IBM

VS FORTRAN Version 2

SC26-4222-3

Programming Guide

Release 3

sinh 3x sinh x(4 cosh² x 1)
sinh 4x sinh x cosh x(8 cosh² x 4)
sinh 5x sinh x(1 12 cosh² x 16 cosh⁴ x)
cosh 3x cosh x(4 cosh² x 3)
cosh 4x 1 8 cosh² x 8 cosh⁴ x
cosh 5x cosh x(5 20 cosh² x 16 cosh⁴ x)
a1
a2
c10
c21
x = 2y-b-a / b-a
f(1+a2)1/2

IBM

VS FORTRAN Version 2

SC26-4222-3

Programming Guide

Release 3

Fourth Edition (March 1988)

This is a major revision of, and makes obsolete, SC26-4222-2.

This edition applies to Release 3 of VS FORTRAN Version 2, Program Numbers 5668-805 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 the preface ("About This Manual"). Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any republication of the page affected. 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.

About This Book

This book explains how to compile and run programs using VS FORTRAN Version 2. It also contains information on advanced coding topics. This book is not intended as a tutorial on the FORTRAN language. Rather, it assumes you have basic FORTRAN knowledge and now want to learn to use VS FORTRAN Version 2.

How This Book Is Organized

This book is organized as follows:

- **Chapter 1, "Overview of VS FORTRAN Version 2,"** gives an overview of the VS FORTRAN Version 2 language, compiler, library, and Interactive Debug.
- **Chapter 2, "Compiling Your Program,"** explains how to compile your program under VM, MVS, and TSO.
- **Chapter 3, "Using the Compiler 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 VM, MVS, and 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, "Performing Input/Output Operations,"** explains how to use READ, WRITE, and other input/output statements.
- **Chapter 7, "Associating Data,"** explains how to associate data between calling and called programs by means of passed arguments and common data areas. It also explains how to use the intercompilation analysis feature.
- **Chapter 8, "Optimizing Your Program,"** suggests ways to make your programs run faster and the best way to use the OPTIMIZE compiler option.
- **Chapter 9, "Vectorizing Your Program,"** explains how to code programs that make use of the IBM System 3090 Vector Facility.
- **Chapter 10, "Creating Reentrant Programs,"** explains the advantages and limitations of reentrant programs and gives an overview of how to create and use them.
- **Chapter 11, "Using VSAM with VS FORTRAN Version 2,"** discusses considerations for using VSAM files with VS FORTRAN Version 2.
- **Appendix A, "Assembler Language Considerations,"** explains how to call FORTRAN subprograms and main programs from assembler programs.
- **Appendix B, "Object Module Records and Statement Table,"** describes the structure of the object module and contents of the SYM object module record. It also describes the statement table, which contains information on the internal statement numbers in the program.

- **Appendix C, "Compatibility Considerations,"** discusses compatiblity of VS FORTRAN Version 2 with VS FORTRAN Version 1 as well as with earlier IBM FORTRAN products.

- **Appendix D, "Internal Limits in VS FORTRAN Version 2,"** describes the specifications for the maximum sizes and lengths for various VS FORTRAN statements and contructs.

- **Appendix E, "The Multitasking Facility (MTF),"** explains how to use the VS FORTRAN Version 2 multitasking facility (MTF) under MVS.

- **Appendix F, "Vector Report Diagnostic Messages,"** describes the diagnostic messages that appear in the vector report listing.

- **Appendix G, "What Determines File Existence,"** describes the conditions that VS FORTRAN Version 2 uses to determine the existence of input/output files.

- **Appendix H, "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.

- **Appendix I, "Sample Compiler Listing with Double-Byte Characters,"** shows a compiler listing containing double-byte Kanji characters.

How to Use This Book

For the task of 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 on the VS FORTRAN Version 2 language and library.

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, however, 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 may appear one or more times in succession.

- Braces ({ }) group items from which you must choose one.

- Square brackets ([ ]) group optional items from which you may choose none, one, or more.

- OR signs (|) indicate you may 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 may be significant. Code non-FORTRAN statements exactly as shown.

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 an EBCDIC double-byte character (X'42')
kk	represents a double-byte character not in the EBCDIC double-byte character set

Summary of the VS FORTRAN Version 2 Publications

The following table lists the VS FORTRAN Version 2 publications and the tasks they support.

Task	VS FORTRAN Version 2 Publications	Order Numbers
Evaluation and Planning	General Information	GC46-4219
	Licensed Program Specifications	GC26-4225
Installation and Customization	Installation and Customization for VM	SC26-4339
	Installation and Customization for MVS	SC26-4340
Application Programming	Language and Library Reference	SC26-4221
	Programming Guide	SC26-4222
	Interactive Debug Guide and Reference	SC26-4223
	Reference Summary	SX26-3751
Diagnosis	Diagnosis Guide	LY27-9516

Industry Standards

The VS FORTRAN Version 2 Compiler and Library licensed program is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of March, 1988.

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 *VS FORTRAN Version 2 Language and Library Reference*, these extensions are printed in color.

Summary of Changes

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, more adjustably dimensioned arrays, and more 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 on-line HELP function of Interactive Debug.
  - Simplification of the VECTOR compile-time option.
  - Vectorization is allowed at OPTIMIZE(2) and OPTIMIZE(3).
  - Changes to the way in which the vector feature treats partial sum processing result in a performance improvement.
- 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, I/O, and for Interactive Debug input and output.
    - 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 (DIV) facility of MVS/XA. (You will find the discussion of VS FORTRAN Version 2's support for DIV in the *Language and Library Reference*.)
    - 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 FORTRAN unit numbers that are available by using the UNTANY and UNTNOFD service subroutines.
- Enhancements to the full screen functions of Interactive Debug

Major Changes to This Manual

- Documentation of the major product enhancements has been added.
- Chapters have been reorganized to improve retrievability of information.
- Editorial changes have been made throughout.

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.

Major Changes to This Manual

Documentation of the major product enhancements has been added.

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

Major Changes to This Manual

Documentation of the above product enhancements has been added.

Contents

Figures

6. Source Program Listing Example—SOURCE and SRCFLG Options . . . 40

Part 1. Introduction

Chapter 1. Overview of VS FORTRAN Version 2

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 on the VS FORTRAN Version 2 language, see *VS FORTRAN Version 2 Language and Library Reference*.

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

Compiler

The VS FORTRAN Version 2 compiler analyzes the source program statements and translates them into a machine language program called the object program. The object program can be combined with library routines to form a program which 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 programs generated by the compiler also operate under operating system control and depend on it for similar services.

Information on how to compile your source programs is contained in this book.

Run-Time Library

The VS FORTRAN Version 2 run-time library contains subprograms which 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

VS FORTRAN Version 2 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 VS FORTRAN Version 2 compiler translates FORTRAN source statements into object code and creates an object module for processing. With the following exceptions, you can compile your source program under any supported operating system and then run it under any of the other supported systems:

- Programs with functions unique to MVS, such as Asynchronous I/O, Data-In-Virtual, or MTF, can be compiled under VM, but must be run under MVS.
- Programs with system-dependent file names on I/O statements can be compiled under any operating system but must be run on the appropriate system.
- Programs with a system-dependent file name on the INCLUDE compiler directive must be compiled on the appropriate system.

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

- If you are a VM user, begin with the section that immediately follows.
- If you are an MVS batch user, skip to page 12.
- If you are a TSO user, skip to page 18.

Compiling Your Program under VM

The following sections discuss:

- How to request compilation.
- Using the INCLUDE statement to direct the compiler to read source statements from another file.
- Compiler output with the appropriate VS FORTRAN Version 2 library.

Requesting Compilation

You cannot compile your programs in the CMS/DOS environment. This is because processing of your program in a CMS virtual machine is done in CMS's OS simulation mode; that is, the VS FORTRAN Version 2 run-time service subroutines use the MVS services that are simulated by CMS. If you have been running other programs in CMS/DOS mode, you must issue the command

```
SET DOS OFF
```

before attempting to compile your VS FORTRAN Version 2 programs.

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

Examples are:

```
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 do issue one and the filenames, filetypes, or filemodes specified on FORTVS2 and the FILEDEF command do not agree, FORTVS2 overrides the FILEDEF.

Specifying Compiler Options

You can specify most compiler options on either the FORTVS2 command or the @PROCESS statement (see "Modifying Compiler Options—@PROCESS Statement" on page 36). The option for specifying the disposition of the listing file can be specified only on the FORTVS2 command. 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

Specifying Compile-Time Options

Options on the @PROCESS statement override those of FORTVS2. However, if an option on FORTVS2 conflicts with an option from another source, that is, a default option or @PROCESS option, the option assumed is that shown in Figure 4 on page 36.

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.

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 TAPn (options
```

where *n* is a number from 1 through 4 that corresponds to virtual tape units 181 through 184, 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 TAP*n* 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 compiler options.

For information on invoking the VS FORTRAN compiler from an assembler program see "Requesting Compilation from an Assembler Program" on page 325.

Using the FORTRAN INCLUDE Directive

INCLUDE is a statement that you can code in your source program to direct the compiler to read source statements from another file. When the end of the included file is reached, the compiler resumes processing with the line following the INCLUDE statement. Included files may reside on any accessed minidisk.

The INCLUDE statement has two formats:

1. If you want your program to conform to the Systems Application Architecture, use the following format:

   ```
   INCLUDE char-constant
   ```

   where *char-constant* is a character constant whose value is a CMS file identifier and, optionally, the name of a MACLIB member, as follows:

   fn [*ft* [*fm*]] [(*member-name*)]

   If you omit the file type, FORTRAN is the default, unless the file is a MACLIB member, in which case MACLIB is the default file type. If you omit the file mode, A1 is the default; or if you omit just the number on the file mode, 1 is the default. You may specify an asterisk (*) for the file mode.

   With this format of INCLUDE, you need not have a FILEDEF command in effect for the included file (or 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. Examples of INCLUDE statements coded in this format are:

   ```
   INCLUDE 'CONSTANT'
   INCLUDE 'COMMON PROJ_01'
   INCLUDE 'MASKS-1 INCLUDE Z1'
   INCLUDE 'OLDPROJ MACLIB (CONTROL)'
   ```

2. Alternatively, you can use the following format of the INCLUDE statement:

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

where *member-name* is the name of a MACLIB member. (For information on how to create MACLIB members, see the appropriate CMS User's Guide for your operating system.) The record formats allowed are fixed blocked and fixed unblocked. An example of an INCLUDE statement coded in this format is:

```
INCLUDE (CONST)
```

With this format, 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
  ```

Note that for the second format above, the set of characters allowed for the member name is more restrictive than for the first format. For additional information on coding rules, see *VS FORTRAN Version 2 Language and Library Reference*.

Conditionally Including Files

You can selectively activate INCLUDE statements if they are coded in the second format shown above by assigning them an identification number and referring to them on the CI compiler option. The identification number is coded after the member name, as shown below:

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

For example:

```
INCLUDE (CONST) 1
:
INCLUDE (DATA) 2
:
INCLUDE (FORMULA) 3
```

If you wanted to include CONST and FORMULA but not DATA, you would specify CI(1,3) at compile-time.

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:

- LISTING file
- TEXT file
- PIF file
- Intercompilation Analysis file

LISTING File

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 38 and "Object Module Listing—LIST Option" on page 114.

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
```

TEXT File

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

If the OBJECT compiler 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 file 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 the object code to be placed. To put an object file into MY FILE2 A, for example, issue the following FILEDEF:

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

If the DECK compiler 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 the object code to be placed. To put an object file into MY FILE3 A, for example, issue the following FILEDEF:

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

PIF 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 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.

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 PIF.

ICAFILE File

The Intercompilation Analysis file is produced if you specify the UPDATE sub-option of the ICA compiler option. For information on intercompilation analysis, see "Intercompilation Analysis" on page 201.

Compiling Your Program under MVS

The following sections discuss:

- Job processing and coding the JOB, PROC, EXEC, and DD statements
- How to request compilation
- Defining data sets used by the compiler
- Using the INCLUDE statement to direct the compiler to read source statements from another file
- Compiler output

Job Processing

Three basic steps are taken to process a FORTRAN program:

1. Compiling
2. Link editing
3. Load module processing (go step)

The input to the compile step is called the *source*. The output from the compile step is called an *object module*, which is the input to the link-edit step. The output of the link-edit step is the *load module*, which is one or more object modules with all external references resolved. The load module is the program run in the go step. If the loader is used in place of the linkage editor, the last two steps (link-edit and load module processing) are combined into one step.

Each step is called a *job step*—the processing of one program. Each job step may be run alone or in combination with other job steps as a job—an application involving one or more job steps. Hence, a job may consist of one step, such as FORTRAN compiler processing, or of many steps, such as compiler processing followed by linkage editor processing and load module processing.

You define the requirements of each job 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 (GC28-0692)
MVS/Extended Architecture JCL (GC28-1148)

Identifying a Job—JOB Statement

The JOB statement begins each MVS job you enter into the system:

//jobname JOB [*parameters*]

The *jobname* identifies this job to the system. The jobname must conform to the standards defined in your appropriate JCL manual.

The *parameters* give accounting and processing information that is specific to your site.

Assigning Default Values—PROC Statement

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

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

The *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. You assign a name to the procedure when adding it to the procedure library, for example, 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 your appropriate JCL manual.

Requesting Execution of the Job Step—EXEC Statement

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

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

The *stepname* identifies this job step.

The *procname* is the name of a procedure you want executed. The names of the IBM-supplied cataloged procedures, which are in your appropriate system procedure library, are given in Figure 24 on page 84.

The PARM parameter lets you specify any compiler options.

The *other parameters* specify accounting and processing information specific to your site.

Defining Files—DD Statement

To define a file you may need, you must specify a DD statement:

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

The *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 1 on page 15 below, Figure 2 on page 16, and Figure 20 on page 80.

The *procstep* identifies the procedure step.

The *data-set-name* is the qualified name you've given the data set that contains your file or files.

The *other parameters* specify additional information about the data set, such as its location and space allocation.

Compiling and Running

The simplest way to compile and run your program is to use the IBM-supplied catalog procedure VSF2CLG, as shown in the sample JCL below. VSF2CLG compiles, link-edits, and runs your program.

```
//jobname    JOB
//           EXEC VSF2CLG
//FORT.SYSIN DD *
(source program)
/*
//
```

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, while VSF2C only compiles. For a list of all the cataloged procedures, see Figure 24 on page 84. The cataloged procedures should be located in your appropriate 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 appropriate JCL manual for your operating system.

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 *
(source program)
/*
//
```

For information on invoking the VS FORTRAN compiler from an assembler program see "Requesting Compilation from an Assembler Program" on page 325.

Defining Compiler Data Sets

Figure 1 lists the required and optional data sets used by the compiler. Many of the data sets used by the compiler are defined in cataloged procedures; for those that are not, you must supply DD statements. (Cataloged procedures are discussed under "Requesting Compilation Only" on page 14.)

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
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 the FORTRAN INCLUDE Directive" 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
SYSTERM	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 or UPDATE 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

Figure 1. Compiler Data Sets

Notes to Figure 1:

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 2 lists the DCB default values for compiler data set characteristics.

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

Figure 2. Compiler Data Set DCB Default Values

Note to Figure 2:

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**

  the Program Information File should be named:

  ***userid.module-name*.PIF**

- If the input to the compiler is in a partitioned data set named:

  ***userid.module-name*.FORTRAN(*member-name*)**

  the Program Information File should be named:

  ***userid.member-name*.PIF**

Using the FORTRAN INCLUDE Directive

INCLUDE is a statement that you can code in your source program to direct the compiler to read source statements from another file. When the end of the included file is reached, the compiler resumes processing with the line following the INCLUDE statement.

The INCLUDE statement has two formats:

1. If you want your program to conform to the System Application Architecture, use the following format:

   INCLUDE *char-constant*

   where *char-constant* is a character constant whose value is a fully-qualified data set name or partioned data set member. The data set must be cataloged. With the above format, you need not code a DD statement for the included file. The record formats allowed are the same as those that are allowed for any VS FORTRAN source file. Examples of INCLUDE statements coded in this format are:

```
INCLUDE 'USER.HISTO.PACKAGE'
INCLUDE 'USER.PROJ1.FORT.INCL(COMMON)'
INCLUDE 'USER.PROF2.FORT.CONST(MASKS)'
```

2. Alternatively, you can use the following format of the INCLUDE statement:

   INCLUDE (*member-name*)

   where *member-name* is the name of a partitioned data set member. With this format 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 an INCLUDE statement coded in this format is:

   ```
   INCLUDE (MASKS)
   ```

   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.

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

Conditionally Including Files

You can selectively activate INCLUDE statements if they are coded in the second format shown above by assigning them an identification number and referring to them on the CI compiler option. The identification number is coded after the member name, as shown below:

INCLUDE (*member-name*) *n*

For example:

```
INCLUDE (CONST) 1
:
INCLUDE (DATA) 2
:
INCLUDE (FORMULA) 3
```

If you wanted to include CONST and FORMULA but not DATA, you would specify CI(1,3) at compile-time.

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:

- Compiler output listing
- Object module
- Program Information File
- Intercompilation analysis file

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 38 and "Object Module Listing—LIST Option" on page 114.

Object Module

If you specified the OBJECT compiler option, the object module is directed to the data set defined by the SYSLIN ddname. If you specified the DECK compiler option, the object module is directed to the data set defined by the SYSLIN ddname.

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

This file is produced if you specify the ICA compiler option. For information on intercompilation analysis, see "Intercompilation Analysis" on page 201.

Compiling Your Program under TSO

The following sections discuss:

- How to request compilation
- Allocating compiler data sets
- Compiler output

For additional information on TSO, see:

OS/VS2 MVS TSO Command Language Reference (GC28-0646)
OS/VS2 MVS TSO Terminal User's Guide (GC28-0645)

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 the FORTRAN INCLUDE Directive" 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 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 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 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 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 compiler options, issue the CALL command as follows:

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

or you can request one or more compiler options explicitly:

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

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:

- Compiler output listing
- Object module
- Program Information File
- Intercompilation analysis file

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 38 and "Object Module Listing—LIST Option" on page 114.

Object Module

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.

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

This file is produced if you specify the ICA compiler option. For information on intercompilation analysis, see "Intercompilation Analysis" on page 201.

Chapter 3. Using the Compiler Options

VS FORTRAN Version 2 compiler 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.

Available Compiler Options

Figure 3 lists the compiler options, their abbreviations, and the IBM-supplied defaults. Your system administrator may have changed these defaults for your installation; there is a column in the figure where you can note any change.

Option	Abbreviation	Installation Default
AUTODBL(*value* | NONE)	AD	See note below
CHARLEN(*number* | 500)	CL	
CI(*number1*,*number2*,...)	None	See note below
DBCS | NODBCS	None	
DC(*name1*,*name2*,...)	None	See note below
DECK | NODECK	D | NOD	
DIRECTIVE (*trigger-constant*) | NODIRECTIVE [(*trigger-constant*)] | NODIRECTIVE	DIR NODIR	See note below
FIPS(S | F) | NOFIPS	None	
FLAG(I | W | E | S)	None	
FREE | FIXED	None	
GOSTMT | NOGOSTMT	GS | NOGS	
ICA [(USE(*name1*,*name2*,...) UPDATE(*name*) MXREF | NOMXREF CLEN | NOCLEN CVAR | NOCVAR MSG(NEW | NONE | ALL) MSGON(*number1*,*number2*,...) | MSGOFF(*number1*,*number2*,...))] | NOICA	 UPD MON MOFF	See note below
IL(DIM | NODIM)	None	
LANGLVL (66 | 77)	LVL	
LINECOUNT(*number* | 60)	LC	
LIST | NOLIST	L | NOL	
MAP | NOMAP	None	
NAME(*name* | MAIN)	None	
OBJECT | NOOBJECT	OBJ | NOOBJ	
OPTIMIZE(0 | 1 | 2 | 3) | NOOPTIMIZE	OPT | NOOPT	
RENT | NORENT	None	
SAA | NOSAA	None	

Figure 3 (Part 1 of 2). VS FORTRAN Version 2 Compiler Options

Option	Abbreviation	Installation Default
SDUMP[(ISN | SEQ)] | NOSDUMP	SD | NOSD	
SOURCE | NOSOURCE	S | NOS	
SRCFLG | NOSRCFLG	SF | NOSF	
SXM | NOSXM	None	
SYM | NOSYM	None	
TERMINAL | NOTERMINAL	TERM | NOTERM	
TEST | NOTEST	None	
TRMFLG | NOTRMFLG	TF | NOTF	
VECTOR [(REPORT[(*optionlist*)] | NOREPORT INTRINSIC | NOINTRINSIC IVA | NOIVA REDUCTION | NOREDUCTION SIZE(ANY | LOCAL | *n*))] | NOVECTOR	VEC REP | NOREP INT | NOINT RED | NORED SIZ NOVEC	See note below
XREF | NOXREF	X | NOX	

Figure 3 (Part 2 of 2). VS FORTRAN Version 2 Compiler Options

Note: These options cannot be changed at installation time. The value is always the IBM-supplied default, if any.

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—AUTODBLOption" on page 47.

The value can be:

NONE
Indicates no conversion is to be performed.

DBL
Indicates that promotion of both single and double-precision items is to take place. 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 promotion of only single-precision items is to take place.

DBL8
Indicates that promotion of only double-precision items is to take place.

DBLPAD
Indicates that both promotion and padding are to take place for single and double-precision items. 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 promotion of single-precision items, such as REAL*4 and COMPLEX*8 items, is to take place. Items of other types are padded if they share storage space with promoted items.

DBLPAD8

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

nnnnn

Indicates that the program is to be converted according to the value of *nnnnn*, 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 as follows:

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 all logical 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 all integer 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 all 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.

The following AUTODBL values are equivalent:

AUTODBL(NONE)	is equivalent to	AUTODBL(00000)
AUTODBL(DBL)	is equivalent to	AUTODBL(30000)
AUTODBL(DBL4)	is equivalent to	AUTODBL(10000)
AUTODBL(DBL8)	is equivalent to	AUTODBL(20000)
AUTODBL(DBLPAD)	is equivalent to	AUTODBL(33334)
AUTODBL(DBLPAD4)	is equivalent to	AUTODBL(13336)
AUTODBL(DBLPAD8)	is equivalent to	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 non-converted program to link with a program converted with the option AUTODBL(DBLPAD).

CHARLEN(*number* | <u>500</u>)
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.

CI(*number1,number2,...*)
Specifies the identification numbers of the INCLUDE statements to be processed. For information on INCLUDE statements, see the correct section for your operating system: "Using the FORTRAN INCLUDE Directive" on page 9 (VM) or "Using the FORTRAN INCLUDE Directive" on page 16 (MVS).

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

DBCS | <u>NODBCS</u>
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 single-byte 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 very large 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.

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 VM: When you specify the DC option in the FORTVS2 command, you can pass up to 31 characters to VS FORTRAN Version 2 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.

DECK | <u>NODECK</u>
specifies whether or not the compiler is to write the object module to the data set defined by the ddname SYSPUNCH.

DIRECTIVE (*trigger-constant*) | NODIRECTIVE [(*trigger-constant*)] | <u>NODIRECTIVE</u>
Specifies whether or not the processing of selected comments as vector directive statements is enabled or disabled. Refer to "Using Vector Directives" on page 253 for more information on 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. Refer to "Using Vector Directives" on page 253 for more information on vector directives.

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

FIPS (S | F) | <u>NOFIPS</u>
Specifies whether or not standard language flagging is to be performed, and, if it is, the standard language flagging level: subset or full.

Items not defined in 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 LANGLVL(66) and FIPS flagging at either level, the FIPS option is ignored.

FLAG (<u>I</u> | 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. Thus, if you want to suppress all messages that are warning or informational, specify FLAG(E).

FREE | <u>FIXED</u>
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 | <u>NOGOSTMT</u>
Specifies whether or not 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.

ICA

[(USE(*name1,name2,...*)
UPDATE(*name*)
MXREF | NOMXREF
CLEN | NOCLEN
CVAR | NOCVAR
MSG(NEW | NONE | ALL)
MSGON(*number1,number2,...*) | MSGOFF(*number1,number2,...*))]
| <u>NOICA</u>

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

USE(*name1,name2,...*)
Specifies the names of the ICA files against which a program unit is to be checked. This option can be repeated any number of times as long as the total number of files specified in USE and UPDATE suboptions does not exceed forty (40). For more information about this option, see "Notes on the USE and UPDATE Suboptions" on page 208.

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.

UPDATE(*name*)
Specifies the name of the intercompilation analysis file that is to be created or updated.

name
The intercompilation analysis file name, which must be a sequence of 1 to 8 alphameric characters, beginning with a letter.

For more information about this option, see "Notes on the USE and UPDATE Suboptions" on page 208.

<u>MXREF</u> | NOMXREF
Specifies whether to produce external cross-reference listings.

<u>CLEN</u> | NOCLEN
Specifies whether to check the length of named common blocks.

CVAR | <u>NOCVAR</u>
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. Conversely, 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 ILX0061I.

See "Using the MSGON and MSGOFF Suboptions to Suppress Messages" on page 211 for a list of the message numbers and corresponding message texts.

IL(DIM | NODIM)
Specifies whether the code for adjustably-dimensioned arrays is to be placed inline — IL(DIM), or done via library call — 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 LANGLVL(77) processing.

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 or not 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 114.

MAP | NOMAP
Specifies whether or not 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 43.

NAME(*name* | MAIN)
Can only be specified when LANGLVL(66) is specified. It specifies the name of the CSECT generated in the object module. It only applies to main programs.

OBJECT | NOOBJECT
Under VM, directs the compiler to write the object module to the file associated with the ddname TEXT.

Under MVS, directs the compiler 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 register and branch 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 8, "Optimizing Your Program" on page 217.

RENT | NORENT
Specifies that the object module generated be suitable for use in a shareable area. If you are not planning on running your program in a shareable area, specify NORENT. Otherwise, see Chapter 10, "Creating Reentrant Programs" on page 269.

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

SDUMP[(ISN | SEQ)] | NOSDUMP
Specifies that 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. These effects are described as follows:

- Null sequence numbers are set to 1. Because the compiler does not verify that numbers are unique or in a meaningful sequence, the user must assume responsibility for insuring the integrity of the sequence numbers. 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 or not. 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. For more details, see "Using the SXM Option" on page 42.

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 "SYM Record" on page 335.

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 to a SYSTERM data set. See "Using the Terminal Output Display—TERMINAL and TRMFLG Options" on page 46.

TEST | NOTEST
TEST overrides any optimization level above OPTIMIZE(0). It 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
Causes the initial line of source statements in error and their associated error messages (formatted for the terminal being used) to be displayed 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 46.

VECTOR

> **[(REPORT[(*optionlist*)] | NOREPORT**
> **INTRINSIC | NOINTRINSIC**
> **IVA | NOIVA**
> **REDUCTION | NOREDUCTION**
> **SIZE(ANY | LOCAL | *n*))]**
> **| <u>NOVECTOR</u>**

Invokes the vectorization process, which produces programs that can utilize the speed of the IBM System 3090 Vector Facility. This option instructs the compiler to transform eligible statements in DO 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, certain FORTRAN statements, such as DEBUG or invalid FORTRAN statements, downgrade the optimization level after it has been set in the @PROCESS statement or passed as a run-time option. As a result, the optimization level may be downgraded to 0 and no vectorization occurs.

The vectorization process is described in detail in Chapter 9, "Vectorizing Your Program" on page 227.

The VECTOR compiler option includes the following suboptions:

REPORT[(*optionlist*)] | <u>NOREPORT</u>

Instructs the compiler to either display a report of vectorization information 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 DO loop nesting, flags that identify vectorized and non-vectorized loops, and messages that give more detailed information about the vectorization process. See "Producing Vector Reports" on page 237 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. Acceptable abbreviations are:

TERM	TE
LIST	LI
XLIST	XL
SLIST	SL
STAT	ST

Following is a brief description of the options:

TERM	produces a vector report at the terminal.
LIST	produces a simple format of the vector report on a printed listing.

XLIST	produces an extended format of the vector report on a printed listing.
SLIST	produces a vector report, in a format similar to that produced by the SOURCE compiler option, on a printed listing.
STAT	produces a vector statistics table on a printed 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 reports can be requested individually or in any combination.

The IBM-supplied default is NOREPORT, but if REPORT is specified without the TERM, LIST, XLIST, or SLIST suboptions, XLIST is the IBM-supplied default.

INTRINSIC | NOINTRINSIC

Enables or disables the vectorization of out-of-line intrinsic function references.

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, as there are no vector entry points in the old routines.

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 will be 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 will not vectorize. The scalar math routines of the library you have accessed first will be used.

IVA | NOIVA

Produces 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 15.

REDUCTION | NOREDUCTION
Enables or disables the vectorization of reduction functions.

The translation of vector operations from scalar to vector code may produce different results. Because vector code operations may not be performed in scalar processing order, inherent variations in the data that is accumulated cause the results to be dependent on the order of accumulation.

You should be aware of this when using floating point data because addition is not associative with this type of data.

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 not be as 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 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.

XREF | NOXREF
Specifies whether or not a source cross-reference listing is to be produced. For a description of cross-reference output, see "Cross Reference—XREF Option" on page 44.

Conflicting Compiler Options

The following table lists conflicting compiler options. The table also lists those options that are assumed when conflicting compiler options are specified.

Conflicting Compiler Options		Options Assumed	
DBCS	FIPS	DBCS	NOFIPS
DBCS	SAA	DBCS	NOSAA
FIPS	FLAG¬=I	FIPS	FLAG=I
FIPS	SAA	Installation default	Installation default
FREE	SAA	FREE	NOSAA
FREE	FIPS	FREE	NOFIPS
FREE	SDUMP(SEQ)	FREE	SDUMP(ISN)
LANGLVL(66)	FIPS	LANGLVL(66)	NOFIPS
LANGLVL(66)	SAA	LANGLVL(66)	NOSAA
LANGLVL(66)	DBCS	LANGLVL(66)	NODBCS
LANGLVL(77)	NAME	LANGLVL(77)	Ignore NAME
NODECK	SYM	NODECK	NOSYM
NOOBJ	SYM	NOOBJ	NOSYM
NOTRMFLG	VEC(REP(TERM...))	NOTRMFLG	VEC(REP(...))
TEST	NOSDUMP	TEST	SDUMP(ISN)
TEST	OPT(1), OPT(2), or OPT(3)	TEST	OPT(0)
VEC	OPT(0) or OPT(1)	VEC	OPT(3)
VEC(IVA)	NOSDUMP	VEC(IVA)	SDUMP(ISN)

Figure 4. Conflicting Compiler Options

Modifying Compiler Options—@PROCESS Statement

The options you specify when you request compilation remain in force for all source programs you're compiling, unless you override them with the @PROCESS statement.

Each source program requires its own @PROCESS statement if you wish to override the options specified at compiler invocation. If any source program does not have its own @PROCESS statement, it is compiled according to the compiler-invocation specifications—NOT according to the @PROCESS specifications of the preceding source program in the job stream. (See Chapter 2, "Compiling Your Program" on page 7 for information on how to specify options when you invoke the compiler.)

If an option on the @PROCESS statement conflicts with an option from another source, for example a default option or one you specify at compiler invocation, the option assumed is that shown in Figure 4.

To change the compiler options, place the @PROCESS statement before the first line in the source program. The following coding rules apply:

- @PROCESS must appear in columns 1 through 8 of the statement.
- The @PROCESS statement can be followed by compiler options in columns 9 through 72 of the statement. The options must be separated by commas or blanks.
- Up to 20 @PROCESS statements can be supplied for a program unit. Columns 9 through 72 of a following @PROCESS statement are appended to the previous @PROCESS statement.

  Intervening lines must not appear between @PROCESS statements. A line such as a blank, comment, or INCLUDE will terminate @PROCESS statement input, and subsequent @PROCESS statements will be ignored.

The following restrictions apply to the options you may specify:

- All compiler options except OBJECT, DECK, [DISK, PRINT, and NOPRINT] are permissible (the latter three are available only to CMS users; for details, see "Specifying Compiler Options" on page 8).
- If NODECK or NOOBJ has been specified on the compiler invocation, you cannot specify DECK or OBJ, respectively, on the @PROCESS statement.
- TERMINAL and TRMFLG cannot be specified on the @PROCESS statement if TERMINAL was not specified on the compiler invocation or in the installation defaults.
- Only the NOICA form of the ICA option can be specified in the @PROCESS STATEMENT.

Compiler Output

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

- Compiler output listing
- Object module—a translation of your program in machine code
- Program Information File (PIF)—a file 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*)
- Messages about the results of the compilation

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

Using the Compiler Output Listing

If you use the IBM-supplied default compiler 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 compiler options you can use to modify it.

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:

