,0)
	DROP	15
	USING	routine,12

Figure 155. Subprogram Entry Point Instructions (Extended Linkage Convention)

If an assembler language subprogram may be called from a VS FORTRAN program using either the Standard Linkage Convention or the Extended Linkage Convention, a different entry code instruction sequence can be used. An example of such a "dual-mode" entry sequence is shown in Figure 156 on page 462.

```

routine      SAVE   (14,12),,*
              LR     12,15
              USING  routine,12
              LA     15,save-area
              ST     15,8(0,13)
              L      14,=V(VEIAC#)
              EX     0,0(0,14)
              BZ     standard_mode
* entry in Extended mode
              LAE    15,0(15,0)
              STM    13,14,72(15)
              STAM   1,12,96(15)
              LAE    12,0(12,0)
              MVC    4(4,15),=C'F2SA'
              LAE    1,0(1,0)
              B      common_path
* entry in Standard mode
standard_mode ST    13,4(0,15)
common_path  LR     13,15

```

Note: Although register 12 is established as the program's base register, until the LAE 12,0(12,0) instruction is executed, the code path for the entry in Extended mode should not attempt to reference storage under this base register.

Figure 156. Subprogram Entry Point Instructions (Dual-Mode)

In this entry code example, there is a reference to the external entry VEIAC#. VEIAC# is a special entry in the VS FORTRAN Version 2 library that performs the IAC 14 instruction *only* when running on a machine that contains this instruction. (When running on a machine that does not contain this instruction, this entry performs a SR 14,14 instruction to provide the equivalent function.) To use this entry, load its address into register 14 and issue an EX 0,0(0,14) instruction. Register 14 will be set with the run-time mode in bits 22 and 23 (the value of these bits will be 00 to indicate entry in Standard (Primary) mode or 10 to indicate entry in Extended (Access Register) mode. The condition code is also set to indicate the mode: 0 to indicate Standard (Primary) mode, or 2 to indicate Extended (Access Register) mode.

Extended Save Area Format for Programs Called Using the Extended Linkage Convention

Programs called using the Extended Linkage Convention can use an extended save area to save the caller's access registers and execution environment.

The extended save area has the following characteristics:

1. The save area is 36 words (144 bytes) in size.
2. The save area does not use the second word to locate the caller's save area; rather, the address of the caller's save area is found in the 19th word (displacement +72 bytes) in the extended save area.

An ID string is placed in the second word to identify the save area as an extended format save area. The value of this ID string is: "F2SA" (EBCDIC).

The "F2SA" value tells any program that follows the chain of save areas that the save area is in the Extended Save Area Format.

When a save area containing the "F2SA" value in the second word is encountered, the program should obtain the pointer to the previous save area from the word at +72 in this save area.

3. The 20th word (displacement +76 bytes) contains the caller's execution environment, in the following format:
 - a. Bits 0-21 : Reserved
 - b. Bits 22-23: PSW bits 16 and 17
 - c. Bits 24-31: Reserved
4. The 21st word is not used.
5. The 22nd through the 36th words (displacement +84 bytes through +140 bytes) contain the caller's access registers, in the order:

14, 15, 0, 1, ... , 10, 11, 12

This extended save area is shown in Figure 157.

WORD	DISPLACEMENT (DECIMAL)	CONTENT
0	+0	RESERVED
1	+4	C'F2SA'
2	+8	RESERVED
3	+12	GPR 14
4	+16	GPR 15
5	+20	GPR 0
6	+24	GPR 1
⋮	⋮	⋮
16	+64	GPR 11
17	+68	GPR 12
18	+72	PREVIOUS S.A.
19	+76	CALLER'S MODE
20	+80	NOT USED
21	+84	AR 14
22	+88	AR 15
⋮	⋮	⋮
33	+132	AR 10
34	+136	AR 11
35	+140	AR 12

Note: Access Registers 14, 15 and 0 are not saved when running in the VS FORTRAN Version 2 environment.

Figure 157. Extended Save Area Format

Retrieving Arguments in an Assembler Subprogram

The instructions needed to retrieve arguments passed from a Fortran program to an assembler language subprogram depend on the linkage convention by which the assembler language subprogram was called.

Retrieving Variables from the Argument List with the Standard Linkage Convention

When an assembler subprogram is called using the Standard Linkage Convention, the argument list contains the direct address of each variable. The assembler subprogram can retrieve the variable, using the following instructions:

```
L    Q,x(0,1)
MVC  LOC(y),z(Q)
```

where:

- Q Any general register except 0, 1, 13, or the program's base register.
- LOC The location that will contain the variable.
- x The displacement of the address of the variable from the start of the argument list.
- y The length of the variable itself.
- z Either 0 or the correct displacement for an array element. (Note that z must lie in the range [0,4095]; if the displacement of the desired array element lies outside this range, you must take additional steps to calculate the displacement at run time.)

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(0,1)
MVC  LOC(8),0(5)
```

Retrieving Arrays and Array Elements from the Argument List: When the whole array is passed as an argument, the address of the first element of the array is placed in the argument list. If you must retrieve other elements in the array, you may need to specify in a separate instruction the displacement for that element from the beginning of the array:

```
L    Q,x(1)
L    R,disp
L    S,0(R,Q)
ST   S,LOC
```

where:

- Q, R, S Any general registers except 0, 1, 13, or the program's base register.
- x The displacement of the address of the variable from the start of the argument list.
- disp The displacement of the element within the array.
- LOC The location that will contain the array element.

Retrieving Character Variables from the Argument List: The argument list contains the address of the character variable and the address of the length of the character variable.

The assembler subprogram can retrieve the variable using the following instructions:

```

L   Q,x(0,1)    Get data address
LR  S,1         Get copy of argument list address
S   S,=F'4'     Step back to location of length section offset
L   S,0(0,S)    Get offset to length section
AR  S,1         Set address of length section
L   R,x(0,S)    Get character variable length pointer
L   R,0(0,R)    Get character variable length

```

where:

Q, R, S Any general registers except 0, 1, 13, or the program's base register.

x The displacement of the address of the variable from the start of the argument list.

After the above instructions run, Q contains the address of the character variable and R contains the length of the character variable.

Retrieving Variables from the Argument List with the Extended Linkage Convention

When called using the Extended Linkage Convention, the argument list contains the indirect address of each variable. The assembler program can retrieve the variable, using the following instructions:

```

L   R,x(0,1)
LAM Q,Q,0(R)
L   Q,4(0,R)
MVC LOC(y),z(Q)

```

where:

Q Any general register except 0, 1, 13, or the program's base register.

R Any general register except 0, 1, 13, the program's base register, or the register specified as Q.

LOC The location that will contain the variable.

x The displacement of the address of the variable from the start of the argument list.

y The length of the variable itself.

z Either 0 or the correct displacement for an array element. (Note that z must lie in the range [0,4095]; if the displacement of the desired array element lies outside this range, you must take additional steps to calculate the displacement at run time.)

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   15,4(0,1)
LAM 5,5,0(15)
L   5,4(0,15)
MVC LOC(8),0(5)

```

Retrieving Arrays and Array Elements from the Argument List: When the whole array is passed as an argument, 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 may need to specify the displacement for that element from the beginning of the array in a separate instruction:

```
L    15,x(1)
LAM  Q,Q,0(15)
L    Q,4(0,15)
L    R,disp
L    S,0(R,Q)
ST   S,LOC
```

where:

Q, R, S Any general registers except 0, 1, 13, or the program's base register.
x The displacement of the address of the variable from the start of the argument list.
disp The displacement of the element within the array.
LOC The location that will contain the array element.

Retrieving Character Variables from the Argument List: The argument list contains the address of the character variable and the address of the length of the character variable. The assembler program can retrieve the variable using the following instructions:

```
L    15,x(0,1)    Get address of address of data
LAM  Q,Q,0(15)    Get data address
L    Q,4(0,15)
LR   S,1          Get copy of argument list address
S    S,=F'4'      Step back to location of length section offset
L    S,0(0,S)     Get offset to length section
AR   S,1          Set address of length section
L    R,x(0,S)     Get character variable length pointer
L    R,0(0,R)     Get character variable length
```

where:

Q, R, S Any general registers except 0, 1, 13, or the program's base register.
x The displacement of the address of the variable from the start of the argument list.

After the above instructions run, Q contains the address of the character variable and R contains the length of the character variable.

Returning a Function Value from an Assembler Program

If the assembler language subprogram is called as a function subprogram (i.e., by a function reference within the Fortran program), the assembler language subprogram must return a function value that corresponds to the type of the value that the Fortran program expects to be returned. The Fortran program will have declared the FUNCTION name with a type that corresponds to the type of the value returned (for example, INTEGER TIMER).

The assembler language subprogram must return a value that matches this type, and must return this value according to the following convention, which is different for character and non-character types.

For non-character functions, the value is returned in one of the following registers:

General Register 0

INTEGER or LOGICAL value

Floating-point Register 0

REAL (REAL*4) or DOUBLE PRECISION (REAL*8) value

Floating-point Registers 0,2

Extended precision REAL (REAL*16), COMPLEX*8, or COMPLEX*16. For COMPLEX values, the real part goes in Register 0 and the imaginary part in Register 2.

Floating-point Registers 0,2,4,6

COMPLEX*32; the real part goes in Registers 0 and 2; the imaginary part goes in Registers 4 and 6.

For character functions, the function value is returned in storage. A character function is always passed the character form of an argument list (as described in "Passing Character Arguments Using the Standard Linkage Convention" on page 454 and "Passing Character Arguments Using the Extended Linkage Convention" on page 456). The last argument in this list provides the address and length of the storage area in which the function value will be returned.

For assembler functions called using the Standard Linkage Convention, the function can return a character value using the following instructions (assuming it is at most 256 characters long):

```

L    Q,x(0,1)      Get data address
LR   S,1
S    S,=F'4'
L    S,0(0,S)
AR   S,1
L    R,x(0,S)      Get character variable length pointer
L    R,0(0,R)      Get character variable length
BCTR R,0
EX   R,MOVE
.
.
MOVE MVC 0(0,Q),LOC

```

where:

Q, R, S Any general registers except 0 or 1.

x The displacement of the last entry in the argument list.

LOC The address of the character value to be returned.

Following is an alternative solution to moving the returned character value (especially when it is greater than 256 bytes):

```

L    14,x(0,1)      Receiving field
LR   15,1           Receiving length
S    15,=F'4'
L    15,0(0,15)
AR   15,1
L    15,x(0,15)
L    15,0(0,15)
LA   2,LOC          Character value to move
LR   3,15           Length same as receiving
MVCL 14,2

```

Assembler Considerations

where:

x The displacement of the last entry in the argument list.
LOC The address of the character value to be returned.

Note: If the length of the character value in the subprogram is less than the receiving length, then, in place of the load register (LR 3,15) entry, use:

```
          L      3,LCLLEN      Length of local character value
          ICM    3,X'8',BLANK  Inserts pad character
BLANK     DC     C' '
```

where:

LCLLEN A fullword containing the length of the local character value.

For assembler functions called using the Extended Linkage Convention, the function can return a character value using the following instructions (assuming it is at most

256 characters long):

```
          L      15,x(0,1)      Get address of data address
          LAM    Q,Q,0(15)      Get data address
          L      Q,4(0,15)
          LR     S,1
          S      S,=F'4'
          L      S,0(0,S)
          AR     S,1
          L      R,x(0,S)       Get character variable length pointer
          L      R,0(0,R)       Get character variable length
          BCTR   R,0
          EX     R,MOVE
          .
          .
          MOVE   MVC  0(0,Q),LOC
```

where:

Q, R, S Any general registers except 0 or 1.
x The displacement of the last entry in the argument list.
LOC The address of the character value to be returned.

Following is an alternative solution to moving the returned character value (especially when it is greater than 256 bytes):

```
          L      15,x(0,1)      Get address of data address
          LAM    14,14,0(15)    Receiving field
          L      14,4(0,15)
          LR     15,1           Receiving length
          S      15,=F'4'
          L      15,0(0,15)
          AR     15,1
          L      15,x(0,15)
          L      15,0(0,15)
          LAE    2,LOC          Character value to move
          LR     3,15           Length same as receiving
          MVCL   14,2
```

where:

x The displacement of the last entry in the argument list.
 LOC The address of the character value to be returned.

Note: If the length of the character value in the subprogram is less than the receiving length, then, in place of the load register (LR 3,15) entry, use:

	L	3,LCLLEN	Length of local character value
	ICM	3,X'8',BLANK	Inserts pad character
BLANK	DC	C' '	

where:

LCLLEN A fullword containing the length of the local character value.

Returning to Alternate Return Points

When a statement number is an argument in a CALL to an assembler subprogram, the subprogram cannot access the statement number argument.

To accomplish the same thing as the Fortran statement RETURN i (used in Fortran subprograms to return to a statement other than that immediately following the CALL), the assembler subprogram must place 4*i in register 15 before returning to the calling program.

For example, when the statement:

```
CALL SUB(A,B,&10,&20)
```

is used to call an assembler subprogram, the following instructions would cause the subprogram to return to the proper point in the calling program:

```

:
LA    15,4           To return to 10

BCR   15,14
:
LA    15,8           To return to 20

BCR   15,14
```

Internal Representation of VS FORTRAN Version 2 Data

If you are using VS FORTRAN Version 2 data in your assembler language programs, you should be aware of the formats VS FORTRAN Version 2 uses within the computer.

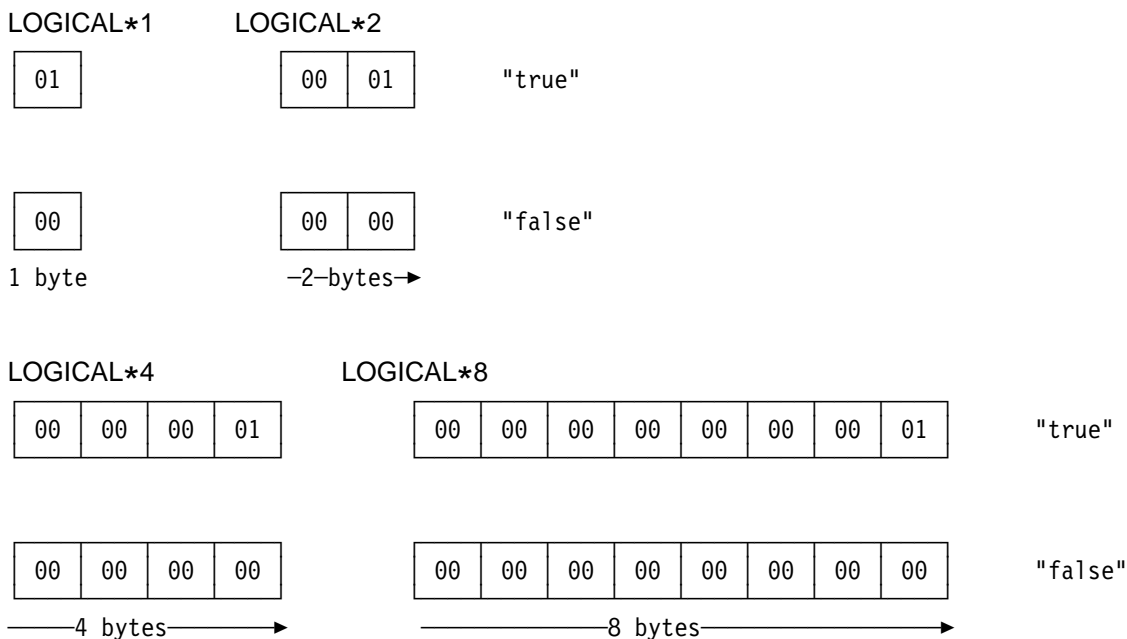
For REAL and COMPLEX items in internal storage, a nonzero floating-point number is said to be normalized if the first hexadecimal digit of its fraction is not zero. The normalized representation of a floating-point zero has sign, characteristic, and fraction all equal to zero.

The following examples show how VS FORTRAN Version 2 data items appear in internal storage. For a complete discussion of the internal representation of data, see *IBM System/370 Principles of Operation*.

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 1, 2, 4, or 8 bytes in length. Their value can be "true" or "false."

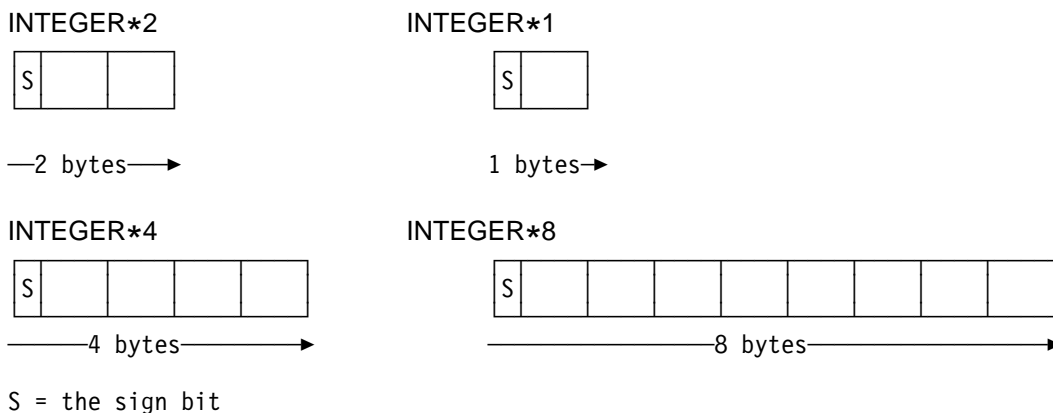
Their internal representation in hexadecimal notation is:



Less than or equal to zero is "false." Greater than zero is "true." If you set a logical value to "true," its internal representation is 01 or 0001. However, if you test the numeric value, it is whatever you set it to be.

Integer Items in Internal Storage: Integer items are treated internally as two's complement binary fixed-point signed operands, 1, 2, 4, or 8 bytes in length.

Their internal representation is:



Bytes Items in Internal Storage: Are identical to INTEGER*1.

Unsigned Items in Internal Storage: Unsigned items are treated internally as binary fixed-point unsigned operands, one byte in length.

Their internal representation is:



1 byte

Real Items in Internal Storage: The compiler converts real items into 4-byte, 8-byte, or 16-byte floating-point numbers.

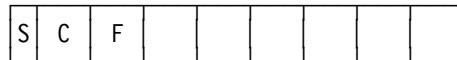
Their internal representation is:

REAL*4



————4 bytes————→

DOUBLE PRECISION (REAL*8)

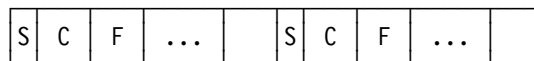


————8 bytes————→

For REAL*4 and DOUBLE PRECISION items, the codes shown are:

- S = sign bit (bit 0)
- C = characteristic, 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

REAL*16 (extended precision)



0 8 64 72

————16 bytes————→

For extended precision items, the codes are:

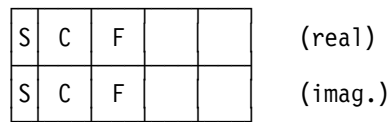
- S = sign bit (sign for the item in bits 0 and 64)
- C = characteristic, in bit positions 1 through 7
and 65 through 71 (the value in bit positions 63 through 71
is 14 less than that in bit positions 1 through 7)
- F = fraction, in bit positions 8 through 63, and 72 through 127

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.

Assembler Considerations

The internal representations of complex numbers are:

COMPLEX*8

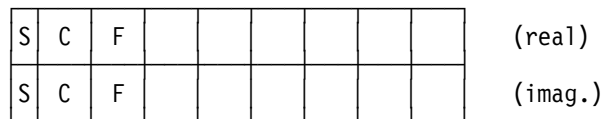


————4 bytes————→

For COMPLEX*8 items, the codes shown are:

- S = sign bit (bit 0)
- C = characteristic, in bit positions 1 through 7
- F = fraction, which occupies bit positions 8 through 31

COMPLEX*16

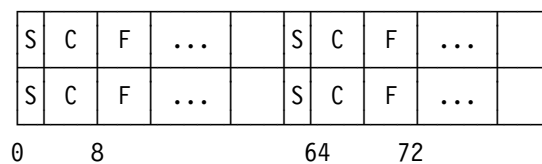


————8 bytes————→

For COMPLEX*16 items, the codes shown are:

- S = sign bit (bit 0)
- C = characteristic, in bit positions 1 through 7
- F = fraction, which occupies bit positions 8 through 63

COMPLEX*32



————16 bytes————→

For COMPLEX*32 Items, the codes are:

- S = sign bit (sign for the item in bits 0 and 64)
- C = characteristic, 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

Requesting Compilation from an Assembler Program

VS FORTRAN Version 2 can be started in either 24-bit or 31-bit addressing mode by using the CALL, ATTACH, or LINK macro instructions in an assembler language program.

The program must supply to the Fortran compiler:

- The information usually specified in the PARM parameter of the EXEC statement (under MVS) or the compiler invocation (under CMS).
- The ddnames of the MVS data sets or CMS files to be used during processing by the Fortran compiler. These can be any valid ddnames.

Name	Operation	Operand
[<i>name</i>]	LINK ATTACH	EP= <i>compiler-name</i> , PARAM=(<i>optionaddr</i> [, <i>ddnameaddr</i>]), VL=1
[<i>name</i>]	CALL	FORTVS2, (<i>optionaddr</i> [, <i>ddnameaddr</i>]), VL

compiler-name

Specifies the program name of the compiler to be invoked. FORTVS2 is specified for VS FORTRAN Version 2.

optionaddr

Specifies the address of a variable-length list containing information usually specified in the PARM parameter.

The option list must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. If there are no parameters, the count must be zero. The option list is free form, with each field separated by a comma. No blanks should appear in the list.

ddnameaddr

Specifies an alternate list of ddnames to be used to refer to data sets used during Fortran compiler processing. This address is supplied by the invoking program. If standard ddnames are used, this operand may be omitted.

The ddname list must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. Each name of fewer than eight bytes must be left-justified and padded with blanks. If an alternate ddname is omitted from the list, the standard name is assumed. If the name is omitted from within the list, the 8-byte entry must contain binary zeros. Names can be completely omitted only from the end of the list.

The sequence of the 8-byte entries in the ddname list is as follows:

Entry	Alternate Name
1	SYSLIN (under MVS), TEXT (under CMS)
2	00000000
3	00000000
4	00000000
5	SYSIN (under MVS), FORTRAN (under CMS)
6	SYSPRINT (under MVS), LISTING (under CMS)
7	SYSPUNCH
8	00000000
9	00000000
10	SYSTEM
11	SYSLIB

VL=1 or VL

Specifies that the sign bit of the last fullword of the address parameter list is to be set to 1.

Figure 158 shows an example of link macro instructions.

```

LINK      CSECT
          USING *,12
          STM  14,12,12(13)
          LR   12,15
          ST   13,SAVE+4
          LA   15,SAVE
          ST   15,8(,13)
          LR   13,15
*
*      INVOKE THE COMPILER
*
          OPEN  (COMPILER)
          LINK  EP=FORTVS2,PARAM=(OPTIONS,DDNAMES),VL=1,DCB=COMPILER
          CLOSE (COMPILER)
          L     13,4(,13)
          LM    14,12,12(13)
          SR    15,15
          BR    14
*
*      CONSTANTS AND SAVE AREA
*
SAVE      DC    18F'0'
OPTIONS   DC    H'24',C'XREF,LIST,GOSTMT,MAP,OBJ'
DDNAMES   DC    H'88',CL8'MYSYSL',3XL8'0000000000000000'
          DC    CL8'MYSYSI',CL8'MYSYSPT',CL8'MYSYSPU'
          DC    2XL8'0000000000000000'
          DC    CL8'MYSYST'
          DC    CL8'MYSYSLIB'
COMPILER  DCB    DDNAME=VSFORT,DSORG=PO,MACRF=R
          END

```

Figure 158. Link Macro Instruction Example

Appendix C. Object Module Records

The object module consists of five types of records, identified by the characters ESD, TXT, RLD, SYM, or END in columns 2 through 4. The first position of each record contains X'02'; positions 73 through 80 contain the first 4 characters of the program name followed by a 4-digit sequence number. The remainder of the record contains program information.

This appendix describes the SYM record only. For information on the other records, see *Assembler H Version 2 Application Programming: Guide*.

If you request the SYM compile-time option, VS FORTRAN Version 2 produces SYM records containing symbolic information for products like TSO TEST. SYM records are similar in form and content to those described in *Assembler H Version 2 Application Programming: Guide*.

SYM records are built for variables and arrays only. The locations of the variables or arrays are either in a LOCAL area (to the module) or in a common area. Note that if the common area is redefined from program unit to program unit, then the SYM records for the common area vary to match the definition in the program unit.

The format of the SYM records is as follows:

Columns Contents

1	X'02'
2-4	SYM
5-10	Blank
11-12	Number of bytes of text in variable or array field (columns 17 through 72)
13-16	Blank
17-72	Variable field (see below)
73-80	Deck ID and/or sequence number. The deck ID is the program name. The name can be 1 to 8 characters long. If the name is fewer than 8 characters long or if there is no name, the remaining columns contain a card sequence number.

The variable field (columns 17 through 72) contains up to 20 bytes of text. The contents of the fields within an individual entry are as follows:

1. Nondata-type SYM card

Type Displacement Name

xx	000000	commons
----	--------	---------

1 byte 3 bytes 1-7 bytes

- Type can identify a CSECT or a common area, and type codes used include the length of the name.
- xx values for CSECTs are:

X'10' indicates CSECT and a name 1 byte long

X'11' indicates CSECT and a name 2 bytes long

X'12' indicates CSECT and a name 3 bytes long
 X'13' indicates CSECT and a name 4 bytes long
 X'14' indicates CSECT and a name 5 bytes long
 X'15' indicates CSECT and a name 6 bytes long
 X'16' indicates CSECT and a name 7 bytes long

- commons is the name of the program (for example, MAIN, A, SUMM, FLEX, and so on).
- xx values for COMMON are:
 - X'30' indicates COMMON and a name 1 byte long
 - X'31' indicates COMMON and a name 2 bytes long
 - X'32' indicates COMMON and a name 3 bytes long
 - X'33' indicates COMMON and a name 4 bytes long
 - X'34' indicates COMMON and a name 5 bytes long
 - X'35' indicates COMMON and a name 6 bytes long
 - X'36' indicates COMMON and a name 7 bytes long
 - X'38' indicates COMMON and no name (blank COMMON)
- commons is the name of the COMMON (for example, X, COMM, ADDIT, FILL, and so on).

2. Data-type SYM card

Type	Displacement	Name	Variable Portion
xx	displc	varname	recordinform

1 byte 3 bytes 1–7 bytes 5–6 bytes

- Type can be for a SCALAR or an ARRAY variable.
- Type codes used include the length of the name and multiplicity.
- xx values for scalars are:
 - X'80' indicates data, no multiplicity, and a 1-byte name
 - X'81' indicates data, no multiplicity, and a 2-byte name
 - X'82' indicates data, no multiplicity, and a 3-byte name
 - X'83' indicates data, no multiplicity, and a 4-byte name
 - X'84' indicates data, no multiplicity, and a 5-byte name
 - X'85' indicates data, no multiplicity, and a 6-byte name
 - X'86' indicates data, no multiplicity, and a 7-byte name
- xx values for array variables are:
 - X'C0' indicates data, multiplicity, and a name 1 byte long
 - X'C1' indicates data, multiplicity, and a name 2 bytes long
 - X'C2' indicates data, multiplicity, and a name 3 bytes long
 - X'C3' indicates data, multiplicity, and a name 4 bytes long
 - X'C4' indicates data, multiplicity, and a name 5 bytes long
 - X'C5' indicates data, multiplicity, and a name 6 bytes long
 - X'C6' indicates data, multiplicity, and a name 7 bytes long
- displc is the displacement of the variable or array into the module.
- varname is the name of the variable or array (for example, X, Y, Z, SUMM, FLEX, and so on).
- recordinform is the variable portion, which contains the length of the data and the multiplicity (1 for a scalar).

- recordinform is the variable portion, which contains the length of the data array element and the multiplicity (number of elements in the array). There is no information concerning dimensionality.

recordinform is further divided as follows:

Data Type	Length	Multiplicity
bb	leng	elemen
1 byte	1 or 2 bytes	3 bytes

The data type field may contain the following values:

bb = X'00' which means CHARACTER
 X'04' which means LOGICAL*1 (hexadecimal)
 X'04' which means LOGICAL*2 (hexadecimal)
 X'04' which means LOGICAL*4 (hexadecimal)
 X'04' which means LOGICAL*8 (hexadecimal)
 X'0C' which means UNSIGNED*1 (byte)
 X'14' which means BYTE (byte)
 X'14' which means INTEGER*1 (byte)
 X'14' which means INTEGER*2 (halfword)
 X'10' which means INTEGER*4 (word)
 X'10' which means INTEGER*8 (D-type)
 X'18' which means REAL*4 (E-type)
 X'1C' which means REAL*8 (D-type)
 X'38' which means REAL*16 (L-type) (extended precision)
 X'18' which means COMPLEX*8 (E-type) (2 E-types)
 X'1C' which means COMPLEX*16 (D-type) (2 D-types)
 X'38' which means COMPLEX*32 (L-type) (2 L-types)

The length value is actually the length code or the actual length minus one. Character and logical items have lengths of 2 bytes. The length field may contain the following values:

leng = X'nnnn' which means CHARACTER with a length of nnnn + 1, where 'nnnn' is the hexadecimal length
 X'0000' which means LOGICAL*1 (hexadecimal)
 X'0001' which means LOGICAL*2 (hexadecimal)
 X'0003' which means LOGICAL*4 (hexadecimal)
 X'0007' which means LOGICAL*8 (hexadecimal)
 X'00' which means UNSIGNED*1 (byte)
 X'00' which means BYTE (byte)
 X'00' which means INTEGER*1 (byte)
 X'01' which means INTEGER*2 (halfword)
 X'03' which means INTEGER*4 (word)
 X'07' which means INTEGER*8 (D-type)
 X'03' which means REAL*4 (E-type)
 X'07' which means REAL*8 (D-type)
 X'0F' which means REAL*16 (L-type) (extended precision)
 X'03' which means COMPLEX*8 (E-type) (2 E-types)
 X'07' which means COMPLEX*16 (D-type) (2 D-types)
 X'0F' which means COMPLEX*32 (L-type) (2 L-types)

elemen = the number of elements of an array (only valid for an array).

3. Punched output format

The SYM record output is part of the text/object file. Each SYM record contains the information for one item. There is one segment of information per record. For example, the information concerning the CSECT is on one record. The information for one variable (scalar or array) is on a record. All the information is tightly packed on each record. The format of the punched record is similar to that provided by the Assembler (F or H) (see general SYM record format above).

A sample hexadecimal representation of a nondata CSECT record is as follows:

```
02E2E8D4404040404040000A4040404015000000C1C2C3C4C5C6
```

where:

02 = X'02'
E2E9D4 = SYM
404040404040 = blanks
000A = 10
40404040 = blanks
15 = CSECT (or Fortran program) with a 6-character name
000000 = displacement from beginning of the CSECT/Fortran program
C1C2C3C4C5C6 = CSECT/program name 'ABCDEF'

Note: The normal record is 80 characters long. The rest is not shown because it is blank, or sequence numbers.

A sample hexadecimal representation of a data variable record is as follows:

```
02E2E8D4404040404040000A40404040850001C0D1D2D3D4D5D61003000001
```

where:

02 = X'02'
E2E8D4 = SYM
404040404040 = blanks
000A = 10
40404040 = blanks
85 = scalar with a 6-character name
0001C0 = displacement from beginning of the CSECT/Fortran program
D1D2D3D4D5D6 = JKLMNO, scalar variable name
10 = INTEGER*4
03 = length of 3 bytes
000001 = multiplicity of 1

Appendix D. Compatibility and Migration Considerations

You must recompile main programs and subprograms that share static common blocks to run them in parallel. You do not need to recompile main programs and subprograms that share dynamic common blocks.

VS FORTRAN Version 2 produces programs that use the vector facility. You can run VS FORTRAN Version 1 programs on the IBM 3090 hardware; however, they cannot utilize the vector facility feature of that system. See Chapter 16, "Using the Vector Feature" on page 277 for a description of this feature.

Processing of the XREF and MAP compiler options changed with VS FORTRAN Version 2 Release 6 and no longer influences the actual mapping of storage. Some differences might be noticed in programs that assumed the particular mapping of previous releases.

With VS FORTRAN Version 2 Release 6, error message numbers 0 through 499 are reserved. Users can use error message numbers 500 through 899. Programs written for prior releases that used error message numbers 302 through 599 as user-defined error message numbers must be modified.

With VS FORTRAN Version 2 Release 6 CMPLXOPT is the default suboption value when VECTOR is specified. This requires COMPLEX*8 data to be aligned on a double-word boundary. Data in programs compiled by prior releases may not be properly aligned (for example, arguments) and may require those programs to be recompiled or new programs interfacing with them to be compiled with the NOCMPLXOPT vector suboption.

Mathematical routines with improved precision and greater speed are incorporated with VS FORTRAN Version 2. For compatibility of results of mathematical computations between VS FORTRAN Version 1 and Version 2, you may want to use the VS FORTRAN Version 2 copies of the standard mathematical routines provided with the VS FORTRAN Version 1. The Version 1 service subroutines displaced by the new routines are provided with Version 2 in the VSF2MATH library and are called the Alternative Mathematical Library Routines. The routines that were in the Alternative Mathematical library (VALTLIB) in VS FORTRAN Version 1 are no longer available in VS FORTRAN Version 2. Figure 159 shows which libraries contain the various scalar mathematical routines for each version.

Figure 159. Libraries Containing Mathematical Routines

Routines	Version 1 Library	Version 2 Library
New scalar math routines	—	VSF2FORT
Old standard scalar math routines	VFORTLIB	VSF2MATH
Old alternative math routines	VALTLIB	Not available

See the *VS FORTRAN Version 2 Language and Library Reference* for a description of the new mathematical routines.

VECTOR(REDUCTION) under Version 2 Release 3 has been enhanced to improve performance. This change may alter the order in which operations are performed and therefore may affect program results.

Callable routines to provide a return code, the current date, and the current time were added to the VS FORTRAN Version 2. They are new capabilities not available with earlier VS FORTRANs. See *VS FORTRAN Version 2 Language and Library Reference* for a description of these routines.

In VS FORTRAN Version 2, mixed case source input is allowed. The compiler interprets it as uppercase. Wherever character data is entered that is interpreted by the compiler or library, its value is recognized independent of case. As an extension of the FORTRAN-77 standard, FIPS flagging is provided. See *VS FORTRAN Version 2 Language and Library Reference* for more information.

VS FORTRAN Version 2 relieves size constraints on address constants for referenced labels, computed GO TO statements, and CALL arguments. This allows larger programs to be compiled, specifically at OPT(0). See *VS FORTRAN Version 2 Language and Library Reference*, which describes the limits of these compiler entities.

The multitasking facility (MTF) was provided with VS FORTRAN Version 1.4.1 SPE, and support for it is carried forward with VS FORTRAN Version 2. MTF allows users to get improved run-times on multiprocessors and attached-processor systems. See Chapter 20, "The Multitasking Facility (MTF)" on page 397 for a description of this feature.

VS FORTRAN Version 2 specifies that an @PROCESS statement must be placed before all other source statements in a compilation unit. Flexibility is provided by allowing an @PROCESS to be preceded by comment lines, EJECT statements, or certain INCLUDE statements. Since Release 3, VS FORTRAN Version 2 restricts some of this flexibility by ignoring the following compile-time options, and providing a level 4 message, if they appear in an @PROCESS statement preceded by comments, EJECT statements, or INCLUDE statements:

- DBCS
- SAA
- FIPS
- LANGLVL
- CHARLEN
- NAME
- VECTOR
- FREE|FIXED
- DIRECTIVE.

Previous to VS FORTRAN Version 2 Release 3, defaults for RECFM, LRECL, and BLKSIZE could not be modified at installation time. If you previously relied on defaults for these options and your site modified the defaults when installing Release 3, your programs may run incorrectly. For such programs, be sure to code these options on the file definition or CALL FILEINF statement in order to avoid problems.

Version 2 libraries detect OPEN/CLOSE inconsistencies not detected in Version 1. For this reason. OCSTATUS and NOOCSTATUS run-time options are introduced

in VS FORTRAN Version 2 to allow load modules compiled with Version 1 to run with Version 2 libraries.

In the case of LRECL for files with record format FB, FBA, VB, or VBA, the IBM-supplied default for formatted I/O differs from previous releases if the block size is greater than 800 for MVS or 80 for VM. In previous releases, the LRECL value for such data sets was always made equal to the block size; in Release 3, the default for LRECL is 800 for MVS and 80 for VM. For more information on installation defaults, see Chapter 12, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 227.

Differences Among Certain IBM FORTRAN Compilers

In VS FORTRAN Versions 1 and 2, logical variables may contain only logical values and may not appear in arithmetic expressions (an error or serious error message is issued). Logical variables may not contain numeric or character values either (an error message is not issued for these conditions however). This is true for both FORTRAN 66 and FORTRAN 77. Under FORTRAN 66 only, logical variables may appear in relational expressions (and a warning message is issued). This nonstandard usage of logical variables was permitted in FORTRAN H Extended and FORTRAN H.

Some of the Extended Language features permitted with the use of the XL option from FORTRAN H and FORTRAN H Extended are similar to functions in VS FORTRAN Versions 1 and 2. For a description of the bit functions, see *VS FORTRAN Version 2 Language and Library Reference*.

Object modules generated with the ALC option by the FORTRAN H Extended compiler, which refer to a common area also referenced in a VS FORTRAN compilation, may not execute as expected when link-edited together with the VS FORTRAN object module. The ALC option is not supported by VS FORTRAN.

In VS FORTRAN Versions 1 and 2, 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 Versions 1 and 2, evaluation of arithmetic expressions involving constants is performed at compile time (including those containing mixed-mode constants).

In VS FORTRAN Versions 1 and 2, the number of arguments is checked in statement function references. The mode of arguments is checked for statement function references under the LONGLVL(77) option only.

In VS FORTRAN Versions 1 and 2, the form of the compile-time option to name a program is NAME(nam) under FORTRAN 66.

Arguments are received only by location (or name) in FORTRAN 77. The default in FORTRAN 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 FORTRAN 77, but is considered user-supplied under FORTRAN 66 and FORTRAN H and FORTRAN H Extended.

The extended range of a DO loop is not part of the VS FORTRAN Versions 1 and 2 language. It is a valid construction under FORTRAN 66. Under FORTRAN 77, branches into the range of a DO loop from outside the range of the loop are diagnosed by the compiler with a warning message issued at OPT(2), OPT(3), or VECTOR.

In VS FORTRAN Versions 1 and 2, when a variable has been initialized with a DATA statement, that variable cannot appear in a subsequent explicit type statement and a level 12 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 FORTRAN 66 and FORTRAN 77. If it is not, VS FORTRAN Versions 1 or 2 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 Versions 1 and 2, all calculations for arrays with adjustable dimensions are performed by a service subroutine called at all entry points that specify such arrays. This method was required for FORTRAN 77 because it permits redefinition of the parameters with adjustable dimensions in the subprogram but requires that the array properties do not change from those existing at the entry point.

In previous implementations, the output form for a real datum whose value was exactly zero was shown as 0.0 (or .0 if the field width specified was not wide enough to contain the leading zero). The VS FORTRAN Versions 1 and 2 libraries follow the ANSI standard exactly and, for a format edit descriptor of kPEw.d or kPGw.d (which is, for this data value, equivalent to kPEw.d), produces the form required for this edit descriptor. For example, for either kPG13.6 or kPE13.6 edit descriptors, VS FORTRAN Versions 1 and 2 produce the form:

```
0.000000E+00
```

(The scale factor has no effect for this data value.)

In previous implementations, the interpretation of the effect of a positive scale factor did not follow the ANSI standard. For a scale factor, k, where $0 < k < d+2$ (d is the number of digits specified in the E, D, or Q edit formats), the output field contains exactly k significant digits to the left of the decimal point and d-k+1 significant digits to the right of the decimal point. In previous implementations, for k>0, only d-k significant digits appeared to the right of the decimal point. For example, for a datum value of .0000137 and a format descriptor of 2PE13.6, VS FORTRAN Versions 1 and 2 produce:

```
13.70000E-06
```

The previous implementation produces:

```
13.7000E-06
```

FORTRAN G1, FORTRAN H Extended, and VS FORTRAN Versions 1 and 2 use slightly different techniques to raise integer and real variables to integer constant powers:

- FORTRAN G1 generates inline code for integer constant powers up through 6 and calls the service subroutine for all values greater than 6.
- FORTRAN H Extended generates inline code for all integer constant powers except when the base is an INTEGER*2 variable, in which case the service subroutine is used.
- VS FORTRAN Versions 1 and 2 generate code inline for all cases.

These differences in implementation yield the same results provided the values produced are valid. For example, the result of raising an INTEGER*2 variable to a constant power must not exceed the value that can be contained in an INTEGER*2 entity.

The VS FORTRAN Version 2 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 see different results from when you run the programs.

Passing Character Arguments

In releases prior to Release 3 of VS FORTRAN Version 1 for FORTRAN 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 FORTRAN 66, because every character argument generates two items in the parameter list. For FORTRAN 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 both languages, only one item is generated in the parameter list for Hollerith arguments.

Every program that had been compiled with versions of VS FORTRAN Version 1, prior to Release 3, and that either references or defines a user subprogram which has character-type arguments or is itself of character type, must be recompiled with VS FORTRAN Version 1, Release 3 or later, or with VS FORTRAN Version 2.

The reason for this is a change in the construction of parameter lists. The new construction provides a means of passing arguments to functions and subroutines in such a manner that the information needed for character-type arguments is “transparent”; that is, the parameter list can be referenced without any regard to the character-type argument information.

The method is to provide a double parameter list for all argument lists that contain any character-type argument, or for any reference to a character-type function. The primary list consists of pointers to the actual arguments; the secondary list consists

of pointers to the lengths of the actual arguments. The high-order bit in the last argument position of each part of the parameter list will be set on. If there are no character-type arguments, or if the function being referenced is not character-type, only a primary list is passed.

The doubling of all parameter lists, except for intrinsic functions that do not involve character arguments, and for implicitly invoked function references, not only implies that the parameter lists themselves are different, but that the prologues of Fortran subprograms are different in order to process these changed parameter lists. Therefore, if any Fortran program compiled prior to VS FORTRAN Version 1, Release 3, and that references subprograms with character-type arguments (or is a character-type function itself), is to be used with a Fortran program that is compiled with VS FORTRAN Version 1, Release 3 or later, or with Version 2, then the old program must also be recompiled with VS FORTRAN Version 1, Release 3 or later, or with VS FORTRAN Version 2.

Extended Common Blocks

A subroutine or function must be recompiled with the EMODE compile-time option when it accepts arguments and any of these arguments include data that reside in extended common blocks.

Any program that defines extended common blocks must be recompiled using the EC compile-time option.

The multitasking facility cannot be used to reference data that reside in extended common blocks. Programs that previously used the MTF and are to be expanded to use extended common blocks should use the facilities for parallel programming support within VS FORTRAN Version 2 Release 5.

The static debug and the interactive debug facilities cannot be used to debug programs that reference extended common blocks.

Using the Current Library with Existing VS FORTRAN Load Modules

Existing VS FORTRAN Version 1 and VS FORTRAN Version 2 load modules are compatible with the VS FORTRAN Version 2 Release 5 library with the following considerations:

Load modules created with Version 1 Releases 2, 3, and 3.1: Load modules using the MVS reentrant I/O library load module (IFYVRENT) will run if the VS FORTRAN Version 2 library is made available during processing. Load modules created with Version 1 prior to Release 2 that use the reentrant I/O library load module must be relinked with the VS FORTRAN Version 2 library.

Load modules created with Version 1 Release 4 or later and Version 2 Release 1 or later: Load modules that ran in load mode will continue to run if the VS FORTRAN Version 2 Release 5 library is made available during processing.

Load modules with nonshareable portions of object modules will run if the load modules that contain the corresponding shareable portions of those object modules are made available during processing. No relinking is necessary.

Load modules created with Version 1 all releases and Version 2 Releases 1 and 1.1: Load module compatibility may be affected by changes to the input/output semantics. To assist in file existence verification, the semantics for OPEN, CLOSE, and INQUIRE have changed, and the run-time options OCSTATUS and NOOCSTATUS have been added.

Migration Considerations

In addition to some fundamental platform differences, such as operating system differences and differences in floating point number representation, you should consider language and language-related differences that exist between VS FORTRAN and other Fortran products before porting source code to, or from VS FORTRAN. The following sections list the differences between VS FORTRAN Version 2.6 and XL FORTRAN Version 2.3 and AIX VS FORTRAN/ESA Release 2.

Language Elements Supported in XL FORTRAN not Supported in VS FORTRAN

The following lists language elements supported in XL FORTRAN that are not supported in VS FORTRAN:

- SELECT constructs
- Internal subprograms
- Interface blocks
- Derived types and structures
- CYCLE and EXIT statements
- VOLATILE statement
- VIRTUAL statement
- IMPLICIT UNDEFINED type statement
- Expressions are allowed in FORMAT statements.
- Recursion
- Tab characters
- Argument keywords
- An actual argument can be specified by an argument list built-in function (%VAL or %REF).
- Named DO, DO WHILE, CASE, and IF constructs
- Infinite DO constructs are DO constructs with no loop control.
- XL FORTRAN recognizes a set of backslash escapes for compatibility with the C language strings.
- A named common block can be initialized in a program unit other than a block data program unit.
- Any missing subscript for an EQUIVALENCE statement is assumed to be the lower bound of the corresponding dimension statement.
- Debug lines (also known as *D lines*) are recognized.

- The GETENV, ABORT, SYSTEM, SIGNAL, RAND, and SRAND subprograms are intrinsic procedures.
- Character constants and Hollerith constants as typeless constants

Language Elements Supported in AIX VS FORTRAN/ESA not Supported in VS FORTRAN

The following lists language elements supported in AIX VS FORTRAN/ESA that are not supported in VS FORTRAN:

- Recursion
- Tab characters
- PRAGMA ALIAS statement
- %VAL argument list built-in functions
- Backslash escapes for compatibility with C
- Debug lines
- "f77" library routines, including (but not limited to) GETENV, ABORT, SYSTEM, SIGNAL, RAND, and SRAND.
- Full support for the AUTOMATIC and STATIC storage classes. All local variables are AUTOMATIC by default.
- Neither supports `_` as the first character of a name. However, in AIX VSFORTRAN/ESA, the PRAGMA ALIAS may be used to specify global names with leading underscores.

Differences between XL FORTRAN and VS FORTRAN

The following list contrasts the differences between XL FORTRAN and VS FORTRAN language elements:

- For XL FORTRAN, the maximum length of a name is 250 characters; for VS FORTRAN, the maximum length of a name is 32 characters.
- For XL FORTRAN, `_` can be used as the first character of a name; for VS FORTRAN, `_` is not allowed as a first character.
- VS FORTRAN does not support the MIXED option; variables names are case insensitive.
- For XL FORTRAN, a statement can have unlimited continuation lines, with a maximum statement length of 6700 bytes; for VS FORTRAN, a maximum of 99 continuation lines is allowed, with a maximum statement length of 6600 characters.
- For XL FORTRAN, the expression to be evaluated in a computed GO TO statement can be any arithmetic expression. If the expression is a noninteger, it is converted to an integer value before use. For VS FORTRAN, only integer expressions are allowed for computed GO TO statements.
- For XL FORTRAN, the NAMELIST names are folded to lowercase when they are written to the internal file, except when the MIXED compiler option is specified. For VS FORTRAN, the NAMELIST names are folded to uppercase; the mixed option is not supported.
- For XL FORTRAN, trigonometric functions take arguments in radians or in degrees. For VS FORTRAN, only radians are allowed for arguments of trigonometric functions.
- For XL FORTRAN, the maximum number of dimensions that can be declared is 20; for VS FORTRAN, the maximum number is 7.

- XL FORTRAN assumes a default of 'NULL' for the BLANK specifier when reading from a unit for which an OPEN statement was not specified; the default for VS FORTRAN is 'ZERO'.
- For VS FORTRAN, characters are passed in two words, one containing the address of the string and the other containing the length. The words containing the lengths of character arguments appear after all words containing addresses.
VS FORTRAN uses a shadow parameter to pass the length of the string.
- XL FORTRAN fully supports storage classes automatic and static; VS FORTRAN supports only the AUTOMATIC and STATIC statements (all storage is static).
- VS FORTRAN supports all lengths of integer data in intrinsic functions that call for integer arguments.
- In VS FORTRAN, file positioning of an existing file for sequential I/O with an explicit OPEN always positions the file at the beginning. XL FORTRAN positions the file at the end for STATUS=OLD. Positioning with STATUS=UNKNOWN depends on the setting of the -qposition.
- For XL FORTRAN the default size of pointers is 4 bytes; for VS FORTRAN it is 8 bytes, which can be changed by an option or overridden by an explicit specification to 4 bytes.