```
MAY 1, 1988        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.

The first page of the listing also shows the compiler options (default and explicit) in effect for this compilation.

Note: An installation option allows you to print the date as:

```
year month day
```

instead of the format shown above.

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 5 on page 39 and Figure 6 on page 40 show examples of the source program listing.

The listing format provides additional information on non-commentary source lines. These are provided as interpretive and debugging aids for large or complex programs:

- 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. The DO-level count can be an aid in understanding a complex program where DO loops may span several pages of the listing.

- 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 can be useful in locating "open" block-IF statements (those missing an ENDIF), as well as an aid to understanding program logic.

- Flagging of source statements which have been read from a source library through the INCLUDE statement. A '+' to the immediate left of a source statement indicates that the statement was read from an "included" file.

The brief example given in Figure 5 illustrates how the source listing flags the IF and DO levels and the INCLUDE code.

```
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)
                   *       ,sgn
          3          Integer*4   func,N,M
             *
          4          Include (codes)
             *
             * Definition of function codes.
             *
          5 +          Integer*4 fadd,fsub,fmul,fdiv
          6 +          Parameter(
            +      *       fadd = 1
            +      *      ,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 fucntion specified for ADDOP')
1        20          Endif
         21          Return
         22          End
*STATISTICS*   SOURCE STATEMENTS = 22, PROGRAM SIZE = 1592 BYTES,
PROGRAM NAME = ADDOP      PAGE:     1.
*STATISTICS*    NO DIAGNOSTICS GENERATED.
**ADDOP** END OF COMPILATION 1 ******
```

Figure 5. Source Program Listing Example—with IF, DO, and INCLUDE Flags

```
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 6. Source Program Listing Example—SOURCE and SRCFLG Options

Note: When this program is compiled, the diagnostic messages shown in Figure 7 on page 41 are produced.

Source Program Listing—SRCFLG Option

The SRCFLG option enables you to 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 6 shows an example of an error that occurred at ISN 14.

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 7 on page 41.

```
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.

*STATISTICS*   SOURCE STATEMENTS = 20, PROGRAM SIZE = 1372 BYTES,
               PROGRAM NAME=GAUSS  PAGE: 1.
*STATISTICS* 2 DIAGNOSTICS GENERATED. HIGHEST SEVERITY CODE IS 8.
**GAUSS** END OF COMPILATION 1 ******
```

Figure 7. 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 compilation, if the compiler detects an S-level error, it inserts a call for a library function instead of generating the code for the statement. During 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.

Using the SXM Option

You can use the SXM compiler 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.

Using the MAP and XREF Options

The MAP and XREF compiler 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 made of 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:

  1. Unreferenced variables
  2. Variables referenced in only one place
- Are all referenced variables set? (Except for variables in common, parameters, initialized variables, etc.)
- 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.)
- Have you inadvertently altered a variable passed to the main entry of a subroutine? (For example, at a subordinate entry point.)

Source Program Map—MAP Option

The following paragraphs describe each area of a storage map, such as that shown in Figure 8 on page 43.

```
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  SIZE OF PROGRAM: 55C HEX BYTES.
 NAME         TYPE  TAG   DISPL.  NAME         TYPE  TAG   DISPL.
 ________     ____  ____  ______  ________     ____  ____  ______
 ABS                I             ERF                I
 EXP                I             GAUSS         R*8  SFR   0001D0
 GUASS        R*4   SF    0001F8  MAX                I
 MU           R*8   F     000208  RANGE_HI      R*8  F     00020C
 RANGE_LO     R*8   F     000210  ROOT_PI       R*8  FK    1846D8
 ROOT_2       R*8   FK    0001C8  SIGMA         R*8  FA    000214
 T_HI         R*8   SFA   0001D8  T_LO          R*8  SFA   0001E0
 TOTAL_AREA   R*8   SF    0001E8  X             R*8  F     000218
 X_SQ         R*8   SF    0001F0

***** NO USER LABELS *****
```

Figure 8. Example of a Storage Map—MAP Option

Name Column

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.

Type Column

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

Tag Column

The third column is headed TAG—it displays use codes for each name and variable.

Displacement Column

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.

Common Block Maps—MAP Option

If your source program contains COMMON statements, you'll also get a storage map for each common block.

The map for a common block contains much the same kind of information as for the main program. The DISPL column shows the displacement from the beginning of the common block.

Any equivalenced common variable is listed with its name followed by (E); its displacement (offset) from the beginning of the block is also given.

Statement Label Map—MAP Option

The MAP option also gives you a statement label map 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.

Cross Reference—XREF Option

The XREF option produces cross references for symbols and statement labels used in the source program. Figure 9 on page 45 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
NAME        TYPE   TAG    DECLARED 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
0
**** NO USER LABELS ****
```

Figure 9. Example of Cross Reference—XREF Option

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

NAME Column: Names are listed in alphabetic order. Names containing double-byte characters are listed in binary value order before EBCDIC names.

TYPE Column: Each name is listed in the same format as for the MAP option described above.

TAG Column: The tag for each name shows its usage. The symbols used in the TAG column are defined at the top of the cross reference.

DECLARED Column: The DECLARED column gives the internal statement number where the data item is defined.

REFERENCED Field: 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.

End of Compilation Message

The last entry of the compiler output listing is the informative message:

MAIN END OF COMPILATION *n* ****** TIME STAMP: YY.DDDHH.MM.SS

where MAIN is the program name and *n* 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 7 on page 41

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 10 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 10. Example of Compile-Time Messages—TRMFLG Option

Figure 11 shows the results of different combinations of the TERMIINAL|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 all messages and statistics at end of listing	X	—	X	—

Figure 11. Results of the TERMINAL|NOTERMINAL and TRMFLG|NOTRMFLG Options

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

Through the FIPS option, you can help ensure that your program conforms to the current FORTRAN standard—American National Standard Programming Language FORTRAN, ANSI X3.9-1978.

You can specify standard language flagging 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 messages are all in the same format as other diagnostic messages described under "Diagnostic Message Listing—FLAG Option" on page 40, and are all at the 4(W) (warning) level.

Using the SAA Flagger—SAA Option

By using the SAA option, you can identify language elements that are not a part of the System Application Architecture. If you specify this option, the compiler issues diagnostic messages at the 4(W) (warning) level for the language elements that do not conform to SAA. For general information about VS FORTRAN diagnostic messages, see "Diagnostic Message Listing—FLAG Option" on page 40. For information about the SAA interface for FORTRAN, see *Systems Application Architecture Common Programming Interface: FORTRAN Reference*, (SC26-4357).

Using the Automatic Precision Increase Facility—AUTODBLOption

The AUTODBL compiler 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 compiler 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.

Precision Conversion Process

The conversion process includes two functions: promotion and padding.

See "Available Compiler Options" on page 23 for the syntax of the AUTODBL option and an explanation of each suboption. Promotion is the process of converting items from one precision to a higher precision; for example, from single precision to double precision. 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.

Promotion

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.

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

- If promotion is requested for the result mode of the specific intrinsic function name being passed, then the promoted function name (if it exists) 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.
- If the AUTODBL option specifies padding only (for a given mode), then the intrinsic function name used as argument is not changed.

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.

See Figure 12 on page 53 and Figure 13 on page 55 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

Integer and logical items (and non-promoted 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.

Effect on 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.

Effect on Initialization with Character Constants

Care should be exercised 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 LANGLVL(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.

Effect on 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)
```

Effect on 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.

Effect of 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.

Effect of Argument Padding on Arrays

When padding is requested with the second position of the AUTODBL option set as either 1 or 3, then all non-promoted 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.

Effect on 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.

Effect on Direct Access Input/Output Processing

When an OPEN statement has been specified (or a DEFINE FILE statement for LANGLVL(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.

Effect on 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.

Effect on 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.

Effect on 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 non-padded 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

The following tables show the promotion of single and double precision intrinsic functions for LANGLVL(77) and LANGLVL(66).

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)

Figure 12 (Part 1 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(77)

Generic Name	Single Precision Function	Corresponding Double Precision Function	Corresponding Extended Precision Function
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)
Note 5	FLOAT (REAL*4)	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)
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 Note 8	AMAX1 (REAL*4) AMAX0 (REAL*4) MAX1 (REAL*4)	DMAX1 (REAL*8) Note 8 (REAL*8) Note 9 (REAL*8)	QMAX1 (REAL*16)
MIN Note 10	AMIN1 (REAL*4) AMIN0 (REAL*4) MIN1 (REAL*4)	DMIN1 (REAL*8) Note 10 (REAL*8) Note 11 (REAL*8)	QMIN1 (REAL*16)

Figure 12 (Part 2 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(77)

Notes for Figure 12 follow Figure 13 because some notes apply to both figures.

Generic Name	Single Precision Function	Corresponding Double Precision Function	Corresponding Extended Precision Function
LOG Note 12	ALOG (REAL*4) CLOG (COMPLEX*8)	DLOG (REAL*8) CDLOG (COMPLEX*16)	QLOG (REAL*16) CQLOG (COMPLEX*32)
LOG10 Note 13	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 14	ARSIN (REAL*4)	DARSIN (REAL*8)	QARSIN (REAL*16)
ACOS Note 15	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 1
LGAMMA Note 16	ALGAMA (REAL*4)	DLGAMA (REAL*8)	Note 1
INT	INT (REAL*4) IFIX (REAL*4) HFIX (REAL*4)	IDINT (REAL*8) IDINT (REAL*8) Note 4 (REAL*8)	IQINT (REAL*16)
Note 5	FLOAT (REAL*4)	DFLOAT (REAL*8)	QFLOAT (REAL*16)
REAL	REAL (COMPLEX*8)	DREAL (COMPLEX*16)	QREAL (COMPLEX*32)
SNGL	Note 18	SNGL (REAL*8)	SNGLQ (REAL*16)
DBLE	DBLE (REAL*4)	Note 2 (REAL*8)	DBLEQ (REAL*16)
QEXT	QEXT (REAL*4)	QEXTD (REAL*8)	Note 6 (REAL*16)
CMPLX	CMPLX (REAL*4)	DCMPLX (REAL*8)	QCMPLX (REAL*16)
IMAG Note 17	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)
SIGN	SIGN (REAL*4)	DSIGN (REAL*8)	QSIGN (REAL*16)
DIM	DIM (REAL*4)	DDIM (REAL*8)	QDIM (REAL*16)
MAX Note 8	AMAX1 (REAL*4) AMAX0 (REAL*4) MAX1 (REAL*4)	DMAX1 (REAL*8) Note 8 (REAL*8) Note 9 (REAL*8	QMAX1 (REAL*16)

Figure 13 (Part 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
MIN Note 10	AMIN1 (REAL*4) AMINO (REAL*4) MIN1 (REAL*4)	DMIN1 (REAL*8) Note 10 (REAL*8) Note 11 (REAL*8)	QMIN1 (REAL*16)

Figure 13 (Part 2 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(66)

Notes to Figure 12 and Figure 13:

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. The alternate function names are chosen depending upon the mode of the function result (listed in this table).
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.
12. LOG is also the specific name of the single-precision function (corresponding to ALOG).
13. LOG10 is also the specific name of the single-precision function (corresponding to ALOG10).
14. ASIN is also the specific name of the single-precision function (corresponding to ARSIN).

Operation	Operand
LIBRARY	*ddname* [(*member-name* [,*member-name*],...)] [,*ddname*[(*member-name* [,*member-name*],...)]]

ddname	indicates the name of a DD statement specifying a library.
member-name	indicates the name of a member of the library.

Linkage Editor Data Sets

The linkage editor generally uses five system data sets; others may be necessary if secondary input is specified. Secondary input is defined by the programmer; cataloged procedures do not supply the secondary input DD statements.

Figure 20 on page 80 lists the function, device types, and allowable device classes for each linkage editor data set.

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

Figure 20. Linkage Editor Data Sets

Note to Figure 20:

1 The **Defined** column indicates whether or not the ddname is defined in cataloged procedures.

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.

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 options you can use, and the loader data sets, are described in the following paragraphs.

Using Linkage Editor Control Statements: You can use the INCLUDE and LIBRARY linkage editor control statements as follows:

> **INCLUDE**—used to specify additional object modules you want included in the output load module.
>
> **LIBRARY**—used to specify additional libraries to be searched for object modules to be included in the load module.

Linkage Editor Control Statements

Linkage editor control statements specify an operation and 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.

INCLUDE Linkage Editor Control Statement: The INCLUDE statement specifies additional programs to be included as part of the 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.

LIBRARY Linkage Editor Control Statement: The LIBRARY statement specifies additional libraries to be searched for object modules to be included in the load module.

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.

VS FORTRAN Version 2 supplies you with cataloged procedures that let you link-edit or load your programs easily. For details, see "Running the Load Module" on page 83.

Using the Linkage Editor

When you use the linkage editor rather than the loader, you have many processing options and optional data sets you can use, depending on the link-edit processing you want done.

Linkage Editor Processing Options: 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 7, "Associating Data" on page 193.)

SIZE—specifies the amount of virtual storage to be used for this link-edit job.

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 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, depending on what you want the linkage editor to do for you, you can, optionally, specify the following data set:

SYSTERM—used for writing error messages. This data set can be on a direct access, magnetic tape, or printer device.

```
//RECOMP  EXEC VSF2CL,
//              PGMLIB='mypds.load',PGMNAME=mylmod
//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(mylmod)
/*
```

In the example, 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.

Running a Link-Edit

You can use two different programs to perform the link-edit: the linkage editor or the loader. Which you use depends upon the output you want produced.

Linkage Editor: Use the linkage editor when you want to reduce storage requirements through overlays, or to use additional libraries as input, or to define the structural segments of the program.

Loader: Use the loader when your input is a small object module that doesn't require overlay, that doesn't require additional linkage editor control statements, and that you'll be running immediately.

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 that you can use to replace all of the run-time library modules in your load modules. You should replace these library modules in this manner whenever your input to the linkage editor is an existing load module containing these 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:

The following example 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 into your load module the corrective service that has been applied to these library modules in your product data sets. In this example, "mypds.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(mylmod)
 NAME mylmod(R)
/*
```

In the example above, only SYS1.VSF2FORT is provided in the SYSLIB DD statement; therefore, running the resulting load module is in load mode.

The following example 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 "mylmod," and "mysrce" is the name of the data set that contains the FORTRAN source program that you want to recompile.

Link-Editing Your Program

You must link-edit any object module before you can run your program, combining this object module with others to construct an executable load module.

Note: FORTRAN 66 object programs are link-edited exactly the same as FORTRAN 77 object programs.

Your input to the linkage editor 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 B, "Object Module Records and Statement Table" on page 335.)

The following sections discuss migration of load modules from previous releases and how to run a link-edit.

Migration of VS FORTRAN Load Modules

The load modules that contain your FORTRAN programs nearly always contain a number of VS FORTRAN Version 2 run-time library modules as well. These library modules are subject to the following restrictions:

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.

Usually when you link-edit a load module, your primary linkage editor input 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 two requirements listed above.

Sometimes 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:

1. You need to recompile some, but not all, of your own FORTRAN routines that are within one of your load modules.
2. You have 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.
3. You want 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 cause either or both of the two requirements listed above to be violated.

Selecting Link Mode or Load Mode

During installation of the VS FORTRAN Version 2 library, your system programmer may have specified the libraries to be used in link mode. (All procedures provided with the product are set up for load mode.) A single environment may have been established for all users, or the selection of load mode or link mode left up to the individual user. The procedures for specifying libraries in link mode or load mode are described below. If you have compiled RENT and separated your text and have link-edited the nonreentrant part for link mode operation, the reentrant parts modules must still be loaded (from LPA or a PDS).

Specifying Libraries in Load Mode

- For operation in load mode, provide VSF2FORT but *not* VSF2LINK to the linkage editor to use when including VS FORTRAN Version 2 library modules. Specify only SYS1.VSF2FORT in the DD statement for SYSLIB in the linkage editor step:

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

- To run a program link-edited in load mode, make VSF2LOAD available for the processing step by performing one of the following steps.

  1. Place the following JOBLIB DD statement in the JCL for the job which runs the VS FORTRAN Version 2 program:

     ```
     //JOBLIB   DD  DSN=SYS1.VSF2LOAD,DISP=SHR
     ```

  2. Or, place one of the following DD statements in the JCL for the step which runs the VS FORTRAN Version 2 program:

     ```
     //STEPLIB  DD  DSN=SYS1.VSF2LOAD,DISP=SHR
     //         DD  DSN=MY.USERLIB,DISP=SHR
     ```

     or

     ```
     //FORTLIB  DD  DSN=SYS1.VSF2LOAD,DISP=SHR
     //         DD  DSN=MY.USERLIB,DISP=SHR
     ```

     (If both FORTLIB and STEBLIB are specified, FORTLIB is searched first.)

     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 *OS/VS2 MVS Supervisor Services and Macro Instructions* (GC28-1114), or *MVS/Extended Architecture System Programming Library: Supervisor Services and Macro Instructions* (GC28-1154), in the discussion of program management.)

Specifying Libraries in Link Mode

- For operation in link mode, concatenate VSF2LINK ahead of VSF2FORT for use by the linkage editor when it includes VS FORTRAN Version 2 library modules. Specify both VSF2LINK and VSF2FORT in the DD statement for SYSLIB in the linkage editor step:

  ```
  //SYSLIB  DD  DSN=SYS1.VSF2LINK,DISP=SHR
  //        DD  DSN=SYS1.VSF2FORT,DISP=SHR
  ```

- A program link-edited in link mode does not require any VS FORTRAN Version 2 libraries at run time.

If you are using shareable modules, you will have to access them with either a FORTLIB DD statement or a STEPLIB DD statement.

If the nonshareable parts of your program run in 31-bit addressing mode, you can recreate the shareable modules under VM/XA 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, you must assign an RMODE value of 24 or an ORIGIN address below the 16-megabyte line.

Running Your Program Under MVS

The following sections discuss:

- Loading library modules at run time
- Link-editing your program
- Using partitioned data sets
- Considerations for sequential files
- Considerations for direct files
- Considerations for I/O files
- Cataloging and overlaying programs
- Considerations for MVS/XA

Loading Library Modules at Run Time

Before you can run a program you must link-edit your object module with the required library modules to create a load module. You can link-edit all library modules except the mathematical routines using either of the following modes:

Mode	**Definition**
Link	Include all of the code for specified service subroutines in your executable module.
Load	Include only the pointers for the required service subroutines in your executable module. At run time, the code for the specified library modules is loaded and run.

Running in load mode:

- Reduces the amount of auxiliary storage required for the load module
- Expedites the compile-link-go process
- In an MVS/XA environment, allows some service subroutines to be placed in the extended link pack area

Running Load Modules Created in Version 1, Releases 1 and 1.1

Any load module originally link-edited under VS FORTRAN Version 1, Releases 1 or 1.1 must be link-edited again using the current release library. The older versions referred to the reentrant I/O service subroutines by the name IFYVRENT; newer versions of VS FORTRAN refer to these routines under the name AFBVRENT.

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 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 continguous 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.

Compatibility with Programs Compiled Under Earlier Releases of VS FORTRAN and CMS

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 nonrelocatable 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.

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.

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.

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 17. Default Values for AMODE on the LOAD 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

Figure 18. Default Values for AMODE on the LKED Command

Overriding the Compile-Time Value for RMODE: To override the RMODE value assigned to your program at compile-time by VS FORTRAN, you can specify:

- The RMODE or ORIGIN option on the LOAD command
- The RMODE option on the LKED command
- The RMODE option on the GENMOD command

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

Note: 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.

On the RMODE option, you can specify 24 or ANY, as shown in Figure 19.

RMODE Value	Program Residence
RMODE 24	Below the 16-megabyte line
RMODE ANY	► Below the 16-megabyte line in a 370-mode virtual machine ► Above the 16-megabyte line in an XA-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)

Figure 19. RMODE Values

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

Figure 15 on page 70 shows the values you can specify for the AMODE option.