Language Elements Supported in VS FORTRAN not Supported in AIX VS FORTRAN/ESA

The following lists language elements supported in VS FORTRAN that are not supported in AIX VS FORTRAN/ESA:

- Multitasking facility
- Data-in-virtual
- Asynchronous input/output
- Data Striping

However, on AIX/ESA, data striping is provided by the system and does not require explicit Fortran support.

- Extended Common
- Double-byte character set (DBCS)
- Keyed access methods
- Integer*1 and Integer*8 data types
- Logical*2 and Logical*8 data types
- Unsigned*1 data type
- Byte data type
- Pointer data type
- Private arrays in parallel constructs
- Dynamic storage allocation statements (ALLOCATE, DEALLOCATE, NULLIFY)
- OPEN specifiers POSITION, PAD, DELIM, CHAR, PASSWORD, and KEYS
- INQUIRE specifiers POSITION, PAD, DELIM, CHAR, KEYED, KEYID, KEYLENGTH, KEYSTART, LASTKEY, LASTRECL, and PASSWORD
- The IGNFHU and IGNFHDD service subroutines

Language Elements Supported in VS FORTRAN not Supported in XL FORTRAN

The following lists language elements supported in VS FORTRAN, that are not supported in XL FORTRAN:

- The following debug statements:
 - AT
 - DEBUG
 - END DEBUG
 - DISPLAY
 - TRACE OFF
 - TRACE ON
- The following intrinsic functions:
 - COTAN
 - ALLOCATED
- The following extended error handling subroutines:
 - ERRMON
 - ERRSAV
 - ERRSET
 - ERRSTR
 - ERRTRA
- The following service and utility subprograms:
 - ARGSTR
 - DVCHK
 - OVERFL
 - DUMP/PDUMP
 - CDUMP/CPDUMP
 - EXIT
 - SDUMP
 - XUFLOW
 - IGNFHU
 - IGNFHDD
- Multitasking facility
- Data-in-virtual
- Asynchronous input/output
- Data Striping
- Keyed access methods
- Integer*8 data type
- Logical*8 data type
- UnSigned*1 data type
- Parallel language constructs and service subprograms
- Dynamic storage allocation statements (ALLOCATE, DEALLOCATE, NULLIFY) for pointee arrays
- OPEN specifiers POSITION, PAD, DELIM, CHAR, ACTION, PASSWORD, and KEYS

- INQUIRE specifiers POSITION, PAD, DELIM, CHAR, ACTION, KEYED, KEYID, KEYLENGTH, KEYSTART, LASTKEY, LASTRECL, PASSWORD, READ, READWRITE, and WRITE
- The WRITE statement can be used to write over an end-of-file and extends the external file. An ENDFILE, BACKSPACE, CLOSE, or REWIND statement will then reinstate the end of file.

Appendix E. Compiler Report Diagnostic Messages

Compiler Report Diagnostic Messages—Vector	492
Compiler Report Diagnostic Messages—Parallel	528
Compiler Report Diagnostic Messages—Vector and Parallel	549

The messages in this appendix are those that can appear in the compiler report listing when either the REPORT(XLIST) or the REPORT(SLIST) suboption of the VECTOR or PARALLEL compile-time option is requested.

Each message has two versions: a short form and a long form. The short form is printed in the right margin on the same line as the statement to which it applies. The long form, which contains a more detailed explanation of the reason for the message, appears in a message summary listing following the program listing.

This appendix contains three sections:

- “Compiler Report Diagnostic Messages—Vector” on page 492.
- “Compiler Report Diagnostic Messages—Parallel” on page 528.
- “Compiler Report Diagnostic Messages—Vector and Parallel” on page 549.

The messages are ordered according to message number. Each entry gives the short and long forms of the message, a brief explanation of what the message means, and a description of any supplemental data (such as lists of variable names or ISNs) that might be inserted into the long form of the message. Where appropriate, there are examples of situations that cause the message, and suggestions for rewriting programs to improve vectorization and parallel code generation.

The following status flags mark the loops printed on the program listing portion of the compiler report:

- Unanalyzable Loop Messages (UNAN flag)¹⁰
- Recurrence Detection Messages (RECR flag)
- Unsupported Operation Messages (UNSP flag)¹⁰
- Vectorizable Statements Messages (ELIG and VECT flags)
- Listing Clarification Messages (SCAL and VECT flags)
- Vector Directive Messages (VDIR flag).

For a complete description of the status flags, see “Producing Compiler Reports” on page 344.

Warning: Because of added vector and parallel capability, the messages in this section are valid for VS FORTRAN Version 2 Release 5 and later and are not necessarily the same for prior releases.

¹⁰ If you specify SDUMP, the ISNs given in the <ilist> for the UNAN and UNSP long messages are the exact ISN for the statement causing the UNAN or UNSP condition. If you specify NOSDUMP, the ISN in the <ilist> is the ISN for the loop.

Compiler Report Diagnostic Messages—Vector

The following messages apply to errors or conditions that arise when the compiler attempts to vectorize your program.

ILX0101I Short Form: NON-INTEGER LOOP CONTROL
Long Form: A LOOP CONTROL PARAMETER IS NOT INTEGER*4.

Explanation: Indicates that a variable that is not INTEGER*4 is used as part of an expression controlling the iteration of a loop or as a DO loop variable.

Possible Response: If the lower bound, upper bound, or increment expressions are not INTEGER*4, replace these expressions with INTEGER*4 variables that have been assigned the appropriate values prior to the loop.

If the DO loop variable is not INTEGER*4 and it can be replaced by one that is, do so.

ILX0102I Short Form: MORE THAN 8 NESTED LOOPS
Long Form: ANALYSIS IS RESTRICTED TO LOOPS AT THE INNERMOST EIGHT LEVELS OF NESTING.

Explanation: Indicates that a loop has not been considered for vectorization because it contains nested loops more than eight levels deep.

ILX0103I Short Form: NESTED LOOP NOT ANALYZABLE
Long Form: SOME NESTED LOOP WAS FOUND TO BE UNANALYZABLE.

Explanation: Indicates that a loop contains a nested loop which was not eligible for vectorization analysis.

Example:

```
C EXAMPLE
C NESTED LOOP NOT ANALYZABLE
  REAL*4 A(100,100)

  DO 10 I = 1,100
    DO 10 J = 1,100
      A(I,J) = A(I,J) ** 2.1
      WRITE(6,*,ERR=999) A(I,J)
    10 CONTINUE
```

In this case, the inner loop is unanalyzable because it contains an I/O statement that is unanalyzable. The outer loop is marked as unanalyzable since it surrounds the unanalyzable inner loop.

Possible Response: Identify the loop causing the rejection and attempt to recode it to eliminate the problem.

Modified Example:

```
C POSSIBLE RESPONSE
C NESTED LOOP NOT ANALYZABLE
  REAL*4 A(100,100)

  DO 10 I = 1,100
    DO 10 J = 1,100
      A(I,J) = A(I,J) ** 2.1
  10 CONTINUE
  WRITE(6,*,ERR=999) A
```

ILX0104I Short Form: I/O OPERATION
Long Form: CERTAIN I/O STATEMENTS AT ISN(S) <ilist> ARE NOT ANALYZABLE.

Explanation: Indicates that a loop contains one or more I/O statements that cannot be analyzed. The following I/O statements are not analyzable:

- Statements other than READ, WRITE or PRINT
- READ and WRITE statements containing END= or ERR= labels
- Statements that specify a NAMELIST
- Statements containing arrays without subscripts
- Statements that specify internal files
- Asynchronous I/O statements
- Implied-DO statements. (Some implied-DO statements may be analyzable).

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```
C EXAMPLE
C I/O OPERATION
  REAL*4 A(100),B(100)

  DO 10 I = 1,100,10
    A(I) = B(I) * 3.3
    WRITE(6,ID=J) A(I)...A(I+9)
  10 CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements are separated from the portions of the original loop that are eligible for vectorization analysis.

Modified Example:

```

C POSSIBLE RESPONSE
C I/O OPERATION
  REAL*4 A(100),B(100)

  DO 10 I = 1,100,10
    A(I) = B(I) * 3.3
10  CONTINUE

  DO 20 I = 1,100,10
    WRITE(6,ID=J) A(I)...A(I+9)
20  CONTINUE

```

This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0105I Short Form: DO WHILE OR IMPLIED DO
Long Form: ONE OR MORE DO WHILE
LOOPS OR I/O STATEMENTS WITH
IMPLIED DO LOOPS AT ISN(S) <ilist>
ARE NOT ANALYZABLE.

Explanation: Indicates the presence of an unanalyzable loop construct contained within an iterative DO loop. This may be caused by either a DO WHILE statement or by an I/O statement that contains an implied DO loop.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C DO WHILE OR IMPLIED DO
  REAL A(50,10),B(50)

  DO 10 I=1,50
    A(I,5) = B(I) * 3.3
    WRITE(6,*) (A(I,J),J=1,10)
10  CONTINUE

```

Possible Response: Break the loop into two or more loops, so that any unanalyzable constructs are separated from the portions of the original loop that are analyzable.

Modified Example:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C DO WHILE OR IMPLIED DO
  REAL A(50,10),B(50)

  DO 10 I=1,50
    A(I,5) = B(I) * 3.3
10  CONTINUE

  DO 20 I=1,50
    WRITE(6,*) (A(I,J),J=1,10)
20  CONTINUE

```

This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0106I Short Form: CHARACTER DATA
Long Form: ONE OR MORE
STATEMENTS USING CHARACTER
DATA OCCUR AT ISN(S) <ilist>.

Explanation: Indicates the presence of character data.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Possible Response: Break the loop into two or more loops, so that any statements that reference character data are separated from the portions of the original loop that are eligible for vectorization analysis. This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0107I Short Form: UNANALYZABLE STOP OR RETURN
Long Form: STOP OR RETURN
STATEMENTS AT ISN(S) <ilist> ARE NOT
ANALYZABLE BECAUSE THIS LOOP
CONTAINS A NESTED LOOP.

Explanation: Loops that contain STOP or RETURN statements are vectorizable only if they are innermost loops. This message indicates that a loop was rejected because it is an outer loop that contains a a STOP or RETURN statement.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C UNANALYZABLE STOP OR RETURN
  REAL*4 A(50,50),B(50,50),C(50,50)

  DO 20 I = 1,50
    DO 10 J = 1,50
      IF (C(I,J) .EQ. 0.0) STOP
      A(I,J) = B(I,J)
10    CONTINUE
20  CONTINUE

```

In this case, the inner loop is eligible for vectorization although the outer loop is not.

Possible Response: You may be able to make the outer loop eligible for partial vectorization if you restructure the code to separate the loop termination test from the rest of the loop. The original nest of loops

may be transformed as follows:

1. Insert a new nest of loops to evaluate the loop termination condition and determine the number of times each loop iterates before terminating. Note that the code within the new nest will remain ineligible for vectorization.
2. Rewrite the original nest to process only the elements that are referenced before the loop terminates. Note that the number of iterations for the inner loop is different on the final iteration of the outer loop than it is on earlier iterations. You will need to duplicate the inner loop to process the final iteration separately.
3. Add a test at the end of the modified code to determine whether the STOP or RETURN statement should be executed.

Modified Example:

```

C POSSIBLE RESPONSE
C UNANALYZABLE STOP OR RETURN
  REAL*4 A(50,50),B(50,50),C(50,50)
  INTEGER IMAX,JMAX
  LOGICAL NEED_STOP

      DO 20 IMAX = 1,50
        DO 10 JMAX = 1,50
          IF (C(IMAX,JMAX) .EQ. 0.0) GOTO 100
10      CONTINUE
20      CONTINUE
100     NEED_STOP = (JMAX.LE.50)

      DO 40 I = 1,IMAX-1
        DO 30 J = 1,50
          A(I,J) = B(I,J)
30      CONTINUE
40      CONTINUE

      IF (NEED_STOP) THEN
        DO 50 J = 1,JMAX - 1
          A(IMAX,J) = B(IMAX,J)
50      CONTINUE
        STOP
      ENDIF

```

Note that this transformation will only be valid if the test that determines whether the loop should terminate is independent of the values that are being computed in the loops. Also note that if the test is not the first statement of that loop, some special processing may be necessary.

ILX0108I Short Form: BRANCH AROUND INNER LOOP
Long Form: THE BRANCH(ES) ORIGINATING AT ISN(S) <ilist> BYPASS ONE OR MORE NESTED LOOPS.

Explanation: Indicates that a loop was rejected because some branch within the loop causes an inner loop to be bypassed.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C BRANCH AROUND INNER LOOP
  REAL*4 A(100),B(100)

      DO 6 I = 1,100
        A(I) = A(I) / 2.0
        IF (A(I) .EQ. 0.0) GO TO 5
        DO 4 J = 1,100
          B(J) = B(J) ** 2.1
4      CONTINUE
5      CONTINUE
6      CONTINUE

```

Possible Response: Break the loop into two or more loops, so that any unanalyzable branches are separated from the portions of the original loop that are eligible for vectorization analysis. This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

Modified Example:

```

C POSSIBLE RESPONSE
C BRANCH AROUND INNER LOOP
  REAL*4 A(100),B(100)

      DO 3 I = 1,100
        A(I) = A(I) / 2.0
3      CONTINUE
C
      DO 6 I = 1,100
        IF (A(I) .EQ. 0.0) GO TO 5
        DO 4 J = 1,100
          B(J) = B(J) ** 2.1
4      CONTINUE
5      CONTINUE
6      CONTINUE

```

ILX0109I Short Form: UNANALYZABLE EXIT BRANCH
Long Form: EXIT BRANCHES ORIGINATING AT ISN(S) <ilist> ARE NOT ANALYZABLE BECAUSE THIS LOOP CONTAINS A NESTED LOOP.

Explanation: Loops that contain exit branches are vectorizable only if they are innermost loops. This message indicates that a loop was rejected because it is an outer loop that contains an exit branch.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C UNANALYZABLE EXIT BRANCH
      REAL*4 A(50,50),B(50,50),C(50,50)

      DO 20 I = 1,50
        DO 10 J = 1,50
          IF (C(I,J) .EQ. 0.0) GOTO 500
          A(I,J) = B(I,J)
10      CONTINUE
20      CONTINUE
      ...
500     CONTINUE

```

In this case, the inner loop is eligible for vectorization, although the outer loop is not.

Possible Response: You may be able to make the outer loop eligible for partial vectorization if you restructure the code to separate the loop termination test from the rest of the loop. The original nest of loops may be transformed as follows:

1. Insert a new nest of loops to evaluate the loop termination condition and determine the number of times each loop iterates before terminating. Note that the code within the new nest will remain ineligible for vectorization.
2. Rewrite the original nest to process only the elements that are referenced before the loops terminate. Note that the number of iterations for the inner loop is different on the final iteration of the outer loop than it is on earlier iterations. You will need to duplicate the inner loop to process the final iteration separately.
3. Add a test at the end of the modified code to determine whether the original branch should be taken.

Modified Example:

```

C POSSIBLE RESPONSE
C UNANALYZABLE EXIT BRANCH
      REAL*4 A(50,50),B(50,50),C(50,50)
      INTEGER IMAX,JMAX
      LOGICAL NEED_BRANCH

      DO 20 IMAX = 1,50
        DO 10 JMAX = 1,50
          IF (C(IMAX,JMAX) .EQ. 0.0) GOTO 100
10      CONTINUE
20      CONTINUE
100     NEED_BRANCH = (JMAX.LE.50)

      DO 40 I = 1,IMAX-1
        DO 30 J = 1,50
          A(I,J) = B(I,J)
30      CONTINUE
40      CONTINUE

      IF (NEED_BRANCH) THEN
        DO 50 J = 1,JMAX - 1
          A(IMAX,J) = B(IMAX,J)
50      CONTINUE
          GOTO 500
        ENDIF
      ...
500     CONTINUE

```

Note that this transformation will only be valid if the test for the loop exit branch is independent of the values that are being computed in the loops. Also note that if the loop exit branch is not the first statement of that loop, some special processing may be necessary.

ILX0110I Short Form: LOOP NOT OPTIMIZABLE
Long Form: VECTORIZATION IS INHIBITED BECAUSE THIS LOOP IS NOT OPTIMIZABLE. THIS MAY BE CAUSED BY AN INDUCTION VARIABLE THAT MAY BE RESET INSIDE THE LOOP OR BY COMPLEX BRANCHING OUTSIDE THE LOOP.

Explanation: Indicates situations where optimization and vectorization of loops are inhibited. This can happen for a variety of reasons:

- When DO loop variables are not guaranteed to behave like standard DO loop variables. For example, this occurs when a DO loop variable is used as a parameter to a subroutine.
- When a DO loop variable is referenced in an EQUIVALENCE statement.
- When certain complicated patterns of branching are used around a DO loop.

Example 1:

```
C EXAMPLE 1
C LOOP NOT OPTIMIZABLE
  EQUIVALENCE (K1,K2)
  REAL*4 A(100)

  DO 10 K1 = 1,100
    A(K2) = A(K2) ** 2.1
10  CONTINUE
```

Possible Response 1: When a loop is not optimizable because the induction variable is used in an EQUIVALENCE statement, attempt to use a different DO loop variable to control the loop iteration.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C LOOP NOT OPTIMIZABLE
  EQUIVALENCE (K1,K2)
  REAL*4 A(100)

  DO 10 K = 1,100
    A(K) = A(K) ** 2.1
10  CONTINUE
```

Example 2:

```
C EXAMPLE 2
C LOOP NOT OPTIMIZABLE
  REAL*4 X(100),Y(100)

  DO 20 K = 1,100
    CALL SUB2(K)
    X(K) = Y(K) ** 2.1
20  CONTINUE
```

Possible Response 2: When a loop is not optimizable because the induction variable is passed as an argument to a subroutine, split the original loop into two or more loops so that the vectorizable statements are not in the same loop as the nonvectorizable statements (in this case, the CALL statement.)

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C LOOP NOT OPTIMIZABLE
  REAL*4 X(100),Y(100)

  DO 19 K = 1,100
    CALL SUB2(K)
19  CONTINUE
  DO 20 K = 1,100
    X(K) = Y(K) ** 2.1
20  CONTINUE
```

This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

Example 3:

```
C EXAMPLE 3
C LOOP NOT OPTIMIZABLE
  REAL*4 Q(100,100),R(100,100)

  DO 160 J=1,100
  DO 160 I=1,100
    L=1
    IF (L.GT.LLIM) GO TO 140
100  DO 120 K=1,100
      Q(J,I) = R(L,K)
120  CONTINUE
      L=L+1
      IF (L.LE.LLIM ) GO TO 100
140  CONTINUE
160  CONTINUE
```

The inner loop is not eligible for optimization due to the complex pattern of branches around that loop.

Possible Response 3: For cases such as this, attempt to redesign the algorithm using structured programming constructs whenever possible.

Modified Example 3:

```
C POSSIBLE RESPONSE 3
C LOOP NOT OPTIMIZABLE
  REAL*4 Q(100,100),R(100,100)

  DO 160 J=1,100
  DO 160 I=1,100
  DO 120 L=1,LLIM
100  DO 120 K=1,100
      Q(J,I) = R(L,K)
120  CONTINUE
160  CONTINUE
```

ILX0111I Short Form: BACKWARD BRANCH
Long Form: ONE OR MORE BACKWARD
BRANCHES TO THE STATEMENT
LABEL(S) <llist> HAVE BEEN FOUND.

Explanation: Indicates the presence of a backward branch or a DO WHILE loop within a DO loop.

Supplemental Data:

<llist> is a list of user defined statement labels that are used to indicate the targets of any backward branches that occur in the loop.

Example:

```
C EXAMPLE
C BACKWARD BRANCH
  REAL*4 A(100),B(1000,100)

  DO 20 I = 1,100
    A(I) = 0.0
    K = 1
10  A(I) = A(I) + B(K,I)
    K = K + 1
    IF (K .LE. 1000) GO TO 10
20  CONTINUE
```


Possible Response: Attempt to replace the backward GOTO with an equivalent DO loop.

Modified Example:

```
C POSSIBLE RESPONSE
C BACKWARD BRANCH
      REAL*4 A(100),B(1000,100)

      DO 20 I = 1,100
        A(I) = 0.0
        DO 15 K = 1,1000
10          A(I) = A(I) + B(K,I)
15          CONTINUE
20          CONTINUE
```

ILX0112I Short Form: INTRINSIC CHARACTER FUNCTION
Long Form: THE CHARACTER MANIPULATION FUNCTION(S) <flist> ARE NOT ANALYZABLE.

Explanation: Indicates that a loop is rejected because it uses one or more of the character manipulation functions (INDEX, LGE, LGT, LLE, and LLT) contained in the VS FORTRAN library.

Supplemental Data:

<flist> is a list consisting of function names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements using character manipulation functions are separated from the portions of the original loop that are eligible for vectorization analysis. This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0113I Short Form: RESTRICTED CONSTRUCT
Long Form: THE LANGUAGE CONSTRUCT(S) <clist> ARE NOT ANALYZED FOR VECTORIZATION.

Explanation: Indicates that a loop is rejected because it contains some language construct that cannot be analyzed by the compiler. These constructs include assigned and computed GOTO statements and NAMELIST statements.

Supplemental Data:

<clist> is a list consisting of the names of the language constructs responsible for the rejection along with the ISNs (internal statement numbers) of the statements in which they are used.

Example:

```
C EXAMPLE
C RESTRICTED CONSTRUCT
      INTEGER*4 TEST(100)
      REAL*4 W(100),X(100),Y(100),Z(100)

      DO 1000 I = 1,100
        GOTO (400,500,600),TEST(I)
        W(I) = 0.0
        GOTO 1000
400      W(I) = X(I)
        GOTO 1000
500      W(I) = Y(I)
        GOTO 1000
600      W(I) = Z(I)
1000    CONTINUE
```

Possible Response: In the case of an assigned or computed GOTO statement, it may be possible to recode the loop so that the same logic structure is achieved using logical and arithmetic IF statements.

Modified Example:

```
C POSSIBLE RESPONSE
C RESTRICTED CONSTRUCT
      INTEGER*4 TEST(100)
      REAL*4 W(100),X(100),Y(100),Z(100)

      DO 1000 I = 1,100
        IF (TEST(I).LT.1 .OR. TEST(I).GT.3) THEN
          W(I) = 0.0
        ELSE IF (TEST(I).EQ.1) THEN
400          W(I) = X(I)
        ELSE IF (TEST(I).EQ.2) THEN
500          W(I) = Y(I)
        ELSE IF (TEST(I).EQ.3) THEN
600          W(I) = Z(I)
        ENDIF
1000    CONTINUE
```

You should be careful when doing this, since if the transformed code fails to vectorize, the resulting scalar program may run more slowly than the original program.

ILX0114I Short Form: USER FUNCTION OR SUBROUTINE
Long Form: THE USER FUNCTION(S) OR SUBROUTINE(S) <flist> ARE NOT ANALYZABLE.

Explanation: Indicates that a loop contains a subroutine call or a reference to an external user defined function.

Supplemental Data:

<flist> is a list consisting of function and subroutine names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements containing a subroutine call or a reference to an external user defined function are separated from the portions of the original loop that

are eligible for vectorization analysis. This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0115I Short Form: NON-MATHEMATICAL IMPLICIT
Long Form: THE IMPLICITLY CALLED NON-MATHEMATICAL SUBPROGRAM(S) <flist> HAVE BEEN USED. THESE SUBPROGRAMS ARE NOT ANALYZABLE.

Explanation: Indicates that some statement in the loop will generate a reference to an implicitly invoked character subprogram (CCMPR#, CMOVE#, or CNCAT#) or to an implicitly invoked utility program (DSPAN#, DSPN2#, DSPN4#, or DYCMN#). These programs are described in *VS FORTRAN Version 2 Language and Library Reference*.

Supplemental Data:

<flist> is a list consisting of function names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements resulting in compiler call(s) to implicitly invoked character subprograms or to implicitly invoked utility programs are separated from the portions of the original loop that are eligible for vectorization analysis. This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0116I Short Form: ARRAY W/O SUBSCRIPTS IN I/O
Long Form: ARRAYS IN I/O WITHOUT SUBSCRIPTS ARE NOT ANALYZABLE.

Explanation: Indicates the presence of I/O statements containing arrays without subscripts (such as PRINT*,A) that cannot be analyzed for vectorization.

Example:

```
C EXAMPLE
C ARRAY W/O SUBSCRIPTS IN I/O
  REAL*4 A(800,800),B(800),C(800)

      DO 20 I=1,800
        READ(5,*) B,C(I)
        DO 10 J=1,800
          A(J,I) = B(J) + C(I)
10      CONTINUE
        S = S + A(I,1)
20      CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements containing full arrays are separated from the portions of the original loop that are eligible for vectorization analysis.

Modified Example:

```
C POSSIBLE RESPONSE
C ARRAY W/O SUBSCRIPTS IN I/O
  REAL*4 A(800,800),B(800),C(800)

      DO 10 I=1,800
        READ(5,*) B,C(I)
        DO 10 J=1,800
          A(J,I) = B(J) + C(I)
10      CONTINUE

      DO 20 I=1,800
        S = S + A(I,1)
20      CONTINUE
```

This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0117I Short Form: EQUIVALENCE OFFSET UNKNOWN
Long Form: THE EQUIVALENCE OFFSET(S) FOR VARIABLE(S) <vlist> COULD NOT BE ANALYZED. AN EQUIVALENCE GROUP IS IRREGULAR, OR HAS ARRAYS WITH DIFFERENT DATATYPES OR SCALAR VARIABLES. THE COMPILER ASSUMES THAT THE VARIABLES CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: To compute dependences between EQUIVALENCE variables, the following must be true:

- The variables must have the same element size.
- The computed equivalence offset for the variables must be a multiple of the element size.
- The variables must be array variables.

If any of these conditions are not true, the variables may be presumed to carry dependences and will not be vectorized.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```

C EXAMPLE
C EQUIVALENCE OFFSET UNKNOWN
    REAL*8 A(100),T
    REAL*4 B(100)
    EQUIVALENCE (A,B,T)

    DO 10 I = 100,1,-1
        A(I) = B(I) ** 2.1 * T
10    CONTINUE

```

Possible Response 1: Make the element sizes of the EQUIVALENCE variables the same. Also, replace the scalar variable with an array.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C EQUIVALENCE OFFSET UNKNOWN
    REAL*8 A(100),B(100),C(1)
    EQUIVALENCE (A,B,C)

    DO 10 I = 100,1,-1
        A(I) = B(I) ** 2.1 * C(1)
10    CONTINUE

```

Possible Response 2: Insert the IGNORE RECRDEPS directive to instruct the compiler to assume that a dependence due to the EQUIVALENCE variables does not occur. Before using this directive, you should analyze the storage mapping and subscript expressions of the variables involved and make sure that the different variables do not reference identical storage locations while the loop is running.

Modified Example 2:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C EQUIVALENCE OFFSET UNKNOWN
    REAL*8 A(100),T
    REAL*4 B(100)
    EQUIVALENCE (A,B,T)

*DIR    IGNORE RECRDEPS
    DO 10 I = 100,1,-1
        A(I) = B(I) ** 2.1 * T
10    CONTINUE

```

ILX0118I Short Form: OFFSET UNKNOWN

Long Form: THE OFFSET NEEDED TO ADDRESS THE ARRAY(S) <vlist> COULD NOT BE ANALYZED. THERE MAY BE AN UNKNOWN TERM IN A SUBSCRIPT OR IN A LOOP LOWER BOUND, OR THE ARRAY(S) MAY HAVE ADJUSTABLE DIMENSIONS. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when some additive term in a subscript computation for a particular array is not an induction variable or a constant. It can

also appear when the DO loop in which an array reference is contained has a variable lower bound. In these situations, recurrent dependences are presumed to exist between the statement in which the subscript computation is used and all other statements that reference the array.

Note that sometimes a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value. However, since the compiler performs vectorization analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```

C EXAMPLE 1
C OFFSET UNKNOWN - ADDITIVE TERM
    REAL*4 A(100),B(100)

    DO 10 I = 1,19
        A(I) = A(I+ISKIP) * B(I) ** 2.1
10    CONTINUE

```

Possible Response 1: Identify the expression or expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C OFFSET UNKNOWN - ADDITIVE TERM
    REAL*4 A(100),B(100)

    DO 10 I = 1,19
        A(I) = A(I+4) * B(I) ** 2.1
10    CONTINUE

```

Example 2:

```

C EXAMPLE 2
C OFFSET UNKNOWN - LOWER BOUND
    REAL*4 C(100)

    DO 20 I = ISTART,50
        C(I) = C(3) ** 2.1
20    CONTINUE

```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C OFFSET UNKNOWN - LOWER BOUND
  REAL*4 C(100)

*DIR    IGNORE RECRDEPS
        DO 20 I = ISTART,50
          C(I) = C(3) ** 2.1
        20    CONTINUE
```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the dependences really do exist, wrong results may be produced. In this case, the IGNORE RECRDEPS directive is valid only if the value of the variable ISTART is greater than 3. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0119I Short Form: STRIDE UNKNOWN
Long Form: THE STRIDE NEEDED TO ADDRESS THE ARRAY(S) <vlist> COULD NOT BE ANALYZED, EITHER BECAUSE OF AN UNKNOWN MULTIPLIER IN THE SUBSCRIPT OR AN UNKNOWN LOOP INCREMENT. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when the multiplier of some induction variable within a subscript computation for a particular array is not a constant. It can also appear when the loop in which an array reference is contained has a variable increment value. In these situations, recurrent dependences are presumed to exist between the statement in which the subscript computation is used and all other statements that reference the array.

Note that sometimes a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value. However, since the compiler performs vectorization analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C STRIDE UNKNOWN
  REAL*4 A(100),B(100)
  INTEGER*4 ISKIP

  I = 1
  DO 10 K = 1,19
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
  10    CONTINUE
```

Possible Response 1: Identify the expression or expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C STRIDE UNKNOWN
  REAL*4 A(100),B(100)
  INTEGER*4 ISKIP
  PARAMETER (ISKIP=4)

  I = 1
  DO 10 K = 1,19
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
  10    CONTINUE
```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C STRIDE UNKNOWN
  REAL*4 A(100),B(100)
  INTEGER*4 ISKIP

  I = 1
*DIR    IGNORE RECRDEPS(A)
        DO 10 K = 1,19
          A(I) = A(I) * B(K) ** 2.1
          I = I + ISKIP
        10    CONTINUE
```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the ignored dependences really do exist, wrong results may be produced. (In this case, the dependence exists only if the value of the variable ISKIP is 0.) See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0120I Short Form: POSSIBLE RECURRENCE
Long Form: THIS LOOP COULD NOT BE VECTORIZED BECAUSE OF A POSSIBLE RECURRENCE INVOLVING THE ARRAY(S) <alist>. THE INFORMATION NEEDED TO DETERMINE WHETHER OR NOT THE RECURRENCE EXISTS WAS NOT AVAILABLE TO THE COMPILER. IF NO RECURRENCE EXISTS, VECTORIZATION CAN BE ACHIEVED WITH AN IGNORE DIRECTIVE.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible to vectorize. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until compile time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization is not performed. If the IGNORE directive is used, the recurrence is assumed to be absent. However, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies cases in which the existence of a recurrence cannot be determined at compile time.

Supplemental Data:

<alist> is a list of the names of arrays that are involved in the recurrence. (This list includes only the variables that could not be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```
C EXAMPLE
C POSSIBLE RECURRENCE
  REAL  A(100),B(100)

      DO 10 I = 1,50
        A(I+N) = A(I) * B(I)
10    CONTINUE
```

In this example, a recurrence exists if an element of the array A that is stored on one iteration of the loop is referenced on some later iteration. This is true if the variable N has a value between 1 and 49.

Possible Response: If the existence of the recurrence depends on some values determined at run time, and if the value will never cause the recurrence to exist, you may use the IGNORE directive to vectorize your program.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C POSSIBLE RECURRENCE
  REAL  A(100),B(100)

*DIR  IGNORE RECRDEPS(A)
      DO 10 I = 1,50
        A(I+N) = A(I) * B(I)
10    CONTINUE
```

ILX0121I Short Form: EQUIVALENCE SCALAR USED
Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR EXPANSION BECAUSE THEY ARE IN AN EQUIVALENCE GROUP OR ARE BASED.

Explanation: Scalar variables that are in an EQUIVALENCE group and are modified in a loop cannot be vectorized, since they are not eligible for scalar expansion. Pointer variables which are modified in the loop cannot be vectorized, since they are not eligible for scalar expansion. For more information on scalar expansion, see "Scalar Expansion" on page 288.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for expansion.

Example:

```
C EXAMPLE
C EQUIVALENCED SCALAR USED
  REAL*4 X(50),Y(50),Z(50),D(50),S
  EQUIVALENCE (S,D(50))

      DO 20 I = 1,50
        S = X(I) + Y(I)
        Z(I) = S
20    CONTINUE
```

Possible Response: It is sometimes possible to avoid this situation simply by replacing references to the original scalar with references to a new scalar variable that is never referenced outside the loop. Note that if the original scalar variable is needed later, you should be careful to make sure that it is assigned the correct value after the loop has completed.

Modified Example:

```
C EXAMPLE
C EQUIVALENCED SCALAR USED
  REAL*4 X(50),Y(50),Z(50),D(50),S,S_NEW
  EQUIVALENCE (S,D(50))

      DO 20 I = 1,50
        S_NEW = X(I) + Y(I)
        Z(I) = S_NEW
20    CONTINUE
      S = S_NEW
```

ILX0122I Short Form: DEFINITE RECURRENCE
Long Form: THIS LOOP COULD NOT BE
VECTORIZED BECAUSE OF A DEFINITE
RECURRENCE INVOLVING THE
VARIABLE(S) <vlist>.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible to vectorize. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization is not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies the cases in which a recurrence definitely exists.

Supplemental Data:

<vlist> is a list of the names of the variables that are involved in the recurrence. (This list includes only those variables that could be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```
C EXAMPLE
C DEFINITE RECURRENCE
  REAL  A(100),B(100)

      DO 10 I = 1,50
        A(I+1) = A(I) * B(I)
10    CONTINUE
```

In this example, a recurrence definitely exists, since the value stored into the array A on each iteration of the loop is used on the following iteration. This loop cannot be vectorized.

ILX0123I Short Form: INTERCHANGE
PREVENTING DEP
Long Form: THE ARRAY(S) <alist>
CARRY FORWARD DEPENDENCES AT
NESTING LEVEL(S) <levlist> THAT MAY
BE INTERCHANGE PREVENTING.

Explanation: This message identifies certain dependences, known as *interchange-preventing dependences*, that restrict vectorization of outer DO loops. When an interchange-preventing dependence exists, the reordering, or interchange, of two loops would cause different results to be produced. Since vectorization of an outer loop has the same effect as moving that loop to the innermost position, the

existence of an interchange-preventing dependence carried by an outer loop prevents vectorization.

To understand how an interchange-preventing dependence comes about, study the following example:

```
      DO 10 I=1,2
        DO 10 J=1,2
10      A(I-1,J+1)=A(I,J)
```

In this code, the element A(1,2) is fetched when I=1 and J=2 and is stored into when I=2 and J=1. When these loops are processed, the fetch will occur before the store. However, if the loops were interchanged, the store would come first and different results would probably be produced.

It is not always possible for the compiler to determine whether or not a dependence is interchange preventing. Unless it can prove otherwise, the compiler will always assume that a given dependence is interchange preventing. This will insure that correct results are always produced after vectorization, even though some potential vectorization may be missed.

This message identifies dependences that were assumed to be interchange preventing because the compiler did not have sufficient information to perform a complete and accurate analysis.

Note that this message often occurs when variables are used for the lower or upper bound of a loop or when variables that are not inductions are used inside of subscripts. Sometimes, such a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value. However, since the compiler performs vectorization analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<alist> is a list of the names of the variables that carry the dependences that are presumed to be interchange preventing.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C INTERCHANGE PREVENTING DEPENDENCY
  REAL*4 U(50,50)

      DO 190 J = 1, JUPPER
        DO 190 I = 1, IUPPER
          U(I,J) = U(I+N,J) + U(I,J+N)
190    CONTINUE
```

Possible Response 1: If it is possible to write the loop bounds and subscript expressions in terms of

compile-time constants and induction variables, this may give the compiler enough information to do an accurate analysis.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C INTERCHANGE PREVENTING DEPENDENCY
  REAL*4 U(50,50)

      DO 190 J = 1, 50
      DO 190 I = 1, 50
        U(I,J) = U(I+1,J) + U(I,J+1)
190  CONTINUE
```

Possible Response 2: It is also possible to increase vectorizability by using the IGNORE RECRDEPS directive. Before using this directive, you should analyze the dependences involved to verify that they really are not interchange preventing.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C INTERCHANGE PREVENTING DEPENDENCY
  REAL*4 U(50,50)

*DIR      IGNORE RECRDEPS
      DO 190 J = 1, JUPPER
      DO 190 I = 1, IUPPER
        U(I,J) = U(I+N,J) + U(I,J+N)
190  CONTINUE
```

ILX0124I Short Form: OPTIMIZER INDUCED DEPENDENCE
Long Form: A COMPILER TEMPORARY INTRODUCED DURING SCALAR OPTIMIZATION HAS CAUSED ONE OR MORE DEPENDENCES IN THE LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message is produced when a statement becomes part of a recurrence solely because of some optimization that had been performed prior to vectorization (for example, common sub-expression elimination).

Supplemental Data:

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(100),D(100),F(100)

      DO 100 J = 2,100
      C(J) = D(J-1)
      D(J) = C(J)
      F(J) = D(J) + 2
100  CONTINUE
```

In the DO loop shown above, the first two statements form a recurrence and cannot be vectorized while the last statement would normally be vectorizable. However, vectorization of this statement may be restricted due to some transformations that have been applied by the compiler prior to vectorization analysis.

In optimizing this loop, the compiler will attempt to reduce the number of load instructions that must be processed. As a result, during vectorization analysis, it will appear as if this loop were rewritten as:

```
      DO 100 J = 2,100
      .temp = D(J-1)
      C(J) = .temp
      D(J) = .temp
      F(J) = .temp + 2
100  CONTINUE
```

where .temp is a compiler generated scalar temporary. In order to vectorize the last statement, it would be necessary to split the original loop into two loops so that this statement is separated from the other, nonvectorizable, statements. The presence of the scalar temporary shared by all the statements in the loop prohibits loop splitting and thus prevents partial vectorization.

Possible Response 1: Since the presence of statement labels inhibits optimization to some degree, it is sometimes possible to achieve partial vectorization in cases such as this simply by introducing additional labels.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(100),D(100),F(100)

      DO 100 J = 2,100
      C(J) = D(J-1)
      D(J) = C(J)
      99  F(J) = D(J) + 2
100  CONTINUE
```

Be careful when using this type of transformation since it may inhibit some important optimizations. If you make this change and partial vectorization still does not occur, the resulting scalar code may run more slowly than the original.

Possible Response 2: The transformation suggested above may or may not increase vectorization. If it is not effective, you should try to replace the original loop with

two separate loops, where one loop contains the nonvectorizable portion while the other loop contains the vectorizable portion of the original loop.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(100),D(100),F(100)

      DO 100 J = 2,100
        C(J) = D(J-1)
        D(J) = C(J)
100    CONTINUE
      DO 101 J = 2,100
        F(J) = D(J) + 2
101    CONTINUE
```

This should be done only when you are absolutely certain that the transformation will not alter the results produced by your program.

ILX0125I Short Form: SUBSCRIPT TOO COMPLEX
Long Form: THE ARRAY(S) <alist> USE SUBSCRIPT COMPUTATIONS THAT COULD NOT BE ANALYZED. THEY MAY INCLUDE INDIRECT ADDRESSING, DATA CONVERSIONS, UNKNOWN STRIDES, OR AUXILIARY INDUCTION VARIABLES. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: Indicates the use of subscript computations for which the compiler could not perform accurate dependence analysis. These cases include the constructs listed below. (In each of the examples given, K is an induction variable for some DO loop that contains the indicated array reference.)

- Indirect addressing, as in $A(\text{INDEX}(K))$, where INDEX is an array of integers.
- Subscripts requiring data conversions, as in $A(K+X)$, where X is a real variable.
- Subscripts where the stride is not known at compile time, as in $A(K*KSTEP)$, where KSTEP is an integer variable. (An unknown stride may also occur if the increment expression of some DO loop is not a compile-time constant.)
- Auxiliary induction variables, as in $A(\text{IVAR})$, where IVAR is incremented explicitly within the DO loop by some statement of the form $\text{IVAR}=\text{IVAR}+\text{INCR}$.

When an array uses any of the above types of subscript, recurrent dependences are often presumed to exist between the statement in which the subscript computation occurs and all other statements that reference the array.

This message often occurs when variables are used for the lower bound, upper bound, or increment of a loop or when variables that are not inductions are used inside of subscripts. Sometimes, such a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value. However, since the compiler performs vectorization analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<alist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```
C EXAMPLE 1
C SUBSCRIPT TOO COMPLEX
  REAL*4 A(20)
  INTEGER*4 INDEX(20)

      DO 10 I = 1,20
        A(INDEX(I)) = A(INDEX(I)) ** 2.1
10    CONTINUE
```

Possible Response 1: For cases involving indirect addressing, it may be possible to introduce additional arrays to hold intermediate results. The loop should be transformed as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

This should be done only if it is absolutely certain that the noninductive subscript expression never selects any element more than once.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C SUBSCRIPT TOO COMPLEX
  REAL*4 A(20),NEW_A(20)
  INTEGER*4 INDEX(20)

      DO 9 I = 1,20
        NEW_A(I) = A(INDEX(I))
9      CONTINUE
      DO 10 I = 1,20
        NEW_A(I) = NEW_A(I) ** 2.1
10     CONTINUE
      DO 11 I = 1,20
        A(INDEX(I)) = NEW_A(I)
11     CONTINUE

```

Due to the overhead involved in copying data to and from the new version of the array, this transformation may not result in any performance benefits, even if vectorization is achieved. You should carefully analyze the performance of this code before and after the transformation is applied to make sure that it is worthwhile.

Example 2:

```

C EXAMPLE 2
C SUBSCRIPT TOO COMPLEX
  REAL*4 B(20),X

      DO 20 I = 1,20
        B(I+X) = B(I+X) ** 2.1
20     CONTINUE

```

Possible Response 2: For cases involving data conversions inside of subscript calculations, recode the computations so that all the inputs hold INTEGER values, whenever possible.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C SUBSCRIPT TOO COMPLEX
  REAL*4 B(20),X
  INTEGER*4 INT_X

      INT_X = X
      DO 20 I = 1,20
        B(I+INT_X) = B(I+INT_X) ** 2.1
20     CONTINUE

```

Example 3:

```

C EXAMPLE 3
C SUBSCRIPT TOO COMPLEX
  REAL*4 C(20),Y
  INTEGER*4 KSTEP

      DO 30 I = 1,20
        KSTEP = KSTEP + INC
        C(KSTEP*I) = Y ** C(I*KSTEP)
30     CONTINUE

```

Possible Response 3: For cases involving unknown strides, identify the expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 3:

```

C POSSIBLE RESPONSE 3
C SUBSCRIPT TOO COMPLEX
  REAL*4 C(20),Y
  INTEGER*4 KSTEP
  PARAMETER (KSTEP=4)

      DO 30 I = 1,20
        C(KSTEP*I) = Y ** C(I*KSTEP)
30     CONTINUE

```

Example 4:

```

C EXAMPLE 4
C SUBSCRIPT TOO COMPLEX
  REAL*4 D(400)

      DO 40 J = 1,20
        D(J) = D(J*J) ** 2.1
40     CONTINUE

```

Possible Response 4: For cases in which none of the previous transformations is appropriate, an IGNORE RECRDEPS directive may be used to cause the compiler to assume that no dependence exists.

Modified Example 4:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 4
C SUBSCRIPT TOO COMPLEX
  REAL*4 D(400)

*DIR      IGNORE RECRDEPS(D)
      DO 40 J = 1,20
        D(J) = D(J*J) ** 2.1
40     CONTINUE

```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the ignored dependences really do exist, wrong results may be produced. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0126I Short Form: SCALAR DEFINED BEFORE LOOP

Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR EXPANSION BECAUSE THEY MAY USE VALUES THAT WERE SET BEFORE THE EXECUTION OF THE CONTAINING LOOP AND THEIR VALUES MAY BE USED AFTER THE EXECUTION OF THE CONTAINING LOOP.

Explanation: Vectorization of scalar variables that are modified within a loop requires a process known as scalar expansion. This involves replacing the scalar with a vector register. This may not be possible in certain cases.

The cases where scalar expansion is not done are where a scalar variable is *nonlocal* to a loop, and where that variable is referenced at different nesting levels or

where it is modified by a conditionally executed statement. A variable is considered *nonlocal* to a loop if it is in COMMON, if it uses a value within the loop that will be used after the loop. For more information on scalar expansion, see “Scalar Expansion” on page 288.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for expansion.

Example:

```
C EXAMPLE
C SCALAR DEFINED BEFORE LOOP
      REAL*4 A(20,20),B(20),C(20),T

      T = 1.1
      DO 11 I = 1,20
        DO 10 J = 1,20
          A(J,I) = T
10      CONTINUE
          T = B(I) + C(I)
11      CONTINUE
```

Possible Response: It may be possible to replace the original scalar variable with an array whose dimension ranges from zero to the number of iterations of the loop in which the scalar resides. The loop should be transformed in the following manner:

- Prior to entering the loop, set the zero element of the new array to the value held by the scalar.
- Prior to the first statement that defines the scalar within the loop, replace all references to that scalar with references to the element of the new array whose position is one less than the current iteration count.
- All other references to the scalar within the loop should be replaced by references to the element of the array that corresponds to the current iteration count.
- Following the loop, set the scalar to the value held by the last element of the new array.

Modified Example:

```
C POSSIBLE RESPONSE
C SCALAR DEFINED BEFORE LOOP
      REAL*4 A(20,20),B(20),C(20),T
      REAL*4 TT(0:20)

      T = 1.1
      TT(0) = T
      DO 11 I = 1,20
        DO 10 J = 1,20
          A(J,I) = TT(I-1)
10      CONTINUE
          TT(I) = B(I) + C(I)
11      CONTINUE
      T = TT(20)
```

Note that this transformation is only valid if the first assignment to the original scalar variable is not a conditionally processed statement.

Also be aware that this transformation may not necessarily increase the the vectorizability of your program and could result in a scalar program that runs more slowly than the original.

ILX0127I Short Form: SCALAR NEEDED AFTER LOOP

Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR EXPANSION BECAUSE THEY MAY BE SET TO VALUES THAT WILL BE USED AFTER THE EXECUTION OF THE CONTAINING LOOP.

Explanation: Vectorization of scalar variables that are modified within a loop requires a process known as scalar expansion. This involves replacing the scalar with a vector register. This may not be possible in certain cases.

The cases where scalar expansion is not done are where a scalar variable is *nonlocal* to a loop, and where that variable is referenced at different nesting levels or where it is modified by a conditionally executed statement. A variable is considered *nonlocal* to a loop if it is in COMMON or if it uses a value within the loop that will be used after the loop. For more information about scalar expansion, see “Scalar Expansion” on page 288.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for expansion.

Example:

```
C EXAMPLE
C SCALAR NEEDED AFTER LOOP
      REAL*4 A(20),B(20,20),T

      DO 22 I = 1,20
        T = A(I)
        DO 22 J = 1,20
          B(J,I) = T
22      CONTINUE
      ...
      WRITE(6,*) T
```

Possible Response 1: It may be possible to replace the original scalar variable with a new scalar variable that is local to the loop. If this is done, it will be necessary to insert an additional assignment after the loop to set the original scalar to its appropriate final value.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(20),B(20,20),T
  REAL*4 T_LOCAL

  DO 22 I = 1,20
    T_LOCAL = A(I)
    DO 22 J = 1,20
      B(J,I) = T_LOCAL
22  CONTINUE
  T = A(20)
  ...
  WRITE(6,*) T

```

Possible Response 2: It may be possible to replace the original scalar variable with an array whose dimension ranges from one to the number of iterations of the loop in which the scalar resides. The loop should be transformed in the following manner:

- If the first occurrence of the scalar within the loop is a reference rather than definition, prior to entering the loop, set the first element of the array to the value held by the scalar.
- Replace all occurrences of the scalar within the loop with references to the element of the array that corresponds to the current iteration count.
- Following the loop, set the scalar to the value held by the last element of the array.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(20),B(20,20),T
  REAL*4 TT(20)

  DO 22 I = 1,20
    TT(I) = A(I)
    DO 22 J = 1,20
      B(J,I) = TT(I)
22  CONTINUE
  T = T0T0
  ...
  WRITE(6,*) T

```

Note that this transformation is only valid if the first assignment to the original scalar variable is not a conditionally processed statement.

Also be aware that this transformation may not necessarily increase the the vectorizability of your program and could result in a scalar program that runs more slowly than the original.

ILX0128I Short Form: NESTED SINGLE TRIP LOOP

Long Form: SOME NESTED LOOP CONSISTS OF A SINGLE ITERATION. VECTORIZATION OF THIS LOOP IS INHIBITED.

Explanation: When the upper and lower bound expressions for a particular loop are identical constant expressions, vectorization of all outer loops is restricted.

Note: If two or more copies of a particular DO statement are printed in the compiler report, this message may appear with each copy, even though it may not be applicable in all cases.

Example:

```

C EXAMPLE
C NESTED SINGLE TRIP LOOP
  REAL*4 A(100,100)

  DO 11 I = 1,20
    DO 11 J = 2*3+5,2*3+5
      A(I,J) = A(I,J) ** 2.1
11  CONTINUE

```

Possible Response: Identify the loop causing the rejection and replace the DO statement with an assignment statement that sets the loop variable to the value that was being used as the lower bound.

Modified Example:

```

C POSSIBLE RESPONSE
C NESTED SINGLE TRIP LOOP
  REAL*4 A(100,100)

  DO 11 I = 1,20
    J = (2*3+5)
    A(I,J) = A(I,J) ** 2.1
11  CONTINUE
    J=J+1

```

(Note that an extra statement has been added after the loop to insure that the variable J is set to the correct value in case it is referenced later.)

ILX0129I Short Form: NESTED NONCONSTANT INDUCTION

Long Form: THE LOOP VARIABLE OF THIS LOOP OR OF SOME NESTED LOOP AFFECTS THE LOOP VARIABLE OR AN AUXILIARY INDUCTION VARIABLE USED BY SOME OTHER NESTED LOOP.

Explanation: When the DO loop parameters of an inner loop are modified by an outer loop, or when the initialization or iteration of an auxiliary induction variable is modified by an outer loop, the outer loop is not eligible for vectorization.

The reason for this restriction is that in these cases, the behavior of the inner loop depends on the value of the induction variable of the outer loop. If the outer loop

were vectorized, it would be replaced by a sectioning loop. The induction variable of the sectioning loop would take on a different set of values than those of the original loop, and therefore, the inner loop would behave differently (and would probably produce different results.)

The same situation can arise with auxiliary induction variables. An auxiliary induction variable can be either a user variable that is explicitly incremented within a loop or an internal compiler temporary that has been generated to hold certain subscript computations. When a compiler temporary is used it may be difficult to pinpoint the statement or statements for which it was generated. Usually, however, they are associated with particularly complex subscript expressions (for example, subscript computations where two loop variables are multiplied together).

Note: If two or more copies of a particular DO statement are printed in the compiler report, this message may appear with each copy, even though it may not be applicable in all cases.

Example 1:

```
C EXAMPLE 1
C NESTED NON-CONSTANT INDUCTION
  REAL*4 A(128,128)

      DO 10 I = 1,128
        DO 10 J = I,128
          A(I,J) = A(I,J) * 2.1
10      CONTINUE
```

Possible Response 1: For cases where the induction variable of an outer loop is used as part of the initial or final calculations for an inner DO loop, it may be possible to rewrite the inner loop so that the range of that loop is independent of the outer loop. This can sometimes be done by introducing IF statements inside the loop to exclude the iterations which would not have been processed in the original code.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C NESTED NON-CONSTANT INDUCTION
  REAL*4 A(128,128)

      DO 10 I = 1,128
        DO 10 J = 1,128
          IF (J.LT.I) GOTO 11
          A(I,J) = A(I,J) * 2.1
10      CONTINUE
```

Be careful when using this transformation. Even though it may increase vectorization, the additional overhead of the conditional code may result in increased run time.

Example 2:

```
C EXAMPLE 2
C NESTED NON-CONSTANT INDUCTION
  REAL*4 B(500,10),C(5000)

      DO 20 I = 1,500
        DO 20 J = 1,10
          B(I,J) = C(I*J) * 2.1
20      CONTINUE
```

Possible Response 2: For cases where this message is generated because of nonlinear expressions inside of subscripts, if vectorization of an outer loop is desired, it may be possible to switch the loops so that the outer loop is moved to the innermost position.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C NESTED NON-CONSTANT INDUCTION
  REAL*4 B(500,10),C(5000)

      DO 20 J = 1,10
        DO 20 I = 1,500
          B(I,J) = C(I*J) * 2.1
20      CONTINUE
```

Be careful when using this transformation since switching loops might change the results produced by the loops. You must be certain that this is not the case before you make this type of change.

ILX0130I Short Form: UNKNOWN UPPER BOUND
Long Form: THE ARRAY(S) <alist> MAY OR MAY NOT BE INVOLVED IN DEPENDENCES, DEPENDING ON THE UPPER BOUND OF SOME CONTAINING LOOP. SINCE THE UPPER BOUND IS NOT KNOWN, THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCE(S) IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when a variable is specified as the upper bound of a loop and when the existence of a dependence depends on the value of the upper bound. In these cases, dependence will always be assumed.

Note that sometimes a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value. However, since the compiler performs vectorization analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<alist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear

on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C UNKNOWN UPPER BOUND
    REAL*4 A(-20:20)

    DO 10 I = 1,N
      A(I) = A(I-20) * 22.1
10    CONTINUE
```

Possible Response 1: Identify the loop that carries the dependence and replace the upper bound of the loop with a compile-time constant, if possible.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C UNKNOWN UPPER BOUND
    REAL*4 A(-20:20)
    PARAMETER (N=20)

    DO 10 I = 1,N
      A(I) = A(I-20) * 22.1
10    CONTINUE
```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C UNKNOWN UPPER BOUND
    REAL*4 A(-20:20)

*DIR    IGNORE RECRDEPS
    DO 10 I = 1,N
      A(I) = A(I-20) * 22.1
10    CONTINUE
```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the dependences really do exist, wrong results may be produced. In this case, the directive is correct only if the value of the variable N is always less than 21. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0131I Short Form: STOP OR RETURN WITH OPT(2)
Long Form: STOP OR RETURN STATEMENT(S) AT ISN(S) <ilist> WERE INELIGIBLE FOR VECTORIZATION BECAUSE THE OPTION "OPTIMIZE(2)" WAS SPECIFIED.

Explanation: Indicates that a loop was rejected

because it contains a STOP or RETURN statement and optimization level 2 was requested. Optimization level 3 is required to vectorize such loops. This is because, in order to vectorize, it may be necessary to evaluate some instances of the loop termination test that would not be evaluated with scalar code. In certain cases, this may lead to interrupts that would not otherwise occur.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement(s) responsible for the rejection.

Example:

```
@PROCESS OPT(2) VECTOR
C EXAMPLE
C STOP OR RETURN WITH OPT(2)
    REAL A(50),B(50),C(50)

    DO 500 I = 1,50
      IF (A(I)/B(I) .EQ. 1.0) STOP
      C(I) = A(I) * B(I)
500    CONTINUE
```

In this case, a divide exception would be raised if B(I) is ever zero. If this is true for some value of I greater than the value at which the loop terminates, then the exception will not occur in the scalar loop, but may occur in the vector loop. This may happen, for example, if A(10) equals B(10) and B(11) equals zero.

Possible Response: If it is known that no exception will occur, optimization level 3 may be used.

Otherwise, you may be able to restructure the code to execute the loop termination test in scalar, and the rest of the loop in vector. This makes the loop eligible for partial vectorization while avoiding the possibility of raising an exception. This can be done with the following transformation:

1. Insert a new loop to evaluate the loop termination condition and determine the number of times each loop iterates before terminating. The code for the new loop will remain ineligible for vectorization.
2. Rewrite the original loop to process only the elements that are referenced before the loop terminates.
3. Add a test at the end of the modified code to determine whether the STOP or RETURN statement should be executed.

Modified Example:

```

@PROCESS OPT(2) VECTOR
C POSSIBLE RESPONSE
C STOP OR RETURN WITH OPT(2)
  REAL  A(50),B(50),C(50)
  INTEGER IMAX
  LOGICAL NEED_STOP

  DO 10 IMAX = 1,50
    IF (A(IMAX)/B(IMAX) .EQ. 1.0) GOTO 100
10  CONTINUE
100 NEED_STOP = (IMAX.LE.50)

  DO 20 I = 1,IMAX - 1
    C(I) = A(I)/B(I)
20  CONTINUE

  IF (NEED_STOP) THEN STOP

```

Note that this transformation is valid only if the test that determines whether the loop should terminate is independent of the values that are computed in the loop. Also, if the test is not the first statement in the loop, some special processing is necessary.

ILX0132I Short Form: EXIT BRANCH WITH OPT(2)
Long Form: EXIT BRANCH(ES) AT
ISN(S) <ilist> WERE INELIGIBLE FOR
VECTORIZATION BECAUSE THE OPTION
"OPTIMIZE(2)" WAS SPECIFIED.

Explanation: Indicates that a loop was rejected because it contains an exit branch and optimization level 2 was requested. Optimization level 3 is required to vectorize such loops. This is because, in order to vectorize, it may be necessary to evaluate some instances of the loop termination test that would not be evaluated with scalar code. In certain cases, this may lead to interrupts that would not otherwise occur.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the exit branches.

Example:

```

@PROCESS OPT(2) VECTOR
C EXAMPLE
C EXIT BRANCH WITH OPT(2)
  REAL  A(50),B(50),C(50)

  DO 10 I = 1,50
    IF (A(I)/B(I) .EQ. 1.0) GOTO 5
    C(I) = A(I)/B(I)
10  CONTINUE
  ...
5  CONTINUE

```

In this case, a divide exception would be raised if B(I) is ever zero. If this is true for some value of I greater than the value at which the loop terminates, then the exception will not occur in the scalar loop, but may occur in the vector loop. This may happen, for example, if A(10) equals B(10) and B(11) equals zero.

Possible Response: If it is known that no exception will occur, optimization level 3 may be used.

Otherwise, you may be able to restructure the code to execute the exit branch in scalar, and the rest of the loop in vector. This makes the loop eligible for partial vectorization while avoiding the possibility of raising an exception. This can be done with the following transformation:

1. Insert a new loop to evaluate the exit branch condition and determine the number of times each loop iterates before terminating. The code for the new loop will remain ineligible for vectorization.
2. Rewrite the original loop to process only the elements that are referenced before the exit branch is taken.
3. Add a test at the end of the modified code to determine whether the exit branch is taken.

Modified Example:

```

@PROCESS OPT(2) VECTOR
C POSSIBLE RESPONSE
C EXIT BRANCH WITH OPT(2)
  REAL  A(50),B(50),C(50)
  INTEGER IMAX
  LOGICAL NEED_BRANCH

  DO 10 IMAX = 1,50
    IF (A(IMAX)/B(IMAX) .EQ. 1.0) GOTO 100
10  CONTINUE
100 NEED_BRANCH = (IMAX.LE.50)

  DO 20 I = 1,IMAX - 1
    C(I) = A(I)/B(I)
20  CONTINUE

  IF (NEED_BRANCH) THEN GOTO 5

  ...

5  CONTINUE

```

Note that this transformation is valid only if the test that determines whether the loop should terminate is independent of the values that are computed in the loop. Also, if the test is not the first statement in the loop, some special processing is necessary.

ILX0133I Short Form: IMPLICIT LIBRARY
FUNCTION
Long Form: THE IMPLICITLY CALLED
MATHEMATICAL SUBPROGRAM(S)
<flist> HAVE BEEN USED. THESE
SUBPROGRAMS ARE NOT SUPPORTED
FOR VECTOR.

Explanation: Indicates that a statement requires the use of an implicitly called mathematical library subprogram that use REAL*16 or COMPLEX*32 arguments. These subprograms are outlined in *VS FORTRAN Version 2 Language and Library Reference*.

Supplemental Data:

<flist> is a list consisting of names of the entry points for each implicit subprogram that has been used.

Example:

```
C EXAMPLE
C IMPLICIT LIBRARY FUNCTION
  REAL*8 B(128),C(128)

  DO 19 I = 1,128
    B(I) = QEXTD(C(I)) ** 2.1
19  CONTINUE
```

ILX0134I Short Form: LOGICAL*1 DATA
Long Form: THE LOGICAL*1
VARIABLE(S) <vlist> CANNOT BE
VECTORIZED.

Explanation: Indicates the presence of LOGICAL*1 data.

Supplemental Data:

<vlist> is a list of the names of the variables with the unsupported data type.

Possible Response: Replace the indicated variables with LOGICAL*4 data whenever possible.

ILX0135I Short Form: UNSUPPORTABLE
DEPENDENCE
Long Form: THIS CODE IS
CONSIDERED UNSUPPORTABLE
BECAUSE IT IS LINKED TO SOME
UNSUPPORTABLE STATEMENT(S)
THROUGH MUTUAL DEPENDENCES.

Explanation: This message is produced when a statement does not contain any unsupported constructs but is forced into an unsupported loop because it is tied to some other unsupported statement through recurrent dependences. These dependences can come about in a number of ways:

- The indicated statement may use a scalar variable that is also used in some other statement that contains an unsupported construct.
- There may be some control flow that creates a control dependence that involves both the indicated statement and some unsupported statement.
- There may be a dependence between the indicated statement and some unsupported statement that came about because the two statements share some common subexpression.

Example:

```
C EXAMPLE
C UNSUPPORTABLE DEPENDENCE
  REAL*8 A(512),B(512),C(512)
  REAL*16 D(512)

  DO 10 I = 1,512
    A(I) = B(I)/C(I)
    D(I) = B(I)/C(I)
10  CONTINUE
```

In this example, the second statement uses the REAL*16 array, D, and therefore is not eligible for vectorization, while the first statement would normally be vectorizable. However, the two statements share a common sub-expression, and during vectorization analysis, this loop would appear to the compiler as if it were written:

```
  DO 10 I = 1,512
    .temp = B(I)/C(I)
    A(I) = .temp
    D(I) = .temp
10  CONTINUE
```

where .temp is a scalar temporary generated by the compiler. In order to vectorize the first statement, it would be necessary to split this loop into two separate loops. However, the presence of the scalar temporary prevents the compiler from doing this.

Possible Response: If it is possible to replace the unsupported part of the loop with equivalent supportable constructs, do so.

Otherwise, try to separate the vectorizable statements which are linked to unsupported statements by restructuring the original loop into two or more loops.

Modified Example:

```
C POSSIBLE RESPONSE
C UNSUPPORTABLE DEPENDENCE
  REAL*8 A(512),B(512),C(512)
  REAL*16 D(512)

  DO 10 I = 1,512
    A(I) = B(I)/C(I)
10  CONTINUE
  DO 11 I = 1,512
    D(I) = B(I)/C(I)
11  CONTINUE
```

This type of restructuring should only be done if you are absolutely certain that it will not alter the results produced by your program.

ILX0136I Short Form: CONDITIONAL INTEGER*2 DATA
Long Form: THE USE OF INTEGER*2 OR LOGICAL*2 VARIABLE(S) <vlist> IN CONDITIONALLY EXECUTED CODE CANNOT BE VECTORIZED.

Explanation: Indicates the presence of INTEGER*2 or LOGICAL*2 data in conditionally processed code.

Note: Some statements that affect the flow of control within a loop may not be reproduced in the compiler report listing produced by the VECTOR(REPORT(XLIST)) option. It may be necessary to refer to the source listing or to the output produced by the VECTOR(REPORT(SLIST)) option to determine the correct control flow.

Supplemental Data:

<vlist> is a list of the names of the variables with the unsupported data type

Example:

```
C EXAMPLE
C CONDITIONAL INTEGER*2 DATA
      INTEGER*4 A(512)
      INTEGER*2 B(512)

      DO 9 I = 1,512
        IF (A(I) .LT. 128) B(I) = B(I) ** 2.1
9      CONTINUE
```

In this case, the array B cannot be vectorized because it is an INTEGER*2 variable that is referenced under the control of an IF statement.

Possible Response: Replace the indicated variables with INTEGER*4 data whenever possible.

ILX0137I Short Form: EXTENDED PRECISION DATA
Long Form: THE EXTENDED PRECISION VARIABLE(S) <vlist> CANNOT BE VECTORIZED.

Explanation: Indicates the presence of REAL*16 and COMPLEX*32 data.

Note: Extended precision data may occur as the result of the specification of the AUTODBL option.

Supplemental Data:

<vlist> is a list of the names of the variables with the unsupported data type

Possible Response: Replace the indicated variables with REAL*8 or COMPLEX*16 data whenever possible.

ILX0138I Short Form: CONDITIONAL NONINDUCTIVE SUB
Long Form: THE ARRAY(S) <alist> ARE USED IN CONDITIONALLY EXECUTED CODE AND HAVE NON-INDUCTIVE SUBSCRIPT EXPRESSIONS.

Explanation: Indicates the use of noninductive subscript expressions that occur in conditionally processed code. A noninductive expression is any expression that is not a linear function of some loop induction variable, for example, indirect addressing, as in A(INDEX(K)), or diagonal traversal of an array, as in DIAG(K,K).

Note: Some statements that affect the flow of control within a loop may not be reproduced in the compiler report listing produced by the VECTOR(REPORT(XLIST)) option. It may be necessary to refer to the source listing or to the output produced by the VECTOR(REPORT(SLIST)) option to determine the correct control flow.

Supplemental Data:

<alist> is a list of the names of the arrays that use noninductive subscripts.

Example 1:

```
C EXAMPLE 1
C CONDITIONAL NON-INDUCTIVE SUB
      REAL*4 B(200),C(200)
      INTEGER*4 INDEX(200)

      DO 10 I = 2,128
        IF (B(I) .GT. 500) C(INDEX(I)) = 0.0
10     CONTINUE
```

Possible Response 1: For cases involving indirect addressing, it may be possible to introduce additional arrays to hold intermediate results. The loop should be transformed as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 B(200),C(200),NEW_C(200)
  INTEGER*4 INDEX(200)

  DO 9 I = 2,128
    NEW_C(I) = C(INDEX(I))
9    CONTINUE
  DO 10 I = 2,128
    IF (B(I) .GT. 500) NEW_C(I) = 0.0
10   CONTINUE
  DO 11 I = 2,128
    C(INDEX(I)) = NEW_C(I)
11   CONTINUE

```

Note that this should only be done if it is absolutely certain that the noninductive subscript expression never selects any element more than once. You should also be aware that due to the overhead involved in copying data, it is possible that no performance benefits will be achieved, even if the transformation does result in increased vectorization.

Example 2:

```

C EXAMPLE 2
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 X(50,50),Y(50)

  DO 20 I = 1,50
    IF (Y(I) .LT. 0.0) X(I,I) = X(I,I) ** 2.1
20   CONTINUE

```

Possible Response 2: For cases of diagonal access to an array, it may also be possible to EQUIVALENCE the original array to a one-dimensional array where the elements that were part of a diagonal in the original are now within a single dimension and are separated by a fixed number of elements. These elements can now be referenced through inductive subscript expressions.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 X(50,50),Y(50),NEW_X(2500)
  EQUIVALENCE (NEW_X(1),X(1,1))

  J = 1
  DO 20 I = 1,50
    IF (Y(I) .LT. 0.0) NEW_X(J) = NEW_X(J) ** 2.1
    J = J + 51
20   CONTINUE

```

ILX0139I Short Form: RESTRICTED NONINDUCTIVE SUB

Long Form: THE ARRAY(S) <alist>
HAVE INTEGER*1, INTEGER*2,
LOGICAL*1, or LOGICAL*2 DATA
TYPES AND HAVE NON-INDUCTIVE
SUBSCRIPT EXPRESSIONS.

Explanation: Indicates the use of noninductive subscript expressions that occur in arrays with data

types of INTEGER*1, INTEGER*2, LOGICAL*1., and LOGICAL*2. A noninductive expression is any expression that is not a linear function of some loop induction variable, for example, indirect addressing, as in A(INDEX(K)), or diagonal traversal of an array, as in DIAG(K,K).

Supplemental Data:

<alist> is a list of the names of the arrays that use noninductive subscripts.

Example 1:

```

C EXAMPLE 1
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 C(128)
  INTEGER*4 INDEX(128)

  DO 10 I = 2,128
    C(INDEX(I)) = 0
10   CONTINUE

```

Possible Response 1: If it is possible to replace the original arrays with arrays that are INTEGER*4 or LOGICAL*4, do so.

Otherwise, for cases involving indirect addressing, it may be possible to introduce additional arrays to hold intermediate results. The loop should be transformed as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 C(128),NEW_C(128)
  INTEGER*4 INDEX(128)

  DO 9 I = 2,128
    NEW_C(I) = C(INDEX(I))
9    CONTINUE
  DO 10 I = 2,128
    NEW_C(I) = 0
10   CONTINUE
  DO 11 I = 2,128
    C(INDEX(I)) = NEW_C(I)
11   CONTINUE

```

This should only be done if it is absolutely certain that the noninductive subscript expression never selects any element more than once. You should also be aware that due to the overhead involved in copying data, it is possible that no performance benefits will be achieved, even if the transformation does result in increased vectorization.

Example 2:

```
C EXAMPLE 2
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 A(128,128)

      DO 20 I = 2,128
        A(I,I) = A(I,I) * 2.1
20    CONTINUE
```

Possible Response 2: For cases of diagonal access to an array, it may also be possible to EQUIVALENCE the original array to a one-dimensional array where the elements that were part of a diagonal in the original are now within a single dimension and are separated by a fixed number of elements. These elements can be referenced through inductive subscript expressions.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 A(128,128),NEW_A(128*128)
  EQUIVALENCE (A(1,1),NEW_A(1))

      J = 1
      DO 20 I = 2,128
        NEW_A(J) = NEW_A(J) * 2.1
        J = J + 129
20    CONTINUE
```

ILX0140I Short Form: IN-LINE INTRINSIC FUNCTION
Long Form: NO VECTOR SUPPORT EXISTS FOR THE IN-LINE INTRINSIC FUNCTION(S) <flist>.

Explanation: Indicates the presence of selected intrinsic functions that cannot be vectorized. This set of functions includes the following:

```
DIM MOD SIGN NINT ANINT
```

Supplemental Data:

<flist> is a list of the names of the functions involved.

ILX0141I Short Form: I/O OPERATIONS
Long Form: I/O OPERATIONS ARE NOT SUPPORTED FOR VECTOR.

Explanation: Indicates the presence of I/O statements in a loop. The loop may have been analyzed and found eligible for partial vectorization; however, the I/O operation remains an unsupported operation for vectorization.

Example:

```
C EXAMPLE
C I/O OPERATIONS
  REAL*4 A(100),B(100)

      DO 10 I = 1,100
        A(I) = B(I) * 3.3
        PRINT*, A(I)
10    CONTINUE
```

Analysis Output:

```
C OUTPUT FROM VECTOR ANALYSIS
C I/O OPERATIONS
                                REAL*4  A(100),B(100)

VECT  +----- DO 10 I = 1,100
      |_____ A(I) = B(I) * 3.3

UNSP  +----- DO 10 I = 1,100
      |_____ PRINT*, A(I)
```

ILX0142I Short Form: RELATIONAL EXPRESSION
Long Form: RELATIONAL EXPRESSIONS ARE NOT ELIGIBLE FOR VECTORIZATION

Explanation: Indicates the presence of relational expressions in an assignment statement. Indicates the presence of an IF statement, the logical expression of which contains .EQV., .NEQV., or a statement function.

Example:

```
C EXAMPLE
C RELATIONAL EXPRESSION
  LOGICAL*4 L(128)
  INTEGER*4 B(128),C(128)

      DO 10 I = 1,128
        L(I) = (B(I) .LT. C(I))
10    CONTINUE
```

Possible Response: If the relational expression is being used in an assignment statement to define some logical variable, replace the assignment with a logical IF statement that tests the expression and sets the variable to .TRUE. if the test succeeds and to .FALSE. if it fails.

Modified Example:

```
C POSSIBLE RESPONSE
C RELATIONAL EXPRESSION
  LOGICAL*4 L(128)
  INTEGER*4 B(128),C(128)

      DO 10 I = 1,128
        IF (B(I) .LT. C(I)) THEN
          L(I) = .TRUE.
        ELSE
          L(I) = .FALSE.
        ENDIF
10    CONTINUE
```

ILX0143I Short Form: NOINTRINSIC OPTION IN EFFECT
Long Form: THE INTRINSIC FUNCTION(S) <flist> HAVE BEEN USED. THE NOINTRINSIC OPTION INHIBITS VECTORIZATION OF THIS CODE.

Explanation: Indicates the presence of intrinsic functions that cannot be vectorized because the VECTOR(NOINTRINSIC) option has been specified.

Supplemental Data:

<flist> is a list of the names of the functions involved.

Possible Response: Specify the VECTOR(INTRINSIC) option on an @PROCESS card or when invoking the compiler.

Note: If you are using the VS FORTRAN Version 1 Library, the VECTOR(INTRINSIC) option may cause your program to produce different answers after vectorization. This is because there are no vector functions in the VS FORTRAN Version 1 Library. When intrinsic functions are vectorized, subprograms from the Version 2 Library will always be invoked, although the equivalent Version 1 subprograms would be used when running the program in scalar mode. Since the Version 2 Library produces more accurate results than does the Version 1 Library, vector and scalar processing of the same program may give slightly different answers in this situation.

ILX0144I Short Form: MISALIGNED DATA
Long Form: THE VARIABLE(S) <vlist> HAVE STORAGE ALIGNMENTS THAT CONFLICT WITH THEIR DATA TYPES.

Explanation: Indicates the usage of conflicting storage alignments. References to arrays containing misaligned data should not be vectorized since they would produce alignment exceptions when the program is run.

Supplemental Data:

<vlist> is a list of the names of the variables with the conflicting alignments.

Example:

```
C EXAMPLE
C MISALIGNED DATA
      REAL*4 A(128),B(128)
      INTEGER*2 DUMMY
      COMMON // A,DUMMY,B

      DO 10 I = 1,128
        A(I) = B(I) ** 2.1
10     CONTINUE
```

Possible Response: Modify the declarations so as to assure proper alignment whenever possible.

Modified Example:

```
C POSSIBLE RESPONSE
C MISALIGNED DATA
      REAL*4 A(128),B(128)
      INTEGER*2 DUMMY
      COMMON // A,B,DUMMY

      DO 10 I = 1,128
        A(I) = B(I) ** 2.1
10     CONTINUE
```

ILX0146I Short Form: UNSUPPORTED CONSTRUCT
Long Form: NO VECTOR SUPPORT EXISTS FOR THIS OCCURRENCE OF THE <flist> CONSTRUCT.

Explanation: Indicates that a particular occurrence of a MAX or MIN intrinsic function reference cannot be vectorized because of the complexity or ordering of its arguments.

Supplemental Data:

<flist> is the name of the function involved.

Possible Response: This happens when a scalar variable appears both on the left side of the equal sign and as a MAX and MIN intrinsic function argument in the same Fortran statement.

If it is possible, simplify and reorder the arguments. Try to make the scalar variable appear as the first argument of the MAX or MIN reference.

Example:

Will not vectorize:

```
CC = MAX(MAX(A(I),CC),2.0,B(I))
```

Will vectorize:

```
CC = MAX(CC,A(I),2.0,B(I))
```

ILX0147I Short Form: UNSUPPORTED INTRINSIC FUNC
Long Form: THE INTRINSIC FUNCTION(S) <flist> DO NOT HAVE VECTOR OR SIMULATED VECTOR VERSIONS.

Explanation: Indicates that the compiler has found a call to an external intrinsic function that does not have a vector or simulated vector version. These are the functions that take REAL*16, COMPLEX*32, and CHARACTER arguments.

Supplemental Data:

<flist> is a list of the names of the functions that are not supported.

ILX0148I Short Form: SCALAR FASTER THAN VECTOR
Long Form: CODE THAT WAS ELIGIBLE TO EXECUTE IN VECTOR MODE WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SCALAR.

Explanation: Identifies loops that were eligible for vectorization but did not vectorize at all because the compiler has determined that the cost of vector processing exceeds the cost of scalar processing.

Example:

```
C EXAMPLE
C SCALAR FASTER THAN VECTOR
  REAL*4 A(20,20)

      DO 10 I = 1,20
        A(I,I) = A(I,I) ** 2.1
10      CONTINUE
```

Possible Response: In some cases, the estimates used by the compiler for comparing vector and scalar run times may not be accurate. If you find that the compiler has misjudged the profitability of vectorizing a particular loop, you can use the PREFER VECTOR directive to override the decision to process this loop in scalar.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C SCALAR FASTER THAN VECTOR
  REAL*4 A(20,20)

*DIR      PREFER VECTOR
      DO 10 I = 1,20
        A(I,I) = A(I,I) ** 2.1
10      CONTINUE
```

Note: Do not rely on "intuition" to determine whether or not PREFER VECTOR should be used. Always verify the appropriateness of its use by taking direct measurements of the run times of the affected loop, both with and without the use of the directive.

ILX0150I Short Form: VECTOR SUM REDUCTION
Long Form: VECTORIZATION WAS DONE USING SUM OR PRODUCT REDUCTION ON THE VARIABLE(S) <vlist>. RESULTS MAY DIFFER FROM SCALAR CODE.

Explanation: Indicates when a statement is vectorized using a sum reduction operation. Since the vector reduction instructions cause partial results to be accumulated in a sequence different from the sequence used for an equivalent scalar program, this

may result in answers that are different from those obtained by running the loop in scalar mode.

Supplemental Data:

<vlist> is a list of names of variables that are used as accumulators in statements vectorized via sum reduction.

Example:

```
C EXAMPLE
C VECTOR SUM REDUCTION
  REAL*4 SUM, A(128)

      SUM = 0.0
      DO 10 I = 1,128
        SUM = SUM + A(I)
10      CONTINUE
```

Possible Response: If there is concern for the consistency of program results between vectorized and nonvectorized programs, the vectorization of reduction functions can be inhibited by specifying the VECTOR(NORED) option on an @PROCESS card or when invoking the compiler.

ILX0151I Short Form: VECTOR INTRINSIC FUNCTION
Long Form: THE INTRINSIC FUNCTION(S) <flist> HAVE BEEN VECTORIZED.

Explanation: Indicates when statements are vectorized using vector intrinsic functions.

Note: If you are using the VS FORTRAN Version 1 Library, the vectorization of intrinsic functions may cause your program to produce different answers after vectorization. This is because there are no vector functions in the VS FORTRAN Version 1 Library. When intrinsic functions are vectorized, subprograms from the Version 2 Library will always be invoked, although the equivalent Version 1 subprograms would be used when running the program in scalar mode. Since the Version 2 Library produces more accurate results than does the Version 1 Library, vector and scalar processing of the same program may give slightly different answers in this situation.

Supplemental Data:

<flist> is a list of the names of the intrinsic functions that have been vectorized.

Possible Response: If there is concern for the consistency of program results between vectorized and nonvectorized programs when the Version 1 service subroutines are being used, the vectorization of intrinsic functions can be inhibited by specifying the VECTOR(NOINTRINSIC) option on an @PROCESS card or when invoking the compiler.

ILX0152I Short Form: VECTOR IMPLICIT ROUTINE
Long Form: THE IMPLICITLY CALLED ROUTINE(S) <flist> HAVE BEEN VECTORIZED.

Explanation: Indicates when statements are vectorized using vector versions of implicitly called routines. (These routines are processed as the result of certain notation appearing in a source statement. For example, if a REAL*4 variable is raised to a REAL*4 power, the compiler generates a reference to a routine that raises a real number to a real power.)

Note: If you are using the VS FORTRAN Version 1 Library, the vectorization of intrinsic functions may cause your program to produce different answers after vectorization. This is because there are no vector functions in the VS FORTRAN Version 1 Library. When intrinsic functions are vectorized, subprograms from the Version 2 Library will always be invoked, although the equivalent Version 1 subprograms would be used when running the program in scalar mode. Since the Version 2 Library produces more accurate results than does the Version 1 Library, vector and scalar processing of the same program may give slightly different answers in this situation.

Supplemental Data:

<flist> is a list of the names of the implicitly called routines that have been vectorized.

Example:

```
C EXAMPLE
C VECTOR IMPLICIT ROUTINE
  REAL*4 A(128)

      DO 10 I = 1,128
        A(I) = A(I) ** 2.1
10    CONTINUE
```

Possible Response: If there is concern for the consistency of program results between vectorized and nonvectorized programs when the Version 1 service subroutines are being used, the vectorization of these routines can be inhibited by specifying the VECTOR(NOINTRINSIC) option on an @PROCESS card or when invoking the compiler.

ILX0153I Short Form: VECTORIZED STOP OR RETURN
Long Form: STOP OR RETURN STATEMENT(S) AT ISN(S) <ilists> HAVE BEEN VECTORIZED. IN SOME CASES, THIS MAY LEAD TO AN EXECUTION TIME INTERRUPT THAT WOULD NOT OCCUR WITH SCALAR CODE.

Explanation: Indicates that a loop that contains a STOP or RETURN statement has been vectorized. Optimization level 3 is required to vectorize loops with STOP or RETURN statements. This is because, in order to vectorize, it may be necessary to evaluate some instances of the loop termination test that would not be evaluated with scalar code. In certain cases, this may lead to interrupts that would not otherwise occur.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) of the STOP or RETURN statements.

Example:

```
@PROCESS OPT(3) VECTOR
C EXAMPLE
C VECTORIZED STOP OR RETURN
  REAL A(50),B(50),C(50)

      DO 10 I = 1,50
        IF (A(I)/B(I) .EQ. 1.0) STOP
        C(I) = A(I)/B(I)
10    CONTINUE
```

In this case, a divide exception would be raised if B(I) is ever zero. If this is true for some value of I greater than the value at which the loop terminates, then the exception will not occur in the scalar loop, but may occur in the vector loop. This may happen, for example, if A(10) equals B(10) and B(11) equals zero.

Possible Response: If it is known that no exception will occur, no action is necessary. Otherwise, you may be able to suppress vectorization of this loop either by specifying the option OPTIMIZE(2) or by using a PREFER SCALAR directive.

Another alternative is to restructure the code to execute the loop termination test in scalar, and the rest of the loop in vector. This makes the loop eligible for partial vectorization while avoiding the possibility of raising an exception. This can be done with the following transformation:

1. Insert a new loop to evaluate the loop termination condition and determine the number of times each loop iterates before terminating. The code for the new loop will remain ineligible for vectorization.
2. Rewrite the original loop to process only the elements that are referenced before the loop terminates.
3. Add a test at the end of the modified code to determine whether the STOP or RETURN statement should be executed.
4. Specify the OPTIMIZE(2) option when the program is compiled, or use a PREFER SCALAR directive on the loop that evaluates the loop termination condition.

Modified Example:

```

@PROCESS OPT(2) VECTOR
C POSSIBLE RESPONSE
C VECTORIZED STOP OR RETURN
  REAL  A(50),B(50),C(50)
  INTEGER IMAX
  LOGICAL NEED_STOP

  DO 10 IMAX = 1,50
    IF (A(IMAX)/B(IMAX) .EQ. 1.0) GOTO 100
10  CONTINUE
100 NEED_STOP = (IMAX.LE.50)

  DO 20 I = 1,IMAX - 1
    C(I) = A(I)/B(I)
20  CONTINUE

  IF (NEED_STOP) THEN STOP

```

Note that this transformation is valid only if the test that determines whether the loop should terminate is independent of the values that are computed in the loop. Also, if the test is not the first statement in the loop, some special processing is necessary.

ILX0154I Short Form: VECTORIZED EXIT BRANCH
Long Form: EXIT BRANCH(ES) AT ISN(S) <ilist> HAVE BEEN VECTORIZED. IN SOME CASES, THIS MAY LEAD TO AN EXECUTION TIME INTERRUPT THAT WOULD NOT OCCUR WITH SCALAR CODE.

Explanation: Indicates that a loop that contains an exit branch has been vectorized. Optimization level 3 is required to vectorize loops with exit branches. This is because, in order to vectorize, it may be necessary to evaluate some instances of the loop termination test that would not be evaluated with scalar code. In certain cases, this may lead to interrupts that would not otherwise occur.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) indicating the exit branch.

Example:

```

@PROCESS OPT(3) VECTOR
C EXAMPLE
C VECTORIZED EXIT BRANCH
  REAL  A(50),B(50),C(50)

  DO 10 I = 1,50
    IF (A(I)/B(I) .EQ. 1.0) GOTO 5
    C(I) = A(I)/B(I)
10  CONTINUE
    ...
5  CONTINUE

```

In this case, a divide exception would be raised if B(I) is ever zero. If this is true for some value of I greater than

the value at which the loop terminates, then the exception will not occur in the scalar loop, but may occur in the vector loop. This may happen, for example, if A(10) equals B(10) and B(11) equals zero.

Possible Response: If it is known that no exception will occur, no action is necessary. Otherwise, you may be able to suppress vectorization of this loop either by specifying the option OPTIMIZE(2) or by using a PREFER SCALAR directive.

Another alternative is to restructure the code to execute the loop termination test in scalar, and the rest of the loop in vector. This makes the loop eligible for partial vectorization while avoiding the possibility of raising an exception. This can be done with the following transformation:

1. Insert a new loop to evaluate the exit branch condition and determine the number of times each loop iterates before terminating. The code for the new loop will remain ineligible for vectorization.
2. Rewrite the original loop to process only the elements that are referenced before the exit branch is taken.
3. Add a test at the end of the modified code to determine whether the exit branch is taken.
4. Specify the OPTIMIZE(2) option when the program is compiled, or use a PREFER SCALAR directive on the loop that evaluates the loop termination condition.

Modified Example:

```

@PROCESS OPT(2) VECTOR
C POSSIBLE RESPONSE
C VECTORIZED EXIT BRANCH
  REAL  A(50),B(50),C(50)
  INTEGER IMAX
  LOGICAL NEED_BRANCH

  DO 10 IMAX = 1,50
    IF (A(IMAX)/B(IMAX) .EQ. 1.0) GOTO 100
10  CONTINUE
100 NEED_BRANCH = (IMAX.LE.50)

  DO 20 I = 1,IMAX - 1
    C(I) = A(I)/B(I)
20  CONTINUE

  IF (NEED_BRANCH) THEN GOTO 5

  ...

5  CONTINUE

```

Note that this transformation is valid only if the test that determines whether the loop should terminate is independent of the values that are computed in the loop. Also, if the test is not the first statement in the loop, some special processing is necessary.

ILX0155I Short Form: VECTORIZED IF LOOP
Long Form: IF LOOP VECTORIZED BY
PROCESSING AS A DO LOOP.

Explanation: Indicates when an IF loop has been vectorized by processing it internally as a DO loop.

Example:

```
C EXAMPLE
C VECTORIZED IF LOOP
  INTEGER N,I
  REAL  A(*),B(*),C(*)

  I = 0
140  CONTINUE
  I = I + 1
  IF (I.GT.N) GOTO 141
  A(I) = B(I) + C(I)
  GOTO 140
141  CONTINUE
```

The IF loop above will be vectorized since it is processed as a DO loop.

ILX0156I Short Form: MASKED VECTOR
STATEMENT
Long Form: VECTORIZATION WAS
PERFORMED ON CONDITIONALLY
EXECUTED CODE. VECTOR MASK
MODE HAS BEEN USED. THE VECTOR
REPORT LISTING MAY FAIL TO
INDICATE THE BRANCH STATEMENT(S)
THAT AFFECT THE EXECUTION OF THIS
REGION.

Explanation: Indicates when a statement or group of statements are processed in mask mode. This message is intended to help clarify the program listing that is produced by the VECTOR(REPORT(XLIST)) option. This is necessary because information about the branch structure in a loop is not always reproduced when that loop is printed in the report.

In particular, the only conditional statements that are printed are logical IF statements that perform conditional assignments. No branch statements and no block IF constructs appear in the report.

You should refer to the source listing or to the listing produced by the VECTOR(REPORT(SLIST)) option in order to identify the control flow constructs that can affect the way the code being flagged by this message runs.

Example:

```
C EXAMPLE
C MASKED VECTOR STATEMENT
  REAL*4 A(128),B(128)

  DO 10 I = 1,128
    IF (A(I).LT.0.0) GOTO 5
    B(I) = 1.1
    GOTO 6
  5    B(I) = 2.2
  6    A(I) = 0.0
  10  CONTINUE
```

The GOTO statements in the above loop will not be printed in the compiler report output produced by the VECTOR(REPORT(XLIST)) option.

ILX0158I Short Form: CONDITIONAL SCALAR
CODE
Long Form: THIS CODE IS
CONDITIONALLY EXECUTED. THE
VECTOR REPORT LISTING MAY FAIL TO
INDICATE THE BRANCH STATEMENT(S)
THAT AFFECT THE EXECUTION OF THIS
REGION.

Explanation: Indicates when a statement or group of statements that has not been vectorized is part of a conditionally processed region of code. This message is intended to help clarify the program listing that is produced by the VECTOR(REPORT(XLIST)) option. This is necessary because information about the branch structure in a loop is not always reproduced when that loop is printed in the report.

In particular, the only conditional statements that are printed are logical IF statements that perform conditional assignments. No branch statements and no block IF constructs appear in the report.

You should refer to the source listing or to the listing produced by the VECTOR(REPORT(SLIST)) option in order to identify the control flow constructs that can affect the way the code being flagged by this message runs.

Example:

```
C EXAMPLE
C CONDITIONAL SCALAR CODE
  REAL*4 A(128),B(128)

  DO 10 I = 2,128
    IF (A(I-1).LT.0.0) GOTO 5
    B(I) = B(I-1)
    GOTO 6
  5    B(I) = B(I+1)
  6    A(I) = B(I)
  10  CONTINUE
```

The GOTO statements in the above loop will not be printed in the compiler report output produced by the VECTOR(REPORT(XLIST)) option.

ILX0159I Short Form: IF LOOP PROCESSED AS DO LOOP
Long Form: THIS IF LOOP HAS BEEN PROCESSED AS DO LOOP BY THE COMPILER SO THAT IT IS ELIGIBLE FOR VECTORIZATION ANALYSIS.

Explanation: Indicates when an IF loop has been processed internally as a DO loop, but will run in scalar mode. This may be caused by a recurrence or unsupportable construct.

Example:

```
C EXAMPLE
C IF LOOP PROCESSED AS DO LOOP
  INTEGER I
  REAL A(100)

  I = 1
500 IF (I .GT. 100) GOTO 505
  PRINT *, 'AT ITERATION ', I, A(I)
  I = I + 1
  GOTO 500
505 CONTINUE
```

ILX0165I Short Form: "PREFER SCALAR" USED
Long Form: THIS LOOP WILL BE EXECUTED IN SCALAR BECAUSE OF THE USE OF A "PREFER SCALAR" DIRECTIVE.

Explanation: This message identifies loops to which PREFER SCALAR directives have been applied. When this directive is specified, a loop that is considered eligible for vectorization by the compiler will not be vectorized. It should be used only when an analysis of the run-time performance of the loop has determined that the loop runs faster when the directive is present.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER SCALAR USED
  REAL*4 A(128,128)

*DIR    PREFER SCALAR
DO 10 I = 1,30
  A(I,I) = A(I,I) ** 2.1
10      CONTINUE
```

Possible Response: If the program has been modified since the directive was initially coded, or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First check whether the loop would be chosen for scalar processing if the directive were not enabled. If so, the directive is redundant and should be removed.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether

or not the performance is better when this directive is used.

ILX0167I Short Form: "PREFER VECTOR" USED
Long Form: A "PREFER VECTOR" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP.

Explanation: This message identifies loops to which PREFER VECTOR directives have been successfully applied. (PREFER VECTOR is successful only if the loop to which it is applied is eligible for vectorization and if no nested eligible loop also has PREFER VECTOR specified.) It should be used only when an analysis of the run-time performance of the loop has determined that the loop runs faster when the directive is present.

Note that after loop distribution has taken place, it is possible that a PREFER VECTOR directive is successful for part of the code within a loop, but is unsuccessful for the rest.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER VECTOR USED
  REAL*4 A(10,10)

*DIR    PREFER VECTOR
DO 10 I = 1,N
  A(I,I) = A(I,I) * 2.1
10      CONTINUE
```

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First check whether the loop would be chosen for vector processing if the directive were not enabled. If so, the directive is redundant and should be removed.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0168I Short Form: INAPPLICABLE "PREFER VECTOR"
Long Form: A "PREFER VECTOR" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP BUT COULD NOT BE HONORED BECAUSE THE LOOP WAS NOT ELIGIBLE FOR VECTORIZATION. THE DIRECTIVE HAS BEEN IGNORED.

Explanation: This message identifies loops for which inapplicable PREFER VECTOR directives have been specified. (PREFER VECTOR is inapplicable if the loop contains a recurrence or an unsupportable construct.)

Note that after loop distribution has taken place, it is possible that a PREFER VECTOR directive is inapplicable for part of the code within a loop, but is successful for the rest.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C INAPPLICABLE PREFER VECTOR
  REAL*4 A(128,128),B(128)

*DIR    PREFER VECTOR
        DO 10 I = 2,128
          A(I,I) = A(I,I) * 2.1
          B(I) = B(I-1)
10      CONTINUE
```

The second statement in this loop cannot be vectorized because it forms a recurrence. Therefore, the PREFER VECTOR directive cannot be applied to this statement. The directive is still applicable to the first statement in the loop.

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

It should first be determined whether the directive has any effect on the vectorization of other parts of the original loop. If not, it is useless and should be removed from the program.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0169I Short Form: OVERRIDDEN "PREFER VECTOR"
Long Form: A "PREFER VECTOR" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP BUT COULD NOT BE HONORED BECAUSE "PREFER VECTOR" WAS ALSO SPECIFIED FOR SOME ELIGIBLE NESTED LOOP. THE DIRECTIVE HAS BEEN IGNORED.

Explanation: This message identifies loops for which overridden PREFER VECTOR directives were specified. (PREFER VECTOR will be overridden if it is specified for more than one mutually nested eligible loop. In this case only the innermost PREFER VECTOR will be successful.)

Note that after loop distribution has taken place, it is possible that a PREFER VECTOR directive is overridden for part of the code within a loop, but is successful for the rest.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C OVERRIDDEN PREFER VECTOR
  REAL*4 A(128,128)

*DIR    PREFER VECTOR
        DO 10 I = 1,128
*DIR    PREFER VECTOR
        DO 10 J = 1,128
          A(I,J) = A(I,J) * 2.1
10      CONTINUE
```

Only one loop in this nest can be vectorized. The PREFER VECTOR directive used on the outer loop will be overridden by the PREFER VECTOR used on the inner loop.

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

It should first be determined whether the directive has any effect on the vectorization of other parts of the original loop. If not, it is useless and should be removed from the program.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0170I Short Form: "ASSUME COUNT" USED
Long Form: THE ITERATION COUNT OF THIS LOOP WAS SPECIFIED AS "<n>" BY AN "ASSUME COUNT" DIRECTIVE.

Explanation: This message identifies loops for which successful ASSUME COUNT directives have been specified. (ASSUME COUNT is successful if the iteration count of the loop to which it applies cannot be determined at compile time.) This directive is used to help the compiler decide whether vectorization of a particular loop will result in a performance improvement.