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

Figure 15. AMODE Values

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

```
LOAD MYPROG ...(AMODE 31 other options...

GENMOD MYPROG ...(AMODE 31 other options...

LKED MYPROG (LIBE libname NAME membname (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 what RMODE you specify on the GENMOD command, and what AMODE, if any, you specified on the LOAD command, as shown in Figure 16.

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

Figure 16. Default Values for AMODE on the GENMOD 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 what you specify on the RMODE or ORIGIN option of the LOAD or the RMODE option of the LKED command, and what was assigned at compile-time by VS FORTRAN, as shown in Figure 17 on page 71 and Figure 18 on page 71.

Creating New Programs

When you compile your program, the VS FORTRAN Version 2 compiler assigns it an AMODE value of ANY, which means your program can run in either 24-bit or 31-bit addressing mode. It also assigns your program 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 15 shows the 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 routines 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.

Using Shareable Load Modules: For information on using shareable load modules in a VM/XA environment, see "Special Considerations for VM/XA" on page 281.

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 and Running It" on page 60).

For Printer Files:

```
FILEDEF FTxxF001 PRINTER [(options]
     or
FILEDEF  xx  PRINTER  [(options]
     or
FILEDEF  fn  PRINTER  [(options]
```

You specify the FTxxF001 field to agree with the FORTRAN unit numbers in the source program:

- For the xx field, see Figure 22 on page 82.
- 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 the type of unit record file you're processing.

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 "Dynamically Allocating Files" on page 165.

For information on defaults for LRECL, BLKSIZE, and RECFM, see Appendix H, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 445.

VM/XA Considerations

The extended architecture support described in this section is available under VM/XA System Product with bimodal CMS. Under VM/XA, you can create VS FORTRAN programs in either a 370-mode or XA-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 virtual machine.

Note that in an XA-mode machine, VS FORTRAN 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 section "Creating New Programs" on page 69 concern creating new programs under VM/XA with VS FORTRAN Version 2. "Compatibility with Programs Compiled Under Earlier Releases of VS FORTRAN and CMS" on page 72 discusses existing programs compiled under earlier releases of VS FORTRAN and CMS.

You specify the FTxxFyyy field to agree with the FORTRAN unit numbers in the source program:

- For the xx field, see Figure 22 on page 82.
- For the yyy field, specify 001 if you are not using multiple files. If you are using multiple files, you can specify 001 through 999.
- For the n field, you specify any valid tape unit (1 through 4).
- 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 tape files.

Defining Terminal Files: To define terminal files, you specify the FILEDEF command as follows:

```
FILEDEF FTxxF001 TERMINAL [(options]
     or
FILEDEF  xx  TERMINAL  [(options]
     or
FILEDEF  fn  TERMINAL  [(options]
```

You specify the FTxxF001 field to agree with the FORTRAN unit numbers in the source program.

- For the xx field, see Figure 22 on page 82.
- 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.

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 FTxxF001 READER [(options]
    or
FILEDEF  xx  READER  [(options]
    or
FILEDEF  fn  READER  [(options]
```

For Card Punch Files:

```
FILEDEF FTxxF001 PUNCH [(options]
    or
FILEDEF  xx  PUNCH  [(options]
    or
FILEDEF  fn  PUNCH  [(options]
```

For a dynamically allocated file, no FILEDEF is necessary. For more information on dynamic allocation, see "Dynamically Allocating Files" on page 165.

To define sequential and direct files on disk, specify the FILEDEF command as follows:

```
FILEDEF FTxxFyyy DISK filename filetype [filemode] [(options]
       or
FILEDEF  xx  DISK filename filetype [filemode] [(options]
  if yyy = 001
```

You specify the FTxxFyyy field to agree with the FORTRAN unit numbers in the source program.

- For the xx field, see Figure 22 on page 82.
- For the yyy field, specify 001 if you're not using multiple files. If you are using multiple files, 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.

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 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 UPDATE-IN-PLACE attribute is available with CMS, Release 3 or later.

The options are any FILEDEF options valid for disk files. In particular, the maximum LRECL and BLKSIZE that can be specified is 32760. See "Defaults for the XTENT, LRECL, BLKSIZE, and RECFM Options" on page 68 for information on defaults.

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 user's CMS directory.

Defining Tape Files: To define tape files, you specify the FILEDEF command as follows:

```
FILEDEF FTxxFyyy TAPn [(options]
     or
FILEDEF  xx  TAPn  [(options]  if yyy = 001
     or
FILEDEF  fn  TAPn  [(options]
```

Relating Physical Files to FORTRAN I/O Files

Running a VS FORTRAN program in VM may require reading and writing several types of files. Chapter 6, "Performing Input/Output Operations" on page 121 explains fully how to use VS FORTRAN I/O statements to process input/output files. The discussion in this section is limited specifically to relating physical files to FORTRAN input/output files under VM.

Files Preconnected to the Standard Input/Output Units

Before you can read or write a FORTRAN file, the file must be associated with—that is, connected to— a unit. "Connecting Files" on page 147 contains a general explanation of how files get connected to units.

Certain files are already defined and connected when the program begins to run. These files are referred to as being preconnected. A subset of preconnected files consists of the files that are read from the "standard input unit" or written to the "standard output unit." "Preconnecting Files" on page 147 discusses the preconnected files associated with the standard system input/output units.

Each standard input/output unit has a fixed unit number. Unless they have been changed when VS FORTRAN was installed at your site, unit numbers 5, 6, and 7 are the default standard input/output unit numbers.

Figure 14 shows how each of the standard input/output units is used.

Standard I/O Unit	Used For:	IBM-Supplied Unit Number	VS FORTRAN FILEDEF Refers To:	Default Record Format and Length
reader	READ (*, ...)	5	TERMINAL	RECFM F BLOCK 80
printer	WRITE (*, ...) PRINT (*, ...) [VS FORTRAN Error Messages]	6	TERMINAL	RECFM UA BLOCK 133
punch	PUNCH (*, ...)	7	PUNCH	RECFM F BLOCK 80

Figure 14. VS FORTRAN Standard Input/Output Units

VS FORTRAN Version 2 supplies the default file characteristics as shown in Figure 14. However, you can provide your own FILEDEF command for any of the standard input/output units. The FILEDEF command you provide is used instead of the default one.

FILEDEF Commands

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.

If you do not use a FILEDEF command, the **default** filename, filetype, and filemode for unit number xx are:

```
FILE FTxxF001 A1
```

where xx is the unit number.

Note: If FILE=*fn* is specified in the OPEN statement and no FILEDEF has been issued, the default filetype is *fn* instead of FTxxF001.

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

The LIBRARY statement has the following form:

```
LIBRARY tlibdef(ename, ...)
```

This causes the library referred to by the ddname *tlibdef* to be searched for the members listed as *ename* if the subprograms of those names are not already included in the load module either from the TEXT file input to the LKED command, or by having been 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 tlibdef DISK tlibname TXTLIB fm
FILEDEF textdef DISK textname TEXT fm
```

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

```
GLOBAL LOADLIB VSF2LOAD libname
```

where *libname* is the filename of the CMS LOADLIB into which your load module was placed as a member by the LKED command. You must also issue the following GLOBAL command if the simulation of extended precision (REAL*16 or COMPLEX*32) floating-point instructions is required on a machine that does not have these instructions:

```
GLOBAL TXTLIB CMSLIB
```

Issue the following OSRUN command to run your program:

```
OSRUN membname
```

where *membname* is the name of the member that contains the load module created with the LKED command.

Specifying Run-Time Options

The available run-time options are listed in "Available Run-Time Options" on page 101. You can specify a run-time option as follows:

- When running your program that was created with a LOAD command:

  ```
  START * option [option ...]
  ```

- When running a program that was created with a GENMOD command:

  ```
  modname option [option ...]
  ```

  where modname is the name of your VS FORTRAN Version 2 program, and option is one of the run-time options.

- When running a program that is stored as a member of a CMS LOADLIB:

  ```
  OSRUN membname PARM='option[,option...]'
  ```

If your program runs in load mode, issue the following command:

```
GLOBAL LOADLIB VSF2LOAD
```

To run your program that is stored as a nonrelocatable (MODULE) file, issue the following command:

```
modname
```

where *modname* is the filename of your MODULE file as specified in the GENMOD command.

Using the LKED Command

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

Using LKED for a program to be run in 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.

Using LKED for a program to be run in 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 COMMAND, issue the LKED command:

```
LKED myprog (LIBE libname NAME membname
```

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.
membname	is the name of the member in the LOADLIB file designated by *libname*, above, into which the resulting load module is to be placed.

If your program calls subprograms with object code stored as a separate TEXT file or as a member of a text library, your TEXT file, which is the input to the LKED command, must contain linkage editor INCLUDE or LIBRARY statements that specify the locations of the object code for these subprograms. The INCLUDE statement has two forms:

```
INCLUDE tlibdef(mname,...)
INCLUDE textdef
```

Using the LOAD, INCLUDE, and GENMOD Commands

Use a series of LOAD, INCLUDE, and GENMOD commands to create an executable program that is stored as a nonrelocatable (MODULE) file on your CMS disk. 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. First, you must provide access to the appropriate VS FORTRAN Version 2 Library text libraries, as well as to your own text libraries by means of a GLOBAL TXTLIB command. When you run a program in link mode, use one of these forms of the GLOBAL TXTLIB command:

- If VSF2LINK and VSF2FORT are 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 library, you do not need to specify VSF2FORT in the GLOBAL TXTLIB command.

  You can use the following coding:

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

When you are running a program in load mode, use the following form of the GLOBAL TXTLIB command:

```
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.

Next, you must create a temporary copy of your executable program in virtual storage. To do this, issue one LOAD command followed optionally by one or more INCLUDE commands as follows:

```
LOAD MYPROG ...
INCLUDE subprog ...
```

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. You should 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.

To create the nonrelocatable (MODULE) file on your CMS disk, issue the following GENMOD command:

```
GENMOD modname
```

This command builds a file with a filename of *modname* and a filetype of MODULE. This program may be run at any time.

You may be required to issue one or more GLOBAL commands prior to running your program. You must issue the following command if the simulation of extended precision (REAL*16 or COMPLEX*32) floating-point instructions is required on a machine that does not have these instructions:

```
GLOBAL TXTLIB CMSLIB
```

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

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

  ```
  GLOBAL TXTLIB VSF2LINK VSF2FORT CMSLIB 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 coding:

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

If you are running a program in load mode, use this form of the GLOBAL TXTLIB statement:

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

The text library CMSLIB is part of the VM/SP product; you need to specify it only if the simulation of extended precision (REAL*16 or COMPLEX*32) floating-point instructions is required on a machine that does not have these instructions. 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.

In order 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:

```
LOAD MYPROG ...
INCLUDE subprog ...
```

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. You should 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.

Before running the temporary copy of your executable program, you must issue the following GLOBAL command if your program is to run in load mode:

```
GLOBAL LOADLIB VSF2LOAD
```

For your convenience, you may issue this GLOBAL command prior to issuing the LOAD command.

To run the temporary copy of your program that has been built in virtual storage, issue the following START command:

```
START *
```

making the appropriate combination of libraries available when you create your executable program from your TEXT files.

Creating an Executable Program and Running It

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

1. By using the LOAD, and possibly INCLUDE, commands to produce an executable program within virtual storage. You run the program using the START command. No permanent copy of the executable program is made. Processing can be in link mode or load mode.
2. By using the LOAD, possibly the INCLUDE, and the GENMOD commands to build an executable program that is stored as a nonrelocatable (MODULE) file on a CMS disk. You may run the program later by invoking the file name of the MODULE file as a command. Processing can be in link mode or load mode.
3. By using the LKED command to create—that is, to link-edit—an executable program that is stored as a load module in a member of a CMS LOADLIB. You may run the program later by using the OSRUN command.

The following paragraphs show how to use each of these three methods for creating executable programs and running them. In order for you to do this, your system programmer must have made the following libraries available to you:

- 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 self-contained library, or you may have to use it in conjunction with VSF2FORT.
- 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 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.

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.

Using the LOAD, INCLUDE, and START Commands

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. You must first provide access to the appropriate VS FORTRAN Version 2 text libraries as well as your own text libraries, available by means of a GLOBAL command.

Chapter 4. Running Your Program

When you run the load module, you can run it directly as output from the link-edit (or loader) step, or specify that it be called from a library of load modules.

When you run a load module, you may need many different files. For information about these files, see the appropriate section that explains considerations for your particular operating system:

- ► If you are a VM user, begin with the section that immediately follows.
- ► If you are an MVS batch user, skip to page 73.
- ► If you are a TSO user, skip to page 93.

Running Your Program Under VM

Running your program in a CMS virtual machine is done in CMS's OS simulation mode; that is, the VS FORTRAN Version 2 run-time service subroutines use the MVS services that are simulated by CMS. Because of this, 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.

| Programs with functions unique to MVS, such as Asynchronous I/O, Data-In-
| Virtual, or MTF cannot run under VM, nor can programs with MVS data set names in I/O statements.

The following sections discuss:

- ► Selecting load mode or link mode
- ► Creating an executable program and running it
- ► Specifying run-time options
- ► Relating physical files to FORTRAN I/O files
- | ► Considerations for VM/XA

Selecting Load Mode or Link Mode

All library modules, other than the mathematical routines, can be either included as part of your executable program along with the compiler-generated code, or loaded dynamically when your program is run. 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.

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

15. ACOS is also the specific name of the single-precision function (corresponding to ARCOS).

16. LGAMMA is also the specific name of the single-precision function (corresponding to ALGAMA).

17. IMAG is also the specific name of the single-precision function (corresponding to AIMAG).

18. There is no intrinsic function for LANGLVL(66) for a REAL*4 argument.

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—lets you specify the name of the entry point of the program being loaded.

PRINT|NOPRINT—specifies whether or not loader messages are to be listed in the data set defined by the SYSLOUT DD statement.

RES|NORES—specifies whether or not the link pack area is to be searched to resolve external references.

SIZE—specifies the amount of storage to be allocated for loader processing; this size includes the size of your load module.

Loader Data Sets: 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 21. This figure lists the function, device types, and allowable device classes for each data set.

ddname	Function	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
FT07Fyyy	Punched output data	Card punch	B	Yes
FTxxFyyy[2]	User-defined data set	Unit record Magnetic tape Direct access	SYSSQ A,B SYSDA	No

Figure 21. Loader Data Sets

Notes to Figure 21:

[1] The **Defined** column indicates whether or not the ddname is defined in cataloged procedures.

[2] xx is the unit number (00 through 99), and
yyy is the file sequence number (001 through 999).

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 22 on page 82 lists the function and device types for each data set.

FORTRAN Reference Number	ddname	Function	Device Type
5	FT05Fyyy	Input data set to load module	Card reader Magnetic tape Direct access
6	FT06F001	Printed output data	Printer Magnetic tape Direct access
7	FT07F001	Punched output data	Card punch Magnetic tape Direct access
0-4 8-99	FTxxFyyy	Sequential data set	Unit record Magnetic tape Direct access
0-4 8-99	FTxxFyyy	Direct access data set	Direct access
0-4 8-99	FTxxFyyy	Partitioned data set member using sequential access	Direct access

Figure 22. Load Module Execution Data Sets

DCB Default Values:

Sequential Data Sets: Figure 23 lists the DCB default values for load module execution **sequential** data sets. These default values also apply to dummy data sets.

ddname	RECFM[1]	LRECL[2]	BLKSIZE	DEN	BUFNO
FT05Fyyy	F	80	80	–	2
FT06Fyyy	UA	133	133	–	2
FT07Fyyy	F	80	80	–	2
all others	U	–	800	2	2

Figure 23. Load Module Execution Sequential Data Set DCB Default Values

Notes to Figure 23:

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.

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.

The following sections describe the data sets you may need, and outline the job control language you must use to run your programs.

Providing Access to the VS FORTRAN Version 2 Library at Run Time: A program to be run under MVS 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.

Operating in Load Mode:

You can run an executable module in load mode using any of the following DD statements:

JOBLIB DD Statement: To provide access to the VS FORTRAN Version 2 library for all job steps, place a JOBLIB DD statement for the load module immediately after the JOB statement.

```
//JOBLIB  DD DSN=SYS1.VSF2LOAD,DISP=SHR
```

STEPLIB DD Statement: To provide access to the VS FORTRAN Version 2 library for a single job step, include a STEPLIB DD statement in the DD statements for that job step.

```
//STEPLIB DD DSN=SYS1.VSF2LOAD,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.

FORTLIB DD Statement: 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
```

Specifying Run-Time Options: A complete list of the available run-time options is in "Available Run-Time Options" on page 101. To specify a run-time option, use the following method:

```
//GO  EXEC PGM=MAIN,PARM='option[,option...]'
```

Running the Load Module: How you run the load module depends on the kind of job you're running: run only, link-edit and run, or compile link-edit and run.

IBM-supplied cataloged procedures are available that let you compile, link-edit or load, and/or run easily. A list of all the nonreentrant cataloged procedures is given in Figure 24 on page 84. The cataloged procedures should be located in your appropriate system procedure library.

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

Figure 24. IBM-Supplied Non-reentrant Cataloged Procedures

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
```

Information on interpreting dumps is found in the appropriate debugging guide for your system.

Using Partitioned Data Sets

A partitioned data set (PDS) consists of groups of sequential data called members of the data set. 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 would contain the output data corresponding to one set of input data.

Partitioned data set members can be created, retrieved, and rewritten with VS FORTRAN Version 2, using I/O statements for formatted sequential access files.

Creating Members of a New PDS: For formatted sequential access files, the WRITE statement and the REWIND or CLOSE statements create PDS members. The FORTRAN program must handle each member written as if it were a separate sequential file. After a member is written, a CLOSE or REWIND statement must be specified for the unit representing the member before another member is written. This closes the PDS after each member is created so that the end-of-file (EOF) record is supplied correctly for that member. A different DD statement with a different unit (FORTRAN reference number) is required for each member created in the same program.

Retrieving Members from an Existing PDS: For formatted sequential access files, the READ statement with the END= parameter retrieves multiple members of a PDS under one unit number, when the PDS is referenced only for input. The end-of-data transfer specified by the END= statement label increases the file sequence number. Thus, members can be read, one-by-one, as if they were sequential tape files. A separate DD statement is required for each member being read with the appropriate file sequence number. Also, specify the LABEL parameter and its subparameter IN in the DD statement when reading members as described in the following material.

Rewriting Members of an Existing PDS: Existing members of a PDS can be rewritten, or new members can be written in an existing PDS, by using the method described in "Creating Members of a New PDS." Members can be read, written, and/or rewritten in the same FORTRAN program unit, if:

- The PDS is closed between references to different members by either a CLOSE or a REWIND statement.
- Each PDS member is represented by a different unit and DD statement.

Processing Mode and PDS Input/Output: The JCL parameter LABEL and its subparameters IN and OUT may be used to preset the processing mode to INPUT or OUTPUT in the DD statements for a PDS. This usage is recommended because it enforces correct PDS member handling. As described in "Creating Members of a New PDS" on page 84, correct handling occurs when each PDS member is fully processed and its unit closed before another member is opened and processed.

When the JCL does not preset the processing mode, VS FORTRAN Version 2 assumes:

- INOUT, if the statement that opened the data set is a READ
- OUTIN, if the statement that opened the data set is a WRITE

Input/Output—System Considerations

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 "Job Processing" on page 12.

For additional detail on magnetic tape label processing, see *OS/VS Tape Labels* (GC26-3795).

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 appropriate Data Management Services Guide.

Defining FORTRAN Records

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 168. VS FORTRAN also supplies defaults for I/O data sets, as described under "Installation Defaults for I/O Data Set Characteristics" on page 88.

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:

```
◄──────────────────────100 bytes──────────────────────►

◄──────────────────────block 1────────────────────────►
|   |◄────record 1────►|◄────record 2────►|                 |
|BDW|SDW|  (26 bytes)  |SDW|  (26 bytes)  |(36 unused bytes)|

◄──────────────────────block 2────────────────────────►
|   |◄────record 3────►|                                    |
|BDW|SDW|  (26 bytes)  |          (66 unused bytes)         |
```

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 contain double-byte character data) as formatted or unformatted; that is, you may or may not define them in a FORMAT statement. List-directed I/O statements are considered formatted.

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.

If you're processing records using asynchronous input/output, the records must be variable spanned and unblocked.

Use blocked records wherever possible; blocked records reduce processing time substantially.

Sequential ISCII/ASCII Data Sets: ISCII/ASCII data sets may have sequential organization only. For system considerations, see the documentation for the system you're using.

FORTRAN records in an ISCII/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 characterstics 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 Appendix H, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 445.

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.

Keep in mind that, although overlays reduce storage, they also can drastically increase program run time. In other words, you probably shouldn't use overlays unless they're absolutely necessary. In addition, modules compiled with the RENT compiler option are not executable in MVS as overlays.

The SAVE statement has no effect on overlaid programs. That is, when a program is overlaid by another, variable values in the overlaid program become undetermined.

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.

The linkage-editor control statements you use to process an overlay load module in OS are:

- OVERLAY linkage-editor control statement—which indicates the beginning of an overlay segment and gives the symbolic name of the relative origin.

  OVERLAY control statements are followed by object decks, INSERT control statements, or INCLUDE control statements.

- INSERT linkage-editor control statement—which positions previously compiled routines, when the object decks are not available, within the overlay structure.

  The INSERT control statement gives the names of one or more control sections (CSECTs) that are to be inserted.

  To place the control section in the root segment, position the INSERT control statement before the first OVERLAY control statement.

- INCLUDE linkage-editor control statement—which includes control sections from libraries, if the control sections reside in partitioned data sets or sequential data sets.

  When you use an INCLUDE control statement in an overlay program, you should position it in the input stream at the point where the control section to be included is required.

  The control sections added by an INCLUDE control statement can be manipulated through use of the INSERT control statement.

- ENTRY linkage-editor control statement—which specifies the first instruction of the program to be run, giving the name of an instruction in the root segment. Usually, that name will be either MAIN or the name you've given it in the PROGRAM statement (if specified).

These control statements appear in the input stream after the //SYSLIN DD statement (or after the //LKED.SYSLIN DD statement if you use a cataloged procedure).

MVS/XA Considerations

Every program that runs under MVS/XA is assigned two new attributes: AMODE (addressing mode) and RMODE (residency 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. Generally, the program is also designed to run only in that mode, 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 is expected to reside in virtual storage: 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 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. These program units can still be used by themselves or link-edited with VS FORTRAN Version 2 subprograms to be run under MVS/XA. 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).

MVS/XA Linkage Editor Attributes

To take advantage of 31-bit addressing, a program must be link-edited by the MVS/XA linkage editor and have no 24-bit addressing dependencies. The MVS/XA linkage editor 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 the following table:

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

FORTRAN and MVS/XA 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 "MVS/XA Linkage Editor Attributes" on page 90.

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 compiler 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 "Loading Library Modules at Run Time" on page 73.

  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/Extended Architecture System Programming Library: Supervisor Services and Macro Instructions* (GC28-1154).

Overriding AMODE/RMODE Attributes

To override the default link-edit attributes, specify AMODE and/or RMODE as follows:

- The linkage editor or loader EXEC statement

  ```
  //LKED  EXEC  PGM=programname,
  //            PARM='AMODE=xxx,RMODE=yyy'
  ```

  For additional detail, see *MVS/Extended Architecture Linkage Editor and Loader* (GC26-4011).

- The linkage editor MODE control statement

`MODE`	`AMODE(xxx),RMODE(yyy)`

  For additional detail, see *MVS/Extended Architecture Linkage Editor and Loader*, GC26-4011.

- 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

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 Blank and Named Common (Static and Dynamic)" on page 200.

In order to use the extra storage available with MVS/XA, 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.

Example:

```
@PROCESS DC(CMN1,
@PROCESS 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. Note the continuation of the DC option across two @PROCESS statements.

Extended Architecture Hints for FORTRAN Users

The following list contains helpful information for VS FORTRAN Version 2 users.

- All modules that perform input and output and all input/output buffers and control blocks must reside below the 16-megabyte line, because Data Management does not support callers in 31-bit addressing mode.
- The VS FORTRAN Version 2 Library run-time I/O routines switch addressing mode when system services are needed. The addressing mode can be switched only in a program residing below the 16-megabyte line.
- The maximum size of a load module is 16 megabytes.
- 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 TSO

To link-edit and run 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), and then use the CALL command to run the load module.

The input object module must be OBJ data sets; for example:

```
userid.name.OBJ
```

The following sections discuss:

- Selecting link mode or load mode
- Link-editing your program

- Running a load module
- Fixing run-time errors
- Using CLISTs

Note: The + at the end of the TSO command lines indicate a continuation on the next line.

Selecting Link Mode or Load Mode

As in MVS, you can run your program in either link mode or load mode. See "Loading Library Modules at Run Time" on page 73.

The following example illustrates sample TSO coding to run a program in link mode. The coding in the first example calls in the standard mathematical routines. In the second example, the coding calls alternative mathematical routines.

Example 1: Using standard mathematical routines

```
LINK myprog LOAD(myprog(T)) PRINT(myprog) LET LIST MAP+
     LIB('SYS1.VSF2LINK','SYS1.VSF2FORT')
ALLOCATE FILE(FT06f001) DA(*)
CALL myprog(T)
```

Example 2: Using alternative mathematical routines

```
LINK myprog LOAD(myprog(A)) PRINT(myprog) LET LIST MAP+
     LIB('SYS1.VSF2MATH','SYS1.VSF2LINK','SYS1.VSF2FORT')
ALLOCATE FILE(FT06f001) DA(*)
CALL myprog(A)
```

When you run a program in load mode, the executable, or load, module does not include the code for the required library modules. The program must provide access to the SYS1.VSF2LOAD modules in one of the following ways:

- The systems programmer can
  - Add SYS1.VSF2LOAD to your system link list.
  - Concatenate SYS1.VS2LOAD with the data sets named in the STEPLIB DD statement in your logon procedure.

  or
- You can
  - Associate the data set SYS1.VSF2LOAD with FORTLIB in an ALLOCATE command.

The following examples show the TSO coding you can use to link-edit and run a program in load mode. The coding in the first example calls in the standard mathematical routines. In the second example, the coding calls alternative mathematical routines.

Example 1: Using standard mathematical routines

```
LINK myprog LOAD(myprog(T)) PRINT(myprog) LET LIST MAP+
     LIB('SYS1.VSF2FORT')
ALLOCATE FILE(FT06F001) DA(*)
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD') SHR
CALL myprog(T)
```

Example 2: Using alternative mathematical routines

```
LINK myprog LOAD(myprog(A)) PRINT(myprog) LET LIST MAP+
     LIB('SYS1.VSF2MATH','SYS1.VSF2FORT')
ALLOCATE FILE(FT06F001) DA(*)
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD') SHR
CALL myprog(A)
```

Link-Editing Your Program—TSO LINK Command

You use the LINK command to create and run 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 load and run the OBJ data sets for *myprog* and its subprogram *subprog*, you specify:

For load mode:

```
LINK (myprog,subprog) LOAD(myprog) LIB('SYS1.VSF2FORT')
```

or

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VSF2MATH','SYS1.VSF2FORT')
```

For link mode:

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VSF2LINK','SYS1.VSF2FORT')
```

or

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VSF2LINK','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 last example, you are, therefore, requesting a search of SYS1.VSF2MATH and SYS1.VSF2FORT.

Linkage Editor Listings—TSO LINK Command

You can also use the LINK command to specify linkage editor options. In the above example, you can request the listings to be printed, either on the system printer or at your terminal:

On the System Printer:

```
LINK (myprog,subprog) LIB('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.