Supplemental Data:

<n> is the value of the loop iteration count that has been used for vector cost analysis.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C ASSUME COUNT USED
  REAL*4 A(128,128)

*DIR    ASSUME COUNT(10)
        DO 10 I = 1,N
          A(I,I) = A(I,I) ** 2.1
10      CONTINUE
```

Possible Response: If the program or data has undergone revisions since ASSUME COUNT was initially coded, it may be necessary to verify its correctness.

To do this, conduct a run-time analysis of the loop to determine whether the number specified by the directive approximates the average iteration count observed for the processing loop.

ILX0171I Short Form: INVALID "ASSUME COUNT" USED

Long Form: AN "ASSUME COUNT" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP. THE COMPILER HAS DETERMINED THAT THE ACTUAL ITERATION COUNT IS "<n>". THE DIRECTIVE HAS BEEN IGNORED.

Explanation: This message identifies loops for which invalid ASSUME COUNT directives have been specified. (ASSUME COUNT is invalid if the iteration count of the loop to which it applies can be determined at compile time.) An invalid ASSUME COUNT directive will have no effect on the compilation process.

Supplemental Data:

<n> is the value of the loop iteration count that has been used for vector cost analysis.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C INVALID ASSUME COUNT
  REAL*4 A(128,128)
  PARAMETER (N=128)

*DIR    ASSUME COUNT(10)
        DO 10 I = 1,N
          A(I,I) = A(I,I) ** 2.1
10      CONTINUE
```

Possible Response: Consider removing the directive from the code.

**ILX0172I Short Form: IGNORE RECRDEPS" USED
Long Form: AN "IGNORE RECRDEPS" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP.**

Explanation: This message identifies loops to which IGNORE RECRDEPS directives have been applied. It does not necessarily imply that the directive had an effect on the vectorization of the loop, although this will often be the case.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C IGNORE RECRDEPS USED
  COMMON // N
  REAL*4 A(128)

*DIR    IGNORE RECRDEPS
        DO 10 I = 64,128
          A(I) = A(I-N) ** 2.1
10      CONTINUE
```

Possible Response: Determine whether the directive affects vectorization of the loop. If not, it should be removed to avoid unexpected side effects in the event that the loop is modified in the future.

If the directive does alter vectorization, try to determine why this occurred, that is, determine which potential dependences have been ignored because of the directive. (These dependences are identified by other compiler report messages that appear with the statements within a loop.)

If it is possible to determine the run-time conditions under which these potential dependences actually arise, code should be inserted prior to the loop to check for these conditions and to issue a message and/or stop the program when these conditions are met. (In this case, the dependence will exist if the value of the variable N is between 1 and 64.)

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C IGNORE RECRDEPS USED
  COMMON // N
  REAL*4 A(128)

  IF (N.GE.1 .AND. N.LE.64) THEN
    PRINT *, 'INCORRECT IGNORE DIRECTIVE'
    STOP
  ENDIF

*DIR    IGNORE RECRDEPS
        DO 10 I = 64,128
          A(I) = A(I-N) ** 2.1
10      CONTINUE
```

ILX0173I Short Form: UNBREAKABLE RECURRENCE
Long Form: AN IGNORE DIRECTIVE WAS APPLIED TO THIS LOOP BUT DID NOT LEAD TO VECTORIZATION. THIS IS BECAUSE A DEFINITE RECURRENCE EXISTS INVOLVING THE VARIABLE(S) <vlist>.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible to vectorize. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization is not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies loops in which a recurrence definitely exists, even if the IGNORE directive is used.

Note that for some loops, both types of recurrences may be present. Therefore, when an IGNORE directive is used, other messages may indicate that the directive was applied even though the loop did not vectorize.

Supplemental Data:

<vlist> is a list of the names of the variables that are involved in the recurrence. (This list includes only the variables that could be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example 1:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE 1
C UNBREAKABLE RECURRENCE
  REAL  A(100),B(100)

*DIR  IGNORE RECRDEPS(A)
  DO 10 I = 1,50
    A(I+1) = A(I) * B(I)
  10  CONTINUE
```

In this example, a recurrence definitely exists because the value assigned into the array A on each iteration is used to compute the value for the next iteration. The IGNORE directive is not honored in this case, and it is not possible to vectorize.

Example 2:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE 2
C UNBREAKABLE RECURRENCE
  REAL  A(100),B(100),C(100)

*DIR  IGNORE RECRDEPS(A,C)
  DO 10 I = 1,50
    IF (B(I) > 0.0) THEN
      A(I+1) = A(I) * B(I)
      C(I+N) = C(I) * B(I)
    ENDIF
  10  CONTINUE
```

In this example, there are two recurrences. The first involves the array A, and the second involves the array C. The first recurrence definitely exists, while the second might exist depending on the value of N; if N is less than 1 or greater than 49, no recurrence exists. Although the second recurrence can be eliminated with the IGNORE directive, the first one cannot.

Possible Response 2: Partial vectorization may be performed only if you rewrite the loop so that the vectorizable and nonvectorizable statements are in separate IF blocks.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C UNBREAKABLE RECURRENCE
  REAL  A(100),B(100),C(100)

*DIR  IGNORE RECRDEPS(A,C)
  DO 10 I = 1,50
    IF (B(I) > 0.0) THEN
      A(I+1) = A(I) * B(I)
    ENDIF
    IF (B(I) > 0.0) THEN
      C(I+N) = C(I) * B(I)
    ENDIF
  10  CONTINUE
```

In this modified example, the second IF block is eligible for vectorization. The first IF block still contains a definite recurrence and cannot be vectorized.

ILX0176I Short Form: IGNORABLE RECURRENCE
Long Form: AN IGNORE DIRECTIVE WAS APPLIED TO THIS LOOP BUT DID NOT LEAD TO VECTORIZATION. THIS IS BECAUSE A POSSIBLE RECURRENCE EXISTS INVOLVING THE ARRAY(S) <alist>. THESE ARRAYS WERE NOT SPECIFIED IN THE DIRECTIVE. IF NO RECURRENCE EXISTS, VECTORIZATION CAN ACHIEVED BY MODIFYING THE DIRECTIVE.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible to vectorize. The compiler detects recurrences by analyzing scalar variables, array

subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization is not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies loops in which a possible recurrence was not eliminated, even when the IGNORE directive was used. It indicates that the variables responsible for the possible recurrence were not specified in the list of variables to which the directive should be applied.

Supplemental Data:

<alist> is a list of the names of arrays that are involved in the possible recurrence. (This list includes only the variables that could be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C IGNORABLE RECURRENCE
  REAL  A(100),B(100),C(100)

*DIR  IGNORE RECRDEPS(A)
DO 10 I = 1,50
  IF (B(I) > 0.0) THEN
    A(I+N) = A(I) * B(I)
    C(I+M) = C(I) * B(I)
  ENDIF
10 CONTINUE
```

In this example, there are two possible recurrences. The first involves the array A, and the second involves the array C. Whether the recurrences exist depends on the values of M and N; if these values are less than 1 or greater than 49, no recurrence exists.

The IGNORE directive is specified for array A only, and it is assumed that a recurrence involving A does not exist. However, it is assumed that a recurrence involving C does exist. Since partial vectorization is not possible for statements in the same IF block, no vectorization can be performed.

Possible Response 1: If you are certain that a recurrence will never exist (in this example, that M is always less than 1 or greater than 49), vectorization can be achieved by modifying the IGNORE directive so that it applies to the indicated array or arrays.

Modified Example 1:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 1
C IGNORABLE RECURRENCE
  REAL  A(100),B(100),C(100)

*DIR  IGNORE RECRDEPS(A,C)
DO 10 I = 1,50
  IF (B(I) > 0.0) THEN
    A(I+N) = A(I) * B(I)
    C(I+M) = C(I) * B(I)
  ENDIF
10 CONTINUE
```

Possible Response 2: If it is possible that a recurrence may exist (in this example, that M may be greater than or equal to 1 or less than or equal to 49), you may rewrite the loop so that the vectorizable and nonvectorizable statements are in separate IF blocks.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C IGNORABLE RECURRENCE
  REAL  A(100),B(100),C(100)

*DIR  IGNORE RECRDEPS(A)
DO 10 I = 1,50
  IF (B(I) > 0.0) THEN
    A(I+N) = A(I) * B(I)
  ENDIF
  IF (B(I) > 0.0) THEN
    C(I+M) = C(I) * B(I)
  ENDIF
10 CONTINUE
```

In this modified example, since the directive applied only to the array A, the first IF block is eligible for vectorization. The second IF block will execute in scalar mode.

**ILX0177I Short Form: POTENTIAL RECRDEP
ELIMINATED
Long Form: POTENTIAL BACKWARD
DEPENDENCE(S) INVOLVING THE
ARRAY(S) <alist> HAVE BEEN IGNORED
BECAUSE OF AN "IGNORE RECRDEPS"
DIRECTIVE APPLIED TO THE LOOP(S)
AT NESTING LEVEL(S) <levlist>.**

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive has caused the compiler to ignore some backward dependence that would otherwise have been assumed to exist. It does not necessarily imply that the directive had an effect on the vectorization of the loop, although this will often be the case.

Supplemental Data:

<alist> is a list of the names of the arrays involved in the ignored dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C POTENTIAL RECRDEP ELIMINATED
COMMON // N
REAL*4 A(-200:200)

*DIR    IGNORE RECRDEPS
DO 10 I = 1,128
      A(I) = A(I+N) ** 2.1
10      CONTINUE
```

Possible Response: Determine whether the directive affects vectorization of the loop in which the statement occurs. If not, it should be removed to avoid unexpected side effects in the event that the loop is modified in the future.

If the directive does alter vectorization, try to determine the run time conditions under which these potential backward dependences actually arise. (In this case, there will be a dependence if the value of the variable N is between -127 and -1.) Code should be inserted prior to the loop to check for these conditions and to issue a message and/or stop the program when these conditions are met.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C POTENTIAL RECRDEP ELIMINATED
COMMON // N
REAL*4 A(-200:200)

IF (N.LT.0 .AND. N.GT.-128) THEN
  PRINT *, 'INCORRECT IGNORE DIRECTIVE'
  STOP
ENDIF
*DIR    IGNORE RECRDEPS
DO 10 I = 1,128
      A(I) = A(I+N) ** 2.1
10      CONTINUE
```

ILX0179I Short Form: ACTUAL RECRDEP NOT IGNORED
Long Form: BACKWARD DEPENDENCES INVOLVING THE ARRAY(S) <alist>, WHICH WERE IN THE RANGE OF "IGNORE RECRDEPS" DIRECTIVE(S) APPLIED TO THE LOOP(S) AT LEVEL(S) <levlist> HAVE NOT BEEN IGNORED BECAUSE THE COMPILER HAS DETERMINED THAT THESE

DEPENDENCES ARE ALWAYS PRESENT.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive could have been applied but where the compiler has chosen not to do so because the subject dependences are always present. (This directive is only honored when the compiler determines that there is a potential dependence but that the dependence will not arise under certain run time conditions.) Even if this message is present, it is possible that the directive had some effect on the vectorization of the loop, since some other backward dependences may have been ignored.

Supplemental Data:

<alist> is a list of the names of the variables that carry the dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C ACTUAL RECRDEP NOT IGNORED
REAL*4 A(128)

*DIR    IGNORE RECRDEPS
DO 10 I = 2,128
      A(I) = A(I-1) ** 2.1
10      CONTINUE
```

The IGNORE directive cannot be honored in this case since a recurrence definitely exists.

Possible Response: Determine whether the directive has any effect on the vectorization of other statements used in the loop. If not, remove the directive.

ILX0180I Short Form: POTENTIAL RECRDEP MODIFIED
Long Form: DEPENDENCES INVOLVING THE ARRAY(S) <alist> WERE PRESUMED NOT TO BE INTERCHANGE PREVENTING BECAUSE OF AN "IGNORE RECRDEPS" DIRECTIVE APPLIED TO THE LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive has caused the compiler to assume that some forward dependence is not interchange preventing. This happens only when the compiler is not certain whether or not a dependence is really interchange preventing.

Usually, the presence of an interchange-preventing dependence restricts vectorization. When an interchange-preventing dependence exists, the reordering, or interchange, of two loops would cause different results to be produced. Since vectorization of an outer loop has the same effect as moving that loop to the innermost position, the existence of an interchange-preventing dependence carried by an outer loop prevents vectorization.

To understand how an interchange-preventing dependence comes about, study the following example:

```
DO 10 I=1,2
DO 10 J=1,2
10   A(I-1,J+1)=A(I,J)
```

In this code, the element A(1,2) is fetched when I=1 and J=2 and is stored into when I=2 and J=1. When these loops are processed, the fetch will occur before the store. However, if the loops were interchanged, the store would come first and different results would probably be produced.

It is not always possible for the compiler to determine whether or not a dependence is interchange preventing. Unless it can prove otherwise, the compiler will normally assume that a given dependence is interchange preventing. This will insure that correct results are always produced after vectorization, even though some potential vectorization may be missed.

In cases where the compiler is unable to determine whether or not a particular dependence is interchange preventing, the IGNORE RECRDEPS directive allows you to force the compiler to assume that the dependence is not interchange preventing.

Supplemental Data:

<alist> is a list of the names of the arrays involved in the modified dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C POTENTIAL RECRDEP MODIFIED
      REAL*4 U(100,100,100)

*DIR      IGNORE RECRDEPS
      DO 190 K = 1, 19
      DO 190 J = 1, 19
      DO 190 I = 1, 19
      U(I,J,K) = U(I+N,J,K) + U(I,J+N,K) + U(I,J,K+N)
190   CONTINUE
```

Possible Response: Determine whether the directive affects vectorization of the loop in which the statement occurs. If not, it should be removed to avoid unexpected side effects in the event that the loop is modified in the future.

If the directive does alter vectorization, try to determine the run time conditions under which these dependences might be interchange preventing. Code should be inserted prior to the loop to check for these conditions and to issue a message and/or stop the program when these conditions are met.

Note: The table of ignored dependences that appears after the compiler report message listing can help identify these dependences.

ILX0181I Short Form: ACTUAL RECRDEP NOT MODIFIED
Long Form: INTERCHANGE PREVENTING DEPENDENCES INVOLVING THE ARRAY(S) <alist> THAT WERE IN THE RANGE OF "IGNORE RECRDEPS" DIRECTIVE(S) APPLIED TO THE LOOP(S) AT LEVEL(S) <levlist> HAVE BEEN PRESERVED BECAUSE THE COMPILER HAS DETERMINED THAT THESE DEPENDENCES DEFINITELY EXIST.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive could have been applied to some interchange-preventing dependences but where the compiler has chosen not to do so because the subject dependences are always present.

Usually, the presence of an interchange-preventing dependence restricts vectorization. When an interchange-preventing dependence exists, the reordering, or interchange, of two loops would cause different results to be produced. Since vectorization of an outer loop has the same effect as moving that loop to the innermost position, the existence of an interchange-preventing dependence carried by an outer loop prevents vectorization.

To understand how an interchange-preventing dependence comes about, study the following example:

```
DO 10 I=1,2
DO 10 J=1,2
10    A(I-1,J+1)=A(I,J)
```

In this code, the element A(1,2) is fetched when I=1 and J=2 and is stored into when I=2 and J=1. When these loops are processed, the fetch will occur before the store. However, if the loops were interchanged, the store would come first and different results would probably be produced.

It is not always possible for the compiler to determine whether or not a dependence is interchange preventing. Unless it can prove otherwise, the compiler will normally assume that a given dependence is interchange preventing. This will insure that correct results are always produced after vectorization, even though some potential vectorization may be missed.

Normally, an IGNORE RECRDEPS directive would cause the compiler to assume that a dependence is not interchange preventing. However, in cases where the compiler is absolutely certain that a dependence is interchange preventing, the existence of the IGNORE RECRDEPS directive will have no effect on the analysis of a program.

Note that even if this message is present, it is possible that the directive had some effect on the vectorization of the loop, since some other backward dependences may have been ignored.

Supplemental Data:

<alist> is a list of the names of the variables that carry the dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C ACTUAL RECRDEP NOT MODIFIED
REAL*4 A(128,128)

*DIR    IGNORE RECRDEPS
DO 10 I = 1,100
DO 10 J = 2,100
    A(I,J) = A(I+1,J-1) ** 2.1
10    CONTINUE
```

The IGNORE directive cannot be honored in this case since the dependence is definitely interchange preventing.

Possible Response: Determine whether the directive has any effect on the vectorization of other statements used in the loop. If not, remove the directive.

ILX0200I Short Form: POINTER VARIABLE
Long Form: ONE OR MORE POINTER
VARIABLES ARE MODIFIED IN THE
LOOP.

Explanation: This message identifies a loop where a pointer variable is modified. The compiler cannot determine dependencies for associated pointee variables or arrays.

Example:

```
Subroutine Sub(dis)
Pointer (p1, pteel(100))
Integer disp
Real A(100), B(100), C(100)
p1 = LOC(A)
Do I=1,100
    pteel(1) = B(i) * c(i)
    p1 = p1 + disp
Enddo
```

Possible Response: Rewrite the reference to eliminate pointer arithmetic by transforming it to subscripting arithmetic as a function of the induction variable.

Compiler Report Diagnostic Messages—Parallel

The following messages apply to errors or conditions that arise when the compiler attempts to generate parallel code for your program. They will be produced if you specify the REPORT suboption of the PARALLEL compile-time option.

ILX0301I Short Form: NON-INTEGER LOOP CONTROL
Long Form: A LOOP CONTROL PARAMETER IS NOT INTEGER*4. THE LOOP IS NOT ANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates that a variable that is not INTEGER*4 is used as part of an expression controlling the iteration of a loop or as a DO loop variable.

Possible Response: If the lower bound, upper bound, or increment expressions are not INTEGER*4, replace these expressions with INTEGER*4 variables that have been assigned the appropriate values prior to the loop.

If the DO loop variable is not INTEGER*4 and it can be replaced by one that is, do so.

ILX0302I Short Form: MORE THAN 8 NESTED LOOPS
Long Form: PARALLELIZATION ANALYSIS IS RESTRICTED TO LOOPS AT THE INNERMOST EIGHT LEVELS OF NESTING.

Explanation: Indicates that a loop has not been considered for parallel code generation because it contains nested loops more than eight levels deep.

ILX0303I Short Form: NESTED LOOP NOT ANALYZABLE
Long Form: SOME NESTED LOOP WAS FOUND TO BE UNANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates that a loop contains a nested loop which was not eligible for parallel code generation analysis.

Example:

```
C EXAMPLE
C NESTED LOOP NOT ANALYZABLE
  REAL*4 A(1000,1000)

  DO 10 I = 1,1000
    DO 10 J = 1,1000
      A(I,J) = A(I,J) ** 2.1
      WRITE(6,*,ERR=999) A(I,J)
10  CONTINUE
```

In this case, the inner loop is unanalyzable because it contains an I/O statement that is unanalyzable. The

outer loop is marked as unanalyzable since it surrounds the unanalyzable inner loop.

Possible Response: Identify the loop causing the rejection and attempt to recode it to eliminate the problem.

Modified Example:

```
C POSSIBLE RESPONSE
C NESTED LOOP NOT ANALYZABLE
  REAL*4 A(1000,1000)

  DO 10 I = 1,1000
    DO 10 J = 1,1000
      A(I,J) = A(I,J) ** 2.1
10  CONTINUE
  WRITE(6,*,ERR=999) A
```

ILX0304I Short Form: I/O OPERATION
Long Form: CERTAIN I/O STATEMENTS AT ISN(S) <ilist> ARE NOT ANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates that a loop contains one or more I/O statements that cannot be analyzed. The following I/O statements are not analyzable:

- Statements other than READ, WRITE or PRINT
- READ and WRITE statements containing END= or ERR= labels
- Statements that specify a NAMELIST
- Statements containing arrays without subscripts
- Statements that specify internal files
- Asynchronous I/O statements
- Implied-DO statements. (Some implied-DO statements may be analyzable).

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicates the locations of the statement or statements responsible for the rejection.

Example:

```
C EXAMPLE
C I/O OPERATION
  REAL*4 A(1000),B(1000)

  DO 10 I = 1,1000,1
    A(I) = B(I) * 3.3
    WRITE(6,ID=J) A(I)...A(I+999)
10  CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements are separated from the portions of the original loop that are eligible for parallel code generation analysis.

Modified Example:

```

C POSSIBLE RESPONSE
C I/O OPERATION
  REAL*4 A(1000),B(1000)

  DO 10 I = 1,1000,1
    A(I) = B(I) * 3.3
10  CONTINUE

  DO 20 I = 1,1000,1
    WRITE(6,ID=J) A(I)...A(I+999)
20  CONTINUE

```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0305I Short Form: DO WHILE OR IMPLIED DO
Long Form: ONE OR MORE DO WHILE
LOOPS OR I/O STATEMENTS WITH
IMPLIED DO LOOPS AT ISN(S) <ilist>
ARE NOT ANALYZABLE FOR
PARALLELIZATION.

Explanation: Indicates the presence of an unanalyzable loop construct contained within an iterative DO loop. This may be caused by either a DO WHILE statement or by an I/O statement that contains an implied DO loop.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicates the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C DO WHILE OR IMPLIED DO
  REAL A(1000,1000),B(1000)

  DO 10 I=1,1000
    A(I,5) = B(I) * 3.3
    WRITE(6,*) (A(I,J),J=1,1000)
10  CONTINUE

```

Possible Response: Break the loop into two or more loops, so that any unanalyzable constructs are separated from the portions of the original loop that are analyzable.

Modified Example:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C DO WHILE OR IMPLIED DO
  REAL A(1000,1000),B(1000)

  DO 10 I=1,1000
    A(I,5) = B(I) * 3.3
10  CONTINUE

  DO 20 I=1,1000
    WRITE(6,*) (A(I,J),J=1,1000)
20  CONTINUE

```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0306I Short Form: CHARACTER DATA
Long Form: ONE OR MORE
STATEMENTS USING CHARACTER
DATA OCCUR AT ISN(S) <ilist> AND ARE
UNANALYZABLE FOR
PARALLELIZATION.

Explanation: Indicates the presence of character data.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicates the locations of the statement or statements responsible for the rejection.

Possible Response: Break the loop into two or more loops, so that any statements that reference character data are separated from the portions of the original loop that are eligible for parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0307I Short Form: UNANALYZABLE STOP OR
RETURN
Long Form: STOP OR RETURN
STATEMENTS AT ISN(S) <ilist> ARE
NOT ANALYZABLE FOR
PARALLELIZATION BECAUSE THIS
LOOP CONTAINS A NESTED LOOP.

Explanation: Loops that contain STOP or RETURN statements are not eligible for parallel code generation. This message indicates that a loop was rejected because it is an outer loop that contains a STOP or RETURN statement.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicates the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C UNANALYZABLE STOP OR RETURN
  REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)

  DO 20 I = 1,1000
    DO 10 J = 1,1000
      IF (C(I,J) .EQ. 0.0) STOP
      A(I,J) = B(I,J)
10  CONTINUE
20  CONTINUE

```

Possible Response: To make the outer loops eligible for partial parallel code generation, restructure the code to separate the loop termination test from the rest

of the loop. Transform the original nest of loops as follows:

1. Insert a new nest of loops to evaluate the loop termination condition and determine the number of times each loop repeats before terminating. Note that the code within the new nest will remain ineligible for parallel code generation.
2. Rewrite the original nest to process only the elements that are referenced before the loops terminate. Note that the number of iterations for the inner loop is different on the final iteration of the outer loop than it is on earlier iterations. You will need to duplicate the inner loop to process the final iteration separately.
3. Add a test at the end of the modified code to determine whether the STOP or RETURN statement should be processed.

Modified Example:

```

C POSSIBLE RESPONSE
C UNANALYZABLE STOP OR RETURN
  REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)
  INTEGER IMAX,JMAX
  LOGICAL NEED_STOP

      DO 20 IMAX = 1,1000
        DO 10 JMAX = 1,1000
          IF (C(IMAX,JMAX) .EQ. 0.0) GOTO 100
10      CONTINUE
20      CONTINUE
        DO 40 I = 1,IMAX-1
          DO 30 J = 1,1000
            A(I,J) = B(I,J)
30          CONTINUE
40          CONTINUE
          IF (NEED_STOP) THEN
            DO 50 J = 1,JMAX-1
              A(IMAX,J) = B(IMAX,J)
50          CONTINUE
            STOP
          ENDIF

```

Note that this transformation will only be valid if the test that determines whether the loop should terminate is independent of the values that are being computed in the loops. Also note that if the test is not the first statement of that loop, some special processing may be necessary.

ILX0308I Short Form: BRANCH AROUND INNER LOOP
Long Form: THE BRANCH(ES) ORIGINATING AT ISN(S) <ilist> BYPASS ONE OR MORE NESTED LOOPS. THE LOOP(S) CONTAINING THE BRANCH(ES) ARE NOT ANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates that a loop was rejected because some branch within the loop causes an inner loop to be bypassed.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicates the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C BRANCH AROUND INNER LOOP
  REAL*4 A(1000),B(1000)

      DO 6 I = 1,1000
        A(I) = A(I) / 2.0
        IF (A(I) .EQ. 0.0) GO TO 5
        DO 4 J = 1,1000
          B(J) = B(J) ** 2.1
4          CONTINUE
5          CONTINUE
6          CONTINUE

```

Possible Response: Break the loop into two or more loops, so that any unanalyzable branches are separated from the portions of the original loop that are eligible for parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

Modified Example:

```

C POSSIBLE RESPONSE
C BRANCH AROUND INNER LOOP
  REAL*4 A(1000),B(1000)

      DO 3 I = 1,1000
        A(I) = A(I) / 2.0
3      CONTINUE
C
      DO 6 I = 1,1000
        IF (A(I) .EQ. 0.0) GO TO 5
        DO 4 J = 1,1000
          B(J) = B(J) ** 2.1
4          CONTINUE
5          CONTINUE
6          CONTINUE

```

ILX0309I Short Form: UNANALYZABLE EXIT BRANCH
Long Form: EXIT BRANCHES ORIGINATING AT ISN(S) <ilist> ARE NOT ANALYZABLE FOR PARALLELIZATION BECAUSE THIS LOOP CONTAINS A NESTED LOOP.

Explanation: Loops that contain exit branches are not eligible for parallel code generation. This message indicates that a loop was rejected because it is an outer loop that contains an exit branch.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicates the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C UNANALYZABLE EXIT BRANCH
      REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)

      DO 20 I = 1,1000
        DO 10 J = 1,1000
          IF (C(I,J) .EQ. 0.0) GOTO 500
          A(I,J) = B(I,J)
10      CONTINUE
20      CONTINUE
      ...
500     CONTINUE

```

Possible Response: You may be able to make the outer loops eligible for partial parallel code generation if you restructure the code to separate the loop termination test from the rest of the loop. The original nest of loops may be transformed as follows:

1. Insert a new nest of loops to evaluate the loop termination condition and determine the number of times each loop repeats before terminating. Note that the code within the new nest will remain ineligible for parallel code generation.
2. Rewrite the original nest to process only the elements that are referenced before the loops terminate. Note that the number of iterations for the inner loop is different on the final iteration of the outer loop than it is on earlier iterations. You will need to duplicate the inner loop to process the final iteration separately.
3. Add a test at the end of the modified code to determine whether the original branch should be taken.

Modified Example:

```

C POSSIBLE RESPONSE
C EXIT BRANCH
      REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)
      INTEGER IMAX,JMAX
      LOGICAL NEED_BRANCH

      DO 20 IMAX = 1,1000
        DO 10 JMAX = 1,1000
          IF (C(IMAX,JMAX) .EQ. 0.0) GOTO 100
10      CONTINUE
20      CONTINUE
100     NEED_BRANCH = (JMAX.LE.1000)
      DO 40 I = 1,IMAX-1
        DO 30 J = 1,1000
          A(I,J) = B(I,J)
30      CONTINUE
40      CONTINUE
      IF (NEED_BRANCH) THEN
        DO 50 J = 1,JMAX-1
          A(IMAX,J) = B(IMAX,J)
50      CONTINUE
        GOTO 500
      ...
500     CONTINUE

```

Note that this transformation may not increase the ability of the compiler to generate parallel code for your program and could result in a scalar program that runs more slowly than the original.

ILX0310I Short Form: LOOP NOT OPTIMIZABLE
Long Form: PARALLELIZATION IS INHIBITED BECAUSE THIS LOOP IS NOT OPTIMIZABLE. THIS MAY BE CAUSED BY AN INDUCTION VARIABLE THAT MAY BE RESET INSIDE THE LOOP OR BY COMPLEX BRANCHING OUTSIDE THE LOOP.

Explanation: Indicates situations where optimization and parallel code generation for loops are inhibited. This can happen for a variety of reasons:

- When DO loop variables are not guaranteed to behave like standard DO loop variables. For example, this occurs when a DO loop variable is used as a parameter to a subroutine.
- When a DO loop variable is referenced in an EQUIVALENCE statement.
- When certain complicated patterns of branching are used around a DO loop.

Example 1:

```

C EXAMPLE 1
C LOOP NOT OPTIMIZABLE
      EQUIVALENCE (K1,K2)
      REAL*4 A(1000)

      DO 10 K1 = 1,1000
        A(K2) = A(K2) ** 2.1
10      CONTINUE

```

Possible Response 1: When a loop is not optimizable because the induction variable is used in an EQUIVALENCE statement, attempt to use a different DO loop variable to control the loop iteration.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C LOOP NOT OPTIMIZABLE
      EQUIVALENCE (K1,K2)
      REAL*4 A(1000)

      DO 10 K = 1,1000
        A(K) = A(K) ** 2.1
10      CONTINUE

```

Example 2:

```

C EXAMPLE 2
C LOOP NOT OPTIMIZABLE
      REAL*4 Q(1000,1000),R(1000,1000)

      DO 160 J=1,1000
      DO 160 I=1,1000
      L=1
      IF (L.GT.LLIM) GO TO 140
100   DO 120 K=1,1000
      Q(J,I) = R(L,K)
120   CONTINUE
      L=L+1
      IF (L.LE.LLIM ) GO TO 100
140   CONTINUE
160   CONTINUE

```

The inner loop is not eligible for optimization due to the complex pattern of branches around that loop.

Possible Response 2: For cases such as this, attempt to redesign the algorithm using structured programming constructs whenever possible.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C LOOP NOT OPTIMIZABLE
      REAL*4 Q(1000,1000),R(1000,1000)

      DO 160 J=1,1000
      DO 160 I=1,1000
      DO 120 L=1,LLIM
100   DO 120 K=1,1000
      Q(J,I) = R(L,K)
120   CONTINUE
160   CONTINUE

```

ILX0311I Short Form: BACKWARD BRANCH
Long Form: ONE OR MORE BACKWARD BRANCHES TO THE STATEMENT LABEL(S) <llist> HAVE BEEN FOUND. THE CONTAINING LOOP(S) ARE NOT ANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates the presence of a backward branch or a DO WHILE loop within a DO loop.

Supplemental Data:

<llist> is a list of user defined statement labels that are used to indicate the targets of any backward branches that occur in the loop.

Example:

```

C EXAMPLE
C BACKWARD BRANCH
      REAL*4 A(1000),B(10000,1000)

      DO 20 I = 1,1000
      A(I) = 0.0
      K = 1
10    A(I) = A(I) + B(K,I)
      K = K + 1
      IF (K .LE. 1000) GO TO 10
20    CONTINUE

```

Possible Response: Attempt to replace the backward GOTO with an equivalent DO loop.

Modified Example:

```

C POSSIBLE RESPONSE
C BACKWARD BRANCH
      REAL*4 A(1000),B(1000,1000)

      DO 20 I = 1,1000
      A(I) = 0.0
      DO 15 K = 1,1000
10    A(I) = A(I) + B(K,I)
15    CONTINUE
20    CONTINUE

```

ILX0312I Short Form: INTRINSIC CHARACTER FUNCTION
Long Form: THE CHARACTER MANIPULATION FUNCTION(S) <flist> ARE NOT ANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates that a loop is rejected because it uses one or more of the character manipulation functions (INDEX, LGE, LGT, LLE, and LLT) contained in the VS FORTRAN library.

Supplemental Data:

<flist> is a list consisting of function names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements using character manipulation functions are separated from the portions of the original loop that are eligible for parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

**ILX0313I Short Form: RESTRICTED CONSTRUCT
Long Form: THE LANGUAGE
CONSTRUCT(S) <clist> ARE NOT
ANALYZED FOR PARALLELIZATION.**

Explanation: Indicates that a loop is rejected because it contains some language construct that cannot be analyzed by the compiler. These constructs include assigned and computed GOTO statements and NAMELIST statements.

Supplemental Data:

<clist> is a list consisting of the names of the language constructs responsible for the rejection along with the ISNs (internal statement numbers) of the statements in which they are used.

Example:

```
C EXAMPLE
C RESTRICTED CONSTRUCT
  INTEGER*4 TEST(1000)
  REAL*4 W(1000),X(1000),Y(1000),Z(1000)

  DO 1000 I = 1,1000
    GOTO (400,500,600),TEST(I)
    W(I) = 0.0
    GOTO 1000
400   W(I) = X(I)
    GOTO 1000
500   W(I) = Y(I)
    GOTO 1000
600   W(I) = Z(I)
1000  CONTINUE
```

Possible Response: In the case of an assigned or computed GOTO statement, try to recode the loop to achieve the same logic structure using logical and arithmetic IF statements.

Modified Example:

```
C POSSIBLE RESPONSE
C RESTRICTED CONSTRUCT
  INTEGER*4 TEST(1000)
  REAL*4 W(1000),X(1000),Y(1000),Z(1000)

  DO 1000 I = 1,1000
    IF (TEST(I).LT.1 .OR. TEST(I).GT.3) THEN
      W(I) = 0.0
    ELSE IF (TEST(I).EQ.1) THEN
400   W(I) = X(I)
    ELSE IF (TEST(I).EQ.2) THEN
500   W(I) = Y(I)
    ELSE IF (TEST(I).EQ.3) THEN
600   W(I) = Z(I)
    ENDIF
1000  CONTINUE
```

Note that if parallel code is not generated for the transformed code, the resulting scalar program may run more slowly than the original program.

**ILX0314I Short Form: USER FUNCTION IN PARA
LOOP
Long Form: THE USER FUNCTION(S)
OR SUBROUTINE(S) <flist> ARE NOT
ANALYZABLE FOR VECTORIZATION.**

Explanation: Indicates that a loop contains a subroutine call or a reference to an external user-defined function.

Supplemental Data:

<flist> is a list consisting of function and subroutine names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements containing a subroutine call or a reference to an external user defined function are separated from the portions of the original loop that are eligible for vectorization analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

**ILX0315I Short Form: NON-MATHEMATICAL
IMPLICIT
Long Form: THE IMPLICITLY CALLED
NON-MATHEMATICAL SUBPROGRAM(S)
<flist> HAVE BEEN USED. THESE
SUBPROGRAMS ARE NOT ANALYZABLE
FOR PARALLELIZATION.**

Explanation: Indicates that some statement in the loop will generate a reference to an implicitly invoked character subprogram (CNCAT#) or to an implicitly invoked utility program (DSPAN#, DSPN2#, DSPN4#, or DYCMN#). These programs are described in *VS FORTRAN Version 2 Language and Library Reference*.

Supplemental Data:

<flist> is a list consisting of function names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements resulting in compiler call(s) to implicitly invoked character subprograms or to implicitly invoked utility programs are separated from the portions of the original loop that are eligible for parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0316I Short Form: ARRAY W/O SUBSCRIPTS IN I/O
Long Form: ARRAYS IN I/O WITHOUT SUBSCRIPTS ARE NOT ANALYZABLE FOR PARALLELIZATION.

Explanation: Indicates the presence of I/O statements containing arrays without subscripts (such as PRINT*,A) that cannot be analyzed for parallel code generation.

Example:

```
C EXAMPLE
C ARRAY W/O SUBSCRIPTS IN I/O
  REAL*4 A(800,800),B(800),C(800)

  DO 20 I=1,800
    READ(5,*) B,C(I)
    DO 10 J=1,800
      A(J,I) = B(J) + C(I)
10    CONTINUE
      S = S + A(I,1)
20    CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements containing full arrays are separated from the portions of the original loop that are eligible for parallel code generation analysis.

Modified Example:

```
C POSSIBLE RESPONSE
C ARRAY W/O SUBSCRIPTS IN I/O
  REAL*4 A(800,800),B(800),C(800)

  DO 10 I=1,800
    READ(5,*) B,C(I)
    DO 10 J=1,800
      A(J,I) = B(J) + C(I)
10    CONTINUE

  DO 20 I=1,800
    S = S + A(I,1)
20    CONTINUE
```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0317I Short Form: EQUIVALENCE OFFSET UNKNOWN
Long Form: THE EQUIVALENCE OFFSET(S) FOR VARIABLE(S) <vlist> COULD NOT BE ANALYZED FOR PARALLELIZATION. AN EQUIVALENCE GROUP IS IRREGULAR, OR HAS ARRAYS WITH DIFFERENT DATATYPES OR SCALAR VARIABLES. THE COMPILER ASSUMES THAT THE VARIABLES CARRY DEPENDENCES IN

LOOP(S) AT NESTING LEVEL(S)
<levlist>.

Explanation: To compute dependences between EQUIVALENCE variables, the following must be true:

- The variables must have the same element size.
- The computed equivalence offset for the variables must be a multiple of the element size.
- The variables must be array variables.

If any of these conditions are not true, the variables may be presumed to carry dependences and parallel code will not be generated for your loop.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They may, however, differ from the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C EQUIVALENCE OFFSET UNKNOWN
  REAL*8 A(1000),T
  REAL*4 B(1000)
  EQUIVALENCE (A,B,T)

  DO 10 I = 1000,1,-1
    A(I) = B(I) ** 2.1 * T
10    CONTINUE
```

Possible Response 1: Make the element sizes of the EQUIVALENCE variables the same. Also, replace the scalar variable with an array.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C EQUIVALENCE OFFSET UNKNOWN
  REAL*8 A(1000),B(1000),C(1)
  EQUIVALENCE (A,B,C)

  DO 10 I = 1000,1,-1
    A(I) = B(I) ** 2.1 * C(1)
10    CONTINUE
```

Possible Response 2: Insert the IGNORE RECRDEPS directive to instruct the compiler to assume that a dependence due to the EQUIVALENCE variables does not occur. Before using this directive, you should analyze the storage mapping and subscript expressions of the variables involved and make sure that the different variables do not reference identical storage locations while the loop is running.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C EQUIVALENCE OFFSET UNKNOWN
    REAL*8 A(1000),T
    REAL*4 B(1000)
    EQUIVALENCE (A,B,T)

*DIR    IGNORE RECRDEPS
        DO 10 I = 1000,1,-1
          A(I) = B(I) ** 2.1 * T
10      CONTINUE
```

ILX0318I Short Form: OFFSET UNKNOWN
Long Form: THE OFFSET NEEDED TO ADDRESS THE ARRAY(S) <vlist> COULD NOT BE ANALYZED FOR PARALLELIZATION. THERE MAY BE AN UNKNOWN TERM IN A SUBSCRIPT OR IN A LOOP LOWER BOUND, OR THE ARRAY(S) MAY HAVE ADJUSTABLE DIMENSIONS. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when some additive term in a subscript computation for a particular array is not an induction variable or a constant. It can also appear when the DO loop in which an array reference is contained has a variable lower bound. In these situations, recurrent dependences are presumed to exist between the statement in which the subscript computation is used and all other statements that reference the array.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```
C EXAMPLE 1
C OFFSET UNKNOWN - ADDITIVE TERM
    REAL*4 A(1000),B(1000)

    DO 10 I = 1,800,2
      A(I) = A(I+ISKIP) * B(I) ** 2.1
10    CONTINUE
```

Possible Response 1: Identify the expression or expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C OFFSET UNKNOWN - ADDITIVE TERM
    REAL*4 A(1000),B(1000)

    DO 10 I = 1,800,2
      A(I) = A(I+1) * B(I) ** 2.1
10    CONTINUE
```

Example 2:

```
C EXAMPLE 2
C OFFSET UNKNOWN - LOWER BOUND
    REAL*4 C(1000)

    DO 20 I = ISTART,1000
      C(I) = C(3) ** 2.1
20    CONTINUE
```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C OFFSET UNKNOWN - LOWER BOUND
    REAL*4 C(1000)

*DIR    IGNORE RECRDEPS
        DO 20 I = ISTART,1000
          C(I) = C(3) ** 2.1
20      CONTINUE
```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the dependences really do exist, wrong results may be produced. In this case, the IGNORE RECRDEPS directive is valid only if the value of the variable ISTART is greater than 3. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0319I Short Form: STRIDE UNKNOWN
Long Form: THE STRIDE NEEDED TO ADDRESS THE ARRAY(S) <vlist> COULD NOT BE ANALYZED FOR PARALLELIZATION, EITHER BECAUSE OF AN UNKNOWN MULTIPLIER IN THE SUBSCRIPT OR AN UNKNOWN LOOP INCREMENT. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when the multiplier of some induction variable within a subscript computation for a particular array is not a constant. It can also appear when the loop in which an array reference is contained has a variable increment value. In these situations, recurrent dependences are

presumed to exist between the statement in which the subscript computation is used and all other statements that reference the array.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C STRIDE UNKNOWN
  REAL*4 A(1000),B(1000)
  INTEGER*4 ISKIP

  I = 1
  DO 10 K = 1,800
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
10  CONTINUE
```

Possible Response 1: Identify the expression or expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C STRIDE UNKNOWN
  REAL*4 A(1000),B(1000)
  INTEGER*4 ISKIP
  PARAMETER (ISKIP=4)

  I = 1
  DO 10 K = 1,800
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
10  CONTINUE
```

Possible Response 2: If you can determine that the indicated dependences will not occur at run time, you can cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C STRIDE UNKNOWN
  REAL*4 A(1000),B(1000)
  INTEGER*4 ISKIP

  I = 1
*DIR  IGNORE RECRDEPS(A)
  DO 10 K = 1,800
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
10  CONTINUE
```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the ignored dependences really do exist, wrong results may be produced. (In this case, the dependence exists only if the value of the variable ISKIP is 0.) See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0320I Short Form: POSSIBLE RECURRENCE
Long Form: THIS LOOP COULD NOT BE PARALLELIZED BECAUSE OF A POSSIBLE RECURRENCE INVOLVING THE ARRAY(S) <alist>. THE INFORMATION NEEDED TO DETERMINE WHETHER OR NOT THE RECURRENCE EXISTS WAS NOT AVAILABLE TO THE COMPILER. IF NO RECURRENCE EXISTS, PARALLELIZATION CAN BE ACHIEVED WITH AN IGNORE DIRECTIVE.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible for parallel code to be generated for the loop. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until compile time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and parallel code generation is not performed. If the IGNORE directive is used, the recurrence is assumed to be absent. However, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies cases in which the existence of a recurrence cannot be determined at compile time.

Supplemental Data:

<alist> is a list of the names of arrays that are involved in the recurrence. (This list includes only the variables that could not be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```

C EXAMPLE
C POSSIBLE RECURRENCE
  REAL  A(2000),B(2000)

      DO 10 I = 1,1000
        A(I+N) = A(I) * B(I)
10    CONTINUE

```

In this example, a recurrence exists if an element of the array A that is stored on one iteration of the loop is referenced on some later iteration. This is true if the variable N has a value between 1 and 999.

Possible Response: If the existence of the recurrence depends on some values determined at run time, and if the value will never cause the recurrence to exist, you can use the IGNORE directive in conjunction with the PREFER PARALLEL directive to direct the compiler to generate parallel code for your program.

Modified Example:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C POSSIBLE RECURRENCE
  REAL  A(2000),B(2000)

*DIR  PREFER PARALLEL
      DO 10 I = 1,1000
        A(I+N) = A(I) * B(I)
10    CONTINUE

```

ILX0321I Short Form: EQUIVALENCE SCALAR USED
Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR PRIVATIZATION BECAUSE THEY ARE IN AN EQUIVALENCE GROUP. PARALLELIZATION IS INHIBITED.

Explanation: Parallel code cannot be generated for scalar variables that are in an EQUIVALENCE group and are modified in a loop, since they are not eligible for privatization.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for privatization.

Example:

```

C EXAMPLE
C EQUIVALENCED SCALAR USED
  REAL*4 X(1000),Y(1000),Z(1000),D(1000),S
  EQUIVALENCE (S,D(1000))

      DO 20 I = 1,1000
        S = X(I) + Y(I)
        Z(I) = S
20    CONTINUE

```

Possible Response: It is sometimes possible to avoid this situation simply by replacing references to the original scalar variables with references to a new scalar

variable that is never referenced outside the loop. Note that if the original scalar variable is needed later, you should be careful to make sure that it is assigned the correct value after the loop has completed.

Modified Example:

```

C EXAMPLE
C EQUIVALENCED SCALAR USED
  REAL*4 X(1000),Y(1000),Z(1000),D(1000),S,S_NEW
  EQUIVALENCE (S,D(1000))

      DO 20 I = 1,1000
        S_NEW = X(I) + Y(I)
        Z(I) = S_NEW
20    CONTINUE
    S = S_NEW

```

ILX0322I Short Form: DEFINITE RECURRENCE
Long Form: THIS LOOP COULD NOT BE PARALLELIZED BECAUSE OF A DEFINITE RECURRENCE INVOLVING THE VARIABLE(S) <vlist>.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible for the compiler to generate parallel code for the loop. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and parallel code generation is not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies the cases in which a recurrence definitely exists.

Supplemental Data:

<vlist> is a list of the names of the variables that are involved in the recurrence. (This list includes only those variables that could be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```

C EXAMPLE
C DEFINITE RECURRENCE
  REAL  A(1000),B(1000)

      DO 10 I = 1,999
        A(I+1) = A(I) * B(I)
10    CONTINUE

```

In this example, a recurrence definitely exists, since the value stored into the array A on each iteration of the

loop is used on the following iteration. Parallel code cannot be generated for this loop.

ILX0324I Short Form: OPTIMIZER INDUCED DEPENDENCE
Long Form: A COMPILER TEMPORARY INTRODUCED DURING SCALAR OPTIMIZATION HAS CAUSED ONE OR MORE DEPENDENCES IN THE LOOP(S) AT NESTING LEVEL(S) <levlist>. THESE LOOP(S) CANNOT BE PARALLELIZED.

Explanation: This message is produced when a statement becomes part of a recurrence solely because of some optimization that had been performed prior to parallel code generation (for example, common sub-expression elimination).

Supplemental Data:

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(1000),D(1000),F(1000)

      DO 100 J = 2,1000
        C(J) = D(J-1)
        D(J) = C(J)
        F(J) = D(J) + 2
100    CONTINUE
```

In the DO loop shown above, the first two statements form a recurrence so that the compiler cannot generate parallel code for them; the last statement would normally be eligible for parallel code generation. However, parallel code generation for this statement may be restricted due to some transformations that have been applied by the compiler prior to parallel code generation analysis.

In optimizing this loop, the compiler will attempt to reduce the number of load instructions that must be processed. As a result, during parallel code generation analysis, it will appear as if this loop were rewritten as:

```
      DO 100 J = 2,1000
        .temp = D(J-1)
        C(J) = .temp
        D(J) = .temp
        F(J) = .temp + 2
100    CONTINUE
```

where .temp is a compiler-generated scalar temporary. To generate parallel code for the last statement, split the original loop into two loops so that this statement is

separated from the other statements that are ineligible for parallel code generation. The presence of the scalar temporary shared by all the statements in the loop prohibits loop splitting and thus prevents partial parallel code generation.

Possible Response 1: Since the presence of statement labels inhibits optimization to some degree, it is sometimes possible to achieve partial parallel code generation in cases such as this simply by introducing additional labels.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(1000),D(1000),F(1000)

      DO 100 J = 2,1000
        C(J) = D(J-1)
        D(J) = C(J)
    99  F(J) = D(J) + 2
100    CONTINUE
```

Be careful when using this type of transformation because it may inhibit some important optimizations. If you make this change and partial parallel code generation still does not occur, the resulting scalar code may run more slowly than the original.

Possible Response 2: The transformation suggested above may or may not increase parallel code generation. If it is not effective, try to replace the original loop with two separate loops: one loop containing the portion that is not eligible for parallel code generation; the other loop containing the portion of the original loop that is eligible for parallel code generation.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(1000),D(1000),F(1000)

      DO 100 J = 2,1000
        C(J) = D(J-1)
        D(J) = C(J)
100    CONTINUE
      DO 101 J = 2,1000
        F(J) = D(J) + 2
101    CONTINUE
```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0325I Short Form: SUBSCRIPT TOO COMPLEX
Long Form: THE ARRAY(S) <alist> USE SUBSCRIPT COMPUTATIONS THAT COULD NOT BE ANALYZED FOR PARALLELIZATION. THEY MAY INCLUDE INDIRECT ADDRESSING, DATA CONVERSIONS, UNKNOWN STRIDES, OR AUXILIARY INDUCTION VARIABLES.

**THE COMPILER HAS ASSUMED THAT
THESE ARRAYS CARRY DEPENDENCES
IN LOOP(S) AT NESTING LEVEL(S)
<levlist>.**

Explanation: Indicates the use of subscript computations for which the compiler could not perform accurate dependence analysis. These cases include the constructs listed below. (In each of the examples given, K is an induction variable for some DO loop that contains the indicated array reference.)

- Indirect addressing, as in $A(\text{INDEX}(K))$, where INDEX is an array of integers.
- Subscripts requiring data conversions, as in $A(K+X)$, where X is a real variable.
- Subscripts where the stride is not known at compile time, as in $A(K*KSTEP)$, where KSTEP is an integer variable. (An unknown stride may also occur if the increment expression of some DO loop is not a compile-time constant.)
- Auxiliary induction variables, as in $A(\text{IVAR})$, where IVAR is incremented explicitly within the DO loop by some statement of the form $\text{IVAR}=\text{IVAR}+\text{INCR}$.

When an array uses any of the above types of subscript, recurrent dependences are often presumed to exist between the statement in which the subscript computation occurs and all other statements that reference the array.

This message often occurs when variables are used for the lower bound, upper bound, or increment of a loop or when variables that are not inductions are used inside of subscripts.

Supplemental Data:

<alist> is a list of the names of the arrays that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```
C EXAMPLE 1
C SUBSCRIPT TOO COMPLEX
  REAL*4 A(1000)
  INTEGER*4 INDEX(1000)

  DO 10 I = 1,1000
    A(INDEX(I)) = A(INDEX(I)) ** 2.1
10  CONTINUE
```

Possible Response 1: For cases involving indirect addressing, it may be possible to introduce additional arrays to hold intermediate results. The loop should be transformed as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

Do this only if you are absolutely certain that the noninductive subscript expression never selects any element more than once.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C SUBSCRIPT TOO COMPLEX
  REAL*4 A(1000),NEW_A(1000)
  INTEGER*4 INDEX(1000)

  DO 9 I = 1,1000
    NEW_A(I) = A(INDEX(I))
9  CONTINUE
  DO 10 I = 1,1000
    NEW_A(I) = NEW_A(I) ** 2.1
10 CONTINUE
  DO 11 I = 1,1000
    A(INDEX(I)) = NEW_A(I)
11 CONTINUE
```

Due to the overhead involved in copying data to and from the new version of the array, this transformation may not result in any performance benefits, even if parallel code generation is achieved. You should carefully analyze the performance of this code both before and after applying the transformation to make sure that it is worthwhile.

Example 2:

```
C EXAMPLE 2
C SUBSCRIPT TOO COMPLEX
  REAL*4 B(2000),X

  DO 20 I = 1,1000
    B(I+X) = B(I+X) ** 2.1
20  CONTINUE
```

Possible Response 2: For cases involving data conversions inside of subscript calculations, recode the computations so that all the inputs hold INTEGER values whenever possible.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C SUBSCRIPT TOO COMPLEX
  REAL*4 B(2000),X
  INTEGER*4 INT_X

  INT_X = X
  DO 20 I = 1,1000
    B(I+INT_X) = B(I+INT_X) ** 2.1
20  CONTINUE
```

Example 3:

```

C EXAMPLE 3
C SUBSCRIPT TOO COMPLEX
  REAL*4 C(2000),Y
  INTEGER*4 KSTEP

      DO 30 I = 1,1000
        KSTEP = KSTEP + INC
        C(KSTEP*I) = Y ** C(I*KSTEP)
30    CONTINUE

```

Possible Response 3: For cases involving unknown strides, identify the expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 3:

```

C POSSIBLE RESPONSE 3
C SUBSCRIPT TOO COMPLEX
  REAL*4 C(2000),Y
  INTEGER*4 KSTEP
  PARAMETER (KSTEP=4)

      DO 30 I = 1,1000
        C(KSTEP*I) = Y ** C(I*KSTEP)
30    CONTINUE

```

Example 4:

```

C EXAMPLE 4
C SUBSCRIPT TOO COMPLEX
  REAL*4 D(2000*2000)

      DO 40 J = 1,1000,N
        D(J) = D(J*J) ** 2.1
40    CONTINUE

```

Possible Response 4: For cases in which none of the previous transformations is appropriate, directives may be used to cause the compiler to assume that no dependence exists.

Modified Example 4:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 4
C SUBSCRIPT TOO COMPLEX
  REAL*4 D(2000*2000)

*DIR    IGNORE RECRDEPS(D)
*DIR    PREFER PARALLEL
      DO 40 J = 1,2000,N
        D(J) = D(J*J) ** 2.1
40    CONTINUE

```

Be careful when applying IGNORE RECRDEPS because if this directive is used and if the ignored dependences really do exist, wrong results may be produced. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive. Note that the PREFER PARALLEL directive is also required to direct the compiler to generate parallel code for the loop.

ILX0326I Short Form: SCALAR DEFINED BEFORE LOOP

Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR PARALLELIZATION BECAUSE THE VALUES OF THE VARIABLES MAY CHANGE DURING THE EXECUTION OF THE LOOP CONTAINING THE SCALAR VARIABLE(S) (OR CONTAINING LOOP). PROPER EXECUTION OF THE CODE FOLLOWING THE LOOP DEPENDS ON THE UNCHANGED, PRESET VALUES OF THE SCALAR VARIABLE(S).

Explanation: Parallel code generation for scalar variables that are modified within a loop requires a process known as privatization. This involves replacing the scalar variable with a private variable, and can only be performed on variables that can be proven to be local. You can prove a scalar variable is local to a given loop if the values that it holds during the processing of the loop could not have been set before the loop began processing and will never be used after the loop finishes processing.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for privatization.

Example:

```

C EXAMPLE
C SCALAR DEFINED BEFORE LOOP
  REAL*4 A(1000),B(1000),C(1000),T

      T = 1.1
      DO 11 I = 1,1000
        A(I) = T
        T = B(I) + C(I)
11    CONTINUE

```

Possible Response: It may be possible to replace the original scalar variable with an array whose dimension ranges from zero to the number of iterations of the loop in which the scalar resides. The loop should be transformed in the following manner:

- Prior to entering the loop, set the zero element of the new array to the value held by the scalar.
- Prior to the first statement that defines the scalar within the loop, replace all references to that scalar with references to the element of the new array whose position is one less than the current iteration count.
- All other references to the scalar within the loop should be replaced by references to the element of the array that corresponds to the current iteration count.
- Following the loop, set the scalar to the value held by the last element of the new array.

Modified Example:

```

C POSSIBLE RESPONSE
C SCALAR DEFINED BEFORE LOOP
  REAL*4 A(1000),B(1000),C(1000),T
  REAL*4 TT(0:1000)

  T = 1.1
  TT(0) = T
  DO 11 I = 1,1000
    A(I) = TT(I-1)
    TT(I) = B(I) + C(I)
11  CONTINUE
  T = TT(1000)

```

Note that this transformation is only valid if the first assignment to the original scalar variable is not a conditionally processed statement.

Also be aware that this transformation may not necessarily increase the the ability of the compiler to generate parallel code for your program, and could result in a scalar program that runs more slowly than the original.

ILX0327I Short Form: SCALAR NEEDED AFTER LOOP
Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR PARALLELIZATION BECAUSE THEY MAY BE SET TO VALUES THAT WILL BE USED AFTER THE EXECUTION OF THE CONTAINING LOOP.

Explanation: Parallel code generation for scalar variables that are modified within a loop requires a process known as privatization. This involves replacing the scalar with a private variable. This may not be possible in certain cases.

The cases where privatization is not done are where a scalar variable is *nonlocal* to a loop, and where that variable is referenced at different nesting levels or where it is modified by a conditionally executed statement. A variable is considered *nonlocal* to a loop if it is in COMMON or if it uses a value within the loop that will be used after the loop.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for privatization.

Example:

```

C EXAMPLE
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(1000),B(1000),T

  DO 22 I = 1,1000
    T = A(I)
    DO 22 J = 1,1000
      B(J,I) = T
22  CONTINUE
  ...
  WRITE(6,*) T

```

Possible Response 1: It may be possible to replace the original scalar variable with a new scalar variable that is local to the loop. If this is done, it will be necessary to insert an additional assignment after the loop to set the original scalar to its appropriate final value.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(1000),B(1000,1000),T
  REAL*4 T_LOCAL

  DO 22 I = 1,1000
    T_LOCAL = A(I)
    DO 22 J = 1,1000
      B(J,I) = T_LOCAL
22  CONTINUE
  T = A(1000)
  ...
  WRITE(6,*) T

```

Possible Response 2: It may be possible to replace the original scalar variable with an array whose dimension ranges from one to the number of iterations of the loop in which the scalar variable resides. The loop should be transformed in the following manner:

- If the first occurrence of the scalar variable within the loop is a reference rather than definition, prior to entering the loop, set the first element of the array to the value held by the scalar variable.
- Replace all occurrences of the scalar variable within the loop with references to the element of the array that corresponds to the current iteration count.
- Following the loop, set the scalar variable to the value held by the last element of the array.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(1000),B(1000,1000),T
  REAL*4 TT(1000)

  DO 22 I = 1,1000
    TT(I) = A(I)
    DO 22 J = 1,1000
      B(J,I) = TT(I)
22  CONTINUE
  T = TT(1000)
  ...
  WRITE(6,*) T

```

Note that this transformation is only valid if the first assignment to the original scalar variable is not a conditionally processed statement.

Also be aware that this transformation may not necessarily increase the ability of the compiler to generate parallel code for your program, and could result in a scalar program that runs more slowly than the original.

ILX0330I Short Form: UNKNOWN UPPER BOUND
Long Form: THE ARRAY(S) <alist> MAY OR MAY NOT BE INVOLVED IN DEPENDENCES, DEPENDING ON THE UPPER BOUND OF SOME CONTAINING LOOP. SINCE THE UPPER BOUND IS NOT KNOWN, THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCE(S) IN LOOP(S) AT NESTING LEVEL(S) <levlist> AND CANNOT BE PARALLELIZED.

Explanation: This message occurs when a variable is specified as the upper bound of a loop and when the existence of a dependence depends on the value of the upper bound. In these cases, dependence will always be assumed.

Supplemental Data:

<alist> is a list of the names of the arrays that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```

C EXAMPLE
C UNKNOWN UPPER BOUND
  REAL*4 A(-1000:1000)

  DO 10 I = 1,N
    A(I) = A(I-1000) * 22.1
10  CONTINUE

```

Possible Response 1: Identify the loop that carries the dependence and replace the upper bound of the loop with a compile-time constant, if possible.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C UNKNOWN UPPER BOUND
  REAL*4 A(-1000:1000)
  PARAMETER (N=1000)

  DO 10 I = 1,N
    A(I) = A(I-1000) * 22.1
10  CONTINUE

```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C UNKNOWN UPPER BOUND
  REAL*4 A(-1000:1000)

*DIR  IGNORE RECRDEPS
  DO 10 I = 1,N
    A(I) = A(I-1000) * 22.1
10  CONTINUE

```

Care should be used in applying IGNORE RECRDEPS since if this directive is used and if the dependences really do exist, wrong results may be produced. In this case, the directive is correct only if the value of the variable N is always less than 21. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0335I Short Form: UNSUPPORTABLE DEPENDENCE
Long Form: THIS CODE IS CONSIDERED UNSUPPORTABLE FOR PARALLELIZATION BECAUSE IT IS LINKED TO SOME UNSUPPORTABLE STATEMENT(S) THROUGH MUTUAL DEPENDENCES.

Explanation: This message is produced when a statement does not contain any unsupportable constructs but is forced into an unsupportable loop because it is tied to some other unsupportable

statement through recurrent dependences. These dependences can come about in a number of ways:

- The indicated statement may use a scalar variable that is also used in some other statement that contains an unsupportable construct.
- There may be some control flow that creates a control dependence that involves both the indicated statement and some unsupportable statement.
- There may be a dependence between the indicated statement and some unsupportable statement that came about because the two statements share some common subexpression.

Example:

```
C EXAMPLE
C UNSUPPORTABLE DEPENDENCE
      REAL*8 A(1000),B(1000),C(1000),D

      DO 10 I = 1,1000,1
        PRINT*, B(I)*C(I)
        A(I) = B(I)*C(I)
10     CONTINUE
```

In this example, the first statement uses unsupportable I/O and therefore is not eligible for parallel code generation. Under normal conditions, parallel code could be generated for the second statement. The two statements share a common subexpression, $B(I)*C(I)$, which is assigned to a scalar temporary by the compiler. To generate parallel code for the second statement, it would be necessary to split this loop into two separate loops, an action prevented by the presence of a scalar temporary.

Possible Response: If possible, replace the unsupportable part of the loop with equivalent supportable constructs.

Otherwise, try to separate the statements that are eligible for parallel code generation which are linked to unsupportable statements, by restructuring the original loop into two or more loops.

Modified Example:

```
C POSSIBLE RESPONSE
C UNSUPPORTABLE DEPENDENCE
      REAL*4 A(1000),B(1000),C(1000)

      DO 10 I = 1,1000,1
        PRINT*, B(I)*C(I)
10     CONTINUE

      DO 11 I = 1,1000,1
        A(I) = B(I)*C(I)
11     CONTINUE
```

Use this type of restructuring only if you are absolutely certain that it will not alter the results of your program.

ILX0341I Short Form: I/O OPERATIONS

Long Form: I/O OPERATIONS ARE NOT SUPPORTED FOR PARALLELIZATION.

Explanation: Indicates the presence of I/O statements in a loop. The loop may have been analyzed and found eligible for partial parallel code generation; however, the I/O operation remains an unsupported operation for parallel code generation.

Example:

```
C EXAMPLE
C I/O OPERATIONS
      REAL*4 A(1000),B(1000)

CDIR PREFER PARALLEL
      DO 10 I = 1,1000
        A(I) = B(I) * 3.3
        PRINT*, A(I)
10     CONTINUE
```

Analysis Output:

```
C OUTPUT FROM PARALLEL ANALYSIS
C I/O OPERATIONS
      REAL*4 A(1000),B(1000)

PARA +----- DO 10 I = 1,1000
      |          A(I) = B(I) * 3.3

UNSP +----- DO 10 I = 1,1000
      |          PRINT*, A(I)
```

ILX0346I Short Form: UNSUPPORTED CONSTRUCT

Long Form: NO PARALLEL SUPPORT EXISTS FOR THIS OCCURRENCE OF THE <flist> CONSTRUCT.

Explanation: Indicates parallel code cannot be generated for a particular occurrence of a MAX or MIN intrinsic function reference, because of the complexity or ordering of its arguments.

Supplemental Data:

<flist> is the name of the function involved.

Possible Response: This happens when a scalar variable appears both on the left side of the equal sign and as a MAX and MIN intrinsic function argument in the same Fortran statement.

If it is possible, simplify and reorder the arguments. Try to make the scalar variable appear as the first argument of the MAX or MIN reference.

Example:

The compiler will not generate parallel code:

```
CC = MAX(MAX(A(I),CC),2.0,B(I))
```

The compiler will generate parallel code:

```
CC = MAX(CC,A(I),2.0,B(I))
```

ILX0348I Short Form: SERIAL FASTER THAN PARALLEL
Long Form: CODE THAT WAS ELIGIBLE TO EXECUTE IN PARALLEL MODE WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SERIAL.

Explanation: Identifies loops that were eligible for parallel code generation but for which parallel code was not generated because the compiler has determined that the cost of parallel code generation exceeds the cost of scalar processing.

Example:

```
C EXAMPLE
C SERIAL FASTER THAN PARALLEL
  REAL*4 A(1000)

      DO 10 I = 1,10
        A(I) = A(I) ** 2.1
10     CONTINUE
```

Possible Response: In some cases, the estimates used by the compiler for comparing parallel and serial run times may not be accurate. If you find that the compiler has misjudged the profitability of parallelizing a particular loop, you can use the PREFER PARALLEL directive to override the decision to compile this loop for serial processing.

Modified Example:

```
@PROCESS DIRECTIVE ('DIR')
C EXAMPLE
C SERIAL FASTER THAN PARALLEL
  REAL*4 A(1000)

*DIR    PREFER PARALLEL
      DO 10 I = 1,10
        A(I) = A(I) ** 2.1
10     CONTINUE
```

Note: Do not rely on intuition to determine whether PREFER PARALLEL should be used. Always verify the appropriateness of its use by taking direct measurements of the run times of the affected loop, both with and without the use of the directive.

ILX0349I Short Form: PARALLELIZED EXTERNAL REF
Long Form: PARALLELIZATION WAS PERFORMED ON THE USER EXTERNAL SUBPROGRAM REFERENCE(S) <flist>.

Explanation: Indicates when a user external subprogram reference is processed for parallel code generation.

Supplemental Data:

<flist> is a list consisting of the user external subprogram reference(s) and the ISNs (internal statement numbers) of the statements in which they are used.

Example:

```
C EXAMPLE
C PARALLELIZED EXTERNAL REF
  REAL*4 A(1000)

*DIR PREFER PARALLEL
*DIR IGNORE CALLDEPS
      DO 10 I = 1,1000
        CALL SUB1(A(I))
10     CONTINUE
```

ILX0350I Short Form: PARALLEL SUM REDUCTION
Long Form: PARALLELIZATION WAS DONE USING SUM OR PRODUCT REDUCTION ON THE VARIABLE(S) <vlist>. RESULTS MAY DIFFER FROM ONE EXECUTION TO THE NEXT EXECUTION.

Explanation: Indicates when parallel code is generated for a statement using a sum reduction operation. Parallel code generation can produce different results than serial. Because parallel code generation cannot be performed in serial order, inherent variations in the data that is accumulated cause the results to be dependent on the order of accumulation, so that results may differ from one run of the program to another.

Supplemental Data:

<vlist> is a list of names of variables that are used as accumulators in statements for which parallel code was generated via sum reduction.

Example:

```
C EXAMPLE
C PARALLEL SUM REDUCTION
  REAL*4 SUM, A(1000)

      SUM = 0.0
      DO 10 I = 1,1000
        SUM = SUM + A(I)
10     CONTINUE
```


Possible Response: If there is concern for the consistency of results between programs for which parallel code is generated and programs compiled for serial processing, the parallel code generation for reduction functions can be inhibited by specifying the PARALLEL(NOREDUCTION) option on an @PROCESS card or when invoking the compiler.

ILX0355I Short Form: PARALLELIZED IF LOOP
Long Form: IF LOOP PARALLELIZED BY PROCESSING AS A DO LOOP.

Explanation: Indicates when an IF loop has been compiled for parallel code generation as a DO loop.

Example:

```
C EXAMPLE
C PARALLELIZED IF LOOP
  INTEGER N,I
  REAL  A(*),B(*),C(*)

  I = 0
140  CONTINUE
  I = I + 1
  IF (I.GT.N) GOTO 141
  A(I) = B(I) + C(I)
  GOTO 140
141  CONTINUE
```

Parallel code will be generated for the above IF loop since it is processed as a DO loop.

ILX0356I Short Form: CONDITIONAL PARALLEL STATEMENT
Long Form: PARALLELIZATION WAS PERFORMED ON CONDITIONALLY EXECUTED CODE. THE PARALLEL REPORT LISTING MAY FAIL TO INDICATE THE BRANCH STATEMENT(S) THAT AFFECT THE EXECUTION OF THIS REGION.

Explanation: Indicates when a statement or group of statements are processed conditionally. This message is intended to help clarify the program listing that is produced by the PARALLEL(REPORT(XLIST)) option. This is necessary because information about the branch structure in a loop is not always reproduced when that loop is printed in the report.

In particular, the only conditional statements that are printed are logical IF statements that perform conditional assignments. No branch statements and no block IF constructs appear in the report.

Refer to the source listing or to the listing produced by the PARALLEL(REPORT(SLIST)) option to identify the control flow constructs that can affect the way the code being flagged by this message runs.

Example:

```
C EXAMPLE
C CONDITIONAL PARALLEL STATEMENT
  REAL*4 A(1000),B(1000)

  DO 10 I = 1,1000
    IF (A(I).LT.0.0) GOTO 5
    B(I) = 1.1
    GOTO 6
  5    B(I) = 2.2
  6    A(I) = 0.0
  10   CONTINUE
```

The GOTO statement at ISN 6 in the above loop will not be printed in the compiler report output produced by the PARALLEL(REPORT(XLIST)) option.

ILX0358I Short Form: CONDITIONAL SERIAL CODE
Long Form: THIS CODE IS CONDITIONALLY EXECUTED. THE PARALLEL REPORT LISTING MAY FAIL TO INDICATE THE BRANCH STATEMENT(S) THAT AFFECT THE EXECUTION OF THIS REGION.

Explanation: Indicates when a statement or group of statements that have been compiled for parallel code generation is part of a conditionally processed region of code. This message is intended to help clarify the program listing produced by the PARALLEL(REPORT(XLIST)) option. This is necessary because information about the branch structure in a loop is not always reproduced when that loop is printed in the report.

In particular, the only conditional statements that are printed are logical IF statements that perform conditional assignments. No branch statements and no block IF constructs appear in the report.

Refer to the source listing or to the listing produced by the PARALLEL(REPORT(SLIST)) option to identify the control flow constructs that can affect the way the code being flagged by this message runs.

Example:

```
C EXAMPLE
C CONDITIONAL SERIAL CODE
  REAL*4 A(1000),B(1000)

  DO 10 I = 2,999
    IF (A(I-1).LT.0.0) GOTO 5
    B(I) = B(I-1)
    GOTO 6
  5    B(I) = B(I+1)
  6    A(I) = B(I)
  10   CONTINUE
```

The GOTO statement at ISN 6 in the above loop will not be printed in the compiler report output produced by the PARALLEL(REPORT(XLIST)) option.

ILX0359I Short Form: IF LOOP PROCESSED AS DO LOOP
Long Form: THIS IF LOOP HAS BEEN PROCESSED AS DO LOOP BY THE COMPILER SO THAT IT IS ELIGIBLE FOR PARALLELIZATION ANALYSIS.

Explanation: Indicates when an IF loop has been processed internally as a DO loop, but will run in serial mode. This may be caused by a recurrence, unsupportable construct, and/or economic reasons.

Example:

```
C EXAMPLE
C IF LOOP PROCESSED AS DO LOOP
  INTEGER I
  REAL A(1000)

  I = 1
500 IF (I .GT. 1000) GOTO 505
  PRINT *, 'AT ITERATION ', I, A(I)
  I = I + 1
  GOTO 500
505 CONTINUE
```

ILX0360I Short Form: I/O OPERATION
Long Form: THE LANGUAGE CONSTRUCT "PARALLEL DO" OR "PARALLEL SECTIONS" HAS BEEN SERIALIZED BECAUSE OF CERTAIN I/O OPERATIONS AT ISN(S) <ilist>.

Explanation: Indicates that the language construct PARALLEL DO or PARALLEL SECTIONS has been marked for serial processing because of one or more I/O statements that cannot be analyzed. The I/O statements other than READ, WRITE, or PRINT are not analyzable.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```
C EXAMPLE
C I/O OPERATION
  REAL*4 A(1000),B(1000)

  PARALLEL DO 10 I = 1,1000,1
    A(I) = B(I) ** 2.1
  IF (I.LT.1000) GO TO 10
  BACKSPACE(05,ERR=100)
  READ(5,*) B
10 CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements are separated from the portions of the original loop that are eligible for parallel analysis.

Modified Example:

```
C POSSIBLE RESPONSE
C I/O OPERATION
  REAL*4 A(1000),B(1000)

  PARALLEL DO 10 I = 1,1000,1
    A(I) = B(I) ** 2.1
10 CONTINUE
  DO 20 I = 1,1000,1
    IF (I.LT.1000) GO TO 10
    BACKSPACE(05,ERR=100)
    READ(5,*) B
20 CONTINUE
```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0361I Short Form: "PREFER CHUNK" USED
Long Form: THE RECOMMENDED NUMBER OF SUBDIVISIONS FOR THIS LOOP WAS SPECIFIED AS <n> BY A "PREFER CHUNK" DIRECTIVE FOR PARALLELIZATION ANALYSIS.

Explanation: This message identifies loops for which PREFER CHUNK directives have been specified. The run-time environment will try to use the number of loop subdivisions recommended in this directive when it determines how many chunks to use for the loop to gain favorable performance.

Supplemental Data:

<n> is the value of the recommended number of loop subdivisions.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER CHUNK USED
  REAL*4 A(12800,12800)

  *DIR PREFER CHUNK(32)
  DO 10 I = 1,12800
    A(I,I) = A(I,I) ** 2.1
10 CONTINUE
```

Possible Response: If the program or data has undergone revisions since PREFER CHUNK was initially coded, it may be necessary to verify its correctness.

To do this, conduct a run-time analysis of the loop to determine whether the number that, when specified on the PREFER CHUNK directive, results in the most favorable performance.

ILX0362I Short Form: "PREFER MINIMUM CHUNK" USED
Long Form: THE MINIMUM RECOMMENDED NUMBER OF SUBDIVISIONS FOR THIS LOOP WAS SPECIFIED AS <n> BY A "PREFER CHUNK" DIRECTIVE FOR PARALLELIZATION ANALYSIS.

Explanation: This message identifies loops for which PREFER CHUNK directives have been specified. The run-time environment will try to use the minimum number of loop subdivisions recommended in this directive when it determines how many chunks to use for the loop to gain favorable performance.

Supplemental Data:

<n> is the value of the minimum recommended number of loop subdivisions.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER MINIMUM CHUNK USED
  REAL*4 A(12800,12800)

*DIR PREFER CHUNK(1:)
  DO 10 I = 1,12800
    A(I,I) = A(I,I) ** 2.1
  10 CONTINUE
```

Possible Response: If the program or data has undergone revisions since PREFER CHUNK was initially coded, it may be necessary to verify its correctness.

To do this, conduct a run-time analysis of the loop to determine whether the number that, when specified on the PREFER CHUNK directive, results in the most favorable performance.

ILX0363I Short Form: "PREFER MAXIMUM CHUNK" USED
Long Form: THE MAXIMUM RECOMMENDED NUMBER OF SUBDIVISIONS FOR THIS LOOP WAS SPECIFIED AS <n> BY A "PREFER CHUNK" DIRECTIVE FOR PARALLELIZATION ANALYSIS.

Explanation: This message identifies loops for which PREFER CHUNK directives have been specified. The run-time environment will try to use the maximum number of loop subdivisions recommended in this directive when it determines how many chunks to use for the loop to gain favorable performance.

Supplemental Data:

<n> is the value of the minimum recommended number of loop subdivisions.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER MAXIMUM CHUNK USED
  REAL*4 A(12800,12800)

*DIR PREFER CHUNK(:64)
  DO 10 I = 1,12800
    A(I,I) = A(I,I) ** 2.1
  10 CONTINUE
```

Possible Response: If the program or data has undergone revisions since PREFER CHUNK was initially coded, it may be necessary to verify its correctness.

To do this, conduct a run time analysis of the loop to determine whether the number that, when specified on the PREFER CHUNK directive, results in the most favorable performance.

ILX0365I Short Form: "PREFER SERIAL" USED
Long Form: THIS LOOP WILL BE EXECUTED IN SERIAL BECAUSE OF THE USE OF A "PREFER SERIAL" DIRECTIVE.

Explanation: This message identifies loops to which PREFER SERIAL directives have been applied. When this directive is specified, parallel code is not generated for a loop that is considered eligible for parallel code generation by the compiler. Use this directive only when an analysis of the run-time performance of the loop determines that the loop runs faster when the directive is present.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER SERIAL USED
  REAL*4 A(12800)

*DIR PREFER SERIAL
  DO 10 I = 1,12800
    A(I) = A(I) ** 2.1
  10 CONTINUE
```

Possible Response: If the program has been modified since the directive was initially coded, or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First check whether the loop would be chosen for serial processing if the directive were not enabled. If so, the directive is redundant and should be removed.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether the performance is better when this directive is used.

ILX0367I Short Form: "PREFER PARALLEL" USED
Long Form: A "PREFER PARALLEL" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP.

Explanation: This message identifies loops to which PREFER PARALLEL directives have been successfully applied. (PREFER PARALLEL is successful only if the loop to which it is applied is eligible for parallel code generation.) Use the directive only when an analysis of the run-time performance of the loop determined that the loop runs faster when the directive is present.

Note that after loop distribution has taken place, it is possible that a PREFER PARALLEL directive is successful for part of the code within a loop, but is unsuccessful for the rest.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER PARALLEL USED
  REAL*4 A(1000,1000)

*DIR    PREFER PARALLEL
        DO 10 I = 1,N
          A(I,I) = A(I,I) * 2.1
10      CONTINUE
```

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First check whether the loop would be chosen for parallel code generation if the directive were not enabled. If so, the directive is redundant and should be removed.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0368I Short Form: INAPPLCBL "PREFER PARALLEL"
Long Form: A "PREFER PARALLEL" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP BUT COULD NOT BE HONORED BECAUSE THE LOOP WAS NOT ELIGIBLE FOR PARALLELIZATION. THE DIRECTIVE HAS BEEN IGNORED.

Explanation: This message identifies loops for which inapplicable PREFER PARALLEL directives have been

specified. (PREFER PARALLEL is inapplicable if the loop contains a recurrence or an unsupportable construct.)

Note that after loop distribution has taken place, it is possible that a PREFER PARALLEL directive is inapplicable for part of the code within a loop, but is successful for the rest.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C INAPPLICABLE PREFER PARALLEL
  REAL*4 A(1000,1000),B(1000)

*DIR    PREFER PARALLEL
        DO 10 I = 2,1000
          A(I,I) = A(I,I) * 2.1
          B(I) = B(I-1)
10      CONTINUE
```

Parallel code cannot be generated for the second statement in this loop because it forms a recurrence. Therefore, the PREFER PARALLEL directive cannot be applied to this statement. The directive is still applicable to the first statement in the loop.

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

It should first be determined whether the directive has any effect on the parallel code generation for other parts of the original loop. If not, it is useless and should be removed from the program.

Otherwise, analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0370I Short Form: "ASSUME COUNT" USED
Long Form: THE ITERATION COUNT OF THIS LOOP WAS SPECIFIED AS "<n>" BY AN "ASSUME COUNT" DIRECTIVE FOR PARALLELIZATION ANALYSIS.

Explanation: This message identifies loops for which successful ASSUME COUNT directives have been specified. (ASSUME COUNT is successful if the iteration count of the loop to which it applies cannot be determined at compile time.) This directive is used to help the compiler decide whether parallel code generation for a particular loop will result in a performance improvement.

Compiler Report Diagnostic Messages—Vector and Parallel

The following messages apply to errors or conditions that arise when the compiler attempts to vectorize and generate parallel code for your program.

ILX0501I Short Form: NON-INTEGER LOOP CONTROL
Long Form: A LOOP CONTROL PARAMETER IS NOT INTEGER*4. THE LOOP IS NOT ANALYZABLE FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates that a variable that is not INTEGER*4 is used as part of an expression controlling the iteration of a loop or as a DO loop variable.

Modified Example: If the lower bound, upper bound, or increment expressions are not INTEGER*4, replace these expressions with INTEGER*4 variables that have been assigned the appropriate values prior to the loop.

If the DO loop variable is not INTEGER*4 and it can be replaced by one that is, do so.

ILX0502I Short Form: MORE THAN 8 NESTED LOOPS
Long Form: PARALLELIZATION AND VECTORIZATION ANALYSIS IS RESTRICTED TO LOOPS AT THE INNERMOST EIGHT LEVELS OF NESTING.

Explanation: Indicates that a loop has not been considered for parallel code generation or vectorization because it contains nested loops more than eight levels deep.

ILX0503I Short Form: NESTED LOOP NOT ANALYZABLE
Long Form: SOME NESTED LOOP WAS FOUND TO BE UNANALYZABLE FOR PARALLELIZATION AND VECTORIZATION.

Explanation: Indicates that a loop contains a nested loop which was not eligible for parallel or vector analysis.

Example:

```
C EXAMPLE
C NESTED LOOP NOT ANALYZABLE
  REAL*4 A(1000,1000)

  DO 10 I = 1,1000
    DO 10 J = 1,1000
      A(I,J) = A(I,J) ** 2.1
      WRITE(6,*,ERR=999) A(I,J)
10  CONTINUE
```

In this case, the inner loop is unanalyzable because it contains an I/O statement that is unanalyzable. The

outer loop is marked as unanalyzable since it surrounds the unanalyzable inner loop.

Possible Response: Identify the loop causing the rejection and attempt to recode it to eliminate the problem.

Modified Example:

```
C POSSIBLE RESPONSE
C NESTED LOOP NOT ANALYZABLE
  REAL*4 A(1000,1000)

  DO 10 I = 1,1000
    DO 10 J = 1,1000
      A(I,J) = A(I,J) ** 2.1
10  CONTINUE
  WRITE(6,*,ERR=999) A
```

ILX0504I Short Form: I/O OPERATION
Long Form: CERTAIN I/O STATEMENTS AT ISN(S) <ilist> ARE NOT ANALYZABLE FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates that a loop contains one or more I/O statements that cannot be analyzed. The following I/O statements are not analyzable:

- Statements other than READ, WRITE or PRINT
- READ and WRITE statements containing END= or ERR= labels
- Statements that specify a NAMELIST
- Statements containing arrays without subscripts
- Statements that specify internal files
- Asynchronous I/O statements
- Implied-DO statements (Some implied-DO statements may be analyzable).

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```
C EXAMPLE
C I/O OPERATION
  REAL*4 A(1000),B(1000)

  DO 10 I = 1,1000,1
    A(I) = B(I) * 3.3
    WRITE(6,ID=J) A(I)...A(I+999)
10  CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements are separated from the portions of the original loop that are eligible for parallel or vector analysis.

Modified Example:

```

C POSSIBLE RESPONSE
C I/O OPERATION
  REAL*4 A(1000),B(1000)

  DO 10 I = 1,1000,1
    A(I) = B(I) * 3.3
10  CONTINUE

  DO 20 I = 1,1000,1
    WRITE(6,ID=J) A(I)...A(I+999)
20  CONTINUE

```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0505I Short Form: DO WHILE OR IMPLIED DO
Long Form: ONE OR MORE DO WHILE
LOOPS OR I/O STATEMENTS WITH
IMPLIED DO LOOPS AT ISN(S) <ilist>
ARE NOT ANALYZABLE FOR
PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates the presence of an unanalyzable loop construct contained within an iterative DO loop. This may be caused by either a DO WHILE statement or by an I/O statement that contains an implied DO loop.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C DO WHILE OR IMPLIED DO
  REAL A(1000,1000),B(1000)

  DO 10 I=1,1000
    A(I,5) = B(I) * 3.3
    WRITE(6,*) (A(I,J),J=1,1000)
10  CONTINUE

```

Possible Response: Break the loop into two or more loops, so that any unanalyzable constructs are separated from the portions of the original loop that are analyzable.

Modified Example:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C DO WHILE OR IMPLIED DO
  REAL A(1000,1000),B(1000)

  DO 10 I=1,1000
    A(I,5) = B(I) * 3.3
10  CONTINUE

  DO 20 I=1,1000
    WRITE(6,*) (A(I,J),J=1,1000)
20  CONTINUE

```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0506I Short Form: CHARACTER DATA
Long Form: ONE OR MORE
STATEMENTS USING CHARACTER
DATA OCCUR AT ISN(S) <ilist> AND ARE
UNANALYZABLE FOR
PARALLELIZATION AND
VECTORIZATION.

Explanation: Indicates the presence of character data.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Possible Response: Break the loop into two or more loops, so that any statements that reference character data are separated from the portions of the original loop that are eligible for vectorization and parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0507I Short Form: UNANALYZABLE STOP OR
RETURN
Long Form: STOP OR RETURN
STATEMENTS AT ISN(S) <ilist> ARE NOT
ANALYZABLE FOR VECTORIZATION OR
PARALLELIZATION BECAUSE THIS
LOOP CONTAINS A NESTED LOOP.

Explanation: Loops that contain STOP or RETURN statements are vectorizable only if they are innermost loops and are never eligible for parallel code generation. This message indicates that a loop was rejected because it is an outer loop that contains a STOP or RETURN statement.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```

C EXAMPLE
C UNANALYZABLE STOP OR RETURN
  REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)

  DO 20 I = 1,1000
    DO 10 J = 1,1000
      IF (C(I,J) .EQ. 0.0) STOP
      A(I,J) = B(I,J)
10  CONTINUE
20  CONTINUE

```

In this case, the inner loop is eligible for vectorization although the outer loop is not. Neither loop is eligible for parallel code generation.

Possible Response: You may be able to make the outer loops eligible for partial vectorization and parallel code generation if you restructure the code to separate the loop termination test from the rest of the loop. The original nest of loops can be transformed as follows:

1. Insert a new nest of loops to evaluate the loop termination condition and determine the number of times each loop iterates before terminating. Note that the code within the new nest will remain ineligible for vectorization and parallel code generation.
2. Rewrite the original nest to process only the elements that are referenced before the loop terminates. Note that the number of iterations for the inner loop is different on the final iteration of the outer loop than it is on earlier iterations. You will need to duplicate the inner loop to process the final iteration separately.
3. Add a test at the end of the modified code to determine whether the STOP or RETURN statement should be run.

Modified Example:

```
C POSSIBLE RESPONSE
C UNANALYZABLE STOP OR RETURN
  REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)
  INTEGER IMAX,JMAX
  LOGICAL NEED_STOP

  DO 20 IMAX = 1,1000
    DO 10 JMAX = 1,1000
      IF (C(IMAX,JMAX) .EQ. 0.0) GOTO 100
10    CONTINUE
20  CONTINUE
100 NEED_STOP = (JMAX.LE.1000)

  DO 40 I = 1,IMAX-1
    DO 30 J = 1,1000
      A(I,J) = B(I,J)
30    CONTINUE
40  CONTINUE

  IF (NEED_STOP) THEN
    DO 50 J = 1,JMAX - 1
      A(IMAX,J) = B(IMAX,J)
50    CONTINUE
    STOP
  ENDIF
```

Note that this transformation will be valid only if the test that determines whether the loop should terminate is independent of the values that are being computed in the loops. Also note that if the test is not the first statement of that loop, some special processing may be necessary.

ILX0508I Short Form: BRANCH AROUND INNER LOOP
Long Form: THE BRANCH(ES) ORIGINATING AT ISN(S) <ilist> BYPASS ONE OR MORE NESTED LOOPS. THE LOOP(S) CONTAINING THE BRANCH(ES) ARE NOT ANALYZABLE FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates that a loop was rejected because some branch within the loop causes an inner loop to be bypassed.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```
C EXAMPLE
C BRANCH AROUND INNER LOOP
  REAL*4 A(1000),B(1000)

  DO 6 I = 1,1000
    A(I) = A(I) / 2.0
    IF (A(I) .EQ. 0.0) GO TO 5
    DO 4 J = 1,1000
      B(J) = B(J) ** 2.1
4    CONTINUE
5    CONTINUE
6    CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any unanalyzable branches are separated from the portions of the original loop that are eligible for vectorization or parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

Modified Example:

```
C POSSIBLE RESPONSE
C BRANCH AROUND INNER LOOP
  REAL*4 A(1000),B(1000)

  DO 3 I = 1,1000
    A(I) = A(I) / 2.0
3  CONTINUE
C
  DO 6 I = 1,1000
    IF (A(I) .EQ. 0.0) GO TO 5
    DO 4 J = 1,1000
      B(J) = B(J) ** 2.1
4    CONTINUE
5    CONTINUE
6    CONTINUE
```

ILX0509I Short Form: UNANALYZABLE EXIT BRANCH
Long Form: EXIT BRANCHES ORIGINATING AT ISN(S) <ilist> ARE NOT ANALYZABLE FOR VECTORIZATION OR PARALLELIZATION BECAUSE THIS LOOP CONTAINS A NESTED LOOP.

Explanation: Loops that contain EXIT BRANCH statements are eligible for vectorization only if they are innermost loops and are never eligible for parallel code generation. This message indicates that a loop was rejected because it is an outer loop that contains an EXIT BRANCH statement.

Supplemental Data:

<ilist> is a list of ISNs (internal statement numbers) that indicate the locations of the statement or statements responsible for the rejection.

Example:

```
C EXAMPLE
C UNANALYZABLE EXIT BRANCH
  REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)

  DO 20 I = 1,1000
    DO 10 J = 1,1000
      IF (C(I,J) .EQ. 0.0) GOTO 500
      A(I,J) = B(I,J)
10    CONTINUE
20    CONTINUE
    ...
500  CONTINUE
```

In this case, the inner loop is eligible for vectorization, although the outer loop is not. Neither loop is eligible for parallel code generation

Possible Response: You may be able to make the outer loop eligible for partial vectorization and parallel code generation if you restructure the code to separate the loop termination test from the rest of the loop. The original nest of loops can be transformed as follows:

1. Insert a new nest of loops to evaluate the loop termination condition and determine the number of times each loop iterates before terminating. Note that the code within the new nest will remain ineligible for vectorization and parallel code generation.
2. Rewrite the original nest to process only the elements that are referenced before the loops terminate. Note that the number of iterations for the inner loop is different on the final iteration of the outer loop than it is on earlier iterations. You will need to duplicate the inner loop to process the final iteration separately.
3. Add a test at the end of the modified code to determine whether the original branch should be taken.

Modified Example:

```
C POSSIBLE RESPONSE
C UNANALYZABLE EXIT BRANCH
  REAL*4 A(1000,1000),B(1000,1000),C(1000,1000)
  INTEGER IMAX,JMAX
  LOGICAL NEED_BRANCH

  DO 20 IMAX = 1,1000
    DO 10 JMAX = 1,1000
      IF (C(IMAX,JMAX) .EQ. 0.0) GOTO 100
10    CONTINUE
20    CONTINUE
100  NEED_BRANCH = (JMAX.LE.1000)

  DO 40 I = 1,IMAX-1
    DO 30 J = 1,1000
      A(I,J) = B(I,J)
30    CONTINUE
40    CONTINUE

  IF (NEED_BRANCH) THEN
    DO 50 J = 1,JMAX - 1
      A(IMAX,J) = B(IMAX,J)
50    CONTINUE
      GOTO 500
    ENDIF
    ...
500  CONTINUE
```

Note that this transformation will only be valid if the test for the loop exit branch is independent of the values that are being computed in the loops. Also note that if the loop exit branch is not the first statement of that loop, some special processing may be necessary.

ILX0510I Short Form: LOOP NOT OPTIMIZABLE
Long Form: PARALLELIZATION AND VECTORIZATION ARE INHIBITED BECAUSE THIS LOOP IS NOT OPTIMIZABLE. THIS MAY BE CAUSED BY AN INDUCTION VARIABLE THAT MAY BE RESET INSIDE THE LOOP OR BY COMPLEX BRANCHING OUTSIDE THE LOOP.

Explanation: Indicates situations where optimization, vectorization, and parallel code generation for loops are inhibited. This can happen for a variety of reasons:

- When DO loop variables are not guaranteed to behave like standard DO loop variables. For example, this occurs when a DO loop variable is used as a parameter to a subroutine.
- When a DO loop variable is referenced in an EQUIVALENCE statement.
- When certain complicated patterns of branching are used around a DO loop.

Example 1:

```

C EXAMPLE 1
C LOOP NOT OPTIMIZABLE
  EQUIVALENCE (K1,K2)
  REAL*4 A(1000)

      DO 10 K1 = 1,1000
        A(K2) = A(K2) ** 2.1
10    CONTINUE

```

Possible Response 1: When a loop is not optimizable because the induction variable is used in an EQUIVALENCE statement, attempt to use a different DO loop variable to control the loop iteration.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C LOOP NOT OPTIMIZABLE
  EQUIVALENCE (K1,K2)
  REAL*4 A(1000)

      DO 10 K = 1,1000
        A(K) = A(K) ** 2.1
10    CONTINUE

```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

Example 2:

```

C EXAMPLE 2
C LOOP NOT OPTIMIZABLE
  REAL*4 Q(1000,1000),R(1000,1000)

      DO 160 J=1,1000
        DO 160 I=1,1000
          L=1
          IF (L.GT.LLIM) GO TO 140
100    DO 120 K=1,1000
          Q(J,I) = R(L,K)
120    CONTINUE
          L=L+1
          IF (L.LE.LLIM ) GO TO 100
140    CONTINUE
160    CONTINUE

```

The inner loop is not eligible for optimization due to the complex pattern of branches around that loop.