Running a Load Module under TSO

You can run a program under TSO using either of the following commands:

- TSO CALL command—to run the load module.
- TSO LOADGO command—to invoke the loader program to link-edit and run your program in one step

Required 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 following 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 LNKLST00 for the libraries on the list).
- If you are using the IFYVRENT routines, AFBVRENT (with alias of IFYVRENT) must also be in a library on the system link list.
- 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.

  If you are uncertain whether your executable module has access to the correct library modules, see your system programmer.

Using the TSO 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') SHR
CALL myprog
```

After the program has run, you should delete any data sets that you won't be using again. Use the TSO DELETE command to delete unneeded data sets named in the ALLOCATE and CALL commands. See the following command.

```
DELETE (myprog.indata myprog.outdata myprog.workfil)
```

Specifying Run-Time Options: The same run-time options that are available in MVS are available in TSO. "Available Run-Time Options" on page 101 contains a list of these options and describes how to use each option. In TSO, you specify run-time options with the TSO CALL command. This command has the following form:

```
CALL pgmname 'option[,option...]'
```

where *pgmname* is the name of your VS FORTRAN Version 2 program, and *option* is a run-time option.

TSO 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.

Using the System Loader Program to Load and Run a 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.

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. Provide access to the library modules in SYS1.VSF2LOAD using one of the following methods.
   - The system programmer can
     - 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.

     or
   - You can
     - Associate the data set SYS1.VSF2LOAD with FORTLIB in an ALLOCATE command.
3. Run the LOADGO command.

Examples of LOADGO Commands: This section contains examples of the form of the LOADGO command to use when running a program. The examples show the LOADGO command as you can use it to link-edit and run an object module name *myprog*

Running a Program in Link Mode: Example 1:

```
LOADGO (myprog) LIB('SYS1.VSF2LINK','SYS1.VSF2FORT')
```

Example 2:

```
LOADGO (myprog) LIB('SYS1.VSF2MATH','SYS1.VSF2LINK','SYS1.VSF2FORT')
```

Running a Program in Load Mode: Example 1

```
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD')
LOADGO (myprog) LIB('SYS1.VSF2FORT')
```

Example 2:

```
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD')
LOADGO (myprog) LIB('SYS1.VSF2MATH','SYS1.VSF2FORT')
```

In these examples the LIB operand provides access to the required data sets. The loader program resolves any external references in *myprog* and loads the required object modules.

Using LOADGO for a Link-Edited Module: You can also use LOADGO to run a link-edited load module; for example:

```
LOADGO myprog(tempname)
```

Using LOADGO to Specify 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(*)
```

Specifying PRINT(*)causes the loader listings to display on your terminal.

Fixing Run-Time Errors

In TSO you can use all the FORTRAN debugging aids described in Chapter 4, "Running Your Program" on page 59. You can also use the TSO TEST command with its associated subcommands to debug your object program. Using TEST, you can determine exactly where in the program the abnormal termination occurred.

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(*)
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD')SHR
LOADGO &NAME LIB('SYS1.VSF2FORT')
```

CLIST 2:

```
PROC 1 NAME
LINK &NAME LOAD(temp(MAIN)) LIB('SYS1.VSF2FORT') LET MAP
ALLOCATE FILE(FT06F001) DA(*)
ALLOCATE FILE(FORTLIB) DA('SYS1.VSF2LOAD') SHR
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. It may be more desirable to batch process such programs in the background thus freeing your terminal for other tasks. For information about cataloged procedures for background processing, see "Running the Load Module" on page 83.

Chapter 5. Using the Run-Time Options and Identifying Run-Time Errors

This chapter describes the run-time options and explains how to identify run-time errors.

Available Run-Time Options

The following run-time options are available:

The default values for the options described below may be established for a single program or for an entire installation by establishing default options tables. See "Establishing a Default Run-Time Options Table" on page 106 for instructions on how to set up this table for a single program. The installation-wide default options table is set up by your systems programmer. If the IBM-supplied defaults are not overridden, the defaults are those defaults given in the following option descriptions.

If you code conflicting run-time options (for example, STAE and NOSTAE), the last value specified takes precedence.

For more information on using the run-time options, see "Specifying Run-Time Options" on page 64 (VM), "Specifying Run-Time Options" on page 83 (MVS), "Specifying Run-Time Options" on page 97 (TSO), or *VS FORTRAN Version 2 Interactive Debug Guide and Reference.*

ABSDUMP | <u>NOABSDUMP</u>
These options specify whether post-ABEND symbolic dump information will be printed in the event of an abnormal termination.

AUTOTASK(*loadmod,ntasks*) | <u>NOAUTOTASK</u> (MVS only)
These options specify whether the multitasking facility (MTF) is enabled for your program. See Appendix E, "The Multitasking Facility (MTF)" on page 349 for more information on the multitasking facility.

AUTOTASK(*loadmod,ntasks*)

loadmod
is the name of the load module that contains the parallel subroutines, which are to be run in the subtasks that are created by MTF.

ntasks
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.

DEBUG | <u>NODEBUG</u>
These options specify whether VS FORTRAN Version 2 Interactive Debug will be invoked. They can be specified by CMS and TSO users at runtime. For more information, see *VS FORTRAN Version 2 Interactive Debug Guide and Reference.*

DEBUNIT(*s1*[,*s2*,...]) | <u>NODEBUNIT</u> (MVS Format)

DEBUNIT(*s1*[*s2 s3*]) | <u>NODEBUNIT</u> (VM Format)

DEBUNIT allows you to specify 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.

<u>INQPCOPN</u> | NOINQPCOPN

These options control 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.

<u>IOINIT</u> | NOIOINIT

These options specify 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 VM, 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.

<u>OCSTATUS</u> | NOOCSTATUS

These options control 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. Some exceptions follow:

- 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.
- On devices where deletion is not possible, CLOSE with STATUS='DELETE' will close the file as if STATUS='KEEP' had been specified. If the ERR or IOSTAT specifiers are not used, a warning message will be given to indicate that the requested deletion is not allowed.

In addition, when a preconnected file is disconnected by a CLOSE statement, an OPEN statement is required to reestablish the connection 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', the following occurs:

If 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 reestablish the connection to a unit. Before the connection is reestablished, 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.

<u>SPIE</u> | NOSPIE

These options specify 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 running of the SPIE (or ESPIE) macro insstruction. 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
These options specify 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 statement 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 ABSDUMP 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 TSO Test to debug your program, if you use the QUIT command in an attention exit to terminate the program, 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 statement 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

These options specify 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.

NOXUFLOW

suppresses the program interrupt caused by an exponent underflow. The hardware provides the fix-up.

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. However, if the library processes the error, the run time consumed is considerable during interrupt handling. This is true for all occurrences of an exponent underflow even though messages are printed only for the first few occurrences. Therefore, if exponent underflows and the corresponding fix-up are acceptable, you should specify the NOXUFLOW option to save processor time.

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 invokes your program. For MVS, see "Specifying Run-Time Options" on page 83; for VM, see "Specifying Run-Time Options" on page 64; and for TSO, see "Specifying Run-Time Options" on page 97.
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 systems 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, then do 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 VM, place 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 do the assembly as follows:

```
GLOBAL MACLIB VSF2MAC
HASM VLPRM1
```

This produces a text file called VLPRM1 TEXT.

3. Include the object module produced from the assembly of the VSF2PARM macro instruction when you create your executable program.

   If you are running 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)
   ```

   If you are running under VM, 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, see *VS FORTRAN Version 2 Language and Library Reference*.

Using the Optional Traceback Map

Whenever you get a library diagnostic message, you can, optionally, get 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 111.

To cause ISNs to appear in a traceback map, you must have compiled with the GOSTMT or the SDUMP compiler option (see "Available Compiler Options" on page 23).

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 sample traceback map in Figure 25 lists the names of called routines, internal statement numbers (ISNs) within routines, and the arguments received by each subroutine.

```
TRACEBACK OF CALLING ROUTINES; MODULE ENTRY ADDRESS = 020000
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 SUBB (020830) CALLED BY SUBA (0205C8) AT ISN 4 AT OFFSET 0001C2.
  ARGUMENT LIST AT 020760.
    ARG. NO.  ADDRESS     INTEGER      REAL        CHAR   HEXADECIMAL
          1   00020330 :        0  0.000000E+00  '....'   00000000
          2   80020330 :        0  0.000000E+00  '....'   00000000
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 SUBA (0205C8) CALLED BY MAIN (020000) AT ISN 10 AT OFFSET 000442.
  ARGUMENT LIST AT 0201E4.
    ARG. NO.  ADDRESS     INTEGER      REAL        CHAR   HEXADECIMAL
          1   00020330 :        0  0.000000E+00  '....'   00000000
          2   80020330 :        0  0.000000E+00  '....'   00000000
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 MAIN (020000) CALLED BY OPERATING SYSTEM.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 25. Sample Traceback Map

MODULE ENTRY ADDRESS = *address*
shows the entry point of the earliest routine entered.

routine (*address*)
lists the names of all routines entered in the current calling sequence with the routine entry address. In Figure 25, the final routine that ran is SUBB, which begins at hexadecimal address 00020830.

Names are shown with the last routine called at the top and the first routine called at the bottom of the listing.

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 25, SUBA, which began at address 000205C8, called routine SUBB, which began at address 00020830. Calls to the main program from the operating system are indicated by the CALLED BY OPERATING SYSTEM format.

AT ISN *nnnn*
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 or, for some errors, GOSTMT.

OFFSET (*address*)
lists the hexadecimal offset within the routine that made the call.

ARGUMENT LIST AT (*address*)
shows the address of the argument list passed to the called routine or the message, NO ARGUMENT PASSED TO SUBROUTINE.

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`

The control program runs its own routine to recover from the error, and 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.

The summary of errors printed at the end of the listing tells you how many times an error was encountered.

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 6 on page 40. Using the ISN, you can locate the source statement within the calling module.

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 can be found 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 one of the following forms:

yy n

'message'

0

Character	Meaning
yy	message identification number assigned by the system.
n	string of 1 through 5 decimal digits you specified in the PAUSE or STOP statement. For the STOP statement, this number is placed in register 15.
'message'	character constant you specified in the PAUSE or STOP statement.
0	printed when a PAUSE statement containing no characters is 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 subroutine.

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 an 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 following actions take place when an error occurs:

1. The FORTRAN error monitor (ERRMON) receives control.

2. The error monitor prints the necessary diagnostic and informative messages:

   - A short message, along with an error identification number.

     The data in error (or some other associated information) is printed as part of the message text. For more information on run-time messages, see *VS FORTRAN Version 2 Language and Library Reference*.

   - The error count, telling you how many times each error occurred.

   - A traceback map (optional), tracing the subroutine flow back to the main program, after each error occurrence.

3. Then the error monitor takes one of the following actions:

   - Terminates the job.

   - Returns control to the calling routine, which takes a standard corrective action and then continues to run.

   - Calls a user-written closed subroutine, possibly to correct the data in error, and then returns to the routine that detected the error, which then continues to run.

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 107).

- 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 changed in the VS FORTRAN Version 2 library permanently 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. Caution should be used when changing the values of any variables in the common area while in a closed user error handling routine under optimization levels of 1, 2, or 3. Certain control flow and variable usage information are not known to the optimizer, 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 from the optimizer in the context of the program. Therefore, the optimizer assumes that RETCDE is not changed between the READ and the GOTO 1 in the above example. It is kept in a register that causes an incorrect result. If you specify IOSTAT in the READ statement, as follows:

   ```
   1     READ(5,*,END=100,IOSTAT=RETCDE) I
   ```

   the optimizer can optimize it correctly.

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 commmand 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 compiler 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.

Figure 26 shows how you can use VS FORTRAN Version 2 debug statements to obtain the information you specify for your use in determining the cause of an error.

Program Code with Debug Statements:

```
        DEBUG SUBCHK(ARRAY1), TRACE, INIT(ARRAY1)
        AT 10
        TRACE ON
         (procedural code for debugging)
        AT 40
        TRACE OFF
        DISPLAY I, J, K, L, M, N, ARRAY1
        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)
```

How Each Debug Statement Is Used: 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.

AT 10 begins the first debug packet.

TRACE ON turns on program tracing at statement label 10, so that subsequent statements with FORTRAN labels are traced.

(Procedural debugging code contains valid FORTRAN statements to aid in debugging; for example, to initialize variables.)

AT 40 ends the first debug packet and begins the second.

TRACE OFF turns off program tracing at statement label 40, so that subsequent statements after 40 are not traced.

The DISPLAY statement writes the values of I, J, K, L, M, N, and ARRAY1.

END DEBUG ends the second (and last) debug packet.

Figure 26. Using Static Debug Statements

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 27 on page 115. (Some of the information in the listing has been realigned to fit the dimensions of the page.)

If the SDUMP or TEST option has been specified, in addition to the LIST option, the ISN number is printed before each source statement in the pseudo-assembler listing, including statements in vectorized DO loops.

Each line of the listing is formatted (from left to right) as follows:

- The relative address of the instruction or the data item in hexadecimal.
- The next area shows the storage representation of the instruction or initialized data item, in hexadecimal format.
- The next area (not always present) shows names and statement labels, which may be either those appearing in the source program or those generated by the compiler (compiler-generated labels are in the form *nn nnn nnnnnnn*).
- The next area shows the pseudo-assembler language format for each statement.
- The last area shows the source program items referred to by the instruction, such as entry points of subprograms, variable names, or other statement labels.

Figure 27 on page 115 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 27 on page 115; 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.

```
ENTRY CODE
  000000  47F0 F020           GAUSS  BC     15,32(0,15)
  000004  17                         DC     XL1'17'
  000005  C7C1E4E2E240404040         DC     CL9'GAUSS
  00000E  F8F74BF0F7F7               DC     CL6'87.077'
  000014  F1F04BF3F64BF3F4           DC     CL8'10.36.34'
  00001C                             DS     A(PIB)
  000020  90EC D00C                  STM    14,12,12(13)
  000024  184D                       LR     4,13
  000026  98CD F034                  LM     12,13,52(15)
  00002A  5040 D004                  ST     4,4(0,13)
  00002E  50D0 4008                  ST     13,8(0,4)
  000032  07FC                       BR     12
  000034                             DS     A(PROLOG)
  000038                             DS     A(SAVEAREA)
ENTRY TABLE
  00003C                             DS     9F
PROGRAM INFORMATION BLOCK
  000060                             DS     22F
COMPILER PROPERTIES TABLE
  0000C0                             DS     2F
IAD WORK AREA
  0000C8                             DS     4F
SAVE AREA
  0000D8                             DS     18F
REGISTER 12 ADCON
  000120                             DS     A(REG12)
LOCATION OF EPILOG ADDRESS
  000124                             DS     A
IAD WORK AREA
  00012C                             DS     23F
TEMPORARY FOR FIX/FLOAT OR NOT
  000188  0000000000000000           DC     XL8'0000000000000000'
CONSTANTS
  000190  4F08000000000000           DC     XL8'4F08000000000000'
  000198  3C10C6F7A0B5ED8D           DC     XL8'3C10C6F7A0B5ED8D'
  0001A0  4080000000000000           DC     XL8'4080000000000000'
  0001A8  41281B26263AE000           DC     XL8'41281B26263AE000'
   .
   .
   .
   .
ARITHMETIC AND LOGICAL VARIABLES
  0001D0  NO INITIAL DATA      TOTAL_AREA
                                     DS     D
  0001D8  NO INITIAL DATA      X_SQ   DS     D
  0001E0  NO INITIAL DATA      T_HI   DS     D
  0001E8  NO INITIAL DATA      T_LO   DS     D
  0001F0  NO INITIAL DATA      GAUSS  DS     D
  0001F8  NO INITIAL DATA      GUASS  DS     F
ADCONS FOR EXTERNAL REFERENCES
  000208  00000000                   DC     AL4(00000000)       RAN GE_HI
  00020C  00000000                   DC     AL4(00000000)       RAN GE_LO
  000210  00000000                   DC     AL4(00000000)       MU
  000214  00000000                   DC     AL4(00000000)       SIG MA
  000218  00000000                   DC     AL4(00000000)       X
  00021C  00000000                   DC     AL4(00000000)       D#E RF
  000220  00000000                   DC     AL4(00000000)       D#E XP
PROGRAM CODE
*  ISN     5
  00026C  5870 D13C           2.001  L      7,316(0,13)
  000270  6800 7000                  LD     0,0(0,7)        SIGMA
  000274  6900 D0E8                  CD     0,232(0,13)     00000000 0000000
  000278  5850 D17C                  L      5,380(0,13)     7.000#
  00027C  0775                       BCR    7,5
```

Figure 27 (Part 1 of 3). Object Module Listing Example—LIST Compiler Option

```
* ISN     6
  00027E  6800 D0E8           6.000  LD    0,232(0,13)      00000000 00000000
  000282  6000 D118                  STD   0,280(0,13)      GAUSS
  000286  5850 D18C                  L     5,396(0,13)      18.000#
  00028A  07F5                       BCR   15,5
  .
  .
  .
  .
EPILOGUE CODE
  000406  0700                       BCR   0,0
  000408  0700                       BCR   0,0
  00040A  6800 D118                  LD    0,280(0,13)
  00040E  58D0 D004                  L     13,4(0,13)
  000412  58E0 D00C                  L     14,12(0,13)
  000416  92FF D00C                  MVI   12(13),255
  00041A  982C D01C                  LM    2,12,28(13)
  00041E  07FE                       BCR   15,14
PROLOGUE CODE
  000420  987A 1000                  LM    7,10,0(1)
  000424  5070 D140                  ST    7,320(0,13)      X
  000428  5080 D13C                  ST    8,316(0,13)      SIGMA
  00042C  5090 D138                  ST    9,312(0,13)      MU
  000430  50A0 D134                  ST    10,308(0,13)     RANGE_LO
  000434  5870 1010                  L     7,16(0,1)
  000438  5070 D130                  ST    7,304(0,13)      RANGE_HI
  00043C  58F0 D170                  L     15,368(0,13)
  000440  07FF                       BCR   15,15
ADCON FOR PROLOGUE
  000034  00000420                   DC    XL4'00000420'
ADCON FOR SAVE AREA
  000038  000000D8                   DC    XL4'000000D8'
ADCON FOR EPILOGUE
  000124  00000406                   DC    XL4'00000406'
ADCONS FOR PARAMETER LISTS
  0001FC  80000228                   DC    AL4(X'80000228')     .SO 001
  000200  800001E0                   DC    AL4(X'800001E0')     T_H I
  000204  800001E8                   DC    AL4(X'800001E8')     T_L 0
TEMPORARIES AND  GENERATED CONSTANTS
  000224  00000000                   DC    XL4'00000000'
  000228  00000000                   DC    XL4'00000000'
  00022C  00000000                   DC    XL4'00000000'
  000230  00000000                   DC    XL4'00000000'
  000234  00000000                   DC    XL4'00000000'
  000238  00000000                   DC    XL4'00000000'
ADCONS FOR B BLOCK LABELS
  000248  0000026C                   DC    XL4'0000026C'
  00024C  0000026C                   DC    XL4'0000026C'
  000250  0000027E                   DC    XL4'0000027E'
  000254  0000028C                   DC    XL4'0000028C'
  000258  00000350                   DC    XL4'00000350'
  00025C  0000035E                   DC    XL4'0000035E'
  000260  000003EE                   DC    XL4'000003EE'
  000264  000003F2                   DC    XL4'000003F2'
  000268  000003FE                   DC    XL4'000003FE'
PROGRAM CODE TABLE
  000442  04                         DC    XL1'04'
  000443  B9                         DC    XL1'B9'
  000444  B7                         DC    XL1'B7'
  000445  B2                         DC    XL1'B2'
  000446  C018                       DC    XL2'C018'
  000448  C018                       DC    XL2'C018'
  .
  .
  .
  .
```

Figure 27 (Part 2 of 3). Object Module Listing Example—LIST Compiler Option

```
SYMBOL DICTIONARY
  000460  40407BE2E8D4E5E2            DC      CL8' #SYMVS'
  000468  05C7C1E4E2E20000            DC      CL8' GAUSS  '
  000470  10000000                    DC      XL4'10000000'
  000474  00000420                    DC      XL4'00000420'
  .
  .
  .
  .
PROGRAM INFORMATION BLOCK
  000060  7BD7C9C2E3C1C27B            DC      CL8'#PIBTAB#'
  000068  0060                        DC      XL2'0060'
  00006A  0000                        DC      XL2'0000'
  .
  .
  .
ENTRY TABLE LIST
  00003C  00000000                    DC      XL4'00000000'
  000040  C7C1E4E2E2404040            DC      CL8'GAUSS   '
  000048  00000420                    DC      XL4'00000420'
  .
  .
  .
COMPILER PROPERTIES TABLE
  0000C0  1000000000000000            DC      XL8'1000000000000000'
```

Figure 27 (Part 3 of 3). Object Module Listing Example—LIST Compiler Option

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 User 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 inadvertantly destroy data and instructions.
3. Moving data into an item that's too small for it, resulting in truncation.
4. Making invalid data references to equivalenced 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 call the entry point VFEIN# or VFEIL# when your main program is not a FORTRAN program. See Appendix A, "Assembler Language Considerations" on page 319.
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 XREF compiler option and Intercompilation Analysis (ICA) are VS FORTRAN features that can help you identify many coding errors. See "Using the MAP and XREF Options" on page 42 and "Intercompilation Analysis" on page 201.

Part 3. Advanced Coding Topics

Chapter 6. Performing Input/Output Operations

This chapter explains how to perform input/output (I/O) operations in VS FORTRAN. It is divided into the following main sections:

"Concepts and Terminology" explains concepts and terms unique to FORTRAN input/output, as well as several operating system terms.

"Overview of Input/Output Statements" on page 129 gives an overview of the VS FORTRAN statements used for I/O.

"Reading and Writing Data" on page 130 discusses how to read and write formatted and unformatted data.

"Connecting, Disconnecting, and Reconnecting Files" on page 147 explains how to connect a file to a unit, how to disconnect it, and, if necessary, how to reconnect it within the same program.

"Gathering Useful Information About Units and Files" on page 158 shows how to use the INQUIRE statement to find out certain characteristics about a unit or file.

| **"Dynamically Allocating Files" on page 165** explains how to dynamically allo-
| cate files.

"Monitoring Errors" on page 174 explains how to check for errors and branch to error routines.

"Considerations for Specific Access Methods" on page 174 gives an overview of the access methods used by FORTRAN and explains coding of the I/O statements that is specific to each access method.

| **"Considerations for Double-Byte Data" on page 190** explains coding of the I/O
| statements that is specific to files containing double-byte character data.

For detailed information on the syntax of the VS FORTRAN statements discussed in this chapter, see *VS FORTRAN Version 2 Language and Library Reference*.

Concepts and Terminology

This section explains several concepts and terms associated with FORTRAN files and I/O processing that are used within this chapter.

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.

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 in "Gathering Useful Information About Units and Files" on page 158 and Appendix G, "What Determines File Existence" on page 433.

File Definitions and Dynamic File Allocation

In order to refer to a file on its storage medium and establish the linkage between your program and the file, a *file definition* must be in effect.

How you code a file definition depends on your operating system:

- For VM, you use the FILEDEF command for non-VSAM data sets and the DLBL command for VSAM data sets. Or, for non-VSAM data sets, you may depend on a default file definition when you don't provide one of your own. The FILEDEF command is explained under "FILEDEF Commands" on page 65. The DLBL command is explained in *Virtual Machine/System Product CMS User's Guide*, SC19-6210.
- For MVS, you use the DD statement as explained under "Defining Files—DD Statement" on page 14.
- For TSO, you use the ALLOCATE command as explained under "Allocating Compiler Data Sets" on page 18.

For certain types of files, you can omit coding file definitions and VS FORTRAN will supply them to the system for you. 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 routine, 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.

Dynamic file allocation is discussed in detail under "Dynamically Allocating Files" on page 165.