Possible Response 2: For cases such as this, attempt to redesign the algorithm using structured programming constructs whenever possible.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C LOOP NOT OPTIMIZABLE
  REAL*4 Q(1000,1000),R(1000,1000)

      DO 160 J=1,1000
        DO 160 I=1,1000
          DO 120 L=1,LLIM
100    DO 120 K=1,1000
          Q(J,I) = R(L,K)
120    CONTINUE
160    CONTINUE

```

ILX0511I Short Form: BACKWARD BRANCH
Long Form: ONE OR MORE BACKWARD BRANCHES TO THE STATEMENT LABEL(S) <list> HAVE BEEN FOUND AND ARE UNANALYZABLE FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates the presence of a backward branch or a DO WHILE loop within a DO loop.

Supplemental Data:

<list> is a list of user defined statement labels that are used to indicate the targets of any backward branches that occur in the loop.

Example:

```

C EXAMPLE
C BACKWARD BRANCH
  REAL*4 A(1000),B(1000,1000)

      DO 20 I = 1,1000
        A(I) = 0.0
        K = 1
10     A(I) = A(I) + B(K,I)
        K = K + 1
        IF (K .LE. 1000) GO TO 10
20    CONTINUE

```

Possible Response: Attempt to replace the backward GOTO with an equivalent DO loop.

Modified Example:

```

C POSSIBLE RESPONSE
C BACKWARD BRANCH
  REAL*4 A(1000),B(1000,1000)

      DO 20 I = 1,1000
        A(I) = 0.0
        DO 15 K = 1,1000
10         A(I) = A(I) + B(K,I)
15        CONTINUE
20    CONTINUE

```

ILX0512I Short Form: INTRINSIC CHARACTER FUNCTION
Long Form: THE CHARACTER MANIPULATION FUNCTION(S) <flist> ARE NOT ANALYZABLE FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates that a loop is rejected because it uses one or more of the character manipulation functions (INDEX, LGE, LGT, LLE, and LLT) contained in the VS FORTRAN library.

Supplemental Data:

<flist> is a list consisting of function names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements using character manipulation functions are separated from the portions

of the original loop that are eligible for vectorization and parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0513I Short Form: RESTRICTED CONSTRUCT
Long Form: THE LANGUAGE
CONSTRUCT(S) <clist> ARE NOT
ANALYZABLE FOR PARALLELIZATION
OR VECTORIZATION.

Explanation: Indicates that a loop is rejected because it contains some language construct that cannot be analyzed by the compiler. These constructs include assigned and computed GOTO statements and NAMELIST statements.

Supplemental Data:

<clist> is a list consisting of the names of the language constructs responsible for the rejection along with the ISNs (internal statement numbers) of the statements in which they are used.

Example:

```
C EXAMPLE
C RESTRICTED CONSTRUCT
  INTEGER*4 TEST(1000)
  REAL*4 W(1000),X(1000),Y(1000),Z(1000)

  DO 1000 I = 1,1000
    GOTO (400,500,600),TEST(I)
    W(I) = 0.0
    GOTO 1000
  400   W(I) = X(I)
        GOTO 1000
  500   W(I) = Y(I)
        GOTO 1000
  600   W(I) = Z(I)
  1000 CONTINUE
```

Possible Response: In the case of an assigned or computed GOTO statement, it may be possible to recode the loop so that the same logic structure is achieved using logical and arithmetic IF statements.

Modified Example:

```
C POSSIBLE RESPONSE
C RESTRICTED CONSTRUCT
  INTEGER*4 TEST(1000)
  REAL*4 W(1000),X(1000),Y(1000),Z(1000)

  DO 1000 I = 1,1000
    IF (TEST(I).LT.1 .OR. TEST(I).GT.3) THEN
      W(I) = 0.0
    ELSE IF (TEST(I).EQ.1) THEN
  400   W(I) = X(I)
    ELSE IF (TEST(I).EQ.2) THEN
  500   W(I) = Y(I)
    ELSE IF (TEST(I).EQ.3) THEN
  600   W(I) = Z(I)
    ENDIF
  1000 CONTINUE
```

Be careful when doing this; if the transformed code fails to vectorize or have parallel code generated for it, the resulting scalar program may run more slowly than the original program.

ILX0514I Short Form: USER FUNCTION OR
SUBROUTINE
Long Form: THE USER FUNCTION(S)
OR SUBROUTINE(S) <flist> ARE NOT
ANALYZABLE FOR VECTORIZATION.

Explanation: Indicates that a loop contains a subroutine call or a reference to an external user defined function.

Supplemental Data:

<flist> is a list consisting of function and subroutine names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements containing a subroutine call or a reference to an external user defined function are separated from the portions of the original loop that are eligible for vectorization analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0515I Short Form: NON-MATHEMATICAL
IMPLICIT
Long Form: THE IMPLICITLY CALLED
NON-MATHEMATICAL SUBPROGRAM(S)
<flist> HAVE BEEN USED. THESE
SUBPROGRAMS ARE NOT ANALYZABLE
FOR PARALLELIZATION OR
VECTORIZATION.

Explanation: Indicates that some statement in the loop will generate a reference to an implicitly invoked character subprogram (CNCAT#) or to an implicitly invoked utility program (DSPAN#, DSPN2#, DSPN4#, or DYCMN#). These programs are described in the *VS FORTRAN Version 2 Language and Library Reference*.

Supplemental Data:

<flist> is a list consisting of function names and the ISNs (internal statement numbers) of the statements in which they are used.

Possible Response: Break the loop into two or more loops, so that any statements resulting in compiler call(s) to implicitly invoked character subprograms or to implicitly invoked utility programs are separated from the portions of the original loop that are eligible for vectorization and parallel code generation analysis. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0516I Short Form: ARRAY W/O SUBSCRIPTS IN I/O
Long Form: ARRAYS IN I/O WITHOUT SUBSCRIPTS ARE NOT ANALYZABLE FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates the presence of I/O statements containing arrays without subscripts (such as PRINT*,A) that cannot be analyzed for parallel code generation or vectorization.

Example:

```
C EXAMPLE
C ARRAY W/O SUBSCRIPTS IN I/O
  REAL*4 A(800,800),B(800),C(800)

      DO 20 I=1,800
        READ(5,*) B,C(I)
        DO 10 J=1,800
          A(J,I) = B(J) + C(I)
10      CONTINUE
        S = S + A(I,1)
20    CONTINUE
```

Possible Response: Break the loop into two or more loops, so that any I/O statements containing full arrays are separated from the portions of the original loop that are eligible for vectorization and/or parallel code generation analysis.

Modified Example:

```
C POSSIBLE RESPONSE
C ARRAY W/O SUBSCRIPTS IN I/O
  REAL*4 A(800,800),B(800),C(800)

      DO 10 I=1,800
        READ(5,*) B,C(I)
        DO 10 J=1,800
          A(J,I) = B(J) + C(I)
10      CONTINUE

      DO 20 I=1,800
        S = S + A(I,1)
20    CONTINUE
```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0517I Short Form: EQUIVALENCE OFFSET UNKNOWN
Long Form: THE EQUIVALENCE OFFSET(S) FOR VARIABLE(S) <vlist> COULD NOT BE ANALYZED FOR PARALLELIZATION OR VECTORIZATION. AN EQUIVALENCE GROUP IS IRREGULAR, OR HAS ARRAYS WITH DIFFERENT DATATYPES OR SCALAR VARIABLES. THE COMPILER ASSUMES THAT THE VARIABLES CARRY

DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: To compute dependences between EQUIVALENCE variables, the following must be true:

- The variables must have the same element size.
- The computed equivalence offset for the variables must be a multiple of the element size.
- The variables must be array variables.

If any of these conditions are not true, the variables may be presumed to carry dependences and the compiler will note vectorize or generate parallel code for them.

Supplemental Data:

<vlist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They may, however, differ from the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C EQUIVALENCE OFFSET UNKNOWN
  REAL*8 A(1000),T
  REAL*4 B(1000)
  EQUIVALENCE (A,B,T)

      DO 10 I = 1000,1,-1
        A(I) = B(I) ** 2.1 * T
10      CONTINUE
```

Possible Response 1: Make the element sizes of the EQUIVALENCE variables the same. Also, replace the scalar variable with an array.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C EQUIVALENCE OFFSET UNKNOWN
  REAL*8 A(1000),B(1000),C(1)
  EQUIVALENCE (A,B,C)

      DO 10 I = 1000,1,-1
        A(I) = B(I) ** 2.1 * C(1)
10      CONTINUE
```

Possible Response 2: Insert the IGNORE RECRDEPS directive to instruct the compiler to assume that a dependence due to the EQUIVALENCE variables does not occur. Before using this directive, you should analyze the storage mapping and subscript expressions of the variables involved and make sure that the different variables do not reference identical storage locations while the loop is running.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C EQUIVALENCE OFFSET UNKNOWN
    REAL*8 A(1000),T
    REAL*4 B(1000)
    EQUIVALENCE (A,B,T)

*DIR    IGNORE RECRDEPS
        DO 10 I = 1000,1,-1
          A(I) = B(I) ** 2.1 * T
10      CONTINUE
```

ILX0518I Short Form: OFFSET UNKNOWN
Long Form: THE OFFSET NEEDED TO ADDRESS THE ARRAY(S) <alist> COULD NOT BE ANALYZED FOR PARALLELIZATION OR VECTORIZATION. THERE MAY BE AN UNKNOWN TERM IN A SUBSCRIPT OR IN A LOOP LOWER BOUND, OR THE ARRAY(S) MAY HAVE ADJUSTABLE DIMENSIONS. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when some additive term in a subscript computation for a particular array is not an induction variable or a constant. It can also appear when the DO loop in which an array reference is contained has a variable lower bound. In these situations, recurrent dependences are presumed to exist between the statement in which the subscript computation is used and all other statements that reference the array.

Supplemental Data:

<alist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```
C EXAMPLE 1
C OFFSET UNKNOWN - ADDITIVE TERM
    REAL*4 A(1000),B(1000)

    DO 10 I = 1,800,2
      A(I) = A(I+ISKIP) * B(I) ** 2.1
10    CONTINUE
```

Possible Response 1: Identify the expression or expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C OFFSET UNKNOWN - ADDITIVE TERM
    REAL*4 A(1000),B(1000)

    DO 10 I = 1,800,2
      A(I) = A(I+1) * B(I) ** 2.1
10    CONTINUE
```

Example 2:

```
C EXAMPLE 2
C OFFSET UNKNOWN - LOWER BOUND
    REAL*4 C(1000)

    DO 20 I = ISTART,1000
      C(I) = C(3) ** 2.1
20    CONTINUE
```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C OFFSET UNKNOWN - LOWER BOUND
    REAL*4 C(1000)

*DIR    IGNORE RECRDEPS
        DO 20 I = ISTART,1000
          C(I) = C(3) ** 2.1
20      CONTINUE
```

Be careful when applying IGNORE RECRDEPS because if this directive is used and if the dependences really do exist, wrong results may be produced. In this case, the IGNORE RECRDEPS directive is valid only if the value of the variable ISTART is greater than 3. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0519I Short Form: STRIDE UNKNOWN
Long Form: THE STRIDE NEEDED TO ADDRESS THE ARRAY(S) <alist> COULD NOT BE ANALYZED FOR PARALLELIZATION OR VECTORIZATION, EITHER BECAUSE OF AN UNKNOWN MULTIPLIER IN THE SUBSCRIPT OR AN UNKNOWN LOOP INCREMENT. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message occurs when the multiplier of some induction variable within a subscript computation for a particular array is not a constant. It can also appear when the loop in which an array reference is contained has a variable increment value.

In these situations, recurrent dependences are presumed to exist between the statement in which the subscript computation is used and all other statements that reference the array.

Supplemental Data:

<alist> is a list of the names of the variables that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C STRIDE UNKNOWN
  REAL*4 A(1000),B(1000)
  INTEGER*4 ISKIP

  I = 1
  DO 10 K = 1,800
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
10  CONTINUE
```

Possible Response 1: Identify the expression or expressions involved and replace them with references to values that are known at compile time whenever possible.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C STRIDE UNKNOWN
  REAL*4 A(1000),B(1000)
  INTEGER*4 ISKIP
  PARAMETER (ISKIP=4)

  I = 1
  DO 10 K = 1,800
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
10  CONTINUE
```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C STRIDE UNKNOWN
  REAL*4 A(1000),B(1000)
  INTEGER*4 ISKIP

  I = 1
*DIR  IGNORE RECRDEPS(A)
  DO 10 K = 1,800
    A(I) = A(I) * B(K) ** 2.1
    I = I + ISKIP
10  CONTINUE
```

Be careful when applying IGNORE RECRDEPS because if this directive is used and if the ignored dependences really do exist, wrong results may be produced. (In this case, the dependence exists only if the value of the variable ISKIP is 0.) See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0520I Short Form: POSSIBLE RECURRENCE
Long Form: THIS LOOP COULD NOT BE PARALLELIZED OR VECTORIZED BECAUSE OF A POSSIBLE RECURRENCE INVOLVING THE ARRAY(S) <alist>. THE INFORMATION NEEDED TO DETERMINE WHETHER OR NOT THE RECURRENCE EXISTS WAS NOT AVAILABLE TO THE COMPILER. IF NO RECURRENCE EXISTS, PARALLELIZATION AND VECTORIZATION CAN BE ACHIEVED WITH AN IGNORE DIRECTIVE.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible to process for the compiler to generate parallel code for or vectorize them. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until compile time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization and parallel code generation are not performed. If the IGNORE directive is used, the recurrence is assumed to be absent. However, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies cases in which the existence of a recurrence cannot be determined at compile time.

Supplemental Data:

<alist> is a list of the names of arrays that are involved in the recurrence. (This list includes only the variables that could not be fully analyzed. Other

variables involved in the recurrence might not appear in the list.)

Example:

```
C EXAMPLE
C POSSIBLE RECURRENCE
  REAL  A(2000),B(2000)

      DO 10 I = 1,1000
        A(I+N) = A(I) * B(I)
10    CONTINUE
```

In this example, a recurrence exists if an element of the array A that is stored on one iteration of the loop is referenced on some later iteration. This is true if the variable N has a value between 1 and 999.

Possible Response: If the existence of the recurrence depends on some values determined at run time, and if the value will never cause the recurrence to exist, you can use the IGNORE directive to vectorize and generate parallel code for your program. Note that both the IGNORE and PREFER PARALLEL directives are needed generate parallel code for this loop.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C POSSIBLE RECURRENCE
  REAL  A(2000),B(2000)

*DIR  IGNORE RECRDEPS(A)
*DIR  PREFER PARALLEL VECTOR
      DO 10 I = 1,1000
        A(I+N) = A(I) * B(I)
10    CONTINUE
```

ILX0521I Short Form: EQUIVALENCE SCALAR USED
Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR PRIVATIZATION OR EXPANSION BECAUSE THEY ARE IN AN EQUIVALENCE GROUP. PARALLELIZATION AND VECTORIZATION IS INHIBITED.

Explanation: Indicates scalar variables that are in an EQUIVALENCE group and are modified in a loop that cannot be vectorized and for which parallel code cannot be generated, since they are not eligible for privatization or scalar expansion respectively.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for privatization and scalar expansion.

Example:

```
C EXAMPLE
C EQUIVALENCE SCALAR USED
  REAL*4 X(1000),Y(1000),Z(1000),D(1000),S
  EQUIVALENCE (S,D(1000))

      DO 20 I = 1,1000
        S = X(I) + Y(I)
        Z(I) = S
20    CONTINUE
```

Possible Response: Sometimes, you can avoid this situation by replacing references to the original scalar with references to a new scalar variable that is never referenced outside the loop. Note that if the original scalar variable is needed later, be sure that it is assigned the correct value after the loop has completed.

Modified Example:

```
C EXAMPLE
C EQUIVALENCE SCALAR USED
  REAL*4 X(1000),Y(1000),Z(1000),D(1000),S,S_NEW
  EQUIVALENCE (S,D(1000))

      DO 20 I = 1,1000
        S_NEW = X(I) + Y(I)
        Z(I) = S_NEW
20    CONTINUE
      S = S_NEW
```

ILX0522I Short Form: DEFINITE RECURRENCE
Long Form: THIS LOOP COULD NOT BE PARALLELIZED OR VECTORIZED BECAUSE OF A DEFINITE RECURRENCE INVOLVING THE VARIABLE(S) <vlist>.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible for the compiler to vectorize or generate parallel code for them. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization and parallel code generation are not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies the cases in which a recurrence definitely exists.

Supplemental Data:

<vlist> is a list of the names of the variables that are involved in the recurrence. (This list includes only those variables that could be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```
C EXAMPLE
C DEFINITE RECURRENCE
  REAL  A(1000),B(1000)

      DO 10 I = 1,999
        A(I+1) = A(I) * B(I)
10    CONTINUE
```

In this example, a recurrence definitely exists, since the value stored into the array A on each iteration of the loop is used on the following iteration. Parallel code cannot be generated for this loop, and it cannot be vectorized.

ILX0523I Short Form: INTERCHANGE PREVENTING DEP
Long Form: THE ARRAY(S) <alist> CARRY FORWARD DEPENDENCES AT NESTING LEVEL(S) <levlist> THAT MAY BE INTERCHANGE PREVENTING. THESE LOOP(S) CANNOT BE VECTORIZED.

Explanation: This message identifies certain dependences, known as interchange-preventing dependences, that restrict vectorization of outer DO loops. When an interchange-preventing dependence exists, the reordering, or interchange, of two loops would cause different results to be produced. Since vectorization of an outer loop has the same effect as moving that loop to the innermost position, the existence of an interchange-preventing dependence carried by an outer loop prevents vectorization.

To understand how an interchange-preventing dependence comes about, study the following example:

```
      DO 10 I=1,2
      DO 10 J=1,2
10    A(I-1,J+1)=A(I,J)
```

In this code, the element A(1,2) is fetched when I=1 and J=2 and is stored into when I=2 and J=1. When these loops are processed, the fetch will occur before the store. However, if the loops were interchanged, the store would come first and different results would probably be produced.

The compiler cannot always determine whether a dependence is interchange preventing. Unless it can prove otherwise, the compiler will always assume that a given dependence is interchange preventing. This will insure that correct results are always produced after

vectorization, even though some potential vectorization may be missed.

This message identifies dependences that were assumed to be interchange preventing because the compiler did not have sufficient information to perform a complete and accurate analysis.

Note that this message often occurs when variables are used for the lower or upper bound of a loop or when variables that are not inductions are used inside of subscripts. Sometimes, such a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value. However, since the compiler performs vectorization analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<alist> is a list of the names of the variables that carry the dependences that are presumed to be interchange preventing.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C INTERCHANGE PREVENTING DEPENDENCY
  REAL*4 U(1000,1000)

      DO 190 J = 1, JUPPER
      DO 190 I = 1, IUPPER
        U(I,J) = U(I+N,J) + U(I,J+N)
190    CONTINUE
```

Possible Response 1: If it is possible to write the loop bounds and subscript expressions in terms of compile-time constants and induction variables, this may give the compiler enough information to do an accurate analysis.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C INTERCHANGE PREVENTING DEPENDENCY
  REAL*4 U(1000,1000)

      DO 190 J = 1, 999
      DO 190 I = 1, 999
        U(I,J) = U(I+1,J) + U(I,J+1)
190    CONTINUE
```

Possible Response 2: It is also possible to increase vectorizability by using the IGNORE RECRDEPS directive. Before using this directive, you should analyze the dependences involved to verify that they really are not interchange preventing.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C INTERCHANGE PREVENTING DEPENDENCY
  REAL*4 U(1000,1000)

*DIR      IGNORE RECRDEPS
          DO 190 J = 1, JUPPER
          DO 190 I = 1, IUPPER
            U(I,J) = U(I+N,J) + U(I,J+N)
190      CONTINUE
```

ILX0524I Short Form: OPTIMIZER INDUCED DEPENDENCE
Long Form: A COMPILER TEMPORARY INTRODUCED DURING SCALAR OPTIMIZATION HAS CAUSED ONE OR MORE DEPENDENCES IN THE LOOP(S) AT NESTING LEVEL(S) <levlist>. THESE LOOP(S) CANNOT BE PARALLELIZED OR VECTORIZED.

Explanation: This message is produced when a statement becomes part of a recurrence solely because of some optimization that had been performed prior to vectorization and/or parallel code generation (for example, common subexpression elimination).

Supplemental Data:

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```
C EXAMPLE
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(1000),D(1000),F(1000)

          DO 100 J = 2,1000
            C(J) = D(J-1)
            D(J) = C(J)
            F(J) = D(J) + 2
100      CONTINUE
```

In the DO loop shown above, the first two statements form a recurrence and the compiler cannot vectorize or generate parallel code for them, while the last statement would normally be eligible for both vectorization and parallel code generation. However, parallel code generation or vectorization of this statement may be restricted due to some transformations that have been applied by the compiler prior to vectorization and/or parallel code generation analysis.

In optimizing this loop, the compiler will attempt to reduce the number of load instructions that must be processed.

As a result, during vectorization and/or parallel code generation analysis, it will appear as if this loop were rewritten as:

```
          DO 100 J = 2,1000
            .temp = D(J-1)
            C(J) = .temp
            D(J) = .temp
            F(J) = .temp + 2
100      CONTINUE
```

where .temp is a compiler generated scalar temporary. In order to vectorize and generate parallel code for the last statement, it would be necessary to split the original loop into two loops so that this statement is separated from the other, nonparallelizable and nonvectorizable, statements. The presence of the scalar temporary shared by all the statements in the loop prohibits loop splitting and thus prevents partial parallel code generation or vectorization.

Possible Response 1: Since the presence of statement labels inhibits optimization to some degree, it is sometimes possible to achieve partial vectorization and/or parallel code generation in cases such as this simply by introducing additional labels.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(1000),D(1000),F(1000)

          DO 100 J = 2,1000
            C(J) = D(J-1)
            D(J) = C(J)
          99 F(J) = D(J) + 2
100      CONTINUE
```

Be careful when using this type of transformation since it may inhibit some important optimizations. If you make this change and partial vectorization and/or parallel code generation still does not occur, the resulting serial and scalar code may run more slowly than the original.

Possible Response 2: The transformation suggested above may or may not increase vectorization and/or parallel code generation. If it is not effective, you should try to replace the original loop with two separate loops, where one loop contains the nonparallelizable and nonvectorizable portion while the other loop contains the parallelizable and vectorizable portion of the original loop.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C OPTIMIZER INDUCED DEPENDENCE
  REAL*4 C(1000),D(1000),F(1000)

      DO 100 J = 2,1000
      C(J) = D(J-1)
      D(J) = C(J)
100  CONTINUE
      DO 101 J = 2,1000
      F(J) = D(J) + 2
101  CONTINUE

```

Do this only when you are absolutely certain that the transformation will not alter the results of your program.

ILX0525I Short Form: SUBSCRIPT TOO COMPLEX
Long Form: THE ARRAY(S) <alist> USE SUBSCRIPT COMPUTATIONS THAT COULD NOT BE ANALYZED FOR PARALLEL OR VECTOR. THEY MAY INCLUDE INDIRECT ADDR, DATA CONVERSIONS, UNK STRIDES, OR AUXILIARY INDUCTION VARIABLES. THE COMPILER ASSUMES THAT THE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT LEVEL(S) <levlist>.

Explanation: Indicates the use of subscript computations for which the compiler could not perform accurate dependence analysis. These cases include the constructs listed below. (In each of the examples given, K is an induction variable for some DO loop that contains the indicated array reference.)

- Indirect addressing, as in $A(\text{INDEX}(K))$, where INDEX is an array of integers.
- Subscripts requiring data conversions, as in $A(K+X)$, where X is a real variable.
- Subscripts where the stride is not known at compile time, as in $A(K*KSTEP)$, where KSTEP is an integer variable. (An unknown stride may also occur if the increment expression of some DO loop is not a compile-time constant.)
- Auxiliary induction variables, as in $A(\text{IVAR})$, where IVAR is incremented explicitly within the DO loop by some statement of the form $\text{IVAR}=\text{IVAR}+\text{INCR}$.

When an array uses any of the above types of subscript, recurrent dependences are often presumed to exist between the statement in which the subscript computation occurs and all other statements that reference the array.

This message often occurs when variables are used for the lower bound, upper bound, or increment of a loop or when variables that are not inductions are used inside of subscripts. Sometimes, such a variable will be defined only once inside the program unit in which it is used, and will therefore have a constant value.

However, since the compiler performs vectorization and parallel code generation analysis on a DO loop basis, it may not be able to recognize that such a variable is actually a constant.

Supplemental Data:

<alist> is a list of the names of the arrays that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example 1:

```

C EXAMPLE 1
C SUBSCRIPT TOO COMPLEX
  REAL*4 A(1000)
  INTEGER*4 INDEX(1000)

      DO 10 I = 1,1000
      A(INDEX(I)) = A(INDEX(I)) ** 2.1
10  CONTINUE

```

Possible Response 1: For cases involving indirect addressing, try to introduce additional arrays to hold intermediate results. Transform the loop as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

Do this only if it is absolutely certain that the noninductive subscript expression never selects any element more than once.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C SUBSCRIPT TOO COMPLEX
  REAL*4 A(1000),NEW_A(1000)
  INTEGER*4 INDEX(1000)

      DO 9 I = 1,1000
      NEW_A(I) = A(INDEX(I))
9  CONTINUE
      DO 10 I = 1,1000
      NEW_A(I) = NEW_A(I) ** 2.1
10 CONTINUE
      DO 11 I = 1,1000
      A(INDEX(I)) = NEW_A(I)
11 CONTINUE

```

Due to the overhead involved in copying data to and from the new version of the array, this transformation may not result in any performance benefits, even if

vectorization and/or parallel code generation are achieved. Carefully analyze the performance of this code before and after the transformation is applied to make sure that it is worthwhile.

Example 2:

```
C EXAMPLE 2
C SUBSCRIPT TOO COMPLEX
  REAL*4 B(2000),X

  DO 20 I = 1,1000
    B(I+X) = B(I+X) ** 2.1
20  CONTINUE
```

Possible Response 2: For cases involving data conversions inside of subscript calculations, recode the computations so that all the inputs hold INTEGER values, whenever possible.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C SUBSCRIPT TOO COMPLEX
  REAL*4 B(2000),X
  INTEGER*4 INT_X

  INT_X = X
  DO 20 I = 1,1000
    B(I+INT_X) = B(I+INT_X) ** 2.1
20  CONTINUE
```

Example 3:

```
C EXAMPLE 3
C SUBSCRIPT TOO COMPLEX
  REAL*4 C(2000),Y
  INTEGER*4 KSTEP

  DO 30 I = 1,1000
    KSTEP = KSTEP + INC
    C(KSTEP*I) = Y ** C(I*KSTEP)
30  CONTINUE
```

Possible Response 3: For cases involving unknown strides, identify the expressions involved and replace them whenever possible with references to values that are known at compile time.

Modified Example 3:

```
C POSSIBLE RESPONSE 3
C SUBSCRIPT TOO COMPLEX
  REAL*4 C(2000),Y
  INTEGER*4 KSTEP
  PARAMETER (KSTEP=4)

  DO 30 I = 1,1000
    C(KSTEP*I) = Y ** C(I*KSTEP)
30  CONTINUE
```

Example 4:

```
C EXAMPLE 4
C SUBSCRIPT TOO COMPLEX
  REAL*4 D(2000*2000)

  DO 40 J = 1,2000,N
    D(J) = D(J*J) ** 2.1
40  CONTINUE
```

Possible Response 4: For cases in which none of the previous transformations is appropriate, use an IGNORE RECRDEPS to cause the compiler to assume that no dependence exists.

Modified Example 4:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 4
C SUBSCRIPT TOO COMPLEX
  REAL*4 D(2000*2000)

*DIR    IGNORE RECRDEPS(D)
*DIR    PREFER PARALLEL
  DO 40 J = 1,2000,N
    D(J) = D(J*J) ** 2.1
40  CONTINUE
```

Use care should be used in applying IGNORE RECRDEPS since if this directive is used and if the ignored dependences really do exist, wrong results may be produced. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive. Note the requirement of the PREFER PARALLEL directive to cause the compiler to generate parallel code for this loop.

ILX0526I Short Form: SCALAR DEFINED BEFORE LOOP
Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR VECTORIZATION BECAUSE THEY MAY USE VALUES THAT WERE SET BEFORE THE EXECUTION OF THE CONTAINING LOOP AND THEIR VALUES MAY BE USED AFTER THE EXECUTION OF THE CONTAINING LOOP.

Explanation: Vectorization of scalar variables that are modified within a loop requires a process known as scalar expansion. This involves replacing the scalar with a vector register, which may not be possible in certain cases.

Scalar expansion is not performed when a scalar variable is not local to a loop, and where that variable is referenced at different nesting levels or where it is modified by a conditionally processed statement. A variable is considered nonlocal to a loop if it is in COMMON, or if it uses a value within the loop that will be used after the loop. For more information on scalar expansion, see 288.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for scalar expansion.

Example:

```
C EXAMPLE
C SCALAR DEFINED BEFORE LOOP
  REAL*4 A(1000),B(1000),C(1000),T

  T = 1.1
  DO 11 I = 1,1000
    DO 10 J = 1,1000
      A(J,I) = T
10    CONTINUE
      T = B(I) + C(I)
11  CONTINUE
```

Possible Response: Try to replace the original scalar variable with an array whose dimension ranges from zero to the number of iterations of the loop in which the scalar resides. Transform the loop as follows:

- Prior to entering the loop, set the zero element of the new array to the value held by the scalar.
- Prior to the first statement that defines the scalar within the loop, replace all references to that scalar with references to the element of the new array whose position is one less than the current iteration count.
- Replace all other references to the scalar variable within the loop by references to the element of the array that corresponds to the current iteration count.
- Following the loop, set the scalar to the value held by the last element of the new array.

Modified Example:

```
C POSSIBLE RESPONSE
C SCALAR DEFINED BEFORE LOOP
  REAL*4 A(1000),B(1000),C(1000),T
  REAL*4 TT(0:1000)

  T = 1.1
  TT(0) = T
  DO 11 I = 1,1000
    DO 10 J = 1,1000
      A(J,I) = TT(I-1)
10    CONTINUE
      TT(I) = B(I) + C(I)
11  CONTINUE
  T = TT(1000)
```

Note that this transformation is valid only if the first assignment to the original scalar variable is not a conditionally processed statement.

Also be aware that this transformation may not necessarily increase the the ability of your program to vectorize and could result in a scalar program that runs more slowly than the original.

ILX0527I Short Form: SCALAR NEEDED AFTER LOOP

Long Form: THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR PARALLELIZATION OR VECTORIZATION BECAUSE THEY MAY BE SET TO VALUES THAT WILL BE USED AFTER THE EXECUTION OF THE CONTAINING LOOP.

Explanation: Vectorization of and parallel code generation for scalar variables that are modified within a loop requires a process known as privatization and scalar expansion respectively. This involves replacing the scalar with a private variable and vector register. This may not be possible in certain cases.

The cases where privatization and scalar expansion are not done are where a scalar variable is *nonlocal* to a loop, and where that variable is referenced at different nesting levels or where it is modified by a conditionally processed statement. A variable is considered *nonlocal* to a loop if it is in COMMON or if it uses a value within the loop that will be used after the loop.

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for privatization and/or scalar expansion.

Example:

```
C EXAMPLE
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(1000),B(1000),T

  DO 22 I = 1,1000
    T = A(I)
    DO 22 J = 1,1000
      B(J,I) = T
22  CONTINUE
  ...
  WRITE(6,*) T
```

Possible Response 1: Try to replace the original scalar variable with a new scalar variable that is local to the loop. If this is done, you must also insert an additional assignment after the loop to set the original scalar to its appropriate final value.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(1000),B(1000,1000),T
  REAL*4 T_LOCAL

      DO 22 I = 1,1000
        T_LOCAL = A(I)
        DO 22 J = 1,1000
          B(J,I) = T_LOCAL
22    CONTINUE
      T = A(1000)
      ...
      WRITE(6,*) T

```

Possible Response 2: Try to replace the original scalar variable with an array whose dimension ranges from one to the number of iterations of the loop in which the scalar resides. Transform the loop as follows:

- If the first occurrence of the scalar within the loop is a reference rather than definition, prior to entering the loop, set the first element of the array to the value held by the scalar.
- Replace all occurrences of the scalar within the loop with references to the element of the array that corresponds to the current iteration count.
- Following the loop, set the scalar to the value held by the last element of the array.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C SCALAR NEEDED AFTER LOOP
  REAL*4 A(1000),B(1000,1000),T
  REAL*4 TT(1000)

      DO 22 I = 1,1000
        TT(I) = A(I)
        DO 22 J = 1,1000
          B(J,I) = TT(I)
22    CONTINUE
      T = TT(1000)
      ...
      WRITE(6,*) T

```

Note that this transformation is valid only if the first assignment to the original scalar variable is not a conditionally processed statement.

Also be aware that this transformation may not necessarily increase the ability of the compiler to vectorize and generate parallel code for your program, and could result in a scalar program that runs more slowly than the original.

ILX0529I Short Form: NESTED NONCONSTANT INDUCTION

Long Form: THE LOOP VARIABLE OF THIS LOOP OR OF SOME NESTED LOOP AFFECTS THE LOOP VARIABLE OR AN AUXILIARY INDUCTION VARIABLE USED BY SOME OTHER NESTED LOOP. VECTORIZATION IS INHIBITED.

Explanation: When the DO loop parameters of an inner loop are modified by an outer loop, or when the initialization or iteration of an auxiliary induction variable is modified by an outer loop, the outer loop is not vectorization.

The reason for this restriction is that in these cases, the behavior of the inner loop depends on the value of the induction variable of the outer loop. If the outer loop was vectorized, it would be replaced by a sectioning loop. The induction variable of the sectioning loop would take on a different set of values than those of the original loop, and therefore, the inner loop would behave differently (and would probably produce different results.)

The same situation can arise with auxiliary induction variables. An auxiliary induction variable can be either a user variable that is explicitly incremented within a loop or an internal compiler temporary that has been generated to hold certain subscript computations. When a compiler temporary is used it can be difficult to pinpoint the statement or statements for which it was generated. Usually, however, they are associated with particularly complex subscript expressions (for example, subscript computations where two loop variables are multiplied together).

Note: If two or more copies of a particular DO statement are printed in the compiler report, this message may appear with each copy, even though it may not be applicable in all cases.

Example:

```

C EXAMPLE
C NESTED NONCONSTANT INDUCTION
  REAL*4 A(1000,1000)

      DO 10 I = 1,1000
        DO 10 J = I,1000
          A(I,J) = A(I,J) * 2.1
10    CONTINUE

```

Possible Response: For cases where the induction variable of an outer loop is used as part of the initial or final calculations for an inner DO loop, it may be possible to rewrite the inner loop so that the range of that loop is independent of the outer loop. This can sometimes be done by introducing IF statements inside the loop to exclude the iterations which would not have been processed in the original code.

Modified Example:

```

C POSSIBLE RESPONSE
C NESTED NONCONSTANT INDUCTION
  REAL*4 A(1000,1000)

  DO 10 I = 1,1000
    DO 10 J = 1,1000
      IF (J.LT.I) GOTO 10
      A(I,J) = A(I,J) * 2.1
10    CONTINUE

```

Be careful when you use this transformation. Even though it may increase vectorization, the additional overhead of the conditional code may result in increased run time.

ILX0530I Short Form: UNKNOWN UPPER BOUND
Long Form: THE ARRAY(S) <alist> MAY OR MAY NOT BE INVOLVED IN DEPENDENCES, DEPENDING ON THE UPPER BOUND OF SOME CONTAINING LOOP. SINCE THE UPPER BOUND IS NOT KNOWN, THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCE(S) IN LOOP(S) AT NESTING LEVEL(S) <levlist> AND CANNOT BE PARALLELIZED OR VECTORIZED.

Explanation: This message occurs when a variable is specified as the upper bound of a loop and when the existence of a dependence depends on the value of the upper bound. In these cases, dependence will always be assumed.

Supplemental Data:

<alist> is a list of the names of the arrays that carry the presumed dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They may, however, differ from the nesting level indications that appear on the source listing.

Example 1:

```

C EXAMPLE
C UNKNOWN UPPER BOUND
  REAL*4 A(-1000:1000)

  DO 10 I = 1,N
    A(I) = A(I-1000) * 22.1
10    CONTINUE

```

Possible Response 1: Identify the loop that carries the dependence and replace the upper bound of the loop with a compile-time constant, if possible.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C UNKNOWN UPPER BOUND
  REAL*4 A(-1000:1000)
  PARAMETER (N=1000)

  DO 10 I = 1,N
    A(I) = A(I-1000) * 22.1
10    CONTINUE

```

Possible Response 2: If it can be determined that the indicated dependences will not occur at run time, it is possible to cause the compiler to ignore them by using the IGNORE RECRDEPS directive with the loops at the indicated levels.

Modified Example 2:

```

@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C UNKNOWN UPPER BOUND
  REAL*4 A(-1000:1000)

*DIR    IGNORE RECRDEPS
  DO 10 I = 1,N
    A(I) = A(I-1000) * 22.1
10    CONTINUE

```

Be careful when applying the IGNORE RECRDEPS directive because if this directive is used and if the dependences really do exist, wrong results may be produced. In this case, the directive is correct only if the value of the variable N is always less than 21. See the section "Using Parallel and Vector Directives" on page 353 for details on how to correctly specify and verify this directive.

ILX0535I Short Form: UNSUPPORTABLE DEPENDENCE
Long Form: THIS CODE IS CONSIDERED UNSUPPORTABLE FOR VECTORIZATION BECAUSE IT IS LINKED TO SOME UNSUPPORTABLE STATEMENT(S) THROUGH MUTUAL DEPENDENCES.

Explanation: This message is produced when a statement does not contain any unsupportable constructs but is forced into an unsupportable loop because it is tied to some other unsupportable statement through recurrent dependences. These dependences can come about in a number of ways:

- The indicated statement may use a scalar variable that is also used in some other statement that contains an unsupportable construct.
- There may be some control flow that creates a control dependence that involves both the indicated statement and some unsupportable statement.
- There may be a dependence between the indicated statement and some unsupportable statement that

came about because the two statements share some common subexpression.

Example:

```
C EXAMPLE
C UNSUPPORTABLE DEPENDENCE
  REAL*8 A(512),B(512),C(512)
  REAL*16 D(512)

  DO 10 I = 1,512
    A(I) = B(I)/C(I)
    D(I) = B(I)/C(I)
10  CONTINUE
```

In this example, the second statement uses the REAL*16 array, D, and therefore is not eligible for vectorization; the first statement would normally be eligible for vectorization. However, the two statements share a common subexpression, and during vectorization analysis, this loop would appear to the compiler as if it were written:

```
  DO 10 I = 1,512
    .temp = B(I)/C(I)
    A(I) = .temp
    D(I) = .temp
10  CONTINUE
```

where .temp is a scalar temporary generated by the compiler. In order to vectorize the first statement, it would be necessary to split this loop into two separate loops. However the presence of the scalar temporary prevents the compiler from doing this.

Possible Response: If it is possible to replace the unsupported part of the loop with equivalent supportable constructs, do so.

Otherwise, try to separate the vectorizable statements which are linked to unsupported statements by restructuring the original loop into two or more loops.

Modified Example:

```
C POSSIBLE RESPONSE
C UNSUPPORTABLE DEPENDENCE
  REAL*8 A(512),B(512),C(512)
  REAL*16 D(512)

  DO 10 I = 1,512
    A(I) = B(I)/C(I)
10  CONTINUE
  DO 11 I = 1,512
    D(I) = B(I)/C(I)
11  CONTINUE
```

Do this type of restructuring only if you are absolutely certain that it will not alter the results of your program.

ILX0538I Short Form: CONDITIONAL NONINDUCTIVE SUB
Long Form: THE ARRAY(S) <alist> ARE USED IN CONDITIONALLY EXECUTED CODE AND HAVE NON-INDUCTIVE SUBSCRIPT EXPRESSIONS. THEY CANNOT BE VECTORIZED.

Explanation: Indicates the use of noninductive subscript expressions that occur in conditionally processed code. A noninductive expression is any expression that is not a linear function of some loop induction variable, for example, indirect addressing, as in A(INDEX(K)), or diagonal traversal of an array, as in DIAG(K,K).

Note: Some statements that affect the flow of control within a loop may not be reproduced in the compiler report listing produced by the REPORT(XLIST) suboption of the PARALLEL and VECTOR compile-time options. It may be necessary to refer to the source listing or to the output produced by the REPORT(SLIST) suboption of the PARALLEL and VECTOR compile-time options to determine the correct control flow.

Supplemental Data:

<alist> is a list of the names of the arrays that use noninductive subscripts.

Example 1:

```
C EXAMPLE 1
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 B(1000),C(1000)
  INTEGER*4 INDEX(1000)

  DO 10 I = 2,1000
    IF (B(I) .GT. 1000) C(INDEX(I)) = 0.0
10  CONTINUE
```

Possible Response 1: For cases involving indirect addressing, try to introduce additional arrays to hold intermediate results. Transform the loop as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 B(1000),C(1000),NEW_C(1000)
  INTEGER*4 INDEX(1000)

  DO 9 I = 2,1000
    NEW_C(I) = C(INDEX(I))
9    CONTINUE
  DO 10 I = 2,1000
    IF (B(I) .GT. 1000) NEW_C(I) = 0.0
10   CONTINUE
  DO 11 I = 2,1000
    C(INDEX(I)) = NEW_C(I)
11   CONTINUE

```

Do this only if it is absolutely certain that the noninductive subscript expression never selects any element more than once. Also be aware that due to the overhead involved in copying data, it is possible that no performance benefits will be achieved, even if the transformation does result in increased vectorization.

Example 2:

```

C EXAMPLE 2
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 X(1000,1000),Y(1000)

  DO 20 I = 1,1000
    IF (Y(I) .LT. 0.0) X(I,I) = X(I,I) ** 2.1
20   CONTINUE

```

Possible Response 2: For cases of diagonal access to an array, it may also be possible to EQUIVALENCE the original array to a one-dimensional array where the elements that were part of a diagonal in the original are now within a single dimension and are separated by a fixed number of elements. These elements can now be referenced through inductive subscript expressions.

Modified Example 2:

```

C POSSIBLE RESPONSE 2
C CONDITIONAL NON-INDUCTIVE SUB
  REAL*4 X(1000,1000),Y(1000),NEW_X(1000*1000)
  EQUIVALENCE (NEW_X(1),X(1,1))

  J = 1
  DO 20 I = 1,1000
    IF (Y(I) .LT. 0.0) NEW_X(J) = NEW_X(J) ** 2.1
    J = J + 1001
20   CONTINUE

```

ILX0539I Short Form: RESTRICTED NONINDUCTIVE SUB
Long Form: THE ARRAY(S) <alist> HAVE INTEGER*1, INTEGER*2, LOGICAL*1, OR LOGICAL*2 DATA TYPES AND HAVE NON-INDUCTIVE SUBSCRIPT EXPRESSIONS. THEY CAN NOT BE VECTORIZED.

Explanation: Indicates the use of noninductive subscript expressions that occur in arrays with data

types of INTEGER*1, INTEGER*2, LOGICAL*1, AND LOGICAL*2. A noninductive expression is any expression that is not a linear function of some loop induction variable, for example, indirect addressing, as in A(INDEX(K)), or diagonal traversal of an array, as in DIAG(K,K).

Supplemental Data:

<alist> is a list of the names of the arrays that use noninductive subscripts.

Example 1:

```

C EXAMPLE 1
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 C(1000)
  INTEGER*4 INDEX(1000)

  DO 10 I = 2,1000
    C(INDEX(I)) = 0
10   CONTINUE

```

Possible Response 1: If it is possible to replace the original arrays with arrays that are INTEGER*4 or LOGICAL*4, do so.

Otherwise, for cases involving indirect addressing, it may be possible to introduce additional arrays to hold intermediate results. The loop should be transformed as follows:

- Select elements of the original array using the noninductive subscript expression and copy them into a new array.
- Replace the noninductive references to the original array with references to the corresponding elements of the new array.
- Copy the contents of the new array back into the correct positions in the original.

Modified Example 1:

```

C POSSIBLE RESPONSE 1
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 C(1000),NEW_C(1000)
  INTEGER*4 INDEX(1000)

  DO 9 I = 2,1000
    NEW_C(I) = C(INDEX(I))
9    CONTINUE
  DO 10 I = 2,1000
    NEW_C(I) = 0
10   CONTINUE
  DO 11 I = 2,1000
    C(INDEX(I)) = NEW_C(I)
11   CONTINUE

```

Do this only if it is absolutely certain that the noninductive subscript expression never selects any element more than once. Also be aware that due to the overhead involved in copying data, it is possible that no performance benefits will be achieved, even if the transformation does result in increased vectorization.

Example 2:

```
C EXAMPLE 2
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 A(1000,1000)

      DO 20 I = 2,1000
        A(I,I) = A(I,I) * 2.1
20    CONTINUE
```

Possible Response 2: For cases of diagonal access to an array, it may also be possible to EQUIVALENCE the original array to a one-dimensional array where the elements that were part of a diagonal in the original are now within a single dimension and are separated by a fixed number of elements. These elements can be referenced through inductive subscript expressions.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C RESTRICTED NON-INDUCTIVE SUB
  INTEGER*2 A(1000,1000),NEW_A(1000*1000)
  EQUIVALENCE (A(1,1),NEW_A(1))

      J = 1
      DO 20 I = 2,1000
        NEW_A(J) = NEW_A(J) * 2.1
        J = J + 1001
20    CONTINUE
```

ILX0541I Short Form: I/O OPERATIONS
Long Form: I/O OPERATIONS ARE NOT SUPPORTED FOR PARALLELIZATION OR VECTORIZATION.

Explanation: Indicates the presence of I/O statements in a loop. The loop may have been analyzed and found eligible for partial vectorization and parallel code generation; however, the I/O operation remains an unsupported operation for vectorization and parallel code generation.

Example:

```
C EXAMPLE
C I/O OPERATIONS
CDIR PREFER PARALLEL VECTOR
  REAL*4 A(1000),B(1000)

      DO 10 I = 1,1000
        A(I) = B(I) * 3.3
        PRINT*, A(I)
10    CONTINUE
```

Analysis Output:

```
C OUTPUT FROM PARALLEL AND VECTOR ANALYSIS
C I/O OPERATIONS

                                REAL*4  A(1000),B(1000)

PAVE  +----- DO 10 I = 1,1000
      |          A(I) = B(I) * 3.3

UNSP  +----- DO 10 I = 1,1000
      |          PRINT*, A(I)
```

ILX0544I Short Form: MISALIGNED DATA
Long Form: THE VARIABLE(S) <vlist> HAVE STORAGE ALIGNMENTS THAT CONFLICT WITH THEIR DATA TYPES AND CANNOT BE VECTORIZED.

Explanation: Indicates the usage of conflicting storage alignments. References to arrays containing misaligned data should not be vectorized since they would produce alignment exceptions when the program is run.

Supplemental Data:

<vlist> is a list of the names of the variables with the conflicting alignments.

Example:

```
C EXAMPLE
C MISALIGNED DATA
  REAL*4 A(1000),B(1000)
  INTEGER*2 DUMMY
  COMMON // A,DUMMY,B

      DO 10 I = 1,1000
        A(I) = B(I) ** 2.1
10    CONTINUE
```

Possible Response: Modify the declarations so as to assure proper alignment whenever possible.

Modified Example:

```
C POSSIBLE RESPONSE
C MISALIGNED DATA
  REAL*4 A(1000),B(1000)
  INTEGER*2 DUMMY
  COMMON // A,B,DUMMY

      DO 10 I = 1,1000
        A(I) = B(I) ** 2.1
10    CONTINUE
```

ILX0546I Short Form: UNSUPPORTED CONSTRUCT
Long Form: NO PARALLEL OR VECTOR SUPPORT EXISTS FOR THIS OCCURRENCE OF THE <flist> CONSTRUCT.

Explanation: Indicates that a particular occurrence of a MAX or MIN intrinsic function reference cannot be processed in parallel or vector mode because of the complexity or ordering of its arguments.

Supplemental Data:

<flist> is the name of the function involved.

Possible Response: This happens when a scalar variable appears both on the left side of the equal sign and as a MAX and MIN intrinsic function argument in the same Fortran statement.

Try to simplify and reorder the arguments and to make the scalar variable appear as the first argument of the MAX or MIN reference.

Example:

The compiler will not vectorize nor generate parallel code:

```
CC = MAX(MAX(A(I),CC),2.0,B(I))
```

The compiler will vectorize and/or generate parallel code:

```
CC = MAX(CC,A(I),2.0,B(I))
```

ILX0555I Short Form: PARALLEL/VECTOR IF LOOP
Long Form: IF LOOP PARALLELIZED AND VECTORIZED BY PROCESSING AS A DO LOOP.

Explanation: Indicates when an IF loop has been vectorized and had parallel code generated for it by processing it internally as a DO loop.

Example:

```
C EXAMPLE
C PARALLELIZED AND VECTORIZED IF LOOP
  INTEGER N,I
  REAL  A(*),B(*),C(*)

  I = 0
140  CONTINUE
      I = I + 1
      IF (I.GT.N) GOTO 141
      A(I) = B(I) + C(I)
      GOTO 140
141  CONTINUE
```

Parallel code will be generated for the above IF loop and it will be vectorized since it is processed as a DO loop.

ILX0556I Short Form: CONDITIONAL PARALLEL/VECTOR STATEMENT
Long Form: PARALLELIZATION AND VECTORIZATION WAS PERFORMED ON CONDITIONALLY EXECUTED CODE. THE PARALLEL/VECTOR REPORT LISTING MAY FAIL TO INDICATE THE BRANCH STATEMENT(S) THAT AFFECT THE EXECUTION OF THIS REGION.

Explanation: Indicates when a statement or group of statements are processed conditionally. This message is intended to help clarify the compiler listing that is produced by the REPORT(XLIST) suboption of the PARALLEL and VECTOR compile-time options. This is necessary because information about the branch structure in a loop is not always reproduced when that loop is printed in the report.

In particular, the only conditional statements that are printed are logical IF statements that perform

conditional assignments. No branch statements and no block IF constructs appear in the report.

Refer to the source listing or to the listing produced by the REPORT(SLIST) suboption of the PARALLEL and VECTOR compile-time options to identify the control flow constructs that can affect the way the code being flagged by this message runs.

Example:

```
C EXAMPLE
C CONDITIONAL PARALLEL/VECTOR STATEMENT
  REAL*4 A(1000),B(1000)

CDIR PREFER PARALLEL VECTOR
DO 10 I = 1,1000
  IF (A(I).LT.0.0) GOTO 5
  B(I) = 1.1
  GOTO 6
5    B(I) = 2.2
6    A(I) = 0.0
10   CONTINUE
```

The GOTO statement at ISN 6 in the above loop will not be printed in the compiler report output produced by the REPORT(XLIST) suboption of the PARALLEL and VECTOR compile-time options.

ILX0558I Short Form: CONDITIONAL SCALAR CODE
Long Form: THIS CODE IS CONDITIONALLY EXECUTED. THE PARALLEL/VECTOR REPORT LISTING MAY FAIL TO INDICATE THE BRANCH STATEMENT(S) THAT AFFECT THE EXECUTION OF THIS REGION.

Explanation: Indicates when a statement or group of statements that has not been vectorized nor had parallel code generated for it, is part of a conditionally processed region of code. This message is intended to help clarify the compiler listing that is produced by the REPORT(XLIST) suboption of the PARALLEL and VECTOR compile-time options. This is necessary because information about the branch structure in a loop is not always reproduced when that loop is printed in the report.

In particular, the only conditional statements that are printed are logical IF statements that perform conditional assignments. No branch statements and no block IF constructs appear in the report.

Refer to the source listing or to the listing produced by the REPORT(SLIST) suboption of the PARALLEL and VECTOR compile-time options to identify the control flow constructs that can affect the way the code being flagged by this message runs.

Example:

```

C EXAMPLE
C CONDITIONAL SCALAR CODE
      REAL*4 A(1000),B(1000)

      DO 10 I = 2,999
        IF (A(I-1).LT.0.0) GOTO 5
        B(I) = B(I-1)
        GOTO 6
5       B(I) = B(I+1)
6       A(I) = B(I)
10      CONTINUE

```

The GOTO statement at ISN 6 in the above loop will not be printed in the compiler report output produced by the REPORT(XLIST) suboption of the PARALLEL and VECTOR compile-time options.

ILX0559I Short Form: IF LOOP PROCESSD AS DO LOOP
Long Form: THIS IF LOOP HAS BEEN PROCESSED AS DO LOOP BY THE COMPILER SO THAT IT IS ELIGIBLE FOR PARALLELIZATION AND VECTORIZATION ANALYSIS.

Explanation: Indicates when an IF loop has been processed internally as a DO loop, but will run in serial/scalar mode. This can be caused by a recurrence, unsupportable construct, and/or economic reasons.

Example:

```

C EXAMPLE
C IF LOOP PROCESSD AS DO LOOP
      INTEGER I
      REAL A(1000)

      I = 1
500    IF (I .GT. 1000) GOTO 505
      PRINT *, 'AT ITERATION ', I, A(I)
      I = I + 1
      GOTO 500
505    CONTINUE

```

ILX0565I Short Form: PREFER SERIAL SCALAR USED
Long Form: THIS LOOP WILL BE EXECUTED IN SCALAR BECAUSE OF THE USE OF "PREFER SERIAL SCALAR" DIRECTIVE.

Explanation: This message identifies loops to which PREFER SERIAL SCALAR directive has been applied. When this directive is specified, a loop that is considered eligible for parallel code generation or vectorization by the compiler will not be vectorized or have parallel code generated for it. Use the directive only when an analysis of the run-time performance of the loop determines that the loop runs faster when the directive is present.

Example:

```

@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER SERIAL SCALAR USED
      REAL*4 A(100000)

*DIR    PREFER SERIAL SCALAR
        DO 10 I = 1,100000
          A(I) = A(I) ** 2.1
10      CONTINUE

```

Possible Response: If the program has been modified since the directive was initially coded, or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First check whether the loop would be chosen for scalar processing if the directive was not enabled. If so, the directive is redundant and should be removed.

Otherwise, analyze the run time of the loop with and without this directive, and determine whether the performance is better when this directive is used.

ILX0567I Short Form: PREFER PARALLEL VECTOR USED
Long Form: "PREFER PARALLEL VECTOR" HAS BEEN SPECIFIED FOR THIS LOOP.

Explanation: This message identifies loops to which a PREFER PARALLEL VECTOR directive has been successfully applied. "PREFER PARALLEL VECTOR" is successful only if the loop to which it is applied is eligible for vectorization and parallel code generation. Use this directives only when an analysis of the run-time performance of the loop determines that the loop runs faster when the directive is present.

Note that after loop distribution has taken place, it is possible that PREFER PARALLEL VECTOR directive is successful for part of the code within a loop, but is unsuccessful for the rest.

Example:

```

@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER PARALLEL VECTOR USED
      REAL*4 A(1000,1000)

*DIR    PREFER PARALLEL VECTOR
        DO 10 I = 1,N
          A(I,I) = A(I,I) * 2.1
10      CONTINUE

```

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First check whether the loop would be chosen for vectorization and parallel code generation if the directive was not enabled. If so, the directive is redundant and should be removed.

Otherwise, analyze the run time of the loop with and without the directive, and determine whether the performance is better when the directive is used.

ILX0568I Short Form: INAP PREFER PARALLEL VECTOR
Long Form: "PREFER PARALLEL VECTOR" HAS BEEN SPECIFIED FOR THIS LOOP BUT COULD NOT BE HONORED BECAUSE THE LOOP WAS NOT ELIGIBLE FOR PARALLELIZATION OR VECTORIZATION. THE DIRECTIVES HAS BEEN IGNORED.

Explanation: This message identifies loops for which an inapplicable "PREFER PARALLEL VECTOR" directive has been specified. (PREFER PARALLEL VECTOR is inapplicable if the loop contains a recurrence or an unsupportable construct.)

Note that after loop distribution, the PREFER PARALLEL VECTOR directive may be inapplicable for part of the code within a loop but successful for the rest.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C INA PREFER PARALLEL VECTOR
  REAL*4 A(1000,1000),B(1000)

*DIR    PREFER PARALLEL VECTOR
        DO 10 I = 2,1000
          A(I,I) = A(I,I) * 2.1
          B(I) = B(I-1)
10      CONTINUE
```

The second statement in this loop cannot be vectorized nor have parallel code generated for it because it forms a recurrence. Therefore, the PREFER PARALLEL VECTOR directive cannot be applied to this statement. The directive is still applicable to the first statement in the loop.

Possible Response: If the program has been modified since the directive was initially coded or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

First determine whether the directive affects the vectorization and/or parallel code generation of other parts of the original loop. If not, remove them from program.

Otherwise, analyze the run time of the loop with and without the directives, and determine whether the performance is better when the directive is used.

ILX0570I Short Form: "ASSUME COUNT" USED
Long Form: THE ITERATION COUNT OF THIS LOOP WAS SPECIFIED AS "<n>" BY AN "ASSUME COUNT" DIRECTIVE FOR PARALLELIZATION AND VECTORIZATION ANALYSIS.

Explanation: This message identifies loops for which successful ASSUME COUNT directives have been specified. (ASSUME COUNT is successful if the iteration count of the loop to which it applies cannot be determined at compile time.) This directive is used to help the compiler decide whether vectorization of and/or parallel code generation for a particular loop will result in a performance improvement.

Supplemental Data:

<n> is the value of the loop iteration count that has been used for parallel and/or vector cost analysis.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C ASSUME COUNT USED
  REAL*4 A(12800,12800)

*DIR    ASSUME COUNT(12800)
        DO 10 I = 1,N
          A(I,I) = A(I,I) ** 2.1
10      CONTINUE
```

Possible Response: If the program or data has undergone revisions since ASSUME COUNT was initially coded, verify its correctness.

To do this, conduct a run-time analysis of the loop to determine whether the number specified by the directive approximates the average iteration count observed for the processing loop.

ILX0571I Short Form: INVALID "ASSUME COUNT" USED
Long Form: AN "ASSUME COUNT" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP FOR PARALLELIZATION AND VECTORIZATION ANALYSIS. THE COMPILER HAS DETERMINED THAT THE ACTUAL ITERATION COUNT IS "<n>". THE DIRECTIVE HAS BEEN IGNORED.

Explanation: This message identifies loops for which invalid ASSUME COUNT directives have been specified. (ASSUME COUNT is invalid if the iteration count of the loop to which it applies can be determined at compile time.) An invalid ASSUME COUNT directive does not affect the compilation process.

Supplemental Data:

<n> is the value of the loop iteration count that has been used for parallel and/or vector cost analysis.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C INVALID ASSUME COUNT
      REAL*4 A(12800,12800)
      PARAMETER (N=12800)

*DIR      ASSUME COUNT(1000)
          DO 10 I = 1,N
              A(I,I) = A(I,I) ** 2.1
          10      CONTINUE
```

Possible Response: Consider removing the directive from the code.

ILX0572I Short Form: "IGNORE RECRDEPS" USED
Long Form: AN "IGNORE RECRDEPS" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP FOR PARALLELIZATION AND VECTORIZATION ANALYSIS.

Explanation: This message identifies loops to which IGNORE RECRDEPS directives have been applied. It does not necessarily imply that the directive had an effect on the vectorization of and parallel code generation for the loop, although this will often be the case.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C IGNORE RECRDEPS USED
      COMMON // N
      REAL*4 A(3000),B(3000)

*DIR      IGNORE RECRDEPS
*DIR      PREFER PARALLEL
          DO 10 I = 64,1064
              IF (I.LE.128)
+          A(I) = A(I-N) ** 2.1
              B(I) = B(I) ** 2.1
          10      CONTINUE
```

Possible Response: Determine whether the directive affects vectorization and/or parallel code generation for the loop. If not, remove it to avoid unexpected side effects if the loop is modified in the future.

If the directive does alter vectorization and/or parallel code generation, try to determine why this occurred, that is, determine which potential dependences have been ignored because of the directive. (These dependences are identified by other compiler report messages that appear with the statements within a loop.)

Try to determine the run-time conditions under which these potential dependences actually arise. If you can, insert code prior to the loop to check for these conditions and to issue a message and/or stop the program when these conditions are met. (In this case, the dependence will exist if the value of the variable N is

between 1 and 64 or N is between -1 and -1000.) Note the PREFER PARALLEL directive is required for the compiler to generate parallel code for this loop.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C IGNORE RECRDEPS USED
      COMMON // N
      REAL*4 A(3000),B(3000)

          IF ((N.GE.1 .AND. N.LE.64) .OR.
+ (N.LE.-1 .AND. N.LE.-1000)) THEN
              PRINT *, 'INCORRECT IGNORE DIRECTIVE'
              STOP
          ENDIF

*DIR      IGNORE RECRDEPS
*DIR      PREFER PARALLEL
          DO 10 I = 64,1064
              IF (I.LE.128)
+          A(I) = A(I-N) ** 2.1
              B(I) = B(I) ** 2.1
          10      CONTINUE
```

ILX0573I Short Form: UNBREAKABLE RECURRENCE
Long Form: AN IGNORE DIRECTIVE WAS APPLIED TO THIS LOOP BUT DID NOT LEAD TO PARALLELIZATION OR VECTORIZATION. THIS IS BECAUSE A DEFINITE RECURRENCE EXISTS INVOLVING THE VARIABLE(S) <vlist>.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible to vectorize or generate parallel code for it. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization and/or parallel code generation are not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies loops in which a recurrence definitely exists, even if the IGNORE directive is used.

Note that for some loops, both types of recurrences can be present. Therefore, when an IGNORE directive is used, other messages may indicate that the directive was applied even though the loop did not vectorize or have parallel code generated for it.

Supplemental Data:

<vlist> is a list of the names of the variables that are involved in the recurrence. (This list includes

only the variables that could be fully analyzed.
Other variables involved in the recurrence might not appear in the list.)

Example 1:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE 1
C UNBREAKABLE RECURRENCE
  REAL  A(1000),B(1000)

*DIR  IGNORE RECRDEPS(A)
DO 10 I = 1,999
  A(I+1) = A(I) * B(I)
10  CONTINUE
```

In this example, a recurrence definitely exists because the value assigned into the array A on each iteration is used to compute the value for the next iteration. The IGNORE directive is not honored in this case, and it is not possible to vectorize or generate parallel code.

Example 2:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE 2
C UNBREAKABLE RECURRENCE
  REAL  A(2000),B(2000),C(2000)

*DIR  IGNORE RECRDEPS(A,C)
DO 10 I = 1,1000
  IF (B(I) > 0.0) THEN
    A(I+1) = A(I) * B(I)
    C(I+N) = C(I) * B(I)
  ENDIF
10  CONTINUE
```

In this example, there are two recurrences. The first involves the array A, and the second involves the array C. The first recurrence definitely exists, while the second might exist depending on the value of N; if N is less than 1 or greater than 999, no recurrence exists. Although the second recurrence can be eliminated with the IGNORE directive, the first one cannot.

Possible Response 2: Vectorization and/or parallel code generation can be performed only if you rewrite the loop so that the statements that are eligible for parallel code generation and vectorization, and the statements that are *not* eligible for parallel code generation and vectorization are in separate IF blocks.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C UNBREAKABLE RECURRENCE
  REAL  A(2000),B(2000),C(2000)

*DIR  IGNORE RECRDEPS(A,C)
*DIR  PREFER PARALLEL
DO 10 I = 1,1000
  IF (B(I) > 0.0) THEN
    A(I+1) = A(I) * B(I)
  ENDIF
  IF (B(I) > 0.0) THEN
    C(I+N) = C(I) * B(I)
  ENDIF
10  CONTINUE
```

In this modified example, the second IF block is eligible for vectorization and parallel code generation. The first IF block still contains a definite recurrence and cannot be vectorized or have parallel code generated for it. Note the use of the PREFER PARALLEL directive in conjunction with the IGNORE directive.

ILX0576I Short Form: IGNORABLE RECURRENCE

Long Form: AN IGNORE DIRECTIVE WAS APPLIED TO THIS LOOP BUT DID NOT LEAD TO PARALLELIZATION OR VECTORIZATION. THIS IS BECAUSE A POSSIBLE RECURRENCE EXISTS INVOLVING THE ARRAY(S) <alist>. THESE ARRAYS WERE NOT SPECIFIED IN THE DIRECTIVE. IF NO RECURRENCE EXISTS, PARALLELIZATION AND VECTORIZATION CAN ACHIEVED BY MODIFYING THE DIRECTIVE.

Explanation: A recurrence is a group of one or more statements in a loop that use data in a way that makes it impossible for the compiler to vectorize or generate parallel code for the loop. The compiler detects recurrences by analyzing scalar variables, array subscripts, EQUIVALENCE relationships, IF statements, loop bounds, and loop increments.

The presence or absence of a recurrence sometimes depends on data that cannot be determined until run time; for example, a dummy argument may be referenced inside a subscript. In such cases, a recurrence is assumed and vectorization and parallel code generation are not performed. If the IGNORE directive is used, the recurrence is assumed to be absent; however, the IGNORE directive has no effect if a recurrence definitely exists.

This message identifies loops in which a possible recurrence was not eliminated, even when the IGNORE directive was used. It indicates that the variables responsible for the possible recurrence were not specified in the list of variables to which the directive should be applied. Note the use of the PREFER

PARALLEL directive in conjunction with the IGNORE directive.

Supplemental Data:

<alist> is a list of the names of arrays that are involved in the possible recurrence. (This list includes only the variables that could be fully analyzed. Other variables involved in the recurrence might not appear in the list.)

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C IGNORABLE RECURRENCE
  REAL  A(2000),B(2000),C(2000)

*DIR  IGNORE RECRDEPS(A)
*DIR  PREFER PARALLEL
DO 10 I = 1,1000
  IF (B(I) > 0.0) THEN
    A(I+N) = A(I) * B(I)
    C(I+M) = C(I) * B(I)
  ENDIF
10 CONTINUE
```

In this example, there are two possible recurrences. The first involves the array A, and the second involves the array C. Whether the recurrences exist depends on the values of M and N; if these values are less than 1 or greater than 999, no recurrence exists.

The IGNORE directive is specified for array A only, and it is assumed that a recurrence involving A does not exist. However, it is assumed that a recurrence involving C does exist. Since partial vectorization and/or parallel code generation are not possible for statements in the same IF block, no parallel code generation or vectorization can be performed.

Possible Response 1: If you are certain that a recurrence will never exist (in this example, that M is always less than 1 or greater than 999), you can achieve vectorization and parallel code generation by modifying the IGNORE directive so that it applies to the indicated array or arrays.

Modified Example 1:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 1
C IGNORABLE RECURRENCE
  REAL  A(2000),B(2000),C(2000)

*DIR  IGNORE RECRDEPS(A,C)
*DIR  PREFER PARALLEL
DO 10 I = 1,1000
  IF (B(I) > 0.0) THEN
    A(I+N) = A(I) * B(I)
    C(I+M) = C(I) * B(I)
  ENDIF
10 CONTINUE
```

Possible Response 2: If a recurrence can exist (in this example, that M may be greater than or equal to 1

or less than or equal to 999), rewrite the loop so that the statements that are eligible for parallel code generation and vectorization, and the statements that are *not* eligible for parallel code generation and vectorization are in separate IF blocks.

Modified Example 2:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE 2
C IGNORABLE RECURRENCE
  REAL  A(2000),B(2000),C(2000)

*DIR  IGNORE RECRDEPS(A)
*DIR  PREFER PARALLEL
DO 10 I = 1,1000
  IF (B(I) > 0.0) THEN
    A(I+N) = A(I) * B(I)
  ENDIF
  IF (B(I) > 0.0) THEN
    C(I+M) = C(I) * B(I)
  ENDIF
10 CONTINUE
```

In this modified example, since the directive applied only to the array A, the first IF block is eligible for parallel code generation or vectorization. The second IF block will run in scalar/serial mode. Note that the PREFER PARALLEL directive is needed to generate parallel code for this loop.

ILX0577I Short Form: POTENTIAL RECRDEP ELIMINATE
Long Form: POTENTIAL BACKWARD DEPENDENCE(S) INVOLVING THE ARRAY(S) <alist> HAVE BEEN IGNORED FOR PARALLELIZATION AND VECTORIZATION BECAUSE OF AN "IGNORE RECRDEPS" DIRECTIVE APPLIED TO THE LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive has caused the compiler to ignore some backward dependence that it would otherwise have assumed to exist. The message does not necessarily imply that the directive affected the vectorization of and parallel code generation for the loop, although this will often be the case.

Supplemental Data:

<alist> is a list of the names of the arrays involved in the ignored dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They may, however, differ from the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C POTENTIAL RECRDEP ELIMINATE
COMMON // N
REAL*4 A(-1000:2000)

*DIR    IGNORE RECRDEPS
*DIR    PREFER PARALLEL
DO 10 I = 1,1000
    A(I) = A(I+N) ** 2.1
10      CONTINUE
```

Possible Response: Determine whether the directive affects vectorization of and/or parallel code generation for the loop in which the statement occurs. If not, remove it to avoid unexpected side effects if the loop is modified in the future.

If the directive does alter parallel code generation or vectorization, try to determine the run-time conditions under which these potential backward dependences actually arise. (In this case, there will be a dependence if the value of the variable N is between -999 and -1.) Insert code prior to the loop to check for these conditions and to issue a message and/or stop the program when these conditions are met. Note that the PREFER PARALLEL directive is needed to generate parallel code for this loop.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C POTENTIAL RECRDEP ELIMINATED
COMMON // N
REAL*4 A(-1000:2000)

IF (N.LT.0 .AND. N.GT.-1000) THEN
    PRINT *, 'INCORRECT IGNORE DIRECTIVE'
    STOP
ENDIF
*DIR    IGNORE RECRDEPS
*DIR    PREFER PARALLEL
DO 10 I = 1,1000
    A(I) = A(I+N) ** 2.1
10      CONTINUE
```

ILX0579I Short Form: ACTUAL RECRDEP NOT IGNORED
Long Form: BACKWARD DEPENDENCES INVOLVING THE ARRAY(S) <alist> WHICH WERE IN THE RANGE OF "IGNORE RECRDEPS" DIRECTIVE(S) APPLIED TO THE LOOP(S) AT NESTING LEVEL(S) <levlist> HAVE NOT BEEN IGNORED FOR PARALLELIZATION OR VECTORIZATION BECAUSE THE COMPILER HAS DETERMINED THAT THESE DEPENDENCES ARE ALWAYS PRESENT.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive could have been applied but where the compiler has chosen not to do so because the subject dependences are always present. (This directive is only honored when the compiler determines that there is a potential dependence but that the dependence will not arise under certain run-time conditions.) Even if this message is present, the directive may have affected the parallel code generation or vectorization of the loop, since some other backward dependences may have been ignored.

Supplemental Data:

<alist> is a list of the names of the variables that carry the dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They may, however, differ from the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C ACTUAL RECRDEP NOT IGNORED
REAL*4 A(1000)

*DIR    IGNORE RECRDEPS
*DIR    PREFER PARALLEL
DO 10 I = 2,1000
    A(I) = A(I-1) ** 2.1
10      CONTINUE
```

The IGNORE directive cannot be honored in this case since a recurrence definitely exists. Note that the PREFER PARALLEL directive is needed to generate parallel code for this loop.

Possible Response: Determine whether the directive affected the vectorization of and/or parallel code generation for other statements used in the loop. If not, remove the directive.

ILX0580I Short Form: POTENTIAL RECRDEP MODIFIED
Long Form: DEPENDENCES INVOLVING THE ARRAY(S) <alist> WERE PRESUMED NOT TO BE INTERCHANGE PREVENTING FOR VECTORIZATION BECAUSE OF AN "IGNORE RECRDEPS" DIRECTIVE APPLIED TO THE LOOP(S) AT NESTING LEVEL(S) <levlist>.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive has caused the compiler to assume that some forward dependence is not interchange preventing. This happens only when

the compiler is uncertain whether a dependence is really interchange preventing.

Usually, the presence of an interchange-preventing dependence restricts vectorization. When an interchange-preventing dependence exists, the reordering, or interchange, of two loops would cause different results to be produced. Since vectorization of an outer loop has the same effect as moving that loop to the innermost position, the existence of an interchange-preventing dependence carried by an outer loop prevents vectorization.

To understand how an interchange-preventing dependence comes about, study the following example:

```
      DO 10 I=1,2
      DO 10 J=1,2
10      A(I-1,J+1)=A(I,J)
```

In this code, the element A(1,2) is fetched when I=1 and J=2 and is stored into when I=2 and J=1. When these loops are processed, the fetch will occur before the store. However, if the loops were interchanged, the store would come first and different results would probably be produced.

It is not always possible for the compiler to determine whether or not a dependence is interchange preventing. Unless it can prove otherwise, the compiler will normally assume that a given dependence is interchange preventing. This will insure that correct results are always produced after vectorization, even though some potential vectorization may be missed.

In cases where the compiler is unable to determine whether or not a particular dependence is interchange preventing, the IGNORE RECRDEPS directive allows you to force the compiler to assume that the dependence is not interchange preventing.

Supplemental Data:

<alist> is a list of the names of the arrays involved in the modified dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C POTENTIAL RECRDEP MODIFIED
      REAL*4 U(1000,1000,1000)

*DIR      IGNORE RECRDEPS
          DO 190 K = 1, 999
          DO 190 J = 1, 999
          DO 190 I = 1, 999
            U(I,J,K) = U(I+N,J,K) + U(I,J+N,K)
          +          U(I,J,K+N)
190      CONTINUE
```

Possible Response: Determine whether the directive affects vectorization of the loop in which the statement occurs. If not, remove the directive to avoid unexpected side effects if the loop is modified in the future.

If the directive does alter vectorization, try to determine the run-time conditions under which these dependences might be interchange preventing. Insert code prior to the loop to check for these conditions and to issue a message and/or stop the program when these conditions are met.

Note: The table of ignored dependences that appears after the compiler report message listing can help identify these dependences.

ILX0581I Short Form: ACTUAL RECRDEP NOT MODIFIED
Long Form: INTERCHANGE PREVENTING DEPENDENCES INVOLVING THE ARRAY(S) <alist> THAT WERE IN THE RANGE OF "IGNORE RECRDEPS" DIRECTIVE(S) APPLIED TO THE LOOP(S) AT LEVEL(S) <levlist> HAVE BEEN PRESERVED FOR VECTORIZATION BECAUSE THE COMPILER HAS DETERMINED THAT THESE DEPENDENCES DEFINITELY EXIST.

Explanation: This message identifies the statements where an IGNORE RECRDEPS directive could have been applied to some interchange-preventing dependences but where the compiler has chosen not to do so because the subject dependences are always present.

Usually, the presence of an interchange-preventing dependence restricts vectorization. When an interchange-preventing dependence exists, the reordering, or interchange, of two loops would cause different results to be produced. Since vectorization of an outer loop has the same effect as moving that loop to the innermost position, the existence of an interchange-preventing dependence carried by an outer loop prevents vectorization.

To understand how an interchange-preventing dependence comes about, study the following example:

```

      DO 10 I=1,2
      DO 10 J=1,2
10      A(I-1,J+1)=A(I,J)

```

In this code, the element A(1,2) is fetched when I=1 and J=2 and is stored into when I=2 and J=1. When these loops are processed, the fetch will occur before the store. However, if the loops were interchanged, the store would come first and different results would probably be produced.

It is not always possible for the compiler to determine whether or not a dependence is interchange preventing. Unless it can prove otherwise, the compiler will normally assume that a given dependence is interchange preventing. This will insure that correct results are always produced after vectorization, even though some potential vectorization may be missed.

Normally, an IGNORE RECRDEPS directive would cause the compiler to assume that a dependence is not interchange preventing. However, in cases where the compiler is absolutely certain that a dependence is interchange preventing, the existence of the IGNORE RECRDEPS directive will have no effect on the analysis of a program.

Note that even if this message is present, the directive may have affected the vectorization of the loop, since some other backward dependences may have been ignored.

Supplemental Data:

<alist> is a list of the names of the variables that carry the dependences.

<levlist> is a list of the relative nesting levels of the loops that carry the dependences.

Note: These levels correspond to the nesting indicated by the nesting level brackets that appear on the compiler report. They may, however, differ from the nesting level indications that appear on the source listing.

Example:

```

@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C ACTUAL RECRDEP NOT MODIFIED
      REAL*4 A(1000,1000)

*DIR      IGNORE RECRDEPS
      DO 10 I = 1,999
      DO 10 J = 2,1000
      A(I,J) = A(I+1,J-1) ** 2.1
10      CONTINUE

```

The IGNORE directive cannot be honored in this case since the dependence is definitely interchange preventing.

Possible Response: Determine whether the directive affected the vectorization of other statements used in the loop; if not, remove the directive.

ILX0602I Short Form: PREFER SERIAL VECTOR USED Long Form: THIS LOOP WILL BE EXECUTED IN SERIAL AND VECTOR BECAUSE OF THE USE OF A "PREFER SERIAL VECTOR" DIRECTIVE.

Explanation: This message identifies loops to which a PREFER SERIAL VECTOR directive has been applied. When this directive is specified, a loop that is considered eligible for parallel code generation or vectorization by the compiler will be vectorized only. It should be used only when an analysis of the run-time performance of the loop has determined that the loop runs faster when the directive is present.

Example:

```

@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER SERIAL VECTOR USED
      REAL*4 A(1000000)

*DIR      PREFER SERIAL VECTOR
      DO 10 I = 1,1000000
      A(I) = A(I) ** 2.1
10      CONTINUE

```

Possible Response: If the program has been modified since the directive was initially coded, or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

Analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0603I Short Form: PREFER PARALLEL SCALAR USED Long Form: THIS LOOP WILL BE EXECUTED IN PARALLEL AND SCALAR BECAUSE OF THE USE OF A "PREFER PARALLEL SCALAR" DIRECTIVE.

Explanation: This message identifies loops to which a PREFER PARALLEL SCALAR directive has been applied. When this directive is specified, a loop that is considered eligible for parallel code generation or vectorization by the compiler will be run in parallel mode only. It should be used only when an analysis of the run-time performance of the loop has determined that the loop runs faster when the directive is present.

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C PREFER PARALLEL SCALAR USED
  REAL*4 A(100000)

*DIR    PREFER PARALLEL SCALAR
        DO 10 I = 1,100000
          A(I) = A(I) ** 2.1
10      CONTINUE
```

Possible Response: If the program has been modified since the directive was initially coded, or if the program is being compiled on a release of the compiler different from the one on which it was originally developed, it may be important to verify the appropriateness of this directive.

Analyze the run time of the loop with and without this directive specified, and determine whether or not the performance is better when this directive is used.

ILX0604I Short Form: USER EXTERNAL REFERENCE Long Form: THIS LOOP COULD NOT BE VECTORIZED BECAUSE OF THE USER EXTERNAL REFERENCE(S) <flist>.

Explanation: Indicates that a loop contains reference(s) to external reference(s) and as a consequence is not eligible for vectorization.

Supplemental Data:

<flist> is a list consisting of function and subroutine names and the ISNs (internal statement numbers) of the statements in which they are used.

Example:

```
C EXAMPLE
C EXTERNAL USER REFERENCE
  REAL *4 B(1000)
  COMMON /AC/ A(1000)

        DO 170 I=1,1000
          A(I) = A(I) ** 2.1
170      CALL SUB1
```

Note that because of the use of COMMON the above loop could not be automatically split into two loops to obtain partial vectorization.

Possible Response: Break the loops into two or more loops so that any condition(s) that are not eligible for vectorization are separated from the portions of the original loop that are eligible for vectorization. Do this only when you are absolutely certain that the transformation will not alter the results of your program.

Modified Example:

```
C POSSIBLE RESPONSE
C EXTERNAL USER REFERENCE
  REAL *4 B(1000)
  COMMON /AC/ A(1000)

        DO 170 I=1,1000
          A(I) = A(I) ** 2.1

        DO 180 I=1,1000
180      CALL SUB1
```

ILX0624I Short Form: INAPPLICABLE "PREFER VECTOR" Long Form: A "PREFER VECTOR" DIRECTIVE HAS BEEN SPECIFIED FOR THIS "PARALLEL DO" LANGUAGE CONSTRUCT BUT COULD NOT BE HONORED BECAUSE IT IS NOT THE INNERMOST LOOP. THE DIRECTIVE HAS BEEN IGNORED.

Explanation: This message identifies "PARALLEL DO" language constructs for which inapplicable PREFER VECTOR directives have been specified. (PREFER VECTOR is inapplicable if the language construct is not the innermost loop in the nest of loops.)

Example:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C INAPPLICABLE PREFER VECTOR
  REAL*4 A(10000,10000),B(10000,10000)

*DIR    PREFER VECTOR
        PARALLEL DO 10 I = 1,10000
          PARALLEL DO 10 J = 1,10000
            A(I,J) = A(I,J) * 2.1
            B(I,J) = B(I,J)
10      CONTINUE
```

Possible Response: There are two possible responses to this case. The directive could be switched to the innermost loop, or the outermost loop along with the directive could be switched to become the innermost loop. One should perform an analysis of the program to ensure that the semantics are maintained and that optimal run-time performance is obtained.

Modified Example:

```
@PROCESS DIRECTIVE('DIR')
C POSSIBLE RESPONSE
C INAPPLICABLE PREFER VECTOR
  REAL*4 A(10000,10000),B(10000,10000)

        PARALLEL DO 10 J = 1,10000
*DIR    PREFER VECTOR
          PARALLEL DO 10 I = 1,10000
            A(I,J) = A(I,J) * 2.1
            B(I,J) = B(I,J)
10      CONTINUE
```

**ILX0625I Short Form: PARALLEL DO
VECTORIZED Long Form: THE
LANGUAGE CONSTRUCT "PARALLEL
DO" HAS BEEN PARALLELIZED AND
VECTORIZED.**

Explanation: Indicates when a "PARALLEL DO"
language construct has been compiled for vectorization
and parallel code generation.

Example:

```
C EXAMPLE
C PARALLEL DO VECTORIZED
      REAL  A(10000,10000),B(10000,10000)