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 MVS data set name or CMS file identifier. For example:

Open statement with MVS data set name:

```
OPEN (UNIT=9, FILE='/WRITER.DATA')
```

Open statement with CMS file identifier:

```
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 MVS and CMS file definitions. The ddname is MYINPUT.

OPEN statement with ddname:

```
OPEN (UNIT=9, FILE='MYINPUT')
```

MVS file definition:

```
//MYINPUT DD  DSNAME=WRITER.DATA,DISP=OLD
```

CMS file definition:

```
FILEDEF MYINPUT DISK WRITER DATA A
```

Note that VS FORTRAN identifies a dynamically allocated named file only through its file name. Thus, if another file definition refers to the same MVS data set name or CMS file identifier as that specified on the OPEN statement for a dynamically allocated file, the file might be treated as two different files. For an example of this, see "Dynamically Allocating Files" on page 165.

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). Before FORTRAN 77, all files were unnamed. 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.

The ddname for an unnamed file takes one of the following forms:

- For sequential and direct access: FT*nn*F001, where *nn* is the 2-digit unit number. (For a file connected for sequential access, there can be additional subfiles; the forms FT*nn*F002, FT*nn*F003, and so on, are used in this case.)
- For keyed access: FT*nn*K*kk*, 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 FT*nn*K01, FT*nn*K02, 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: FTERR*sss*, 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: FTPRT*sss* where *sss* is the MTF subtask number.

Thus, if you code the following OPEN statement:

```
OPEN (UNIT=9, ACCESS='SEQUENTIAL')
```

the ddname FT09F001 is used.

Access methods are discussed under "Access Methods Used By FORTRAN" on page 125.

Note that VS FORTRAN identifies an unnamed file only through the file's ddname. Thus, if two file definitions refer to the same physical file, FORTRAN sees these as different files.

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 FT*nn*F001 is used. You may also supply file characteristics by means of the FILEINF service routine. Under MVS, VS FORTRAN creates a file definition with file characteristics that are determined by installation defaults. Under VM, the normal VM default file definition is used.
- Under VM, preconnected files directed to any of the standard I/O units (see "Units and File Connection" on page 125 for an explanation of connecting files to units).

  For these files, the default ddname FT*nn*F001 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 FT*nn*F001 is used. VS FORTRAN creates a file definition with file characteristics that are determined by installation defaults.

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 connect 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.

Access Methods Used By FORTRAN

The term *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 access methods: sequential access, direct access, and keyed access. You specify the 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 access method. Later in this chapter a separate section for each access method describes the access method 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 the record whose record number is 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 181 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 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. For information about VSAM files, see Chapter 11, "Using VSAM with VS FORTRAN Version 2" on page 299.

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, 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 "Access Methods Used By FORTRAN" on page 125), 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 with VS FORTRAN Version 2" on page 299.)

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.

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:

- **For direct access**: fixed-length unblocked
- **For sequential access**:
  - for unformatted I/O: variable-length spanned
  - for formatted I/O, under MVS, including TSO: undefined
  - for formatted I/O, under VM: 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 MVS, use the DCB parameter on the DD statement. For VM, use the RECFM option on the FILEDEF command. For TSO, use the RECFM option on the ALLOCATE or ATTRIBUTE command. For dynamically allocated files, use the RECFM parameter on the FILEINF service routine.

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.

Overview of Input/Output Statements

Figure 28 shows the input/output statements available in VS FORTRAN, the access methods to which they apply, and the operations they perform.

I/O Statement	Access Method	Operation
BACKSPACE	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.

Figure 28. VS FORTRAN Input/Output Statements

Reading and Writing Data

The statements available for reading and writing data are:

READ
PRINT
WRITE
REWRITE

You can process formatted records with the PRINT statement and either formatted or unformatted records with the READ, WRITE, and REWRITE statements.

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 and converted to its internal form when you read formatted records.

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.

A file cannot contain both formatted and unformatted records.

The following sections explain the different ways you can read and write formatted and unformatted records. Also, "Using Internal Files" on page 145 explains how to read from and write to internal files.

Reading and 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.

Using List-Directed Formatting

With list-directed formatting, the records read or written consist of a series of constants. The format of the constants is controlled by the type of data that you read or write.

You can use list-directed formatting with files connected for sequential access only.

Writing Data: 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. It has the form:

PRINT *, *list*

where the asterisk specifies list-directed formatting and *list* is a list of output items, such as variable names or array elements, and implied DO lists. (For information about acceptable output items and implied DO lists, see *VS FORTRAN Version 2 Language and Library Reference.*) If you omit the list, a blank record is written.

Data from the items in the list are written to the output record as constants in the order that they appear in the list. The type and length of the data determine how they will be formatted (see Figure 29 on page 131). Real numbers are written in G format (see Figure 32 on page 139). For noncharacter data, the constants are each followed by one space.

If the data type and length are:	The field width will be:	The fraction will be:	The scale factor will be:	The exponent will be:	The minimum number of digits written will be:
Real, length of 4	16 characters	9 digits	0	2 digits	N/A
Real, length of 8	25 characters	18 digits	0	2 digits	N/A
Real, length of 16	42 characters	35 digits	0	2 digits	N/A
Logical, length of 1 or 4	1 character	N/A	N/A	N/A	N/A
Integer, length of 2	6 characters	N/A	N/A	N/A	1 digit
Integer, length of 4	11 characters	N/A	N/A	N/A	1 digit
Complex, length of 8	35 characters	9 digits	0	2 digits	N/A
Complex, length of 16	53 characters	18 digits	0	2 digits	N/A
Complex, length of 32	87 characters	35 digits	0	2 digits	N/A
Character, length of *n*	*n* characters	N/A	N/A	N/A	N/A

Figure 29. List-Directed Output Formats

If you code the following PRINT statement

```
PRINT *, R4VAR, I2ARR, CH4
```

and you have declared the data types for the variables and arrays as follows:

```
   REAL*4        R4VAR       / -12.5E+12 /
   INTEGER*2     I2ARR(2)    / 33, -44 /
   CHARACTER*4   CH4         / 'TWO' /
```

the following record will be written:

```
 -0.125000000E+14      33    -44 TWO
```

The WRITE statement can perform the same function 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. The form of the WRITE statement is:

WRITE (*specifiers*) *list*

where *specifiers* are the specifiers shown in Figure 30 on page 132 and *list* is a list of output items and implied DO lists. As with the PRINT statement, if you omit the list, a blank record is written.

```
UNIT=un
FMT=*
IOSTAT=ios
ERR=st
```

Figure 30. WRITE Statement Specifiers for List-Directed Formatting

For an external file, the UNIT specifier is for indicating the unit to which the file is connected (connecting files to units is discussed under "Connecting Files" on page 147). 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.

For an internal file, you use the UNIT specifier to code the name of an internal storage area containing the file; for example, UNIT=DATA1. Internal files are discussed in more detail under "Using Internal Files" on page 145.

If UNIT is the first specifier, you can omit UNIT=.

FMT=* specifies list-directed formatting. If you omitted UNIT= for the first specifier, you can also omit FMT=. Thus, the following WRITE statements are acceptable and equivalent:

```
WRITE (UNIT=2, FMT=*) R4VAR, I2ARR, CH4

WRITE (2, FMT=*) R4VAR, I2ARR, CH4

WRITE (2, *) R4VAR, I2ARR, CH4
```

Note that the following PRINT and WRITE statements are equivalent:

```
PRINT *, R4VAR, I2ARR, CH4

WRITE (*, *) R4VAR, I2ARR, CH4
```

The IOSTAT and ERR specifiers are available for error checking. These are discussed under "Monitoring Errors" on page 174.

Reading Data: To read data using list-directed formatting, you use the READ statement.

The READ statement has the same specifiers as the WRITE statement, shown in Figure 30, plus an optional END specifier, which allows you to branch to another statement in the same program when the endfile record is encountered.

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:

- 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

  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 may 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
  ```

An example of reading data using list-directed formatting is the following: If your input is:

```
-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 or 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 read or 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.

You can use NAMELIST formatting only with files connected for sequential access.

Reading Data: To read data with NAMELIST formatting, you use a NAMELIST statement in addition to a READ statement.

The NAMELIST statement has the form:

NAMELIST / *name* / *list*

where *name* is the identifier of an I/O list and is sometimes called a NAMELIST name, and *list* is a list of variable or array names. The *name* must *not* be the same as a variable or array name. Also, the variable or array names must not contain any embedded blanks.

The READ statement that you code then refers to the NAMELIST statement.

The form of the READ statement is:

READ (*specifiers*)

where *specifiers* are the specifiers shown in Figure 31. Note that you do not supply an input list on the READ statement because you have already supplied it on the NAMELIST statement.

UNIT=*un*
FMT=*name*
IOSTAT=*ios*
ERR=*st*

Figure 31. READ Statement Specifiers for NAMELIST Formatting

For an external file, the UNIT specifier is for indicating the unit to which the file is connected (connecting files to units is discussed under "Connecting Files" on page 147). To specify the standard input/output unit, you code UNIT=*. To specify another unit, you code the unit number; for example, UNIT=2.

For an internal file, you use the UNIT specifier to code the name of an internal storage area containing the file; for example, UNIT=DATA1. Internal files are discussed in more detail under "Using Internal Files" on page 145.

If UNIT is the first specifier, you can omit UNIT=.

In the FMT specifier, you specify the same name that you coded in the NAMELIST statement. If you omitted UNIT= for the first specifier, you can also omit FMT=.

The IOSTAT and ERR specifiers are available for error checking. These are discussed under "Monitoring Errors" on page 174.

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 records for NAMELIST formatting must be in a particular form. The first record must begin with a blank, followed by an ampersand (&) immediately followed by the NAMELIST *name* and another blank. Following the *name*, you code the input data, separated by commas. To signal the end of the data, you code &END. All records must begin with a blank.

You must code the input data in a special form. The input data must consist of constants identified by the name of their corresponding variable, array element, or array. For a variable or array element, code the name of the variable or array element, followed by an equal sign, followed by the constant. The following example shows the input data for a variable and an array element:

```
R4VAR=12.5E12, I2ARR(2)=-44
```

For an array, code the array name, followed by an equal sign, followed by a series of constants. For example:

```
I2ARR=33, -44
```

The constants in the input data may be integer, real, complex, logical, or character. Listed below are the rules for coding them:

- ► Each constant must agree in type with its corresponding item in the NAMELIST statement, 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

  Note that for integers and exponents, embedded blanks or trailing blanks between the constant and the following comma, if any, are treated as zeros if BLANK='NULL' is not specified on the OPEN statement.

- ► Character constants must be entered within apostrophes. To include an apostrophe within the character data, use two consecutive apostrophes.
- ► If the constant is logical, it may be in the form T or .TRUE. and F or .FALSE..
- ► For an array element, subscripts must be integer constants.
- ► For an array, the number of constants must be less than or equal to the number of elements in the array.
- ► You can specify a repetition factor for any constant. For example:

  ```
  3*2.6
  ```

The names of the data items must appear in the NAMELIST list, but you can specify them in any order that you wish.

For example, if your input data is the following:

```
&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*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: To write data with NAMELIST formatting, you use a NAMELIST statement in addition to a WRITE or PRINT statement.

The WRITE statement has the same form as the READ statement, except that it does not have an END specifier.

The form of the PRINT statement is:

PRINT *name*

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 state ments:

```
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:

```
&MYDATA
R4VAR=-0.125000000E+14,I2ARR=     33,    -44,CH4='TWO '
&END
```

The constants are formatted in the same way as they are for list-directed formatting, as shown in Figure 29 on page 131.

Specifying Your Own Format

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 PRINT statement has the following form:

PRINT *fmt*, *list*

where *fmt* is a format identifier and *list* is a list of output items and implied DO lists. If you omit the list, a blank record is written.

The format identifier can be any of the following:

- A statement label
- A character constant
- A character variable
- An integer variable
- A character array element
- A character array name
- A character expression
- An array name

The most commonly used format identifiers, which are discussed here, are the statement label, character constant, and character variable. For information on the other format identifiers, see *VS FORTRAN Version 2 Language and Library Reference*.

The statement label refers to a FORMAT statement, in which you specify the format in which the data is to be written. You code the FORMAT statement as follows:

FORMAT (*f1*, *f2*, ... *fn*)

where *f1*, *f2*, and so on are format codes and correspond to the items in the output list. The format code *f1* specifies the format for the first item in the output list, *f2* specifies the format for the second item, and so forth. Note that 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.

Some of the format codes and their meanings are shown in Figure 32 on page 139. For a complete list, see *VS FORTRAN Version 2 Language and Library Reference*.

Format Code	Meaning
A*w*	The next *w* characters in the output record will contain character data.
I*w*	The next *w* characters in the output record will contain an integer.
*a*I*w*	This is equivalent to coding the I*w* format code *a* times.
I*w.m*	The next *w* characters in the output record will contain an integer, displaying at least *m* digits.
E*w.d*E*e*	The next *w* characters in the output record will contain a real number, with *d* decimal places, and *e* digits in the exponent field.
F*w.d*	The next *w* characters in the output record will contain a real number, with *d* decimal places, but with no exponent.
G*w.d*E*e*	Depending on the magnitude of the number, the next *w* characters in the output record will contain a real number in either the E format or F format described above.
/	Data transfer on current record is ended. Continue data transfer on next record if more items are in the I/O list. (For sequential and direct access only)
*w*X	For output, fill the next *w* characters with blanks. For input, skip past the next *w* characters.

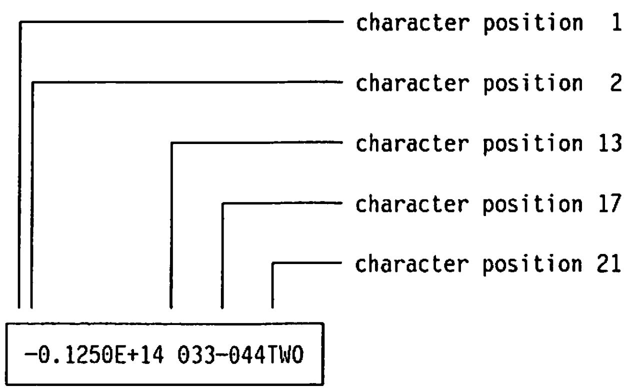
Figure 32. Some Format Codes and Their Meanings

An example of writing formatted data using the FORMAT and PRINT statements is the following: If you run a program with 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:



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
```

The form of the WRITE statement is:

WRITE (*specifiers*) *list*

where *specifiers* are the specifiers shown in Figure 33, plus additional specifiers used for specific access methods. The additional specifiers are discussed under "Considerations for Specific Access Methods" on page 174. The *list* is a list of output items and implied DO lists. If you omit the list, a blank record is written.

UNIT=*un*
FMT=*fmt*
IOSTAT=*ios*
ERR=*st*

Figure 33. WRITE Statement Specifiers When Specifying Your Own Format

For an external file, the UNIT specifier is for indicating the unit to which the file is connected (connecting files to units is discussed under "Connecting Files" on page 147). To specify the standard input/output unit, you code UNIT=*. To specify another unit, you code the unit number; for example, UNIT=2.

For an internal file, you use the UNIT specifier to code the name of an internal storage area containing the file; for example, UNIT=DATA1. Internal files are discussed in more detail under "Using Internal Files" on page 145.

If UNIT is the first specifier, you can omit UNIT=.

In the FMT specifier, you code the format identifier, *fmt*, which can be any of those listed on page 138 for the PRINT statement. If you omitted UNIT= for the first specifier, you can also omit FMT=.

The IOSTAT and ERR specifiers are available for error checking. These are discussed under "Monitoring Errors" on page 174.

For a file connected for keyed access, the FMT specifier and the same format identifiers are also available on the REWRITE statement. This statement is discussed under "Considerations for Specific Access Methods" on page 174.

Reading Data: When reading data, you can specify your own format on the READ statement. The specifiers are the same shown in Figure 33 on page 140, plus additional specifiers used for specific access methods. The additional specifiers are discussed under "Considerations for Specific Access Methods" on page 174.

The format identifiers you code on the FMT specifier can be any of those mentioned above for writing data.

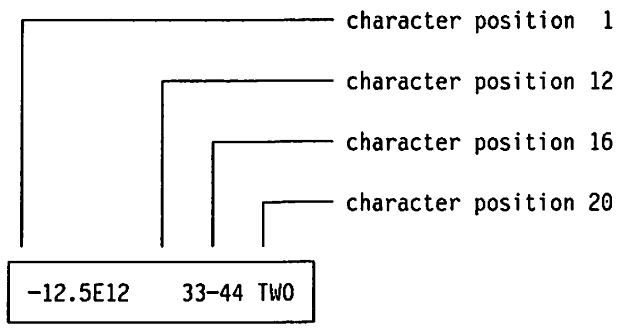
Most of the format codes are the same as those used for writing, except that they, of course, determine how the data will be read rather than written. A few format codes are specific to writing or reading. For example, the format code BN means ignore blanks in input.

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:



```
-12.5E12    33-44 TWO
```

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 and, under MVS, how to perform asynchronous I/O.

Unformatted data can be read from or written to files connected for sequential, direct, or keyed access. Asynchronous I/O is allowed only for sequentially accessed files.

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.

The form of the WRITE statement is:

WRITE (*specifiers*) *list*

where *specifiers* are the specifiers shown in Figure 34, plus additional specifiers used for specific access methods. The additional specifiers are discussed under "Considerations for Specific Access Methods" on page 174. The *list* is a list of output items and implied DO lists.

UNIT=*un*
IOSTAT=*ios*
ERR=*st*
NUM=*n*

Figure 34. WRITE Statement Specifiers When Specifying Your Own Format

The UNIT specifier is for indicating the unit to which the file is connected (connecting files to units is discussed under "Connecting Files" on page 147). If UNIT is the first specifier, you can omit UNIT=.

The IOSTAT and ERR specifiers are available for error checking. These are discussed under "Monitoring Errors" on page 174.

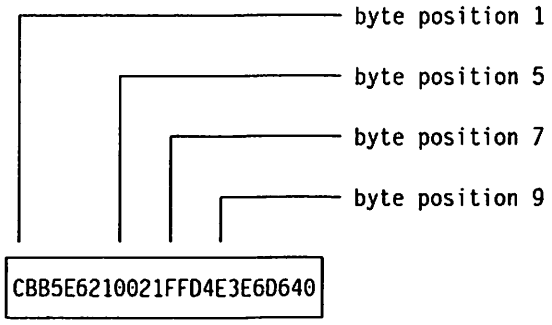
If you code NUM=*n*, the integer variable or integer array element *n* is assigned a value representing the number of bytes transmitted to the output items.

An example of writing unformatted data is the following: 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
```

The REWRITE statement is discussed under "Input/Output Operations for Keyed Access" on page 181.

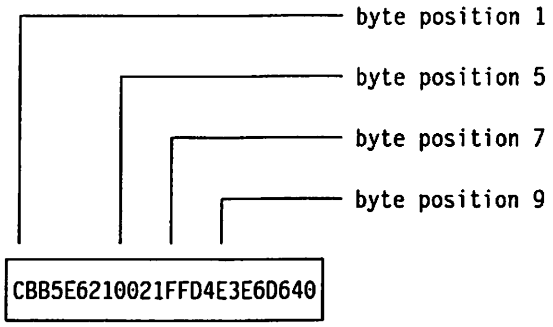
Reading Unformatted Data

To read unformatted data, you use the READ statement. As with the WRITE and REWRITE statements, each READ statement processes only one record.

Similarly, the READ statement has the same specifiers shown in Figure 34 on page 142, plus additional specifiers used for specific access methods. The additional specifiers are discussed under "Considerations for Specific Access Methods" on page 174.

If you code NUM=*n* on the READ statement, the integer variable or integer array element *n* is assigned a value representing the number of bytes transmitted to the input items.

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

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 and must allow variable-length spanned records. In the Multitasking Facility environment, asynchronous I/O is not allowed in parallel subroutines.

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.

The form of the READ statement is:

READ (UNIT=*un*, ID=*id*) *list*

If you code the UNIT specifier first, you can omit UNIT=. In the ID specifier, you code an integer constant or expression that is used to identify the READ statement.

In the input list 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)
```

The WRITE statement has the same form as the READ statement.

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)
```

In addition, the WAIT statement has the optional specifiers COND and NUM. If you specify COND=*i*1, the integer variable *i*1, is assigned a value of 1 if the operation completed successfully, a value of 2 if an error was detected, or a value of 3 if the endfile record was reached. If you specify NUM=*i*2, the integer variable *i*2 is assigned a value indicating the number of bytes transmitted.

Using Internal Files

As explained earlier under "Concepts and Terminology" on page 121, 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 it 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:

```
-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 35, 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
            .
            .  Process integer values
            .
      ELSE IF (RECORD(1:1) .EQ. 'R'·.AND. LEN .GE. 51) THEN
         READ  (RECORD, '(1X, BN, 5E10.3)') REALS
            .
            .  Process real values
            .
      ELSE
            .
            .  Process other types of records
            .
      ENDIF
      GO TO 1
2     CONTINUE
      END
```

Figure 35. Sample Program—Transferring Data to an Internal File

Connecting, Disconnecting, and Reconnecting Files

This section explains how to connect, disconnect, and reconnect files within your program.

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 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.

Preconnecting Files

Before the OPEN statement was introduced in FORTRAN 77, all files were preconnected. To maintain compatability, preconnection is still allowed, but only for certain files. Only unnamed non-VSAM files that are accessed sequentially can be preconnected.

To preconnect a file, you code a file definition with a ddname of FT*nn*F001, where *nn* is the 2-digit number of the unit to which you want to connect the file. For CMS, you can rely on a default FILEDEF command with the ddname FT*nn*F001.

That's all you have to do. When your program begins to run, the file will be automatically connected to the unit.

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 152.

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 PUNCH statement [1]) by using an asterisk rather than the unit number. (The PRINT statement is always directed to a standard output unit.) During installation, either two or three standard output units were defined:

- Two Standard Output Units
  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

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

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.

For dynamically allocated files, VS FORTRAN supplies defaults for the specific devices that will be used for these units. For more information about these defaults, see "Dynamically Allocating Files" on page 165.

Also, under VM, if you don't code a file definition for a file that is not dynamically allocated, VS FORTRAN supplies defaults for specific devices. For more details, see "Files Preconnected to the Standard Input/Output Units" on page 65.

Connecting Files to Units with 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.

The specifiers available on the OPEN statement are shown in Figure 36 on page 149.

[1] The PUNCH statement is a FORTRAN 66 statement that is still supported by VS FORTRAN, but only under the compiler option LANGLVL(66). It is described in *IBM System/360 and System/370 FORTRAN IV Language*, GC28-6515.

```
UNIT=un
FILE=fn
STATUS=sta
ACCESS=acc
FORM=frm
BLANK=blk
ACTION=act
PASSWORD=pwd
RECL=rcl
KEYS=(start:end[,start:end] ...)
IOSTAT=ios
ERR=st
CHAR=chr
```

Figure 36. OPEN statement specifiers

Identifying the Unit and File: 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 152.)

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')
```

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 file is being connected for sequential or direct access, the ddname FT*nn*F001 is used. 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:

```
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 154.

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. For VM, an example of this would be a minidisk that you have been linked to in read-only status. 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. In these cases, you will be unable to connect the file at all if it doesn't already exist.

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 101.) 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.

A system message may be issued showing some certain x13 abend codes. This is normal processing and should be ignored unless a VS FORTRAN message is issued.

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.

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.

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 181.

Choosing Formatted or Unformatted I/O: To specify formatted or unformatted I/O, you 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: You can 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 182. For information on using the ACTION specifier in an MTF environment, see Appendix E, "The Multitasking Facility (MTF)" on page 349.)

Under VM, 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. These specifiers are discussed under "Monitoring Errors" on page 174.

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 "Considerations for Double-Byte Data" on page 190.

Changing Connection Properties

After a file has been connected, either by preconnection or by the OPEN statement, you can 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 37 shows which of these defaults you can override, depending on which I/O statement you use first.

If your first I/O statement for the preconnected 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.

Figure 37 (Part 1 of 2). Overriding Defaults for Preconnected Files

If your first I/O statement for the preconnected 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:**
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	Unformatted	N/A	N/A	Nothing

Figure 37 (Part 2 of 2). Overriding Defaults for Preconnected Files

The following example shows how to change the access method from sequential to direct, using the OPEN statement. As shown in Figure 37 on page 152, 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 FT*nn*F001 is used, and for files connected for keyed access, the ddname FT*nn*K*kk* 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.

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

Figure 38 shows the specifiers available on the CLOSE statement.

UNIT=*un*
STATUS=*sta*
IOSTAT=*ios*
ERR=*st*

Figure 38. CLOSE statement specifiers

Identifying the Unit: On the UNIT specifier, you 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).

You 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 unallocated. 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 39 lists these files and explains how they are actually 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.

Figure 39. Files That Can Be Deleted

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.

You cannot delete any type of file not listed in Figure 39. 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 (VM only)
- An unlabeled tape data set
- A file with the file type TXTLIB, LOADLIB, or MACLIB (VM only)

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 FT*nn*F001, FT*nn*F002, 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 FT*nn*F001) 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 177.

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 under "Monitoring Errors" on page 174.

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.

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 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.

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 example illustrates what happens if you do *not* use the CLOSE statement.

```
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
OPEN (8, BLANK='ZERO')
READ (8, FMT=10) F, G, H
```

Assuming that no previous I/O statements have been directed to unit 8, the first READ statement is directed to the unnamed file associated with unit 8. The second OPEN statement breaks the connection and associates unit 8 with the file named MYDATA. 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
```

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
```

Gathering Useful Information About Units and Files

At times you may want to find out certain characteristics of a unit or file and, based on them, take alternative actions. You can get such information by using the INQUIRE statement.

Forms of INQUIRE

The INQUIRE statement has three different forms:

INQUIRE by file name

INQUIRE by unit

INQUIRE by unnamed file

The first form is for inquiring about a particular named file. If you select this form, you specify the file name on the FILE specifier, in the same manner as you do on the OPEN statement. For files not dynamically allocated, specify the ddname; and, for dynamically allocated files, specify the MVS data set name or CMS file identifier preceded by a slash (/).
For example:

Nondynamically allocated file:

```
INQUIRE (FILE='MYFILE', other specifiers)
```

Dynamically allocated file under CMS:

```
INQUIRE (FILE='/MYDATA OUTPUT A4', other specifiers)
```

Dynamically allocated file under MVS:

```
INQUIRE (FILE='/MYDATA.ON.MVS', other specifiers)
```

You can use the second form to find out whether a particular unit exists and whether any file is connected to it. If a file is connected to the unit at the time, you can also find out selective characteristics of that file. For example, to inquire about a file connected to unit 9, you specify:

```
INQUIRE (UNIT=9, other specifiers)
```

The third form allows you to gather information about an unnamed file. In this form, for the FILE specifier, you code either the file's default ddname or a character expression whose value is blanks (this can be a character constant). For example:

```
INQUIRE (FILE='FT09F001', other specifiers)
```

or:

```
INQUIRE (UNIT=9, FILE=' ', other specifiers)
```

Note that when you provide a blank value in the FILE specifier, you must also code the UNIT specifier. The following conditions, in the order they are listed, determine which file is referred to:

1. If the unit is currently connected to an unnamed file that is internally open, the characteristics for that file are returned.

   Internally open means that a file either has been connected by the OPEN statement; or, in the case of a preconnected file, a READ, WRITE, PRINT, or ENDFILE statement has been successfully issued for the file.

2. If the above condition is not met, and a file definition is in effect for the default ddname FT*nn*K01, the characteristics for that file are returned.

3. Otherwise, the characteristics for the file with the default ddname FT*nn*F001 are returned.

For unnamed files that are dynamically allocated under VM, you may also code the INQUIRE statement as follows. However, in this case, you can inquire only about file existence.

```
INQUIRE (FILE='/FILE FTnnF001 fm', EXIST=EX)
```

A system message may be issued showing some certain x13 abend codes. This is normal processing and should be ignored unless a VS FORTRAN message is issued.

Summary of What You Can Find Out

Figure 40 summarizes the information you can get and the corresponding INQUIRE specifiers.

For VSAM files that are password-protected, you must also code the PASSWORD specifier if you have not already connected the file. Only the file's read password is required.

Other specifiers available on the INQUIRE statement are IOSTAT and ERR. These are discussed in "Monitoring Errors" on page 174.

If you want to know this:	Use this specifier:
Whether the file or unit exists	EXIST=*exs*
Whether the file or unit is connected	OPENED=*opn*
Which unit the file is connected to	NUMBER=*num*
Whether the file has a name	NAMED=*nmd*
What the name of the file is	NAME=*nam*
Whether the file may contain double-byte characters	CHAR=*chr*
Whether the file is currently connected for sequential, direct, or keyed access	ACCESS=*acc*
Whether the file can be connected for sequential access	SEQUENTIAL=*seq*
Whether the file can be connected for direct access	DIRECT=*dir*
Whether the file can be connected for keyed access	KEYED=*kyd*
Whether a file is currently connected for writing only, reading only, or both writing and reading	ACTION=*act*
Whether a file is currently connected for reading only	READ=*ron*
Whether a file is currently connected for writing only	WRITE=*wri*
Whether a file is currently connected for both reading and writing	READWRITE=*rwr*
Whether the file is currently connected for formatted or for unformatted I/O	FORM=*frm*
Whether the file can be connected for formatted I/O	FORMATTED=*fmt*
Whether the file can be connected for unformatted I/O	UNFORMATTED=*unf*
Whether input blanks are treated as zeros or ignored (for formatted I/O)	BLANK=*blk*
What the record length is (for files connected for direct access)	RECL=*rcl*
What the record number of the next record is (for files connected for direct access)	NEXTREC=*nxr*
Which key is in use (for files connected for keyed access)	KEYID=*kid*

Figure 40 (Part 1 of 2). Summary of the information you can get with INQUIRE

If you want to know this:	Use this specifier:
What the length of the key is (for files that can be connected for keyed access)	KEYLENGTH=*kle*
Where the key starts (for files that can be connected for keyed access)	KEYSTART=*kst*
Where the key ends (for files that can be connected for keyed access)	KEYEND=*ken*
What the length of the last record affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement is (for files connected for keyed access)	LASTRECL=*lrl*
What the value of the key of the last record that was affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement is (for files connected for keyed access)	LASTKEY=*lky*

Figure 40 (Part 2 of 2). Summary of the information you can get with INQUIRE

The EXIST specifier allows you to check whether a particular unit or file exists for your program. For INQUIRE by unit, the existence of the specified unit is determined. A unit exists if its unit number is within the range of allowable unit numbers at your site. For INQUIRE by file or INQUIRE by unnamed file, the existence of the specified file is determined. File existence depends on a number of factors, such as what type of device the file is on. Appendix G, "What Determines File Existence" on page 433 discusses these factors in detail. If the unit or file exists, the value true is returned. If it doesn't exist, the value false is returned.

The OPENED specifier is for checking whether a unit or file is connected, either by the OPEN statement or by preconnection.

When you execute an INQUIRE by unit, the value that will be returned for a unit that is preconnected but for which no I/O statements other than INQUIRE have been executed depends on which of the run-time options, INQPCOPN or NOINQPCOPN, is in effect. When INQPCOPN is in effect, a value of true will be returned if the unit is connected (by an OPEN statement or by preconnection); that is, if an I/O statement other than OPEN, CLOSE, or INQUIRE can be executed without first executing an OPEN statement. When NOINQPCOPN is in effect, a value of false will be returned, even for a preconnected unit, if no I/O statements other than INQUIRE have been executed for that unit.

When you specify a default ddname in the FILE specifier, the value true will be returned only if the following conditions are met:

- For the form FT*nn*F*mmm*, the unit number (*nn*) is within the range allowed for your installation and the sequence number (*mmm*) is equal to the number of the particular subfile to which the file is positioned. Subfiles are discussed in more detail under "Processing Subfiles" on page 177.
- For the form FT*nn*K*kk*, the unit number (*nn*) is within the range allowed for your installation and the key number (*kk*) is within the range of the number of keys for the file that is connected.
- For the form FTERR*sss* or FTPRT*sss*, MTF is active and the subtask number (*sss*) is within the range of the number of active MTF subtasks.

The NUMBER, NAMED, and NAME specifiers help you to identify a file. NUMBER gives you the unit number to which the file is connected; NAMED tells you whether the file is named or unnamed; and, NAME gives you the name of the file if it has one. For dynamically allocated named files, NAME returns the CMS file identifier or MVS data set name in the following format:

Under CMS:

'/fn ft fm' **or** *'/fn ft fm (member)'*

where *fn*, *ft*, and *member* are 8 characters and *fm* is 2 characters

Under MVS:

'/dsn' **or** *'/dsn (member)'*

where *dsn* is 44 characters and *member* is 8 characters

Note: If *fn*, *ft*, *fm*, *dsn*, and *member* are shorter than the lengths stated above, they will be padded with blanks. The variable used to hold the name returned by INQUIRE (*nam*) must be declared to be at least as long as 21 characters for CMS files (32 for CMS files that include *member*) and at least as long as 45 characters for MVS files (56 for MVS files that include *member*).

The CHAR specifier is for checking whether a connected file may contain data in a language, such as Japanese, that has double-byte characters. If you coded CHAR='DBCS' on the OPEN statement, the value DBCS is returned; if you coded CHAR='NODBCS', did not code the CHAR specifier, or did not use an OPEN statement, the value NODBCS is returned.

Some INQUIRE specifiers allow you to find out how a file is *currently* connected, whereas others allow you to find out how a file *can be* connected. By coding the ACCESS specifier, you can check which access method is being used for the currently connected file. By coding the SEQUENTIAL, DIRECT, or KEYED specifier, you can find out whether a file can be connected for sequential, direct, or keyed access. Based on the file's characteristics, the character variable or array element given in the specifier is set to the value YES or NO, as shown in Figure 41 on page 163. If the file's characteristics cannot be determined by VS FORTRAN, a value of UNKNOWN is returned.

Specifier	File Characteristics that Result in YES	File Characteristics that Result in NO
SEQUENTIAL	Non-VSAM file VSAM entry-sequenced data set VSAM relative record data set	VSAM key-sequenced data set
DIRECT	Non-VSAM file with fixed unblocked records VSAM relative record data set	Non-VSAM file with other than fixed unblocked records VSAM entry-sequenced data set VSAM key-sequenced data set
KEYED	VSAM key-sequenced data set	Non-VSAM file VSAM entry-sequenced data set VSAM relative record data set

Figure 41. Characteristics that Determine Whether a File Can Be Connected for Sequential, Direct, or Keyed Access

Similarly, to find out whether the currently connected file is connected for formatted or unformatted I/O, you code the FORM specifier. To check whether a file can be connected for formatted or unformatted I/O, you use the FORMATTED or UNFORMATTED specifier. Based on the file's characteristics, the character variable or array element given in the specifier is set to the value YES or NO, as shown in Figure 42. If the file's characteristics cannot be determined by VS FORTRAN, a value of UNKNOWN is returned.

	File Characteristics	File Characteristics
Specifier	that Result in YES	that Result in NO
Specifier	File Character-	File Character-
FORMATTED	Non-VSAM file with other than variable-length spanned records Any VSAM file	Non-VSAM file with variable-length spanned records
UNFORMATTED	Any file	None

Figure 42. Characteristics that Determine Whether a File Can Be Connected for Formatted or Unformatted I/O

If a file is connected for formatted I/O, you can use the BLANK specifier to inquire how blanks in arithmetic input fields are treated. If they are ignored, the value NULL is returned. If they are treated as zeros, the value ZERO is returned.

The ACTION specifier allows you to check whether a currently connected file is connected for reading, writing, or both reading and writing. The value returned is READ, WRITE, or READWRITE. Instead of the ACTION specifier, you could code the READ, WRITE, or READWRITE specifier, for which the value YES or NO is returned.

RECL and NEXTREC are for files connected for direct access. RECL supplies the record length of the file and NEXTREC supplies the number of the next record.

The remaining specifiers are for files connected for keyed access. KEYID lets you find out which key is currently in use for a file. KEYLENGTH, KEYSTART, and KEYEND return more details about that key: its length, where it starts, and where it ends. However, if you use INQUIRE before connecting a file, KEYLENGTH, KEYSTART, and KEYEND return information about the keys in the file referred to by the file definition. LASTRECL and LASTKEY allow you to obtain the record length and KEYID of the last record affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement.

For more detailed information about the INQUIRE specifiers, see *VS FORTRAN Version 2 Language and Library Reference*.

Where In Your Program You Can Code INQUIRE

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 table 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 43 on page 165 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 43. Sample Program—Using the INQUIRE Statement

Dynamically Allocating Files

For certain types of files, you can omit coding file definitions and VS FORTRAN will supply them for you to the system. 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 VM
- Temporary files (connected with STATUS='SCRATCH'), excluding subfiles
- Under VM, 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.

How to Dynamically Allocate a File

To dynamically allocate a file, you omit the file definition for it. In addition, for named files, you must specify the MVS data set name or CMS file identifier, 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 MVS and VM, and for temporary files under MVS, VS FORTRAN creates a file definition with the ddname FT*nn*F001. The file characterstics are determined by installation defaults, discussed below under "File Characteristic Defaults" on page 167, and by the FILEINF service routine, discussed below under "Overriding File Characteristic Defaults" on page 168. For temporary files under VM, the normal VM default file definition is used.

Following are some examples of OPEN statements for named files:

Under MVS:

```
OPEN (9, FILE='/MILLER.MYPDS.ACCOUNT(ABC)')

OPEN (9, FILE='/MILLER.BALANCE.YEAR')
```

Under VM:

```
OPEN (9, FILE='/CALC DATA')

OPEN (9, FILE='/CALC MACLIB A(DATAMBR)'
```

Note that from a VS FORTRAN point of view, the MVS data set name or CMS file identifier 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 MVS:

```
//DD1  DD  DSN=MYFILE.FORT,DISP=OLD

      OPEN (1, FILE='DD1')
      .
      .
      .
      OPEN (2, FILE='/MYFILE.FORT')
```

Under VM:

```
FILEDEF DD1 DISK MYFILE FORT A1

      OPEN (1, FILE='DD1')
      .
      .
      .
      OPEN (2, FILE='/MYFILE FORT A1')
```

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 routine, 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 routine. 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.

VM 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.

File Characteristic Defaults

Most of the defaults for file characterstics (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 Appendix H, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 445. Another installation default, available only for dynamically allocated files, indicates the number of records to allocate for a new DASD file. Under VM, this default is equivalent to the XTENT option on the FILEDEF command. The IBM-supplied default value under VM 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 170. 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 may be modified by means of a file definition).

These are:

- The device for files directed to the standard I/O units. The default devices are:
  - **For VM:**
    - Standard input unit: TERMINAL
    - Standard output unit(s) for error messages, WRITE, and PRINT statements: TERMINAL
    - Standard output unit for PUNCH statement: PUNCH
  - **For MVS:**
    - Standard output unit(s) for error messages, WRITE, and PRINT statements: SYSOUT=A for batch, TERMINAL for TSO
- The default devices for other units under VM. Under VM, the default device for all other units is DISK.

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 routine 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:

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 44 on page 169 shows the file characterstics 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.

File Characteristic	Parameter on CALL FILEINF
VM and MVS:	
Record format	RECFM
Logical record length	LRECL
Block size	BLKSIZE
Number of records	MAXREC
MVS Only:	
Primary space in cylinders	CYL
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 volumes an output data set requires	VOLCNT
Volume serial number	VOLSER
Multiple volume serial numbers	VOLSERS

Figure 44. Parameters on CALL FILEINF

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 About Dynamically Allocated Files" on page 171.) 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.)

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.

    Caution: 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 RECFM, LRECL, and BLKSIZE: Refer to Appendix H, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 445 for information on how these values are processed and the defaults that are supplied.

Considerations for MAXREC: Under VM, 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 45 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.

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

Figure 45. Formulas for Primary Space under MVS

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.

Gathering Information About Dynamically Allocated Files

To inquire about a dynamically allocated file by file name, supply the CMS file identifier or MVS data set name, preceded by a slash (/), on the FILE specifier. Examples are:

```
Under MVS:

INQUIRE (FILE='/PROG1.FORTDATA.RATES', EXIST=EX)

INQUIRE (FILE='/MILLER.MYPDS.ACCOUNT(ABC)', OPENED=OPN)


Under VM:

INQUIRE (FILE='/DEPTJ76 DATAFILE', ACCESS=ACC)

INQUIRE (FILE='/OURFILE MACLIB A(OURMBR)', OPENED=OPN)
```

The form INQUIRE by unit is the same as for nondynamically allocated files. For example:

```
INQUIRE (UNIT=9, other specifiers)
```

Likewise, the form INQUIRE by unnamed file is the same as for nondynamically allocated files; you do *not* code a preceding slash. For example:

```
INQUIRE (FILE='FT09F001', other specifiers)

INQUIRE (UNIT=9, FILE=' ', other specifiers)
```

Under VM, the form shown below for inquiring by unnamed file is also allowed. However, you can use this form only to determine file existence.

```
INQUIRE (FILE='/FILE FTnnF001 fm', EXIST=EX)
```

MVS Notes

- If the file is cataloged or is on the specified volume, VS FORTRAN refers to that file; the file may or may not exist to VS FORTRAN.
- If the file is not cataloged or is not on the specified volume, it is considered not to exist.

VM Notes

- You may omit the file mode from the file name, in which case the default A is used. If the file can't be found on the A disk, it is considered not to exist.
- If you specified * as the file mode, VS FORTRAN refers to the first file having the file name and file type that is found on any disk through the standard search from A-Z disks. If no such file is found, the file is considered not to exist.
- If you specify * as the file mode on the OPEN statement, the NAME specifier on the INQUIRE statement returns the actual file mode of the file.

From a VS FORTRAN point of view, the MVS data set name or CMS file identifier specified on a file definition 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 MVS data set name or CMS file identifier were coded on the OPEN statement, the value would be true.)

Under MVS:

```
//DD1  DD  DSN=MYFILE.FORT,DISP=OLD

       OPEN (1, FILE='DD1')
       .
       .
       .
       INQUIRE (FILE='/MYFILE.FORT', OPENED=OPN)
```

Under VM:

```
FILEDEF DD1 DISK MYFILE FORT A1

       OPEN (1, FILE='DD1')
       .
       .
       .
       INQUIRE (FILE='/MYFILE FORT A1', 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

For example, in Figure 46, the first CALL FILEINF statement sets the RECFM value to FB. The INQUIRE statement then 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.) The second CALL FILEINF statement resets RECFM to FB (because processing of the INQUIRE statement caused it to become ineffective) and also sets values for LRECL, BLKSIZE, and MAXREC.

```
*
* Set record format
*
      CALL FILEINF(IRET,'RECFM','FB')
*
* Check whether the file DEPTJ76 DATAFILE exists and whether it
* can be connected for formatted I/O
*
      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
         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')
         ENDIF
       ENDIF
```

Figure 46. Example of Using the FILEINF Routine with INQUIRE and OPEN Statements

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, the file is considered not to exist.

Monitoring Errors

Two specifiers that are available with most I/O statements that allow for error checking are IOSTAT=*ios* and ERR=*stl*.

When you use the IOSTAT specifier, *ios* is set to one of the following values after the I/O statement has been processed:

- Zero, if no transmission error was detected
- Positive, if an error was detected
- Negative, at sequential end-of-file
- VSAM return and reason codes, for a VSAM file

The ERR specifier allows you to branch to a section of code beginning with the FORTRAN statement labeled *stl* when an error occurs. The section of code that you branch to could close any other open files and display information useful in debugging, such as accumulated totals or current values in selected data items.

Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see the figure on option table defaults in *VS FORTRAN Version 2 Language and Library Reference*.

Extended error handling is also available. For a description of this aid, see "Extended Error Handling" on page 110.

Considerations for Specific Access Methods

This section gives an overview of the access methods used by FORTRAN and explains coding of the I/O statements that is specific to each access method.

Input/Output Operations for 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 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

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.

Reading Data

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.

Writing Data

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.

"Repositioning Files" on page 176 explains how to extend a file by using the BACKSPACE statement together with the WRITE statement.

To write an endfile record, you use the ENDFILE statement. The specifiers for the ENDFILE statement are shown in Figure 47.

UNIT=*un*
ERR=*stl*
IOSTAT=*ios*

Figure 47. ENDFILE Statement Specifiers

For example, 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 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 177.

Repositioning Files

To reposition a file to the beginning of its first record, you 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 Figure 48.

```
UNIT=un
ERR=stl
IOSTAT=ios
```

Figure 48. REWIND Statement Specifiers

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.

To reposition a file to the beginning of the previous record, you use the BACKSPACE statement.

Note: The BACKSPACE statement must not be used with list-directed or NAMELIST formatting. It also must not be used for internal files.

The BACKSPACE statement specifiers are the same as those shown in Figure 48.

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.

Gathering Information About Files

On the INQUIRE statement, the SEQUENTIAL specifier is available to find out whether a file can be connected for sequential access. The characteristics that determine whether a file can be connected for sequential access are shown in Figure 41 on page 163.

Processing Subfiles

An unnamed file connected for sequential access can be composed of subfiles.

Do not confuse a set of subfiles with a set of concatenated data sets under MVS. In either case, FORTRAN treats each set as a single file. However, whereas with concatenated data sets, you do not need to code anything special when writing your program, with subfiles, you do.

To begin with, each subfile must have a separate file definition. The file definition for the first subfile must have the ddname FT*nn*F001, the file definition for the second subfile must have the ddname FT*nn*F002, and so on.

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 49 on page 178).

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
:		

Figure 49. The WRITE Statement Positions the File to the Next Subfile

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 another 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 in the FILE specifier. Figure 50 on page 179 shows, for a sample program, how the change in file position affects the value assigned to the variable given in the OPENED specifier.

```
                                           Subfile at Which   Value Returned
Statements                                 File Is Positioned for OPNn

     :                                     --
     OPEN (8)
111  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
     :
```

Figure 50. Values Returned for OPENED When You Code FILE='FTnnFmmm'

Finally, if you have read or written data beyond the first subfile during the current connection, you cannot delete the file. However, if you attempt to do so, 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 51, 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.

```
                                     Subfile
Statements                           Acted On     Result

     :
     OPEN (8)                        001
5    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
     :
```

Figure 51. Result of Attempting to Delete A Set of Subfiles for which Deletion Is Not Allowed

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.

Input/Output Operations for 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
- 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 may 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.

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.

Gathering Information About Files

To find out whether a file can be connected for direct access, you can use the DIRECT specifier on the INQUIRE statement. The characteristics that determine whether a file can be connected for direct access are shown in Figure 41 on page 163.

Other specifiers on the INQUIRE statement that are unique to direct access are the RECL and NEXTREC specifiers. The RECL specifier supplies the length of the records. The NEXTREC specifier supplies the record number of the next record, that is, the record following the one just read or written. Note that more than one record may have been read or written as a result of using the slash (/) format code. In this case, the record number returned is *not* $n+1$, where n is the number given in the REC specifier.

Input/Output Operations for 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 52. If you omit the ACTION specifier, the default is ACTION='READ'.

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" on page 183 explains when to specify each of these)
Retrieve records	ACTION='READ'
Just update or both update and retrieve records	ACTION='READWRITE'

Figure 52. Coding the ACTION Specifier on the OPEN Statement

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 with VS FORTRAN Version 2" on page 299.)

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 the ddname of its file definition must be FT*nn*K01, where *nn* is the 2-digit unit number. If you specify additional keys, the ddnames of their file definitions must be FT*nn*K02, FT*nn*K03, 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 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'.

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. For *direct retrieval*, you 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.

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 allows you to retrieve the first record whose key value is equal to or greater than your search argument. 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 allows you to retrieve the first record whose key value is greater than your search argument. 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.

Note that for the purpose of comparing key values, the data, regardless of whether you transferred it from character or noncharacter data items, is used in its internal representation (with no editing or conversion) and is interpreted as a string of characters. 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.

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.

For *sequential retrieval*, you 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.

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. An example of this scenario is:

```
   OPEN (8, FORM='FORMATTED', ACCESS='KEYED', KEYS=(16:19, 1:3))
   I=1
   READ (8, FMT=99, KEYGT='D5', KEYID=2, NOTFOUND=100)
     DEPTNO(I), NAME(I), EMPNO(I)
30 I=I+1
   READ (8, FMT=99, NOTFOUND=300)
     DEPTNO(I), NAME(I), EMPNO(I)
   GO TO 30
```

The initial READ statement 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. Assuming that the direct retrieval obtains a record whose key value begins with D6, the sequential READ statement in the loop that is subsequently processed then retrieves all the remaining records whose key values begin 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 Figure 53.

UNIT=*un*
ERR=*stl*
IOSTAT=*ios*
FMT=*fmt*
NUM=*n*
DUPKEY=*stl*

Figure 53. REWRITE Statement Specifiers

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 the same as those shown in Figure 54 on page 188.

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.

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 Figure 54 on page 188.

```
UNIT=un
ERR=stl
IOSTAT=ios
```

Figure 54. REWIND Statement Specifiers

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.

The specifiers on the BACKSPACE statement are the same as those shown in Figure 54.

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 next previous record, as it would if the key were unique.

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.

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.

Gathering Information About Files

The specifiers on the INQUIRE statement that pertain to keyed access are:

- KEYED
- KEYID
- KEYLENGTH
- KEYSTART
- KEYEND
- LASTRECL
- LASTKEY

The KEYED specifier allows you to inquire whether the file can be connected for keyed access. The character variable or array element is set to YES if the file is a VSAM KSDS, NO if the file is not a VSAM KSDS, and UNKNOWN if this cannot be determined.

The KEYID specifier gives you the key of reference.

If you use the INQUIRE statement after you have connected the file, the KEYLENGTH, KEYSTART, and KEYEND specifiers return information about the key of reference. The KEYLENGTH specifier gives you the length of the key, the KEYSTART specifier gives you its beginning position, and the KEYEND specifier gives you its ending position.

If you use the INQUIRE statement before connecting the file, KEYLENGTH, KEYSTART, and KEYEND return information about the keys in the file referred to by the file definition.

The LASTRECL specifier gives you the record length of the last record affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement.

The LASTKEY specifier gives you the key of reference of the last record affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement.

Considerations for Double-Byte Data

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, use NAMELIST formatting, or specify your own format. However, you cannot use internal I/O.

To ensure proper processing of formatted data, connect your file using the OPEN statement, specifying CHAR='DBCS'. CHAR='DBCS' is required for the following:

- List-directed input that might have double-byte text in character constants.
- NAMELIST input that contains 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 routine to preserve the balanced shift characters. For information about the ASSIGNM service routine, see *VS FORTRAN Version 2 Language and Library Reference*.

Note that I/O processing does not check whether your file contains valid double-byte data. For information on valid double-byte data, 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 specifer. 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. For an example of a compiler listing containing double-byte characters, see Appendix I, "Sample Compiler Listing with Double-Byte Characters" on page 451. The syntax notation is described under "Syntax Notation" on page iv.

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 = 'HOUSE<kk>'
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> HOUSE<kk>
```

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<.B>C'
WRITE (10,*) STR1 STR2
```

The following records are written:

```
10 15 FR<.E.D.'.S>
```

```
<kkkk> A<.B>C
```

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><.B>bb
B	AB<.C.'.S>

Chapter 7. Associating Data

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.

General Rules for 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.

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

Items shared in a common area are subject to the same rules as arguments passed in a subprogram argument list (see "General Rules for Arguments" on page 193).

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 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 55. Each column of variables starts at the beginning of the common area. Variables on the same line share the same storage locations.

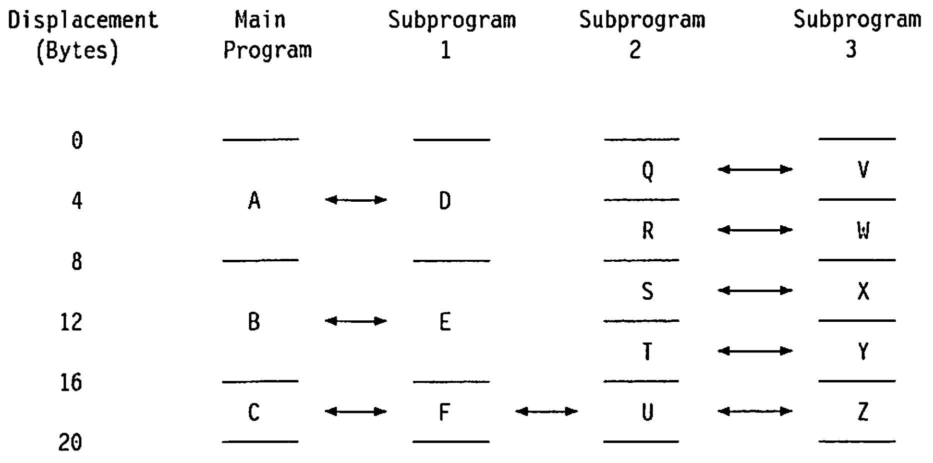


Figure 55. Transmitting Assignment Values between Common Areas

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. The same rules for preserving data values also apply; see especially "General Rules for Arguments" on page 193.

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 330.

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 or REAL
8	REAL
8	COMPLEX or DOUBLE PRECISION
4	REAL, INTEGER, or LOGICAL
2	INTEGER
1	LOGICAL

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
DOUBLE PRECISION	8	Doubleword
COMPLEX	8	Word
LOGICAL	1	Byte
INTEGER	2	Halfword
REAL	8	Doubleword
REAL	16	Doubleword
COMPLEX	16	Doubleword
COMPLEX	32	Doubleword

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
——	0
A	
——	4
K	
——	8
CMPLX	
——	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
——	0
A	
——	2
DV	
——	4
K	
——	8
CMPLX	
——	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 (Static and Dynamic)

There are two forms of common storage you can specify: blank common and named common.

- Blank Common – An unnamed common storage area (common block) is a *blank common* area, when you have not specified a name for the storage area.
- Named Common – When you name common storage areas (or blocks of storage)—they are known as *named common*. Blocks given the same name occupy the same space.

For more information, see "Using Dynamic Common above the 16-Megabyte Line" on page 92.

- You can define only one blank common block in an executable program (although you can specify many COMMON statements defining items in blank common). You cannot assign blank common a name.

  You can define many named common blocks, each with its own name.
- You can define blank common as having different lengths in different program units. You must define a given named common block with the same length in every program unit that uses it.
- You can't initialize values in variables or array elements in blank common using DATA statements.
- In named common, you can initialize values in variables and array elements, through a block data subprogram that contains DATA statements or explicit specification statements.

Dynamic common is useful in the MVS/XA and VM/XA environments for utilizing 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.

If a named common is declared as dynamic common, all program units sharing that common must declare it as dynamic in order for correct program references to the common to be established when the program is run.

For information on using dynamic common with MTF, see Appendix E, "The Multitasking Facility (MTF)" on page 349.

Initializing Named Common: 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.

Intercompilation Analysis

Intercompilation analysis provides a way to identify incompatibilities between program units—particularly in the specification of parameters passed to external procedures.

Introduction

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 60 on page 215), 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 59 on page 214.)

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. In addition, the lengths of named common blocks and the usage of external names are checked for consistency throughout the group of program units.

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
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)
CALL 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)
```

The compiler cannot diagnose certain ambiguous situations; for example, if the calling program unit passes XYZ(1), the compiler cannot tell 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.

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)
```

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.

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.

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. Although violation of this rule may not always produce errors at run time, it is dubious programming practice.