      PARALLEL DO 140 J = 1,10000
*DIR      PREFER VECTOR
      PARALLEL DO 140 I = 1,10000
          A(I,J) = A(I,J) ** 2.1
          B(I,J) = B(I,J) ** 2.1
140  CONTINUE
```

Index

Special Characters

xi

@PROCESS statement

CMS, specifying compile-time options under 7

compatibility with previous releases 480

MVS with TSO, specifying compile-time options under 19

MVS, specifying compile-time options under 14

(pound sign), assembler subprograms and 451

Numerics

0 informational code 48

0, in operator message 123

12 severe error code 48

16 abnormal termination code 48

31-bit addressing 68, 88

4 warning error code 48

8 error code 48

A

abbreviations, for run-time options 106

abnormal termination

dump, requesting 84, 122

ABSDUMP run-time option

description 107

access method services cataloging DEFINE

command 219

access methods

choosing 138, 152

direct 139, 181

keyed 139, 182

operating system 139

sequential 139, 163

access register mode 452

actual argument

rules for use 371

adding data to a file

new records 188

replacing records 189

address

extended 452

AFBTRAC 330

AFBVLPRM 117

AFBVRSEP module 430

AFBVSFST 440

affinity, processor

See processor affinity

ALLOCATE command 135

under TSO 18

alternative mathematical library routines 479

American National Standard (ANS)

flagging for Fortran 50

AMODE attribute 68, 88

antidependence 262

ANZCALL, PARALLEL suboption 34

ANZCALL, VECTOR suboption 36, 39

ARGSTR service subroutine 68, 84

argument

actual

passed by reference 372

array and assembler subprograms 464, 465

assembler programs and 464, 469

assigning values to 372

COMMON statement and 372

dummy 372

general rules 371

passed by reference 372

passing between programs 371

passing character 483

subprograms and 371

transparent, passing 483

variable and assembler subprograms 464, 465

arithmetic

efficiency, for optimization 273

errors, common 129

array

adjustable dimensioned, recommendation

against 272

as actual argument 372

assembler subprograms and 464, 465

efficient common arrangement 375

initializing efficiently 272

initializing, common error 129

optimizing identically dimensioned 272

subscript references invalid, common error 129

ASCII/ISCI encoded file, record format 149

assembler language

common data in 451

extended common and 380

Fortran data 451

initializing run-time environment 457

internal representation of data 469

linkage conventions 453, 455

LIST option listing and 127

register conventions 453, 455

requesting compilation 473

retrieving arguments 464

subprogram 451

calling from Fortran 460

VFEIN# and VFEIL# entry points 457

- assigned name form
 - consequences under CMS 430
 - consequences under MVS 439
 - general description 428
- ASSUME COUNT directive 356
- asynchronous
 - I/O 179
- AUTODBL compile-time option 22, 51
 - conversion values for 24
 - definition of 22
 - equivalent values for 25
 - using automatic precision increase facility with 51
- automatic
 - precision increase facility
 - by means of AUTODBL 51
 - precision conversion process, padding 52
 - precision conversion process, promotion 51
- AUTOMATIC, PARALLEL suboption 33, 297
- AUTOTASK
 - DD statement 416
 - keyword 399, 416
 - run-time option 107
 - PARALLEL run-time option and 107, 113

B

- background command procedure under TSO 97
- BACKSPACE statement
 - invalid for directly accessed VSAM direct file 224
 - keyed access 189
 - sequential access 170
 - sequentially accessed VSAM direct file 223
 - VSAM sequential file considerations 222
- backward dependence 263
- bimodal CMS 68
- blank
 - common block
 - See common block, blank
- block data subprograms
 - coding example 378
 - initializing 378
- block IF statement
 - nested levels 449
- blocked records 140
- blocking 140
- buffers, I/O 175
- BUFNO 175, 199, 200

C

- CALL command under TSO 19, 95
- CALL loader option under MVS 81
- CALL statement
 - IGNFHDD 155
 - IGNFHU 155

- calling and called programs
 - assembler language considerations 451
 - detailed description 371
 - differences among VS FORTRAN Version 2, Version 1, and current implementations 479
 - internal limits in VS FORTRAN Version 2 449
 - invoking the Fortran compiler 473
- card punch file under CMS 146
- card reader file under CMS 146
- cataloged procedures
 - compile-only under MVS 13
 - MVS compiler data set 14
 - using for program output 84
- cataloging
 - and loading alternate index 217
 - entry in a VSAM catalog 219
- CDUMP/CPDUMP routine
 - coordinating parallel use of with locks 319
- character
 - arguments, passing 483
 - data type, internal representation of 470
- CHARLEN compile-time option 25
- chunking for parallel processing
 - definition of 258
 - vector section size and 265
- CI compile-time option 25
- CLEN suboption of ICA compile-time option 30
- CLISTs under TSO 96
- CLOCK routine 368
- CLOCKX routine 368
- CLOSE statement
 - deleting files 157
 - disconnecting files 157
 - NOOCSTATUS run-time option 157
 - OCSTATUS run-time option 157
 - retaining files 157
 - specifiers 157
- CMPLXOPT, VECTOR suboption 36, 40, 284
- CMS considerations
 - compilation 7
 - compile-time options and 8
 - file existence tables 235
 - invoking the VS FORTRAN Version 2 compiler 473
 - linkage conventions 453, 455
 - operating system support xiv
 - specifying run-time options 68
- CMS LOADLIB, changing name of 435
- CNVIOERR run-time option 108
- code independence 407
- codes
 - abnormal termination 48
 - error 48
 - informational 48
 - severe error 48
 - unrecoverable error 48
 - warning error 48

- coding errors, avoiding 129
- coding your program
 - coding errors to avoid 129
 - sharing data 371
- combined link libraries 64, 438
- common block
 - blank
 - description 378
 - must be unnamed 378
 - only one allowed 378
 - coding errors in source 129
 - dynamic
 - 16-megabyte line, using above 70, 91
 - assembler subprograms and 452
 - conflicting storage assignment 387
 - considerations 407
 - DC option and 380
 - description 379
 - obtaining storage for 69
 - expression elimination, OPTIMIZE(3) 255
 - extended
 - assembler subprograms and 452
 - compatibility considerations 484
 - conflicting storage assignment 387
 - description 379
 - EC option and 27, 380
 - ECPACK run-time option and 109
 - EMODE compile-time option and 27
 - RECPAD run-time option and 114
 - named
 - conflicting storage assignment 387
 - description 378
 - length restriction 378
 - parallel program, using in 308
 - static
 - assembler subprograms and 452
 - description 379
 - SC option and 34, 380
 - static, recompilation requirement for use with
 - parallel 309
 - storage maps and 47
- COMMON statement
 - argument usage 372
 - assembler programs and 451, 474
 - blank common 378
 - description of use 371
 - dummy variables for alignment 375
 - efficient data arrangement 375
 - EQUIVALENCE considerations 377
 - fixed order variable alignment 375
 - named common 378
 - passing subroutine arguments using 271
 - referencing shared data 374
 - storage maps and 47
 - transmitting values using 373
 - using efficiently 270

- compilation
 - CMS, requesting under 7
 - identification 41
 - modification of defaults 41
 - MVS with TSO, requesting under 18
 - MVS, requesting under 13
 - statistics in object listing 128
- compile-time
 - options
 - AUTODBL 22, 51
 - CHARLEN 25
 - CI 25
 - CMS, @PROCESS statement and 7
 - conflicting 40
 - DBCS 26
 - DC 26, 379
 - DDIM 26
 - DECK 26
 - defaults for 21
 - DIRECTIVE 26
 - DISK 8
 - DYNAMIC 27, 97
 - EC 27, 379
 - EMODE 27, 379
 - EXEC statement in MVS 13
 - FIPS 28
 - FIXED 28
 - FLAG 28
 - FORTVS2 command 8
 - FREE 28
 - GOSTMT 28
 - HALT 28
 - ICA 28
 - IL 30
 - LANGLVL 66 31
 - LANGLVL 77 31
 - LINECOUNT 31
 - LIST 31
 - MAP 31
 - MVS with TSO, @PROCESS statement and 19
 - MVS, @PROCESS statement and 14
 - NAME 31
 - NODBCS 26
 - NODDIM 26
 - NODECK 26
 - NODIRECTIVE 26
 - NOEMODE 27
 - NOFIPS 28
 - NOGOSTMT 28
 - NOICA 28
 - NOLIST 31
 - NOMAP 31
 - NOOBJECT 31
 - NOOPTIMIZE 31
 - NOPARALLEL 31
 - NOPRINT 8
 - NORENT 34

compile-time (*continued*)

options (*continued*)

- NOSAA 34, 50
- NOSDUMP 35
- NOSOURCE 35
- NOSRCFLG 35
- NOSXM 35
- NOSYM 35
- NOTERMINAL 35
- NOTEST 36
- NOTRMFLG 36
- NOVECTOR 36
- NOXREF 40
- OBJECT 31
- OPTIMIZE 0 31
- OPTIMIZE 1 31
- OPTIMIZE 2 31
- OPTIMIZE 3 31
- PARALLEL 31, 297
- PRINT 8
- PTRSIZE 34
- RENT 34
- SAA 34, 50
- SC 34, 379
- SDUMP 35
- SOURCE 35
- SRCFLG 35
- SXM 35
- SYM 35
- TERMINAL 35
- TEST 36
- TRMFLG 36
- VECTOR 36
- XREF 40

compiler

considerations 253

directives

See parallel, directives

See vector, directives

invoking from assembler 473

messages (appear only online)

See *also* diagnostic messages

format of 47

MVS data sets 14

output

cross reference listing, XREF option 44

default options and 41

dependent on options in effect 17, 20

end-of-compilation message 49

LIST data set under TSO 20

listing 41

LISTING file for CMS 10

listing, header 41

message listing, FLAG option 47

OBJ data set under TSO 20

object module for MVS 17

output file for MVS 17

compiler (*continued*)

output (*continued*)

- program information file (PIF), IVA compiler
 - suboption and 38
- Program Information File (PIF), under MVS 14
- source program listing 42
- standard language flagging 50
- storage map listing, MAP option 44
- TEXT file for CMS 10

report

- defaults for 32, 36
- diagnostic messages 351
- diagnostic messages, parallel 528
- diagnostic messages, vector 491
- diagnostic messages, vector and parallel 549
- printed listing (LIST, SLIST or XLIST) 32, 36
- printed, examples 345
- specifying REPORT option 32, 36
- terminal display (TERM) 32, 36
- terminal display (TERM), examples 344

compiling your source program 7, 21

complex data type

internal representation 471

compound instructions 286

computational independence

definition for MTF 400

definition for parallel processing 258

definition of 296

identifying for MTF 407

parallel processing, using with 296

conflicting compile-time options 40

connecting files and units

changing connection properties 154

direct access 152

file definition statement 150

keyed access 152, 183

named files 150

OCSTATUS run-time option 152

OPEN statement, using an 150

preconnection, using 143

reconnecting

named files 160

preconnected files 161

unnamed files 160

sequential access 143, 152

temporary files 152

unnamed files 151

constant

expressions, how compiler recognizes 272

operands, recognition of 271

restrictions as actual argument 372

construct, definition 258

control

dependence 262

controlled access to data

See data, sharing

- CPUTIME routine
 - using in parallel 324
- critical variables, limitations on optimizing 269
- cross reference dictionary
 - description 45
 - using the 44
 - XREF option requests 40
- cross-compilation, automatic xiv
- current standard, flagging for 50
- CVAR suboption of ICA compile-time option 30

D

- DASD
 - named file I/O under MVS 175
 - unnamed file I/O under MVS 175
- data
 - dependence 262
 - efficient arrangement, common areas 375
 - independence 400, 407
 - sets
 - partitioned 180
 - sharing
 - between programs 371
 - controlled access to 296
 - library lock and event services, controlling with 296
 - parallel processing and 296
 - storage for 371
 - transfer rate, DASD I/O 175
- data space
 - ECPACK run-time option and 109
 - ESA/390 27, 379
 - RECPAD run-time option and 114
- Data Spaces 459
- DATA statement
 - named common and 378
- date
 - in output listing header 41
 - of compilation, compiler default 41
- DBCS
 - See double-byte character set
- DC compile-time option 26, 379, 380
- DCB parameter
 - default values 15
 - default values for load module execution data set 83
 - default values for load module execution direct access data set 83
 - defines MVS record 147
- DCSS
 - and reentrant programs 435
 - loading compiler from 9
- DD statement 135
 - description 13
 - direct access label and 147

- DD statement (*continued*)
 - tape label and 147
 - VSAM file processing 219
- DDIM compile-time option 26
 - with parallel 305
- ddnames
 - direct access 137
 - error message unit 137
 - keyed access 184
 - on FILE specifier 136
 - sequential access 137
 - unnamed files 150, 151
- deadlock, definition of 323
- debug packets 126
- DEBUG run-time option 108
 - CMS, specification under 68
 - description 108
 - MVS, specification under 84
 - TSO, specification under 96
- debug statements 125
- debug, interactive
 - See VS FORTRAN Version 2, interactive debug
- debugging
 - dump, formatted 129
 - extended error handling and 123
 - GOSTMT option and 121
 - static debug example 126
 - static debug statements for 125
- DEBUNIT run-time option 108
- DECK compile-time option 26
- DECLARED column in cross reference 46
- DEF suboption of ICA compile-time option 29, 388
- default
 - name form
 - consequences under CMS 430
 - consequences under MVS 439
 - general description 428
 - run-time options table 117
- defaults
 - extended error handling 123
 - modification of 41
- DEFINE command, VSAM
 - creates catalog entry 213
 - direct file 214
 - keyed file 213
 - processing of 219
 - sequential file 214
- defining
 - record 147
- DELETE statement 189
- dependence
 - classifications
 - direction 263
 - interchangeability 264
 - level 264
 - mode 262
 - type 262

- dependence (*continued*)
 - definition 258
- dependences, table of ignored 352
- determining file existence
 - See file, existence
- diagnostic messages
 - compiler
 - default 41
 - example 48
 - module identifier in 48
 - output and 17, 20
 - compiler report, in 351, 491
 - vector 492
 - GOSTMT option and 121
 - ILX compiler message prefix 48
 - listing, FLAG option 47
 - message number identifies 48
 - operator 122
 - self-explanatory 47
 - severity level in 48
 - traceback map and 119
- differences among VS FORTRAN versions 479
- direct access
 - connecting 150, 181
 - default record formats 141
 - description 139, 181
 - disconnecting 157
 - endfile record 181
 - file organization 139, 181
 - formatted I/O 191
 - reading data 182
 - unformatted I/O 193
 - unnamed files 181
 - writing data 182
- direct address 452
- direct file processing
 - CMS FILEDEF command and 144
 - record format 149
 - VSAM
 - considerations 209
 - direct access for 224
 - sequential access for 223
 - source language 222
 - valid source statements, summary 220
- DIRECTIVE compile-time option 26
- directives
 - parallel
 - See parallel, directives
 - vector
 - See vector, directives
- disconnecting
 - units and files
 - at program termination 134, 159
 - CLOSE statement, using a 157
 - deleting files 157
 - OPEN statement, using an 159
 - retaining files 157
- DISK compile-time option 8
- disk files 231
- displacement column, in storage map 47
- displacement, definition 375
- DISPLAY statement 449
- DLBL command 134
- DO loop
 - extended range of 482
 - validity for parallel processing, VS FORTRAN
 - ensures 294
- DO statement
 - control transfers into, common coding error 129
 - implied 268
 - writing loops inline 273
- DOAFTER statement
 - example of 300
 - inhibits automatic parallel code generation 297
 - lock provided for 299
 - parallel loop and 299
 - processing priority 299
- DOBEFORE statement
 - example of 300
 - inhibits automatic parallel code generation 297
 - lock provided for 299
 - parallel loop and 299
 - processing priority 299
- DOEVERY statement
 - example of 300
 - inhibits automatic parallel code generation 297
 - parallel loop and 299
 - processing priority 299
- double precision
 - data type 375
- double-byte character data
 - See double-byte character set
- double-byte character set
 - CHAR specifier on INQUIRE statement 205
 - CHAR specifier on OPEN statement 153, 205
 - DBCS compile-time option 26
 - defaults for preconnected files 154
 - formatted I/O 205
 - keyed access, considerations for 205
 - MAP compile-time option 47
 - NODBCS compile-time option 26
 - NOMAP compile-time option 47
 - NOXREF compile-time option 46
 - unformatted I/O 205
 - XREF compile-time option 46
- DSPTCH subroutine 399
- dual-execution mode 452
- dummy argument
 - restrictions on assigning values 372
 - rules for use 371
- dummy variables, alignment using 375
- dump
 - formatted 129

- dump (*continued*)
 - requesting 129
- DUMP/PDUMP routine
 - coordinating parallel use of with locks 319
- duplicate expressions, how compiler recognizes 272
- DVCHK routine
 - coordinating parallel use of with locks 319
- dynamic common
 - description 379
- dynamic common block
 - See common block, dynamic
- DYNAMIC compile-time option 27, 63, 75
- dynamic file
 - definition 135
 - defaults for file characteristics 197
 - defaults for file characteristics (CMS) 227
 - defaults for file characteristics (MVS) 149
 - deleting files 157
 - FILEINF routine 198
 - inquiring about 202
 - named 136, 138, 150, 195
 - space calculation under MVS 200
 - unnamed 195
- Dynamically loading modules 64, 65, 76, 77, 78, 92, 96, 97

E

- E error code 48
- EBCDIC
 - data set record format 149
- EC compile-time option 27, 379, 380
- ECPACK run-time option 109
- ELIG status flag 344, 350
- elimination of instructions, OPTIMIZE(3) 255
- EMODE compile-time option 27, 379
- END SECTIONS statement
 - example of 302
 - inhibits automatic parallel code generation 297
 - parallel sections and 301
- end-of-compilation message 49
- endfile record
 - definition 140
 - encountering 164
 - writing 167
- ENDFILE statement 167
 - VSAM files treat as documentation 220
- enhancements to product xiv
- entry point, traceback map lists 120
- entry points
 - VFEIN# and VFEIL# 457
- EP loader option under MVS 81
- EQUIVALENCE statement
 - COMMON statement and 377
 - errors using 377, 378
 - invalid references, common error 129

- EQUIVALENCE statement (*continued*)
 - optimization and 269
- ERRMON subroutine 123
 - using in parallel 325
- error
 - fixing user 129
 - handling
 - effects of VS FORTRAN Version 2 interactive debug on 125
 - extended 123
 - identifying run-time 118
 - message unit 110, 137, 138, 408
 - occurrences, warning on number of 124
 - option table 123
 - parallel threads and 325
 - summary, in traceback map 119
 - to avoid 129
- error messages
 - See diagnostic messages
- ERRSAV subroutine 123
 - using in parallel 325
- ERRSET subroutine
 - changes entry in option table 124
 - requests traceback maps 119
 - using in parallel 325
- ERRSTR subroutine 123
 - using in parallel 325
- ERRTRA subroutine
 - using in parallel 325
- ERRUNIT run-time option 108
- ESA/390 data space 27, 379, 452
- ESDS
 - defining an 214
 - source language considerations 220
 - VSAM sequential file 210
- events, parallel
 - creating 319
 - cycle
 - definition of 319
 - types of 319
 - definition of 317
 - post signals 319
 - uses for 322
 - wait signals 319
- exception codes 121
- EXEC control statement, MVS
 - description 13
 - linkage editor options under MVS 78
 - loader
 - data set 82
 - processing options 81
- execution, compiler
 - See compilation
- existence tables, CMS
 - DASD files 235
 - file definitions specifying DUMMY 240

- existence tables, CMS (*continued*)
 - files on other devices 241
 - library member files 239
 - nonreusable VSAM files 238
 - reusable VSAM files 235
 - tape files 239
 - terminals 239
 - unit record
 - input devices 239
 - output devices 240
- existence tables, MVS
 - basic conditions 241
 - DASD files 243
 - file definitions specifying DUMMY 248
 - files on other devices 249
 - input-stream data sets 246
 - DD 246
 - DD DATA 246
 - labeled tape files 245
 - nonreusable VSAM files 244
 - PDS members 245
 - reusable VSAM files 244
 - system output data sets 247
 - terminals 247
 - unit record
 - input devices 247
 - output devices 248
 - unlabeled tape files 246
- existence, file
 - checking 141
 - definition 141
 - factors determining 233
 - indicating on the OPEN statement 151
 - tables
 - CMS 235
 - MVS 241
- EXIT routine
 - using in parallel 325
- EXIT statement
 - example of 300
 - inhibits automatic parallel code generation 297
 - parallel loop and 299
- explicit type statement
 - type changes using, common coding error 129
- exponent
 - underflow control 116
- expressions
 - common, OPTIMIZE(3) eliminates 255
 - how compiler recognizes duplicate 272
 - internal limits 449
 - restrictions as actual argument 372
 - scaling elimination 273
- extended
 - architecture (XA) considerations 68, 88
 - error handling 123
 - range of the DO 482

- extended common 459
 - data
 - assembler programs and 452
 - linkage conventions for 452
 - description 379
- extended mode 452
- extended parameter list 452
- extensions, IBM
 - documentation of xiii
- external cross reference listing 393
- external file 133

F

- factoring expressions 273
- FAIL run-time option 109
- file
 - characteristics 144
 - conditions 234
 - definition
 - ddname 136
 - description 134
 - existence
 - CMS tables 235
 - disk files on CMS minidisks 231
 - disk files under MVS 231
 - files on terminals 234
 - MVS tables 241
 - striped files 234
 - VSAM files 234
 - ignoring history 155
 - name 136
- file/unit connection
 - See unit/file connection
- FILEDEF command 134, 144
- FILEHIST run-time option 109
- FILEINF subroutine 198
 - DASD I/O under MVS 175, 200
- files
 - definition
 - default for originated tasks 316
 - I/O
 - See I/O files
- FIPS compile-time option
 - description of 28
 - output for 50
- FIPS flagger 50
- fixed
 - length record 140
 - description 147
 - order variable alignment 375
 - point items, conversions of 273
- FIXED compile-time option 28
- fixing user errors 129
- FLAG compile-time option
 - description of 28

- FLAG compile-time option (*continued*)
 - diagnostic message listing 47
 - examples of compiler messages 47
- floating-point
 - numbers, conversions of 273
- foreground command procedure under TSO 96
- format codes 449
 - nested parentheses groups 449
 - repeating code 449
- FORMAT statement
 - reading data 193
 - writing data 191
- formatted data
 - FORMAT statement
 - reading 193
 - writing 192
 - internal files 164, 168, 174
 - list-directed formatting
 - reading data 164
 - writing data 168
 - NAMELIST formatting
 - reading data 166
 - writing data 169
 - OPEN statement 153
- FORTRAN language
 - See VS FORTRAN Version 2
- FORTRAN-supplied functions
 - See intrinsic functions
- FORTVS2 command for CMS 7
- forward dependence 263
- FREE compile-time option 28
- FTERRsss DD statements 417
- FTnnFmmm, optional MVS loader data set 82
- FTPRTsss DD statements 417
 - processing
 - converting MTF programs for use with 418
- functions
 - paired arguments in 371

G

- GENMOD command for CMS 65, 69
- GOSTMT compile-time option 28
 - description 28
- granularity, definition of 332

H

- HALT compile-time option 28
- Hollerith constants 449

I

- I informational code 48
- I/O files
 - access methods 138

I/O files (*continued*)

- CMS considerations 143, 150
- connection
 - changing properties of 154
 - definition 134
 - with the OPEN statement 150
- DASD buffers 175
- definition 134
- disconnection
 - at program termination 159
 - deleting after 157
 - retaining after 157
 - with the CLOSE statement 157
 - with the OPEN statement 159
- dynamically allocating
 - defaults for file characteristics 197
 - defaults for file characteristics (MVS) 149, 227
 - definition 135
 - deleting files 157
 - FILEINF service subroutine 198
 - inquiring about files 202
 - named files 136, 138, 150, 195
 - space calculation under MVS 200
 - unnamed files 195
- existence
 - definition 141
 - indicating 151
- external
 - description 133
 - difference from operating system files 133
- internal
 - description 133
 - reading and writing 174
- MVS considerations 147
- named
 - connection using an OPEN statement 150
 - definition 136
 - originated tasks, using in 317
- optimization and 268
- parallel 176
- parallel task, using between 315
- parallel task, using in 309
- reconnection 159
- striped 176
- temporary 152
- unnamed
 - connection using an OPEN statement 151
 - definition 137
 - originated tasks, using in 317
- I/O statements, overview 163
- I/O terminology
 - connection 134
 - ddname 136
 - direct retrieval 186
 - dynamic file allocation 135
 - external file 133

- I/O terminology (*continued*)
 - file access method 138
 - file definition 134
 - file existence 141, 233
 - internal file 133
 - key of reference 186
 - named files 136
 - preconnection 134
 - records 140
 - sequential retrieval 186
 - subfiles 170
 - unit/file connection 134
 - unnamed files 137
- I/O unit
 - connection
 - changing properties of 154
 - general 134, 150
 - with the OPEN statement 150
 - definition 134
 - disconnection
 - at program termination 159
 - with the CLOSE statement 157
 - with the OPEN statement 159
 - identifier 134
 - preconnection 143
- IBM
 - extensions, documentation of xiii
- ICA compile-time option 28
- ICA feature
 - See intercompilation analysis
- ICA optional MVS compiler data set 14
- IF statement
 - optimization and 274
- IGNFHDD routine 155
- IGNFHU routine 155
- IGNORE directive 358
- ignored dependences 352
- IL compile-time option 30
- ILX, compiler message prefix 48
- IMPLICIT statement
 - type changes using, common coding error 129
- implied DO
 - I/O statements 268
- INCLUDE command for CMS 64, 65
- INCLUDE statement
 - identification numbers 25
 - MVS linkage editor control statement 80
 - under CMS 9
 - under MVS 14, 16
- incompatibilities between programs 380
 - incorrect usage 381
- independence, computational
 - See computational independence
- induction variable, definition 258
- industry standards xiii
- information
 - messages, compile-time 41, 48
- initialization
 - common errors 129
 - run-time environment 457
- input
 - blanks, treatment of 153
- input/output operations
 - See I/O files
- INQPCOPN run-time option 110
- INQUIRE statement
 - response under NOOCSTATUS run-time option 110
 - response under OCSTATUS run-time option 110
 - sample program 201
 - sequential access 172
 - where to code 201
- instruction elimination, OPTIMIZE(3) 255
- instruction scheduling 256
- integer
 - data type
 - internal representation 470
 - optimization efficiency and 269
- INTEGER statement 378
- interactive debug (IAD)
 - extended common and 379
- interactive vectorization aid 38
 - See *also* program, information file (PIF)
- interchange-preventing dependence 264, 279
- intercompilation analysis
 - description 380
 - errors detected
 - common block storage assignment 387
 - conflicting argument and common block association 387
 - conflicting argument usage 381
 - conflicting common block lengths 386
 - conflicting common block storage assignment 386
 - conflicting external name usage 385
 - conflicting function type 385
 - file
 - allocating space for 391
 - considerations in CMS 391
 - considerations in MVS 391
 - number of file names allowed 449
 - search order for 390
 - managing large programs with 387
 - managing small programs with 387
 - messages, suppressing 391
 - MSGOFF suboption 391
 - MSGON suboption 391
 - sample programs compiled with ICA 393
 - external cross reference listing 393
 - UPDATE suboption 390
 - USE suboption 390

- intercompilation analysis (*continued*)
 - using USE, UPDATE, and DEF suboptions 388
 - using with non-Fortran program units 392
 - when to use 387
- internal files 133, 174
- internal limits in VS FORTRAN Version 2 449
- internal statement number (ISN)
 - compile-time messages optionally contain 48
 - source program listing 17, 20, 42
 - traceback map uses 120
- intrinsic functions
 - storage map lists 44
 - TSO usage of 94
 - vectorization of 37, 285
- INTRINSIC, VECTOR suboption 36, 37
- invoking a main program
 - See running programs
- IOINIT run-time option 110
- IOSTAT
 - option and VSAM return code 226
- ISCI/ASCII encoded file, record format 149
- ISN
 - See internal statement number (ISN)
- IVA, VECTOR suboption 38
 - See *also* program, information file (PIF)

J

- job control
 - considerations under MVS 13
 - how to specify for MVS 12
- JOB control statement for MVS 12
- job libraries and run-time loading of library 77
- job processing, MVS 12

K

- keyed access
 - alternate keys 139, 182
 - connecting 150, 183
 - DBCS data 205
 - ddnames 184
 - description 139, 182
 - direct retrieval 186
 - disconnecting 157
 - double-byte data 205
 - file definitions 184
 - formatted I/O 191
 - key of reference 186
 - loading new records 184
 - primary keys 139, 182
 - reading data 185
 - repositioning files 189
 - sequential retrieval 187
 - unformatted I/O 193
 - updating files 188

KSDS

- defining a 213
- VSAM keyed file 210
 - keyed 211
 - relative 211
 - sequential 210

L

- LABEL parameter of DD statement 147
- LANGLVL compile-time option 31
- LANGUAGE, PARALLEL suboption 33
- LET
 - linkage editor option under MVS 78
 - loader option under MVS 81
- level codes
 - 0 (information) 48
 - 12 (serious error) 48
 - 16 (abnormal termination) 48
 - 4 (warning) 48
 - 8 (error) 48
 - description of 48
 - E (error) 48
 - I (information) 48
 - S (serious error) 48
 - U (abnormal termination) 48
 - W (warning) 48
- libraries
 - See *also* run-time, library
 - combined link libraries 64, 438
- library
 - messages
 - See diagnostic messages
 - module run-time loading 62, 75
 - subroutines, mathematical 479
- LIBRARY, MVS linkage editor control statement 80
- limits in VS FORTRAN Version 2 449
- linear data set
 - VSAM
 - defining, JCL for 215
 - description 211
 - processing, JCL for 225
- LINECOUNT compile-time option 31
- link
 - editing
 - MVS linkage editor control statements 80
 - optional MVS linkage editor data sets 79
 - output 81
 - required MVS linkage editor data sets 79
 - TSO listing 94
 - under MVS 78
 - under TSO 91
 - library, combined 64, 438
 - mode
 - selecting 62, 75
 - with reentrant programs 434, 442

- LINK command under TSO 93
- linkage
 - conventions 452, 453, 455
 - editor
 - control statements under MVS 81
 - program under MVS 78
 - editor NAME statement
 - and reentrant programs 428
 - in separation tool output under CMS 430
 - in separation tool output under MVS 439
- LIST
 - compile-time option
 - example of output 128
 - format of listing 128
 - object module listing 127
 - data set under TSO 20
 - linkage editor option under MVS 78
 - report, compiler 349
- list-directed formatting
 - description 164, 168
 - internal files, rules for 165
 - reading data 164
 - writing data 168
- LISTING file for CMS 10
- listing, compiler output
 - See compiler, output
- LKED command for CMS 66, 69
- LOAD command for CMS 64, 65, 69
- load mode
 - selecting 62, 75
 - using multitasking facility (MTF) with 418
 - with reentrant programs 434, 442
- load module
 - execution data set under MVS 82
 - migration under MVS 85
 - run-time output 84
 - running on MVS 84
- loader option under MVS 82
- loader program
 - under MVS 81
 - under TSO 94
- loading
 - library modules under CMS, run-time 62
 - library modules under MVS, run-time 75
 - modules, dynamically 97
 - new records into a file 184
 - VSAM KSDS 219
- LOCAL statement
 - example of 300, 302
 - inhibits automatic parallel code generation 297
 - parallel loop and 299
 - parallel sections and 301
- lock
 - request contention 322
- locks
 - creating 318

- locks (*continued*)
 - definition of 317
 - exclusive mode, using 318
 - shared mode, using 318
 - structured use of, example 323
 - uses for 319
- logic errors, MAP option helps find 44
- logical
 - data type
 - internal representation of 470
 - optimization efficiency and 269
- loop distribution, definition 258
- LOOP qualification for vector processing 277
- LPA
 - reentrant programs 443

M

- main program
- main task for multitasking facility (MTF) 398
- MAP
 - compile-time option
 - description 31, 44
 - example 45, 47
 - using the 44
 - linkage editor option under MVS 78
 - loader option under MVS 81
- map, traceback 119
- mathematical
 - functions, vectorization of 285
 - library routines 479
- maximum
 - efficiency 253
 - LRECL and BLKSIZE 145
- member name, changing 435
- messages
 - See *also* diagnostic messages
 - compiler messages (appear only online) 48
 - format of compiler 47
 - format, operator 123
 - number, compiler 48
 - prefix, compiler 48
 - program interrupt 121
 - programmer-specified text in PAUSE statement 123
 - programmer-specified text in STOP statement 123
- migration, load modules under MVS
 - library module replacement tool 86
 - linkage editor considerations 85
- MODEL, VECTOR suboption 285
- modification of compiler defaults 41
- module identifier, compiler messages 48
- MSG suboption of ICA compile-time option 30
- MSGOFF suboption of ICA compile-time option 30, 391
- MSGON suboption of ICA compile-time option 30, 391

- multiply-and-add 39
- multiply-and-subtract 39
- multitasking facility (MTF)
 - coding for 406
 - compiling 414
 - concepts illustrated 401
 - designing for 406
 - dynamic commons 407, 412, 418
 - examples 409, 412
 - extended common and 379
 - I/O 408, 417
 - independence requirement 407
 - introduction to 398
 - job control language (JCL) for 416
 - linking 414
 - load modules 414
 - passing data 407
 - programs, converting for use with parallel feature 418
 - rules 407, 414
 - running under 416
 - using with load mode 418
- MVS considerations
 - compile-only cataloged procedure 13
 - compile-time options and 13
 - compiler data set 14
 - DASD I/O buffers 175
 - defining a record 147
 - direct access label 147
 - file existence tables 241
 - I/O 147
 - invoking the VS FORTRAN Version 2 compiler 473
 - job control statements 13
 - link-edit processing 78
 - linkage conventions 453, 455
 - linkage editor
 - control statements 81
 - data sets 79
 - use 78
 - load module execution data set 82
 - load modules 79
 - loader
 - data set 82
 - program under TSO 94
 - use 81
 - object modules 14
 - operating system support xiv
 - overlays 85
 - partitioned data sets, using 180
 - requesting an abnormal termination dump 84
 - requesting compilation 13
 - running the load module 84
 - specifying run-time options 84
 - subtasks 459
 - tape label 147
- VSAM
 - DEFINE command 219

- MVS considerations (*continued*)
 - VSAM (*continued*)
 - file creation 219
 - file processing 219
- MVS/ESA
 - considerations 88
- MVS/XA
 - considerations 88
 - dynamic common and 379
- MXREF suboption of ICA compile-time option 30

N

- name
 - of separation tool output under CMS 430
 - of separation tool output under MVS 439
- NAME compile-time option
 - column, in storage map 47
 - description of 31
- named common block
 - See common block, named
- named files
 - definition 136
 - reconnecting 160
- NAMELIST statement
 - description 166, 168
 - reading data 166
 - writing data 169
- names
 - column in cross reference 46
 - cross reference and 46
- NAMESYS macro 437
- NCAL
 - linkage editor option under MVS 78
 - loader option under MVS 81
- nested levels limit 449
- NOABSDUMP run-time option 107
- NOANZCALL, PARALLEL suboption 34
- NOANZCALL, VECTOR suboption 39
- NOAUTOMATIC, PARALLEL suboption 33
- NOAUTOTASK run-time option 107
- NOCLN suboption of ICA compile-time option 30
- NOCMPLXOPT, VECTOR suboption 40
- NOCNVIOERR run-time option 108
- NOCVAR suboption of ICA compile-time option 30
- NODBCS compile-time option 26
- NODDIM compile-time option 26
- NODEBUG run-time option 108
 - CMS, specification under 68
 - description 108
 - MVS, specification under 84
 - TSO, specification under 96
- NODEBUNIT run-time option 108
- NODECK compile-time option 26
- NODIRECTIVE compile-time option 26

- NOECPACK run-time option 109
- NOEMODE compile-time option 27
- NOFILEHIST run-time option 109, 155
- NOFIPS compile-time option 28
- NOGOSTMT compile-time option 28
- NOICA compile-time option 28
- NOINQPCOPN run-time option 110
- NOIOINIT run-time option 110
- NOLANGUAGE, PARALLEL suboption 33
- NOLET loader option under MVS 81
- NOLIST compile-time option 31
- NOMAP compile-time option
 - description of 31
 - loader option under MVS 81
- NOMXREF suboption of ICA compile-time option 30
- noninductive subscript, definition 258
- nonreentrant program 428
- nonshareable part of reentrant program
 - See *also* reentrant programs
 - defined 425
- NOOBJECT compile-time option 31
- NOOCSTATUS run-time option 110
- NOOPTIMIZE compile-time option 31
- NOPARALLEL run-time option 113
- NOPRINT compile-time option 8
- NOPRINT loader option under MVS 82
- NOPTRACE run-time option 111
- NORCHECK suboption of ICA compile-time option 30, 381
- NORECPAD run-time option 114
- NOREDUCTION, PARALLEL suboption 33
- NORENT compile-time option 34
- NOREPORT, PARALLEL suboption 32
- NORES loader option under MVS 82
- NOSAA compile-time option 34
- NOSDUMP compile-time option 35
- NOSOURCE compile-time option 35
- NOSPIE run-time option 114
- NOSRCFLG compile-time option 35
- NOSTAE run-time option 115
- NOSXM compile-time option 35
- NOSYM compile-time option 35
- NOTERMINAL compile-time option 35
- NOTEST compile-time option 36
- NOTRACE, PARALLEL suboption 34
- NOTRMFLG compile-time option 36
- NOVECTOR compile-time option 36
- NOXREF compile-time option 40
- NOXUFLOW run-time option 116
 - CMS, specification under 68
 - description 116
 - MVS, specification under 84
 - TSO, specification under 96
- NPROCS function
 - using 323

- NTASKS subroutine 399
 - using in parallel 324

O

- OBJ data set under TSO 20
- OBJECT compile-time option 31
- object module
 - compiler default 41
 - compiler output 17, 20
 - example of listing 128
 - link-editing 77
 - MVS 14
 - obtaining listing of 127
 - SYM record in 475
- OCSTATUS run-time option
 - description 110
 - effect on connection 152
 - effect on disconnection 157
 - effect on reconnection 159
- OPEN statement
 - access method, choosing 152
 - connecting a file
 - changing connection properties 154
 - direct access 181
 - general 150
 - keyed access 183
 - temporary 152
 - DBCS data, indicating 153
 - disconnecting a file 159
 - error checking 153
 - file existence, indicating 151
 - formatted I/O, choosing 153
 - input blanks, treatment of 153
 - NOOCSTATUS run-time option 152
 - OCSTATUS run-time option 152
 - unformatted I/O, choosing 153
 - VSAM
 - considerations 153
 - direct file considerations 223
 - sequential file considerations 221
- operands, recognition of constant 271
- operating system
 - access methods 139
 - support xiv
- operator
 - message format 123
 - message identification 123
 - messages 122
- OPTIMIZE compile-time option
 - arithmetic conversions, avoiding 273
 - array initialization 272
 - arrays, adjustable dimensioned not recommended 272
 - arrays, optimizing identically dimensioned 272
 - common blocks, using efficiently 270

OPTIMIZE compile-time option (*continued*)

- common expressions, OPTIMIZE(3) eliminates 255
- constant operand recognition 271
- description 31
- difference between OPTIMIZE(2) and OPTIMIZE(3) 254
- double precision conversions and 273
- duplicate expression recognition 272
- efficient accumulator usage 271
- efficient arithmetic constructions and 273
- efficient program size 268
- EQUIVALENCE statement not recommended 269
- higher levels best 267
- IF statement and 274
- instruction elimination, OPTIMIZE(3) 255
- instruction scheduling 256
- integer variables and 269
- logical variables and 269
- OPTIMIZE(3) considerations 255
- passing subroutine arguments in common 271
- scaling elimination 273
- single precision conversions and 273
- source program considerations 253
- unformatted I/O and 268
- variables, optimization limitations 269
- vectorization considerations 257, 292
- writing loops inline 273

option table

- warning on error occurrences 124

options

- compile-time
 - See compile-time, options
- run-time
 - See run-time, options

ORIGINATE statement

- example of 312
- inhibits automatic parallel code generation 297
- task management and 310

originated task, definition of 295

output

- dependence 263
- from separation tool 428
- link-editing 81
- listing
 - header 41
 - using 41, 127

OVERFL routine

- coordinating parallel use of with locks 319

overlays 85

OVLY linkage editor option under MVS 78

P

padding, effect on 52

- array arguments 55
- asynchronous I/O processing 56

padding, effect on (*continued*)

- CALL DUMP or CALL PDUMP 55
- common or equivalence data value 53
- direct access I/O processing 56
- formatted I/O data set 56
- initialization with hexadecimal constant 53
- initialization with literal constant 53
- mode-changing intrinsic function 54
- programs calling subprogram 54
- unformatted I/O data set 56

PARA status flag 344, 345, 349, 350

parallel

- See *also* compiler, report
- chunking 258
- dependences 262, 353
- diagnostic messages 351
- directives
 - applications 353
 - ASSUME COUNT 356
 - format 355
 - global 355
 - IGNORE 358
 - interactions between 354
 - local 355
 - multiple 356
 - PREFER 364
 - verifying correct application 367
- events
 - See events, parallel
- locks
 - See locks
- loop 298
 - statements for creating 299

PARALLEL compile-time option 31, 297

- ANZCALL 34
- AUTOMATIC 33, 297
- LANGUAGE 33
- NOANZCALL 34
- NOAUTOMATIC 33
- NOLANGUAGE 33
- NOREDUCTION 33
- NOREPORT 32
- NOTRACE 34
- REDUCTION 33
- REPORT 32
- TRACE 34, 325

PARALLEL run-time option 113

processing 418

- automatic, eligibility for 297
- automatic, generating with PARALLEL
 - compile-time option 33, 297
- chunking 258
- definition 257
- PARALLEL compile-time option and 31, 297
- PARALLEL run-time option and 113

program 74

- debugging hints 296

- parallel (*continued*)
 - program (*continued*)
 - definition of 294
 - running on MVS 75, 76
 - running on one virtual processor 296
 - running on VM/XA 74
 - report
 - See compiler, report
 - section 301
 - service subroutines 323
 - statements inhibiting automatic parallel code generation 297
 - subroutine 307
 - subroutines for multitasking facility (MTF) 398
 - task
 - coding 310
 - definition of 259, 294
 - environment of 294
 - environment of, example 311
 - I/O, using between 315
 - I/O, using within 309
 - originated 295
 - root 295
 - terminology
 - computational independence 258
 - construct 258
 - dependence 258, 259
 - induction variable 258
 - loop distribution 258
 - noninductive subscript 258
 - parallel task 259
 - parallel thread 259
 - privatization 259
 - recurrence 259
 - unanalyzable loop 261
 - thread
 - definition of 259
 - identifying 295
 - invoking subroutines and functions within 307
 - parallel loop as 298
 - parallel section as 301
 - primary, definition of 295
 - primary, environment of 295
 - primary, scheduled subroutine becomes 310
 - subroutine as 307
 - tracing facility 325
- PARALLEL CALL statement
 - example of 307
 - inhibits automatic parallel code generation 297
 - subroutines as parallel threads, identifying 306
- PARALLEL compile-time option 31, 297
- PARALLEL DO statement
 - events, cannot use within 322
 - example of 300
 - inhibits automatic parallel code generation 297
 - parallel loop as parallel thread, identifying 299
- PARALLEL run-time option 296
 - AUTOTASK run-time option and 107, 113
 - description 113
- PARALLEL SECTIONS statement
 - events, cannot use within 322
 - example of 302
 - inhibits automatic parallel code generation 297
 - parallel sections as parallel threads, identifying 301
- parameters
 - assembler subprograms and 452
 - EMODE compile-time option and 27
- partitioned data sets under MVS 180
- passing arguments
 - between programs 371
 - character 454, 456
- PAUSE statement
 - operator message and 123
- PAVE status flag 344, 345, 349, 350
- PEORIG event routine
 - description of 320
- PEPOST event routine
 - description of 320
- performance considerations
 - description of 426
 - using reentrant programs 425
- period xi
- PETERM event routine
 - description of 320
- PEWAIT event routine
 - description of 320
- PFAFFC subroutine 323
- PFAFFS subroutine 323
- PIF
 - See program, information file (PIF)
- PLCOND lock function
 - description of 318
 - exclusive mode and 318
 - shared mode and 318
- PLFREE lock routine
 - description of 318
- PLLOCK lock routine
 - description of 318
 - exclusive mode and 318
 - shared mode and 318
- PLORIG lock routine
 - description of 318
- PLTERM lock routine
 - description of 318
- Pointer 274, 303
 - association of 274
 - optimization considerations 274
 - private 303
- precision errors, common 129
- preconnected file 143
 - definition 150
 - reconnecting 161

- preconnected file (*continued*)
 - under NOOCSTATUS run-time option 110
 - under OCSTATUS run-time option 110
- PREFER directive
 - CHUNK option 364
 - PARALLEL option 364
 - SCALAR option 364
 - SERIAL option 364
 - VECTOR option 364
- PRINT compile-time option under CMS 8
- PRINT statement
 - list-directed formatting 168
 - NAMELIST formatting 166, 168
 - specifying your own format 191
- printer files under CMS 146
- priority of processing
 - See processing, priority
- privatization
 - definition 259
 - of arrays 304
- processing
 - options for MVS linkage editor 78
 - parallel
 - automatic, generating with PARALLEL
 - compile-time option 33, 297
 - chunking 258
 - compiler report and 344
 - definition 257
 - PARALLEL compile-time option and 31, 297
 - PARALLEL run-time option and 113
 - program, definition of 294
 - under MVS 75
 - under VM/XA 74
 - priority
 - under CMS 227
 - under MVS 228
 - serial
 - program, definition of 294
 - vector
 - compiler report and 344
 - eligibility for 277
 - improving, techniques for 277
 - VECTOR compile-time option and 36
- processor affinity 323, 331, 460
- processor model for vectorization 39
- processor, virtual
 - See virtual, processor
- program
 - coding
 - See coding your program
 - constants
 - See constant
 - information file (PIF)
 - IVA compiler suboption 38
 - under CMS 11
 - under MVS 14, 17
 - under TSO 20

- program (*continued*)
 - interrupt messages 121
 - output, error free 84
 - parallel
 - See parallel, program
 - serial
 - See serial program, definition of
 - units, sharing storage between 371
- PRTUNIT run-time option 111
- PSW 452
- PTRACE run-time option 111, 325
- PTRSIZE compile-time option 34
- publications, summary of
 - related xii
 - VS FORTRAN Version 2 xi
- PUNUNIT run-time option 113
- PYIELD subroutine 323

R

- range of the DO 482
- RCHECK suboption of ICA compile-time option 30, 381
- RDRUNIT run-time option 114
- READ statement
 - direct access 182
 - directly accessed VSAM direct file 224
 - formatted data
 - list-directed 164
 - NAMELIST 166
 - user-specified 193
 - internal files 174
 - keyed access 185
 - sequential access 164
 - sequentially accessed VSAM direct file 223
 - unformatted data 194
 - unformatted record size and 149
 - VSAM sequential file considerations 221
- reading data
 - asynchronous I/O 179
 - direct access 182
 - formatted data
 - list-directed 164
 - NAMELIST 166
 - user-specified 193
 - internal files 174
 - keyed access 185
 - sequential access 164
 - unformatted data 193
- real
 - data type
 - internal representation 471
- reconnecting files
 - named files 160
 - preconnected files 161
 - unnamed files 160

- record
 - definition 147
 - direct access file 149
 - format
 - default values 141
 - fixed-length 140
 - operating system 140
 - specifying 141
 - undefined 140
 - variable-length 140
 - variable-length spanned 140
 - formats under MVS 147
- records
 - blocked 140
 - endfile 140, 167, 181
 - formatted
 - in internal files 174
 - reading and writing 164, 168
 - specifying on the OPEN statement 153
 - unformatted
 - asynchronous I/O 179
 - description 140
 - reading and writing 193
 - specifying on the OPEN statement 153
- RECPAD run-time option 114
- RECR status flag 344
- recurrence, definition 259
- reduction function, vectorization 38
- REDUCTION, PARALLEL suboption 33
- REDUCTION, VECTOR suboption 36, 38
- reentrant programs
 - advantages 425
 - and choice of link mode or load mode 434, 442
 - and residence mode under MVS/XA 443
 - cataloged procedures to separate 441
 - changing name of LOADLIB 435
 - comparison with nonreentrant 423
 - creating individual members of a TXTLIB 434
 - creating under CMS 430
 - creating under MVS 439
 - definition of nonreentrant program 428
 - dynamic loading capability 425
 - in main and auxiliary storage 426
 - installing in a DCSS 435
 - installing in an LPA 443
 - limitations 426
 - nonshareable and shareable parts 425
 - running under CMS 438
 - running under MVS 444
 - sharing 425
 - structure of 424
 - two output forms under CMS 430
 - two output forms under MVS 439
- REFERENCED field in cross reference 46
- referenced variables 449
- register
 - access
 - extended common and 452
 - general
 - extended common and 452
- register conventions 453
- related publications xii
- relative-record data sets
 - See RRDS
- RENT option
 - and separation tool 426
 - description of 34
 - under CMS 430
 - under MVS 439
- report
 - vector
 - See compiler, report
- REPORT, PARALLEL suboption 32
- REPORT, VECTOR suboption 36
- repositioning files
 - BACKSPACE statement 170, 190
 - REWIND statement 169, 189
- RES loader option under MVS 82
- residence mode with reentrant programs 443
- retrieving arguments in assembler programs 464
 - alternate return points 469
 - array and array elements 464, 465
 - character variables 464, 466
 - returning a function value 466
 - variables 464, 465
- return code routines
 - SYSRCS 367
 - SYSRCX 367
- REWIND statement
 - invalid for directly accessed VSAM direct file 224
 - keyed access 189
 - sequential access 169
 - sequentially accessed VSAM direct file 223
 - VSAM sequential file considerations 222
- REWRITE statement 189
- RMODE attribute 68, 88
- root task, definition of 295
- routines
 - alternative mathematical 479
 - listed in traceback map 120
- RRDS
 - defining a 214
 - source language considerations 220
 - VSAM direct file 210
- run-time
 - efficiency and boundary alignment 375
 - environment
 - dedicated, definition of 331
 - nondedicated 332
 - error messages
 - operator 122
 - traceback map with 119

run-time (*continued*)

library

- alternative mathematical routines 479
- making available at run time 75

loading of library

- modules under CMS 62
- modules under MVS 75

options

- abbreviations for 106
- ABSDUMP 105, 107
- AUTOTASK 105, 107
- CMS, specifying under 67
- CNVIOERR 105, 108
- DEBUG 105, 108
- DEBUNIT 105, 108
- ECPACK 109
- FAIL 109
- FILEHIST 105, 109
- INQPCOPN 105, 110
- IOINIT 105, 110
- MVS with TSO, specifying under 95
- MVS, specifying under 84
- NOABSDUMP 105, 107
- NOAUTOTASK 105, 107
- NOCNVIOERR 105, 108
- NODEBUG 105, 108
- NODEBUNIT 105, 108
- NOECPACK 109
- NOFILEHIST 105, 109
- NOINQPCOPN 105, 110
- NOIOINIT 105, 110
- NOOCSTATUS 105, 110, 152, 157
- NOPARALLEL 105, 113
- NORECPAD 114
- NOSPIE 105, 114
- NOSTAE 105, 115
- NOXUFLOW 105, 116
- OCSTATUS 105, 110, 152, 157
- PARALLEL 105, 113, 296
- RECPAD 114
- SPIE 105, 114
- STAE 105, 115
- table 117
- XUFLOW 105, 116

running programs

run-time options

- under CMS, and 67
- under MVS with TSO, and 95
- under MVS, and 84

under CMS 62

under MVS 74

under MVS with TSO 91

S

S severe error code 48

SAA

- compile-time option 34, 50

save area

- extended 452

SC compile-time option 34, 379, 380

SCAL status flag 344, 345, 349

scalar

- expansion 260, 278, 288

SCHEDULE statement

COPYING clause

- example 313

- using 313

COPYINGI clause

- example 314

- using 314

COPYINGO clause

- example 314

- using 314

example of 312

inhibits automatic parallel code generation 297

SHARING clause

- example 312

- using 312

task management and 310

scheduled subroutine, definition of 310

scheduling of instructions 256

SDUMP compile-time option 35

SDUMP routine

- coordinating parallel use of with locks 319

section size for vectorization 38

SECTION statement

- example of 302

- inhibits automatic parallel code generation 297

- parallel sections and 301

sectioning, for vector processing 261

- See *also* vector, sectioning considerations

separation tool

- assigned name form 428

- changing name of output 435

- default name form 428

- forms of output under CMS 430

- forms of output under MVS 439

- invoking under CMS 431

- invoking under MVS 440

- linkage editor NAME statement, and 430, 439

- MVS cataloged procedures 441

- output from 428

- supplied as AFBVSFST 440

- supplied as nonrelocatable file 430

- used to permit dynamic loading 425

- used to permit program sharing 425

- using with multiple object files 427

- using with nonreentrant programs 428

- sequential access
 - asynchronous I/O 179
 - default record formats 141
 - description 139, 163
 - file organization 139
 - list-directed formatting 164, 168
 - NAMelist formatting 166, 168
 - preconnected 138
 - reading data 164
 - repositioning files 169
 - specifying your own format 191
 - subfiles, processing 170
 - unformatted I/O 193
 - VSAM direct file 223
 - writing data 167
- sequential file processing
 - CMS FILEDEF command and 144
 - EBCDIC encoded record 149
 - ISCI/ASCII considerations 149
 - performance 175
 - valid VSAM source statements, summary 220
 - VSAM considerations 209
 - VSAM source language 220
- SERI status flag 344, 349
- serial program, definition of 294
- severity level, compiler messages 48
- shareable
 - part of a reentrant program, defined 425
- shared segments
 - See DCSS
- sharing
 - data between programs 371
- shift-in, documentation xi
- shift-out, documentation xi
- SHRCOM subroutine 407, 414
- single precision, conversions of 273
- single-precision multiply-and-add 39
- single-precision multiply-and-subtract 39
- SIZE
 - linkage editor option under MVS 79
 - loader option under MVS 82
 - parameter for VSAM file processing 220
 - VECTOR suboption 38, 284
- SLIST report, compiler 350
- source code efficiency 253
- SOURCE compile-time option
 - description of 35
 - source program listing 42
 - source program listing example 43
- source program
 - compiling 7, 21
 - listing
 - compiler default 41
 - description 42
 - using MAP and XREF 44
 - map, using the 44
- source statement characters
 - mixed case 480
- spanned record description 148
- SPIE run-time option 114
- SPRECOPT, VECTOR suboption 36
- SRCFLG compile-time option
 - description of 35
 - source program listing 43
 - source program listing example 43
- STAE run-time option 115
- standard
 - input unit 134, 143, 150, 198, 408
 - output unit 134, 137, 143, 150, 198
- standard mode 452
- standard parameter list 452
- standards, industry xiii
- START command for CMS 69
- STAT report, compiler 352
- statement
 - label 449
 - storage maps and 47
- statement function 449
 - arguments 449
 - definition 449
 - nested references 449
 - storage map lists 44
- statements, overview of I/O 163
- static common 378
 - description 379
- static debug
 - example 126
 - statements 125
- statistics
 - table, parallel 352
 - table, vector 352
- step libraries and run-time loading of library 77
- STOP statement
 - causes program termination 123
 - operator message and 123
- storage
 - map description and example
 - column, in storage map 47
- storage, shared
 - See EQUIVALENCE statement
 - See reentrant programs
- stride
 - definition 260
 - minimizing 282
- striped file 176
- subfiles, processing 170
- subprograms
 - arguments in, general rules 371
 - block data 378
 - paired arguments in 371
 - storage map lists 44

- subroutines
 - arguments in 371
 - paired arguments in 371
 - parallel executions controls service 323, 331, 460
 - passing arguments to 371
 - scheduled, definition of 310
- subscripts
 - invalid values for, common coding error 129
 - vectorization, affect on 279
- subset FIPS flagging 50
- summary of errors, in traceback map 119
- SXM compile-time option
 - description of 35
 - using the 45
- SYM compile-time option
 - description of 35
 - record in object module 475
- SYNCRO routine 399
- syntax
 - errors, MAP option helps find 44
 - notation x
- SYSIN
 - required MVS compiler data set 14
- SYSLIB
 - optional for MVS linkage editor 79
 - optional MVS compiler data set 14
 - optional MVS loader data set 82
- SYSLIN
 - MVS loader required data set 82
 - optional MVS compiler data set 14
 - required for MVS linkage editor 79
- SYSMOD 79
- SYSLOUT, optional MVS loader data set 82
- SYSRINT, required for
 - MVS compiler data set 14
 - MVS linkage editor 79
 - MVS loader data set 82
- SYSRCS optional MVS compiler data set 14
- SYSRCS routine 367
 - using in parallel 325
- SYSRCT routine
 - using in parallel 325
- SYSRCX routine 367
 - using in parallel 325
- system
 - support, operating xiv
- Systems Application Architecture
 - See *also* SAA
 - description of 50
- SYSTEM optional MVS compiler data set 14
- SYSUT1, required for MVS linkage editor 79

T

- table of ignored dependences 352

- tag column, in storage map 47
- tape
 - files
 - CMS FILEDEF command and 145
 - ISCI/ASCII considerations 149
- task
 - management statements
 - definition of 295
 - using 310
 - originated
 - See parallel, task
 - parallel 259
 - See *also* parallel, task
- TERM report, compiler 344
- terminal
 - file, CMS FILEDEF command and 146
 - report, compiler 344
- TERMINAL compile-time option 35
- TERMINATE statement
 - example of 312
 - inhibits automatic parallel code generation 297
 - task management and 310
- terminating
 - run-time environment 458
- TEST compile-time option 36
- TEXT file for CMS 10
- thread, parallel
 - See parallel, thread
- Time Sharing Option
 - See TSO (Time Sharing Option)
- time, in output listing header 41
- TRACE, PARALLEL suboption 34
- traceback map 119
- transparent argument passing 483
- TRMFLG compile-time option
 - description of 36
 - output for 49
- true dependence 262
- TSO (Time Sharing Option)
 - ALLOCATE command 18
 - background command procedure under 97
 - CALL command 19
 - CLISTs under 96
 - compilation 18
 - foreground command procedure under 96
 - linkage editor listing 94
 - loader program and 94
 - loading 91
 - running 91
 - TSO, specification under 96
- type
 - column, in storage map 46
 - statement
 - See explicit type statement

U

- U unrecoverable error code 48
- UNAN status flag 344, 345, 349, 350
- unanalyzable loop, definition 261
- undefined
 - length record description 149
 - records 140
- underflow
 - control, exponent 116
- unformatted data
 - asynchronous I/O 179
 - EBCDIC encoded file 149
 - OPEN statement 153
 - reading data 194
 - writing data 193
- unit
 - I/O
 - See I/O unit
 - identifier
 - asterisk 134
 - asterisk (*) 150
 - default value 134, 150
 - external files 168
 - internal files 174
 - record file, CMS FILEDEF command and 146
- unit/file connection
 - changing connection properties 154
 - direct access 152
 - file definition 150
 - keyed access 152, 183
 - named files 150
 - OCSTATUS run-time option 152
 - OPEN statement, using an 150
 - preconnection, using 143
 - reconnecting
 - named files 160
 - preconnected files 161
 - unnamed files 160
 - sequential access 143, 152
 - temporary files 152
 - unnamed files 151
- unnamed common block
 - See common block, blank
- unnamed files
 - connecting 151
 - definition 137
 - preconnecting 143
 - reconnecting 160
 - subfiles 170
- UNSP status flag 344, 350
- unsupportable loop, definition 261
- UNTANY routine
 - coordinating parallel use of with locks 319
- UNTNOFD routine
 - coordinating parallel use of with locks 319

- UPDATE suboption of ICA compile-time option 29, 388
- UPDATE-IN-PLACE attribute 145
- USE suboption of ICA compile-time option 29, 388
- user errors, fixing 129
- user parameters 67, 84, 96
- using VS FORTRAN Version 2
 - under CMS 7
 - under MVS 11
 - under MVS with TSO 18

V

- variable-length records 140
- variables
 - accumulator usage 271
 - and assembler subprograms 464, 465
 - as actual argument 372
 - dummy, for alignment in common 375
 - efficient common arrangement 375
 - fixed order alignment in common 375
 - internal representation 469
 - length record description 148
 - optimization limitations 269
 - recognition when constant 271
 - storage map lists 44
- VECT status flag 344, 345, 349, 350
- vector
 - analysis of loops 280
 - definition 261
 - dependences 258, 262, 353
 - dependences, table of ignored 352
 - diagnostic message reporting 491
 - diagnostic messages 351
- directives
 - applications 353
 - ASSUME COUNT 356
 - format 355
 - global 355
 - IGNORE 358
 - interactions between 354
 - local 355
 - multiple 356
 - PREFER 364
 - verifying correct application 367
- eligibility of loops 277, 280
- enhanced facility 285
- examples
 - compound instructions 286
 - IF conversion 288
 - intrinsic functions 289
 - loop distribution 287
 - loop selection 287
 - printed report 348
 - reduction operations 289
 - scalar expansion 288
 - statement reordering 288

- vector (*continued*)
 - examples (*continued*)
 - terminal report 344
 - interrupts 459
 - intrinsic functions 285
 - mathematical functions 285
 - qualification for 277
 - report
 - See compiler, report
 - restrictions
 - interaction with static debug statements 292
 - math library routines 291
 - subscript values and array bounds 291
 - vector versus scalar summation 290
 - sectioning considerations 265
 - chunk size and 265
 - statistics table 352
 - table of ignored dependences 352
 - techniques for improvement
 - complex data 284
 - compound instructions 286
 - constructs preventing vectorization 277
 - enhanced vector facility 285
 - loops 280
 - program logic 281
 - section size 284
 - statements preventing vectorization 277
 - stride 282
 - subscripts 279
 - vector overhead 283
 - virtual memory 282
 - terminology
 - construct 258
 - dependence 258, 259
 - induction variable 258
 - loop distribution 258
 - noninductive subscript 258
 - recurrence 259
 - scalar expansion 260
 - stride 260
 - unanalyzable loop 261
 - unsupported loop 261
 - vector 261
 - vector section 261
- VECTOR compile-time option 36
 - defaults for 36
 - description of 36
 - INTRINSIC suboption 37
 - IVA suboption 38
 - MODEL suboption 39
 - NOINTRINSIC suboption 37
 - NOIVA suboption 38
 - NOREDUCTION suboption 38
 - NOREPORT suboption 36
 - PARALLEL compile-time option, using with 297
 - REDUCTION suboption 38
- VECTOR compile-time option (*continued*)
 - REPORT suboption 36
 - SIZE suboption 38
 - SPRECOPT suboption 39
- vectorizing loops 280
- VFEIL# entry point 457
- VFEIN# entry point 457
- VFEIS# 415
- VFT2RCL cataloged procedure under MVS 441
- VFT2RCLG cataloged procedure under MVS 441
- VFT2RLG cataloged procedure under MVS 441
- virtual
 - processor
 - definition of 294
- VM
 - See CMS considerations
- VM/XA
 - dynamic common and 379
- VM/XA considerations 68
 - parallel programs and 74
- VS FORTRAN Version 2
 - CMS, using under 7, 67
 - common coding errors 129
 - compiler invocation 473
 - compiling your program 7, 21
 - differences 479
 - fixing user errors 129
 - identifying run-time errors 118
 - interactive debug
 - effects on error handling 125
 - options, specifying 108
 - relationship of TEST and NOSDUMP compile-time options to 36
 - internal limits in 449
 - MVS with TSO, using under 18
 - MVS, using under 13
 - reentrant programs
 - creating under CMS 430
 - creating under MVS 439
 - separation tool, general description 426
 - source language considerations
 - obtaining the VSAM return code—IOSTAT option 226
 - processing VSAM direct file 222
 - processing VSAM sequential file 221
 - subprograms and shared data 371
- VSAM
 - file processing
 - alternate index path 215
 - alternate index terminology 216
 - catalog entry creation 213
 - cataloging and loading alternate index 217
 - example of defining a VSAM file 213
 - file definition 211
 - file organization 210
 - IOSTAT option obtains return code 226
 - operating system data definition statement 217

VSAM (*continued*)
 file processing (*continued*)
 source language considerations 220
 valid source statements, summary 220
 VSAM terminology 210
 VSF2MAC macro library 118
 VSF2RCS 432
 VSF2RSEP 432

W

W warning error code 48
 WAIT FOR statements
 example of 307, 312
 inhibits automatic parallel code generation 297
 parallel subroutines and 306
 task management and 310
 WAIT statement 179
 WRITE statement
 direct access 182
 directly accessed VSAM direct file 224
 formatted data
 list-directed 168
 NAMELIST 169
 user-specified 191
 internal files 174
 keyed access 184, 188
 sequential access 167
 sequentially accessed VSAM direct file 223
 unformatted data 179, 193
 unformatted record size and 149
 VSAM sequential file considerations 222
 writing data
 asynchronous I/O 179
 direct access 182
 formatted data
 list-directed 168
 NAMELIST 169
 user-specified 191
 internal files 174
 keyed access 184, 188
 sequential access 167
 unformatted data 193

X

XA support 68, 88
 XLIST report, compiler 349
 XREF
 compile-time option 40
 cross reference listing 44
 linkage editor option under MVS 78
 source program cross reference 46
 XUFLOW
 routine
 using in parallel 324

XUFLOW (*continued*)
 run-time option 116
 CMS, specification under 68
 description 116
 MVS, specification under 84
 TSO, specification under 96

Y

yy, operator message identifier 123

We'd Like to Hear from You

VS FORTRAN Version 2
Programming Guide for CMS and MVS
Release 6
Publication No. SC26-4222-07

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
 - IBMLink: HLASMPUB at STLVM27
 - Internet: COMMENTS@VNET.IBM.COM

Be sure to include the following with your comments:

- Title and publication number of this book
- Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

VS FORTRAN Version 2
Programming Guide for CMS and MVS
Release 6

Publication No. SC26-4222-07

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Grammatically correct and consistent	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Graphically well designed	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

Company or Organization

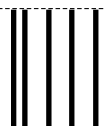
Phone No.



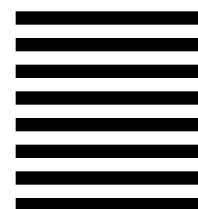
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945



Fold and Tape

Please do not staple

Fold and Tape



File Number: S370-40
Program Number: 5668-805
5668-806
5688-087

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

The VS FORTRAN Version 2 Library

LY27-9516 Diagnosis Guide
GC26-4219 General Information
SC26-4340 Installation and Customization for MVS
SC26-4339 Installation and Customization for CMS
SC26-4420 Installation and Customization for AIX/370
SC26-4223 Interactive Debug Guide and Reference
SC26-4221 Language and Library Reference
GC26-4225 Licensed Program Specifications
SC26-4603 Master Index and Glossary
SC26-4686 Migration from the Parallel FORTRAN PRPQ
SC26-4741 Programming Guide for AIX/370
SC26-4222 Programming Guide for CMS and MVS
SX26-3751 Reference Summary

SC26-4222-07





VS FORTRAN Version 2

Programming Guide for CMS and MVS

Release 6