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

If a named common is declared as dynamic common, all program units sharing that common 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
```

When to Use the Intercompilation Analysis Feature

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.

How to Use Intercompilation Analysis

To use intercompilation analysis, specify the ICA compiler 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.

Notes on the USE and UPDATE Suboptions

The USE suboption specifies the names of intercompilation analysis files containing entries from previous compilations. 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.

The following example shows you how to use the USE and UPDATE suboptions:

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 UPD 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 and UPDATE 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 and UPDATE suboptions

Use care in the way you specify USE and UPDATE 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 and UPDATE suboptions.

Considerations for Intercompilation Analysis Files in CMS: The intercompilation analysis file name is the file name specified in the USE or UPDATE 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.

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 1 on page 15 shows the device type and device class for intercompilation analysis files; Figure 2 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. MSGON and MSGOFF are mutually exclusive: you cannot specify both.

Figure 56 lists the message numbers that you can specify for each MSGON and MSGOFF suboption, the corresponding message text, and an explanation of each message.

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
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
84	UNDEFINED ARGUMENT	Referenced in called; undefined in caller
86	INVALID ARGUMENT MODIFICATION	Modification of arguments sharing storage

Figure 56 (Part 1 of 2). Intercompilation Analysis Messages

Number	Text	Explanation
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

Figure 56 (Part 2 of 2). Intercompilation Analysis Messages

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
      RETURN
      END
```

Sample Programs Compiled with Intercompilation Analysis

Figure 57 on page 213 shows the source language coding for a group of program units to be compiled together, with ICA specified.

```
@PROCESS
C
C PROGRAM TO ILLUSTRATE USE OF THE ICA OPTION
C
      PROGRAM ICATEST
      REAL*8 R8S, R8A(8,3)
      COMPLEX*8 CX8
      CALL ICASUB1 (0,R8S)
      CALL ICASUB2 (CX8,4,R8A)
      CALL ICASUB3 (R8S,4)
      END
@PROCESS
C
      SUBROUTINE ICASUB1(X,Y,Z)
      COMMON /ICACOMM/ ARRAY(3,8), C16
      COMPLEX*16 C16
      REAL*8 Y
      IF (Z .EQ.0) THEN
         X = SIN(Y)
      ELSE
         X = SIN(Y/X)
      ENDIF
      Y = ICASUB2(C16,3,ARRAY) + X
      RETURN
      END
@PROCESS
C
      REAL FUNCTION ICASUB2(X,IA,Z)
      COMMON /ICACOMM/ ARRAY(3,8), C16
      REAL*8 ARRAY
      COMPLEX*8 X
      REAL*4 Z(8,3)
      DO 10 I = 1, IA
10       Z(I,2) = REAL(X) + ARRAY(IA,3)
      ICASUB2 = Z(8,3)
      RETURN
      END
```

Figure 57. ICATEST Input Listing

Figure 58 shows the options specified for ICATEST.

```
| LEVEL 2.3.0 (MAR  1988)       VS FORTRAN          JAN 12, 1988  07:40 :36                       PAGE:     1

| REQUESTED OPTIONS (EXECUTE):  ICA NOTERM

| REQUESTED OPTIONS (PROCESS):

| 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
|                      OPT(0) LANGLVL(77) NOFIPS   FLAG(I)  AUTODBL(NONE)  NAME(MAIN)     LINECOUNT(60)      CHARLEN(500)
```

| Figure 58. 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 59.

```
LEVEL 2.3.0 (MAR  1988)       VS FORTRAN          JAN 12, 1988  11:29:36                          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
                                     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

 NAME    | NO. |     TYPE       | ARGUMENT USAGE |                         REFERENCES
--------------------------------------------------------------------------------------------------------------
ICASUB1  |  2  | SUBROUTINE     | BBF            | ICASUB2

ICASUB2  |  3  | FUNCTION(R*4)  | FFB            |

ICATEST  |  1  | MAIN PROGRAM   |                | ICASUB1 ICASUB2 ICASUB3


 NAME    |     TYPE       |                         IS REFERENCED BY
--------------------------------------------------------------------------------------------------------------
ICACOMM  | COMMON         | ICASUB1(B) ICASUB2(F)

ICASUB1  | SUBROUTINE     | ICATEST

ICASUB2  | FUNCTION(R*4)  | ICASUB1 ICATEST

ICASUB3  | SUBROUTINE     | ICATEST
```

Figure 59. ICATEST Output Listing - External Cross Reference

The external cross reference consists of two tables. The first, at the top of the page, shows:

Column Heading	Contents
NAME	Truncated names of all analyzed program units
NO.	Number indicating the sequence of the module in the compilation. If the number is preceded by F, it identifies the inter-compilation analysis file containing the entries for the name in the NAME column.
TYPE	Program unit type
ARGUMENT USAGE	Argument usage (see the top of the listing for a description of the symbols).
REFERENCES	Truncated names of the program units referenced by each module

The second cross-reference table, the bottom half of the page, shows:

Column Heading	Contents
NAME	Names of subroutines, functions, entry points, block datas, and commons
TYPE	Type of program unit. If the referenced name is an entry in a subprogram, it is so indicated.
IS REFERENCED BY	Truncated names of the program units that reference the program unit specified in NAME

The messages in Figure 60 were generated during the analysis and indicate problems which can produce incorrect results when the application is run.

```
LEVEL 2.3.0 (MAR  1988)       VS FORTRAN         JAN 12, 1988  07:40 :36  NAME:MAIN                    PAGE:     5

*** INTERCOMPILATION MESSAGES. ***
 NUMBER   MODULE  LEVEL   ISN    VS FORTRAN ERROR MESSAGES

ILX0071I   ICIM    4(W)          CONFLICTING COMMON LENGTH -- THE LENGTHS OF COMMON BLOCK ICACOMM IN ICASUB1(COMPILATION 2), AND
                                 ICASUB2(COMPILATION 3) DO NOT AGREE. THE LENGTHS ARE 112 AND 196 BYTES.

ILX0062I   ICIM    4(W)          INCORRECT NO. OF ARGUMENTS -- THE NUMBER OF ARGUMENTS FOR ICASUB1 IN ICATEST(COMPILATION 1) AT
                                 ISN 4 IS 2. IN COMPILATION 2 IT IS SPECIFIED AS 3.

ILX0072I   ICIM    4(W)          MODIFIED CONSTANT ARGUMENT -- ARGUMENT NO. 1 TO ICASUB1 AT ISN 4 IN ICATEST(COMPILATION 1) IS
                                 PASSED AS A CONSTANT OR EXPRESSION. ICASUB1(COMPILATION 2) MAY MODIFY THIS ARGUMENT, "X".

ILX0092I   ICIM    4(W)          CONFLICTING TYPE -- AT THE MAIN ENTRY TO SUBROUTINE, ICASUB1(COMPILATION 2), ARGUMENT NO. 1, "X",
                                 IS EXPECTED TO BE REAL*4; HOWEVER, ICATEST (COMPILATION 1), AT ISN 4 PASSES THIS ARGUMENT, "A
                                 CONSTANT", AS INTEGER*4.

ILX0068I   ICIM    4(W)          CONFLICTING NAME USAGE -- THE NAME, ICASUB2, HAS BEEN REFERENCED AS A SUBROUTINE IN
                                 ICATEST(COMPILATION 1). IT IS DEFINED AS A FUNCTION IN ICASUB2(COMPILATION 3).

ILX0084I   ICIM    4(W)          UNDEFINED ARGUMENT -- ARGUMENT NO. 1, "X",  TO ICASUB2 IS REFERENCED BUT THE ARGUMENT PASSED AS
                                 "CX8" AT ISN 5 IN ICATEST MIGHT NOT BE DEFINED.

ILX0065I   ICIM    4(W)          CONFLICTING TYPE -- AT THE MAIN ENTRY TO FUNCTION, ICASUB2(COMPILATION 3), ARGUMENT NO. 3, "Z",
                                 IS EXPECTED TO BE REAL*4; HOWEVER, ICATEST (COMPILATION 1), AT ISN 5 PASSES THIS ARGUMENT,
                                 "R8A", AS REAL*8.

ILX0093I   ICIM    4(W)          ARGUMENT SIZE -- THE SIZE OF THE ARRAY IN ARGUMENT NO. 3, "R8A", TO ICASUB2 AT ISN 5 IN
                                 ICATEST(COMPILATION 1) DOES NOT AGREE WITH THE SIZE SPECIFIED IN COMPILATION 3 FOR "Z".

ILX0064I   ICIM    4(W)          FUNCTION TYPE -- FUNCTION, ICASUB2, REFERENCED IN ICASUB1 (COMPILATION 2) AT ISN 10 IS
                                 INTEGER*4.  IN COMPILATION 3 IT IS SPECIFIED AS REAL*4.

ILX0084I   ICIM    4(W)          UNDEFINED ARGUMENT -- ARGUMENT NO. 1, "X",  TO ICASUB2 IS REFERENCED BUT THE ARGUMENT PASSED AS
                                 "C16" AT ISN 10 IN ICASUB1 MIGHT NOT BE DEFINED.

ILX0065I   ICIM    4(W)          CONFLICTING TYPE -- AT THE MAIN ENTRY TO FUNCTION, ICASUB2(COMPILATION 3), ARGUMENT NO. 1, "X",
                                 IS EXPECTED TO BE COMPLEX*8; HOWEVER, ICASUB1 (COMPILATION 2), AT ISN 10 PASSES THIS
                                 ARGUMENT, "C16", AS COMPLEX*16.

ILX0074I   ICIM    4(W)          ARRAY DIMENSIONS -- THE SHAPE OF THE ARRAY IN ARGUMENT NO. 3, "ARRAY", TO ICASUB2 AT ISN 10 IN
                                 ICASUB1 (COMPILATION 2) DOES NOT AGREE WITH THAT SPECIFIED IN COMPILATION 3 FOR "Z".

******* INTRACOMPILATION STATISTICS ******* 0 DIAGNOSTICS GENERATED. HIGHEST SEVERITY CODE IS 0.

******* INTERCOMPILATION STATISTICS *(ICA)* 12 DIAGNOSTICS GENERATED.
```

Figure 60. ICATEST Output Listing - Compilation Messages

Chapter 8. Optimizing Your Program

Optimization requires additional compile time but usually results in reduced run time.

The OPTIMIZE compiler option permits selection of no optimization or one of three higher optimization levels. 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).

When you use these higher optimization levels, certain programming practices can help or hinder optimization.

This chapter discusses the optimization levels, debugging optimized programs, and programming practices that affect optimization.

Optimization Levels

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; however this is suppressed if the SDUMP option is in effect.

Optimization Level 1

OPTIMIZE(1) performs register and branch optimization, a modest level of optimization for programs without nested loops. Variables are retained in registers where possible to eliminate unnecessary loads and stores. Branching is improved using RX format branch instructions. 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 done for the entire program. This analysis allows optimizations such as common expression elimination, strength reduction, code motion, and global register assignment. Particular attention is paid to innermost loops and to subscript address calculations.

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.

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 an extra error signal is possible.

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.

Optimization Techniques

Several techniques are used by the optimizer. The more general techniques used by OPTIMIZE(2) and OPTIMIZE(3) are:

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.

Constant propagation and constant folding: Constants used in an expression are combined and new ones generated. Some mode conversions are done and evaluation of some intrinsic functions.

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.

Section oriented branching: The number of required program address registers is reduced by dividing the executable code in a very large program into sections.

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 *Interactive Debug* for more information on debugging optimized code.

Increasing Optimization of Your Program

The following section contains suggestions on how to use the optimization features.

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).

  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 the compiler must use an extra base register as a program address register that would otherwise be used for register optimization at higher optimization levels.
  - 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:

- Blocking a file can result in significant improvements.
- 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. The compiler "bundles" together up to 20 such items, and makes just one call to the I/O library.
- To save processing time, code implied DO loops in I/O statements to produce partial short-lists. The compiler is able to recognize certain combinations of implied-DOs and combine them into a partial short-list. Some examples of I/O statements recognized as partial short-lists 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)
  ```

  In the last example, the implied-DO level containing B is a partial short-list, 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, an efficient code sequence (including MVC or CLC) is generated for the first three statements (1 through 3). A less efficient code sequence (including MVCL or CLCL) 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'
```

Variables

- Use logical variables of length 4 because they can be accessed directly without clearing a register. Avoid using LOGICAL*1 variables.
- Always use integer variables of length 4 for DO loop indexes. Integer variables of length 4 are optimized by strength reduction; they're also generated into branch-on-index instructions. Avoid using INTEGER*2 variables.
- 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 optimized by register optimization in the loops that contain the statements.
  - Equivalenced variables cannot be optimized.
  - Variables in COMMON blocks cannot be optimized across subroutine calls.
  - Variables received as dummy arguments are difficult to optimize. Assign frequently referenced scalar dummy arguments to local vari-

ables. Remember that changing a local variable does not change the argument.

Do not use DO loop indexes in any of the above ways.

- Each reference to a variable in common requires that the address of the common block be in a register. This is the basis for the following recommendations.

  1. Minimize the number of common blocks. Group concurrently referenced variables into the same common block. For example:

Three Registers Required	**One Register Required**
`COMMON /X/ A`	`COMMON /Q/ A,B,C`
`COMMON /Y/ B`	`A=B+C`
`COMMON /Z/ C`	
`A=B+C`	

  2. Place scalar variables before arrays in a given common block. For example:

Two Registers Required	**One Register Required**
`COMMON /Z/ X(5000),Y`	`COMMON /Z/ Y, X(5000)`
`X(1)=Y`	`X(1)=Y`

  3. Place small arrays before large ones. All the scalar variables and the first few arrays can then be addressed through one address constant. The subsequent larger arrays probably each need a separate address constant.

  4. Assign frequently referenced scalar variables in a common block to a local variable. References to the local variable will not require the common block address to be in a register. You should be sure to assign the value back to the common variable at the end of processing.

- When you're accumulating intermediate summations, keep the result in a scalar variable rather than in an array. Array accumulators require load and store instructions; scalar variable accumulators can be maintained in a register.

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. The compiler recognizes only local variables as having a constant value. (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, the compiler calculates one subscript and uses it for all the arrays. The compiler can maintain one index for all the arrays defined as having the same dimensions.
- 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:
  1. 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	**Computation (I*N) Required**
`SUBROUTINE EXEC(Z,N)`	`SUBROUTINE EXEC(Z,N)`
`REAL *8 Z(9,N)`	`REAL *8 Z(N,9)`
`Z(I,5)=A`	`Z(5,I)=A`

  2. 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) compiler 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 compiler does not check that I is not larger than J and K is not larger than L. To request error checking, use the IL(NODIM) option.
- Initialize large arrays using a DO loop. You get faster overall run time and use less storage than if you initialize using a DATA statement. For example, the following statements:

  ```
  DOUBLE PRECISION A(5000)
  DATA A/5000*0.0D0/
  ```

  generate 40000 bytes of object module information—more than 500 TXT records. The 5000 zeros are placed in the object module, placed in the load module, and fetched into storage when you run the program.

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	No Duplicates Recognized
`A=B*(X*Y*Z)`	`A=B*X*Y*Z`
`C=X*Y*Z*D`	`C=X*Y*Z*D`
`E=F+(X+Y)`	`E=F+X+Y`
`G=X+Y+H`	`G=X+Y+H`

  the compiler can recognize X*Y*Z and X+Y as duplicate expressions because they're either coded in parentheses or coded at the left end of the expression.

- 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	Constant Expressions Not Recognized
`V*W*X*(C*D*E)`	`V*W*X*C*D*E`
`C+D+E+V+W+X`	`V+W+X+C+D+E`

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:

Not Using Factoring

```
  SUM=0.0
  DO 1 I=1,9
1 CONTINUE
  SUM=SUM+FAC*ARR(I)
```

Using Factoring

```
  SUM=0.0
  DO 1 I=1,9
1 CONTINUE
  SUM=SUM+ARR(I)
  SUM=SUM*FAC
```

In many programs, you can factor extensively.

Conversions

- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic. For example:

No Conversions Needed

```
      X=1.0
      DO 1 I=1,9
      A(I)=A(I)*X
1     CONTINUE
      X=X+1.0
```

Multiple Conversions Needed

```
      DO 1 I=1,9
1     CONTINUE
      A(I)+A(I)*I
```

When you must use mixed-mode arithmetic, code the fixed-point and floating-point arithmetic in separate computations as much as possible.

- Converting data between single and double precision requires two, or even three, instructions, so avoid conversions if possible.

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
1     CONTINUE
      A(I)=A(I)+Z*B(I)
```

Inefficient

```
      DO 1 I=1,9
1     CONTINUE
      A(I)=A(I)-2.0*B(I)
```

In division operations, do the following:

- For constants, use one of the following constructions:

  ```
  X*(1.0/2.0)
  0.5*X
  ```

  rather than the construction X/2.0.

- For a variable used as a denominator in several places, use the same technique.

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.

Chapter 9. Vectorizing Your Program

The VS FORTRAN Version 2 compiler can produce programs that use the IBM 3090 Vector Facility, a hardware feature that provides high-speed computation particularly suitable for scientific and engineering applications. The compiler transforms eligible statements in DO loops into vector instructions resulting in significantly faster run time.

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 DO loop, and can reduce run time dramatically. Figure 61 illustrates this reduction in run time.

```
   DO  8  K = 1, 90
8  CONTINUE
   A(K)=A(K)+B(K)
```

Traditional scalar (non-vector) 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 61. How Vector Processing Speeds Run Time

To make use of the IBM 3090 Vector Facility, you must specify the VECTOR compiler option. In addition, you must specify optimization at the OPTIMIZE(2) or OPTIMIZE(3) level. (OPTIMIZE(3) is the default.) If an optimization level is not specified, OPTIMIZE(3) will be assumed.

This chapter presents information on vector terminology, coding suggestions, sample output listings, and instructions on using vector directives. For additional information on the compiler options, see Chapter 2, "Compiling Your Program" on page 7. For information on vector report diagnostic messages, see Appendix F, "Vector Report Diagnostic Messages" on page 373.

Terminology

Some of the terms used in discussing vectorization are defined as follows:

vector

a group of elements of an array 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 DO 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),...,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),...,B(K,300) and B(1,M),B(2,M),...,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 DO loop that defines the vector. For example:

```
   DO 50 I = 1,100        DO 60 J = 1,100,10
50    B(I) = 0.060     60    A(J) = 0.0
```

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.

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 N x M array, the stride on the second subscript is N. Addressing FORTRAN arrays in row order is stride N.

Suppose you have a 4 x 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.

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 DO 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.

recurrence

a group of one or more statements forming a cycle of dependences. If a recurrence exists, vectorization is not performed. 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.

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.

loop distribution
the compiler's process of automatically restructuring DO loops so that statements originally contained within a single DO loop are placed into multiple loops, each with the same induction parameters as the original. Loop distribution is possible only when the statements involved are not part of a recurrence carried by the loop being distributed.

unanalyzable loop
a DO loop that is ineligible for vectorization because it contains some statement or construct that prevents the compiler from gathering the information needed for vectorization analysis. For a list of the types of DO loops that are unanalyzable, see "Analysis Eligibility Stage" on page 234.

unsupportable loop
a DO loop or portion of a DO 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. For a list of the types of DO loops that are unsupportable, see "Operations Support Stage" on page 235.

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 DO loop variables (that is, their increments are controlled by assignment statements within a loop) are referred to as *auxiliary induction variables*.

noninductive subscript
a subscript expression in which the array elements referenced on successive iterations of a DO loop are not separated by a constant number of bytes. Examples are:

```
    DO 10 I = 1,N
10    A(I*I) = 0.0
```

and

```
    DO 20 J = 1,N
20    B(INDEX(J)) = 0.0
```

scalar expansion
the compiler's process of automatically replacing references to a scalar variable with references to a vector temporary, thereby allowing 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 247.

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. The compiler converts the control dependence to a data dependence using a technique called IF-conversion.

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 stores 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 is 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 compiler's 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 DO loop, the loop controls are left in place, but changed to increment by Z instead of by 1.

Eligibility of DO Loops for Vectorization

A DO loop must pass four qualification stages before it can be compiled into code which can be run in vector mode:

Analysis Eligibility Stage: The compiler determines whether a DO loop can be analyzed. The compiler analyzes only DO loops. A loop can have more than one inner loop at the next level; however, only the eight innermost levels of a nest are analyzed.

A loop cannot be analyzed if it contains:

- Any branches out of a loop, around an inner loop, or backwards within a loop.
- Loops other than DO loops
- Loops with a loop index or iteration control expression other than INTEGER*4
- Loops with induction variables mentioned in EQUIVALENCE statements
- I/O statements
- ASSIGN, ENTRY, RETURN, PAUSE, or STOP statements
- Computed or assigned GO TO statements
- Subroutine calls
- External, non-intrinsic function references
- CHARACTER data

Recurrence Detection Stage: The compiler determines the vectorization eligibility of statements within any DO loop not previously rejected. The statements are grouped into regions. Each region consists of a group of statements forming a recurrence carried by a particular DO loop. (The extent of these regions is indicated in the XLIST vector report.) A loop is rejected if it contains:

- Interchange-preventing dependences. This restriction does not apply if the loop is already at the innermost level.

- An induction variable modifying inner DO loop parameters or inner auxiliary induction variables.

- Unbreakable recurrences. A recurrence may be breakable if some of its dependences involve variables that are eligible for scalar expansion.

Operations Support Stage: The compiler determines the vectorization eligibility of any region not previously rejected. It bases its decision on whether the constructs found in the region have vector support in the compiler and hardware. Any region that uses in any of the following operations or constructs is rejected for vector processing.

- LOGICAL*1 fetches or stores
- INTEGER*2 fetches or stores governed by an IF-statement
- REAL*16 or COMPLEX*32 operations
- Noninductive subscripts governed by an IF-statement
- Noninductive subscripts to an INTEGER*2 array
- Intrinsic in-line functions from the following families: DIM, MOD, SIGN, NINT, ANINT, or BTEST.
- Some occurrences of intrinsic in-line MIN and MAX functions using INTEGER
- Intrinsic functions using REAL*16 or COMPLEX*32
- Intrinsic functions when NOINTRINSIC is specified
- Relational expressions that need to be stored (for example, L=A.GE.B)
- Misaligned data

Vectorization Selection Stage: The compiler selects for vectorization some of the regions not rejected as ineligible by the above criteria. Selection is based on cost estimates (in CPU cycles) of processing each region in vector and scalar mode. Some regions, though eligible for vectorization, will be run in scalar mode because it is more economical to do so.

Vectorizable Mathematical Functions

Most VS FORTRAN Version 2 intrinsic functions and mathematical operations can be vectorized. The INTRINSIC | NOINTRINSIC suboption of the VECTOR compiler 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 34.

Some of the intrinsic functions and mathematical operations take advantage of the vector hardware (see Figure 62), while others are evaluated using scalar code but accept vector arguments and return vector results (see Figure 63).

Out-of-Line Intrinsic Functions:	SQRT	DLOG	DTAN	CDABS
	DSQRT	SIN	DCOTAN	ALOG10
	EXP	DSIN	ATAN	DLOG10
	DEXP	COS	DATAN	ATAN2
	ALOG	DCOS	CABS	DATAN2
In-Line Intrinsic Functions:	HFIX	IFIX	COMPLX	SNGL
	IABS	INT	DCMPLX	DBLE
	ABS	IOR	CONJG	AMAX1
	DABS	ISHFT	DCONJG	DMAX1
	IAND	NOT	FLOAT	MAX1
	IBCLR	AIMAG	DFLOAT	AMIN1
	IBSET	DIMAG	DPROD	DMIN1
	IDINT	AINT	REAL	MIN1
	IEOR	DINT	DREAL	
Mathematical Operations:	REAL*4 raised to a REAL*4 power			
	REAL*8 raised to a REAL*8 power			

Figure 62. Intrinsic Functions and Mathematical Operations that Use Vector Hardware

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	
	SINH	DLGAMA	CDEXP	
	DSINH	TANH	CLOG	
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			
	COMPLEX*8 divide			
	COMPLEX*16 divide			

Figure 63. Intrinsic Functions and Mathematical Operations Evaluated Using Scalar Code

Producing Vector Reports

This section describes the main features of the output reports produced using the REPORT suboption on the VECTOR compiler option. The VECTOR compile option is discussed on page 33.

Reports can contain several different sections and can be displayed on a terminal or printed, depending on what you specify on the VECTOR option.

Displaying A Report on a Terminal

To display a report on your terminal, specify the REPORT(TERM) suboption on the VECTOR option. The TRMFLG option must also be in effect.

The terminal report, an example of which is shown in Figure 64, displays only selected portions of the vectorized 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.

To the far left of the DO statements are flags indicating the following:

VECT	Loop was selected for vectorization.
SCAL	Loop was analyzed and chosen to run in scalar mode.
UNAN	Loop was unanalyzable.

Also to the left of the statements are nested brackets that mark the beginning and end of each loop in every block of DO loops that was analyzable for vectorization. The brackets indicate how the statements were grouped after the vectorization selection stage (see "Vectorization Selection Stage" on page 235). These brackets are the only way to accurately determine the relative nesting of the statements in the vector report. In a typical report, the DO loop distribution (which involves displaying multiple copies of a single DO statement) makes it impossible to express the nesting level using normal FORTRAN rules. Therefore, to avoid confusion, labels for these statements are not printed.

```
SCAL  +-------       DO 200 J=1,100
      |                SUM=0.0
      |   *DIR         IGNORE RECRDEPS(B)
VECT  |+------         DO 100 I=1,100,2
      ||                 A(I,J) = B(I+N,J+N) + B(I+N,J-N)
      ||                 C(I,J) = B(I-N,J+N) + B(I-N,J-N)
      ||                 B(I,J) = A(I,J) + C(I,J)
      ||______           SUM = SUM + ABS(A(I,J)-C(I,J))
      |_______         DIFSUM(J) = SUM

UNAN                 DO 300 J=1,100

THE DO-LOOPS HAVE BEEN PROCESSED AS INDICATED.
```

Figure 64. Sample Vector Report, Displayed on a Terminal

Printing Reports

Figure 65 on page 243 shows a printed listing with a vector report produced by specifying the following suboptions on the VECTOR option. In addition, the IGNORE directive, discussed on page 258, was present.

```
VECTOR(REPORT(LIST XLIST SLIST STAT)
```

You can specify the suboptions on the VECTOR option in any order but the sections of the report will always be printed in the order shown in the figure.

The listing shown in Figure 65 on page 243 has the following sections:

1 The options requested and the options in effect.

2 A vectorization analysis report produced by REPORT(LIST) on the VECTOR option.

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 vectorization.

Note that optimization takes place before vectorization and alters parts of the program. As a result, when the vector 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 mapping statements in the vector report to statements in the source program listing.
- Flags next to DO statements, indicating the following:

VECT	Loop was selected for vectorization.
SCAL	Loop was analyzed and chosen to run in scalar mode.
UNAN	Loop was unanalyzable.

- Nested brackets marking the beginning and end of each loop in every block of DO loops that was analyzable for vectorization. The brackets indicate how the statements were grouped after the vectorization selection stage (see "Vectorization Selection Stage" on page 235). These brackets are the only way to accurately determine the relative nesting of the statements in the vector report. In a typical report, the DO loop distribution (which involves printing multiple copies of a single DO statement) makes it impossible to express the nesting level using normal FORTRAN rules. Therefore, to avoid confusion, labels for these statements are not printed. Labels are printed with statements that are not analyzable.
- Source statements.

3 An extended vectorization analysis report produced by REPORT(XLIST) on the VECTOR option.

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 vectorized. 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 identified during the recurrence detection stage of vectorization analysis (for information on the stages of vectorization analysis, see "Eligibility of DO Loops for Vectorization" on page 234).

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 vectorization 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 mapping statements in the vector report to statements in the source program listing.
- Flags next to DO statements, indicating the following:

VECT	Loop was selected for vectorization.
ELIG	Loop was eligible for vectorization but was chosen to run in scalar mode (usually because of cost determination).
RECR	Loop was not vectorized because it carries a recurrence.
UNSP	Loop was not vectorized because it contains operations not supported by either the compiler or the vector hardware.
UNAN	Loop was unanalyzable.

- Nested brackets marking the beginning and end of each loop in every block of DO loops that was analyzable for vectorization. The brackets indicate how the statements were grouped after the recurrence detection stage (see "Recurrence Detection Stage" on page 235). These brackets are the only way to accurately determine the relative nesting of the statements in the vector report. In a typical report, the DO loop distribution (which involves printing multiple copies of a single DO statement) makes it impossible to express the nesting level using normal FORTRAN rules. Therefore, to avoid confusion, labels for these statements are not printed. Labels are printed with statements that are not analyzable.
- 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 vectorization analysis report showing the entire source program. This is produced by REPORT(SLIST) on the VECTOR option. 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 the statements were vectorized. In the first column are characters, indicating the following:

V	Loop was partially or completely vectorized.
S	Loop was run in scalar mode.
U	Loop was unanalyzable.
v	Statement was vectorized.
s	Statement was run in scalar mode.

  In the second column are numbers, indicating the vector analysis depth of the loop (or for statements, of the loop chosen as the vector sectioning loop for the statement). The vector analysis depth is 1 for each outermost analyzable loop and increases by 1 for each nested loop up to a maximum depth of 8.

- 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 vector report messages. These are produced by either REPORT(XLIST) or REPORT(SLIST) on the VECTOR option.

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 vectorized. However, some messages, issued for loops that were vectorized, highlight situations where vectorization may change the result of a computation. Other messages clarify certain ambiguities in the vector report listing. Messages also appear when 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 F, "Vector Report Diagnostic Messages" on page 373 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 that might help increase vectorization are discussed. 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 statistics table produced by REPORT(STAT) on the the VECTOR option. The table displays the iteration count for each analyzable DO 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 vector report to statements in the source program listing.
- Array name or DO loop induction variable.
- Up to eight columns, one for each level of DO 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 DO 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**." Strides that cannot be fully determined at compile time are followed by "**?**."

    or:
  - For a reference to a DO 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 258.) 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 the REPORT(XLIST) on the VECTOR option.

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.

  EQUDEP A potential dependence between two variables in an EQUIVALENCE relationship that has been eliminated because of an IGNORE EQUDEPS directive.
- The names of arrays carrying dependences.

- The names of other arrays involved in dependences that have been eliminated by an IGNORE EQUDEPS 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.

8 General diagnostic and informative messages.

1
```
LEVEL 2.3.0 (AUG  1988)        VS FORTRAN          AUG 24, 1988  13:20:37                                  PAGE:    1
REQUESTED OPTIONS (PROCESS):  VEC(REP(TERM LIST XLIST SLIST STAT) IVA) OPT(3)  DIR('DIR')
OPTIONS IN EFFECT: NOLIST NOMAP NOXREF NOGOSTMT NODECK NOSOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) NOSXM IL(DIM) NOTEST NODC NOICA DIRECTIVE NODBCS NOSAA
                      OPT(3) LANGLVL(77) NOFIPS   FLAG(I)  AUTODBL(NONE)  NAME(MAIN)    LINECOUNT(60)      CHARLEN(500)
                      VECTOR (  IVA    INTRINSIC    REDUCTION  SIZE(ANY)     REPORT(LIST TERM XLIST SLIST STAT))
```

2
```
LEVEL 2.3.0 (AUG  1988)        VS FORTRAN          AUG 24, 1988  13:20:37  NAME:EXP01                      PAGE:   2
                                            REPORT(LIST)   VECTORIZATION 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
                    C
0007 SCAL +-------              DO 200 J=1,100
0008      |                       SUM=0.0
          |         *DIR          IGNORE RECRDEPS(B)
0009 VECT |+------                DO 100 I=1,100,2
0010      ||                        A(I,J) = B(I+N,J+N) + B(I+N,J-N)
0011      ||                        C(I,J) = B(I-N,J+N) + B(I-N,J-N)
0012      ||                        B(I,J) = A(I,J) + C(I,J)
0013      ||______                  SUM = SUM + ABS(A(I,J)-C(I,J))
0014      |_______                DIFSUM(J) = SUM
                    C
0015 UNAN                       DO 300 J=1,100
0016                 300          IF (DIFSUM(J).GT.1.0) PRINT *,J,DIFSUM(J)
0018                            RETURN
0019                            END
```

3
```
LEVEL 2.3.0 (AUG  1988)        VS FORTRAN          AUG 24, 1988  13:20:37  NAME:EXP01                      PAGE:    3
                                            REPORT(XLIST)  VECTORIZATION 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 ELIG +-------              DO 200 J=1,100                                   SCALAR FASTER THAN VECTOR   -48
0008      |                       SUM=0.0
          |         *DIR          IGNORE RECRDEPS(B)
0009 VECT |+------                DO 100 I=1,100,2                                "IGNORE RECRDEPS" USED      -72
0013      ||______                  SUM = SUM + ABS(A(I,J)-C(I,J))                VECTOR SUM REDUCTION        -50
0014      |_______                DIFSUM(J) = SUM

0007 RECR +-------              DO 200 J=1,100
          |         *DIR          IGNORE RECRDEPS(B)
0009 VECT |+------                DO 100 I=1,100,2                                "IGNORE RECRDEPS" USED      -72
0010      ||                        A(I,J) = B(I+N,J+N) + B(I+N,J-N)              OFFSET UNKNOWN              -18
          ||                                                                      INTERCHANGE PREVENTING DEP  -23
          ||                                                                      POTENTIAL RECRDEP ELIMINATED-77
0011      ||                        C(I,J) = B(I-N,J+N) + B(I-N,J-N)              OFFSET UNKNOWN              -18
          ||                                                                      INTERCHANGE PREVENTING DEP  -23
          ||                                                                      POTENTIAL RECRDEP ELIMINATED-77
0012      ||______                  B(I,J) = A(I,J) + C(I,J)                      OFFSET UNKNOWN              -18
          |_______                                                                INTERCHANGE PREVENTING DEP  -23
                                                                                  POTENTIAL RECRDEP ELIMINATED-77
                    C
0015 UNAN                       DO 300 J=1,100                                    I/O OPERATION               -04
0016                 300          IF (DIFSUM(J).GT.1.0) PRINT *,J,DIFSUM(J)
0018                            RETURN
0019                            END
```

Figure 65 (Part 1 of 3). Printed Listing Including Vector Report

4

```
LEVEL 2.3.0 (AUG  1988)        VS FORTRAN          AUG 24, 1988  13:20:37  NAME:EXP01                     PAGE:    4
                                             REPORT(SLIST) VECTORIZATION 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
S1               7              DO 200 J=1,100                                             SCALAR FASTER THAN VECTOR   -48
s        1       8                SUM=0.0
                        *DIR      IGNORE RECRDEPS(B)
V2       1       9                DO 100 I=1,100,2                                         "IGNORE RECRDEPS" USED      -72
v2       2      10                  A(I,J) = B(I+N,J+N) + B(I+N,J-N)                       OFFSET UNKNOWN              -18
                                                                                           INTERCHANGE PREVENTING DEP  -23
                                                                                           POTENTIAL RECRDEP ELIMINATED-77
v2       2      11                  C(I,J) = B(I-N,J+N) + B(I-N,J-N)                       OFFSET UNKNOWN              -18
                                                                                           INTERCHANGE PREVENTING DEP  -23
                                                                                           POTENTIAL RECRDEP ELIMINATED-77
v2       2      12                  B(I,J) = A(I,J) + C(I,J)                               OFFSET UNKNOWN              -18
                                                                                           INTERCHANGE PREVENTING DEP  -23
                                                                                           POTENTIAL RECRDEP ELIMINATED-77
v2       2      13    100           SUM = SUM + ABS(A(I,J)-C(I,J))                         VECTOR SUM REDUCTION        -50
s        1      14    200         DIFSUM(J) = SUM
                        C
U               15              DO 300 J=1,100                                             I/O OPERATION               -04
         1      16    300         IF (DIFSUM(J).GT.1.0) PRINT *,J,DIFSUM(J)
                18              RETURN
                19              END
```

5

```
LEVEL 2.3.0 (AUG  1988)        VS FORTRAN          AUG 24, 1988  13:20:37  NAME:EXP01                     PAGE:    5
NUMBER     ISN        FLAG  VS FORTRAN VECTOR REPORT MESSAGES

ILX0148I  0007        ELIG  CODE THAT WAS ELIGIBLE TO EXECUTE IN VECTOR MODE WAS DETERMINED TO EXECUTE MORE EFFICIENTLY IN SCALAR.

ILX0172I  0009        VDIR  AN "IGNORE RECRDEPS" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP.

ILX0118I  0010-0012   RECR  THE OFFSET NEEDED TO ADDRESS THE ARRAY(S) "B" 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) "1".

ILX0123I  0010-0012   RECR  THE ARRAY(S) "B" CARRY FORWARD DEPENDENCES AT NESTING LEVEL(S) "1" THAT MAY BE INTERCHANGE PREVENTING.

ILX0177W  0010-0012   VDIR  POTENTIAL BACKWARD DEPENDENCE(S) INVOLVING THE ARRAY(S) "B" HAVE BEEN IGNORED 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.

ILX0104I  0015        UNAN  ONE OR MORE I/O STATEMENTS OCCUR AT ISN(S) "17".
```

Figure 65 (Part 2 of 3). Printed Listing Including Vector Report

```
6
LEVEL 2.3.0 (AUG  1988)       VS FORTRAN          AUG 24, 1988  13:20:37  NAME:EXP01                           PAGE:    6
                                                   VECTOR STATISTICS TABLE

 ISN ARRAY/INDUCTION               LEVEL 1   LEVEL 2

   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                                 100         2V
     C                                 100         2V
  14 DIFSUM                              1

7
LEVEL 2.3.0 (AUG  1988)       VS FORTRAN          AUG 24, 1988  13:20:37  NAME:EXP01                           PAGE:    7
                 TABLE OF DEPENDENCES ELIMINATED BY IGNORE DIRECTIVES

DIRECTIVE    ARRAY NAME    FROM   TO    DEP     AFFECTED       LOOP
  TYPE      (EQUIV NAME)   ISN    ISN   TYPE    DIMENSIONS     VARIABLES

BACKDEP       B             12     10   TRUE    1              I
BACKDEP       B             12     11   TRUE    1              I

8
LEVEL 2.3.0 (AUG  1988)       VS FORTRAN          AUG 24, 1988  13:20:37  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 = 162688 BYTES, PROGRAM NAME = EXP01    PAGE:    1.
*STATISTICS* 1 DIAGNOSTIC GENERATED. SEVERITY CODE IS 0.
**EXP01** END OF COMPILATION 1 ******                                                                   TIME: 13:20:37
```

Figure 65 (Part 3 of 3). Printed Listing Including Vector Report

Gathering Run-Time Statistics

By using the Interactive Vectorization Aid function of Interactive Debug, you can collect run-time statistics on the vector length and stride of each DO loop. The timing and sampling facilities provide information on the relative efficiency of each DO loop.

To make use of the Interactive Vectorization Aid, you must specify the IVA suboption on the VECTOR compiler option, described on page 34, and the SDUMP compiler option, described on page 31. For more information, see vector tuning information in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Examples of Vectorization

The following examples show how the compiler vectorizes typical sequences of FORTRAN statements. The associated vector 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.

```
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 DO 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
```

The following vector 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, the compiler can place them in 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 array (appearing only in a vector register). As you can see, the entire loop has been vectorized. Scalar expansion cannot occur if a scalar variable uses a value set before the loop, or if it computes a value needed after the loop.

```
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 can be used to convert control dependences into data dependences, allowing analysis for possible vectorization. Following is an example of a vector 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
```

As a result of vectorization, the following vector 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
```

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 =  ... + S + ....` 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 267.

An example of reduction operation recognition is given below. Vector code will be generated for the DO 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).

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
```

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 Vector Directives" on page 253.

Statements Preventing Vectorization

Inside DO loops, avoid using any of the statements or constructs listed under "Eligibility of DO Loops for Vectorization" on page 234.

Subscripts

- When the same array is either both fetched and stored or stored more than once in a loop, make sure that subscripts are simple functions of DO 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 DO 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
```

  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 DO loop indexes.

- Avoid using variables mentioned in EQUIVALENCE statements.
- Avoid using variables that are not induction variables in subscript expressions.

DO loops

- The compiler analyzes only DO loops for vectorization. However, the compiler can recognize some auxiliary induction variables as well as the DO loop control variable. Vectorization of a loop is prevented when it contains an induction variable:
  - Modifying inner DO loop parameters or auxiliary induction variables
  - Whose increment cannot be proven to be non-zero
  - Mentioned in EQUIVALENCE statements
- The compiler does not vectorize any loop with a backward branch, a branch out of the loop, a branch around an inner loop, or branches caused by computed or assigned GOTO statements. Other types of conditional or unconditional forward branches may be used in the loop.
- Vectorization requires preservation of the meaning (semantics) of the original DO 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=1,32,4
                                A(N+0)=B(N+0)*C(N+0)+D(N+0)
                                A(N+1)=B(N+1)*C(N+1)+D(N+1)
  DO 1 N=1,32                   A(N+2)=B(N+2)*C(N+2)+D(N+2)
1 A(N)=B(N)*C(N)+D(N)         1 A(N+3)=B(N+3)*C(N+3)+D(N+3)
```

  When a short DO 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 DO 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 256) 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, the compiler uses the fixed default value 65. For example:

```
     REAL*4 A(20,100,*)
     REAL*4 B(50,500,200)

:
     DO 100 I=1,N1              <== assumed count is 20
     DO 100 J=1,N2              <== assumed count is 10
     DO 100 K=1,N3              <== assumed count is 65 (the default)
        A(I,J,K)     = 0.0
        B(I,J*50,K*M) = 0.0
100  CONTINUE
```

  If the upper limit of an array is large but the actual size is significantly smaller than the limit, it is best to specify the ASSUME COUNT directive to make effective use of vector cost analysis.

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 greater than IHIGH and IHIGH+INC is within the range of the dimensions of B and C, 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
```

Temporary Variables

Scalar variables, used to hold intermediate results, can be vectorized as long as the scalars are local to the loop in which they are used. This means that they are not in COMMON, do not use values that could have been set before the loop, and do not define values that may be used after the loop.

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 IBM 3090 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.) The principal reason for this is that, on the 3090, data is moved between the registers and main memory by way of a high speed buffer, or cache. If the memory reference pattern is such that data remains in the cache for as long as it is needed, good performance will be achieved. If, on the other hand, data must be moved between cache and main memory multiple times during processing of a nest of loops, some performance degradation may be seen.

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 chose 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
5       A(K,J,I) = B(K,J,I)
```

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
    5       A(I,J,K) = B(I,J,K)
```

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. Thus, if the outer loop is vectorized, we will have the largest possible vector length along with the shortest possible stride.

In some cases, non-unit 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. Due to the mechanism that determines how long a piece of data will remain in the cache, 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 hardware (because a vector instruction in an outer loop is processed less frequently than one in 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.

Using Vector Directives

Occasionally the VS FORTRAN Version 2 compiler does not make the desired vectorization decisions. You can use vector directives to override or influence the compiler's decisions. However, while vector directives can improve vectorization, you must use them with caution. If the vectorization performed by the compiler is sufficient, or you can improve vectorization by recoding, do not use vector directives.

The vector directives are:

ASSUME COUNT	specifies a value that is to be used for vector cost analysis when a DO loop iteration count cannot be determined at compile time.
IGNORE	instructs the compiler to ignore specified dependences in a DO loop.
PREFER	specifies that a DO loop be processed in vector, if eligible, or scalar mode, regardless of decisions made by vector cost analysis.

Applications

Problem areas that might benefit from the use of vector directives are:

1. When the iteration count of a loop cannot be determined at compile time, vector cost analysis algorithms might make an incorrect estimate of the loop count. Use the ASSUME COUNT vector directive to specify a value.

2. In determining whether to process a loop in vector or scalar mode, vector cost analysis algorithms might make an undesirable decision. Use the PREFER vector directive to request that particular loops be run in vector or scalar 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 apply dependence testing algorithms. In such cases it must assume that dependences exist. In the example:

```
      DO 10 I = 1,100
10      A(I+IBASE) = A(I)
```

   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 vector 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.

Interactions between 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.

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 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.

Specifying Vector Directives

To specify vector directives, you code them in your source program, preceding the DO loops to which they apply. In addition, you must specify the DIRECTIVE compiler option using an @PROCESS statement. The DIRECTIVE compiler option is discussed on page 28 in Chapter 2, "Compiling Your Program."

To code a 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 compiler 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 vector directive. The syntax for each vector directive is given in the following sections.

The vector 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 compiler option, in which `*VDIR:` is the trigger-constant, and an ASSUME COUNT vector directive:

```
@PROCESS DIRECTIVE('*VDIR:')
⋮
C*VDIR:  ASSUME COUNT(3)
```

Local and Global Vector Directives

A vector directive that affects only the first DO loop that follows it is called a *local vector directive*. A vector directive that affects multiple DO loops is called a *global vector directive*. Only the ASSUME COUNT and PREFER SCALAR vector directives can be used as global vector 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 or PREFER SCALAR vector 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 Vector Directives

The following rules apply when you have multiple 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, and will continue to be in effect for subsequent loops.
- Multiple ASSUME COUNT and PREFER local directives are not allowed for a single DO loop. If two ASSUME COUNT local directives or two PREFER local directives are specified for the same DO 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 258.

ASSUME COUNT Directive

ASSUME COUNT specifies a value that is to be used for vector cost analysis when DO loop iteration counts cannot be determined at compile time.

If you specify ASSUME COUNT for a loop with a known iteration count, the vector 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 code. At worst, an incorrectly specified ASSUME COUNT directive results in increased run time.

ASSUME COUNT only affects the vector cost analysis phase of the compiler. The information is not used to determine the existence of dependences or recurrences.

Syntax

Local 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. 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, vectorization may still not occur, for the following reasons:

- The existence of dependences that can not be overridden.
- The existence of unanalyzable and unsupportable 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).

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 systems programmer to check whether the IGNORE directive is enabled for your installation.

By specifying REPORT(XLIST) on the VECTOR compiler option, you can produce a table of ignored dependences in the vector 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 241.

Syntax

IGNORE { RECRDEPS[(*array-list*)][EQUDEPS[(*array-list*)]] | EQUDEPS[(*array-list*)] [RECRDEPS [(*array-list*)]] }

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 231.

Loops identified by the flag "RECR" in the XLIST vector report are candidates for using this directive to increase vectorization. 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 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. If you omit the array-list, the directive applies to all applicable arrays within the loop.

Examples of IGNORE RECRDEPS

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 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 265.

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
   20   A(K+77) = A(K) * B(K)
```

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)
   40       J = J + M
```

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
   50       A(I+M1) = A(I) * B(I)
```

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
   60       A(I+M1) = A(I+M2) * B(I)
```

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 on page 266.) Verification is possible in this case because the same indexing array is used on both sides of the equation.

```
          IF (DUPIND(J, 1000, 0))
     X      CALL DIRERR(70,72,'Independent Indirect Indexes')

C*VDIR:  IGNORE RECRDEPS(A)
          DO 70 I = 1, 1000
   70       A(J(I)) = A(J(I)) * B(I)
```

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))
     X   CALL DIRERR(80,72,'Independent Indirect Indexes')
        DO 79 I = 1, 1000
          IF (J(I).GT.I .AND. J(I).LE.1000)
     X      CALL DIRERR(80,72,'Independent Indirect Indexes')
   79   CONTINUE

C*VDIR:  IGNORE RECRDEPS(A)
        DO 80 I = 1, 1000
   80     A(J(I)) = A(I) * B(I)
```

Example 5c: ***Indirect Addressing*** In the following example, two different indexing arrays are used.

```
C*VDIR:  IGNORE RECRDEPS(A)
        DO 90 I = 1, 1000
   90     A(JLEFT(I)) = A(JRIGHT(I)) * B(I)
```

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
```

EQUDEPS

instructs the compiler to assume that no dependences arise between variables in an EQUIVALENCE group. Two variables in an EQUIVALENCE group may share common storage. This causes the compiler to assume that every reference to one of the variables is dependent on every reference to the other. IGNORE EQUDEPS instructs the compiler to ignore such dependences.

IGNORE EQUDEPS acts on every applicable array in the loop unless qualified by an array-list.

array-list

is an optional list of array variable names separated by commas. The names must fit on one line and must not extend beyond column 72. If you cannot fit all the names 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, you need 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.

Examples of IGNORE EQUDEPS

Example 1: In the following example, the subscript expressions are such that a value stored into variable A will never be fetched later through variable B.

```
      DIMENSION A(1000), B(1000)
      EQUIVALENCE (A(1), B(1))

C*VDIR:  IGNORE EQUDEPS(A)
         DO 120 K = 1, 100
    120    A(K) = B(K+1) * 10.0
```

Example 2: In the following example, because the loop is too short to reference the storage locations where variables A and B overlap, no vectorization preventing dependence can occur.

```
      DIMENSION A(100), B(1000)
      EQUIVALENCE (A(1), B(101))

C*VDIR:  IGNORE EQUDEPS(A)
         DO 130 K = 1, 100
   130    A(K) = B(K) * 10.0
```

Example 3: 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. To guard against this situation, use verification code as shown in the IGNORE RECRDEPS examples.

```
      DIMENSION A(100), B(1000)
      EQUIVALENCE (A(1), B(1,01))

       IF (.NOT.(N .LE. 100)
     +   CALL DIRERR(140, 73, 'N .LE. 100')

C*VDIR: IGNORE EQUDEPS(A)
       DO 140 K = 1, N
  140    A(K) = B(K) * 10.0
```

PREFER Directive

PREFER specifies that particular DO loops be run in vector or scalar mode, regardless of decisions made by vector cost analysis.

Use PREFER only after carefully studying run times of a loop in vector and scalar 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 Directive:

PREFER {SCALAR | VECTOR}

Global Directive:

PREFER SCALAR ON

PREFER SCALAR OFF

PREFER {SCALAR | VECTOR}
begins a PREFER SCALAR or PREFER VECTOR local directive.

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.

PREFER SCALAR ON
begins a PREFER SCALAR global directive, which specifies that following loops be run in scalar mode.

PREFER SCALAR OFF
cancels the preceding PREFER SCALAR ON statement.

Examples of PREFER

Example 1: In this example, PREFER VECTOR is used to vectorize a loop that the compiler might run in scalar mode.

```
         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
    100    IF(B(I).NE.0) THEN A(I) = B(I)*C(I)* ...
```

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
```

Verifying Correct Application of Directives

To verify that a vector directive is applied correctly, you can write a subroutine similar to the one in Figure 66 on page 266. 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 66. Sample Routine to Report an Invalid Directive

The example in Figure 67 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, JBASE)
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 JBASE is the minimum index value possible.
C All J(*) values are greater than or equal to JBASE.
C
C Bit array B used to check index values:
C dimension is BSIZE, 32 bits per word.
      INTEGER BSIZE
      PARAMETER (BSIZE = 1000)
C Allow for 32000 distinct values.
      INTEGER B(BSIZE)
      INTEGER J(N)
C
      DUPIND = .FALSE.
C Initialize Bit array
      DO 1 K = 1, BSIZE
    1   B(K) = 0
C  Now, test each value of J(*) in turn
      DO 2 K = 1, N
        IF (J(K) .LT. JBASE) THEN
          WRITE (*,7) K, JBASE
  7       FORMAT('- *** Indirect index No.',I6,' is below',I12)
          STOP 'DUPIND 1'
        ENDIF
C Calculate word number M and bit number L
        M = 1 + (J(K) - JBASE) / 32
        IF (M .GT. BSIZE) THEN
          WRITE (*,8) K, 32*BSIZE+JBASE
  8       FORMAT('- *** Indirect index No.',I6,
     X           ' exceeds Max',I12)
          STOP 'DUPIND 2'
        ENDIF
        L = MOD((J(K) - JBASE), 32)
```

Figure 67 (Part 1 of 2). Sample Routine to Check Indexing Arrays for Duplicate Values

```
C See if that value has been previously observed
        IF (BTEST(B(H),L)) THEN
          DUPIND = .TRUE.
          WRITE (*,9) K
  9       FORMAT('- *** Indirect index No.',I6,' is repeated')
          RETURN
        ELSE
C Indicate that value has been observed
          B(H) = IBSET(B(H),L)
        ENDIF
    2 CONTINUE
      END
```

Figure 67 (Part 2 of 2). Sample Routine to Check Indexing Arrays for Duplicate Values

Considerations and Restrictions for Vectorization

Restrictions limiting vectorization are discussed in the following sections.

Vector Versus Scalar Summation

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.

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
1     S = S + A(K,J,I)
```

different sums are possible in scalar, and vector processing for each of the I, J, or K loops. All are algebraically equivalent.

Version 2 Versus Version 1 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, as there are no vector entry points in the old routines.

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 34.

Subscript Values and Array Bounds

The VS FORTRAN Version 2 compiler assumes that subscripts remain inside array dimensions. LANGLVL(77) requires that every array subscript be within its corresponding dimension declaration. LANGLVL(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
1        A(I,2) = A(I,1)
```

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 10. Creating Reentrant Programs

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.

Detailed information about creating reentrant programs under VM is found in "Creating and Using a Reentrant Program under VM" on page 277. Detailed information for MVS is found in "Creating and Using a Reentrant Program under MVS" on page 291.

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 only be used 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 68 shows this.

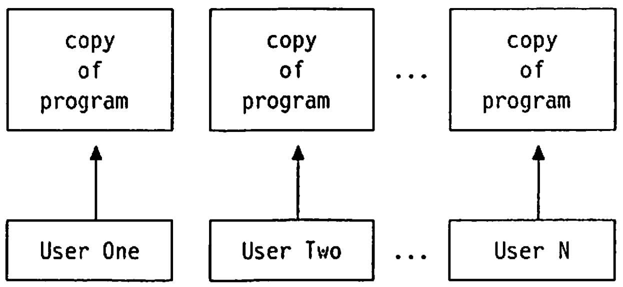


Figure 68. Nonreentrant Program Requires Multiple Copies for Concurrent Use

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 compiler option, you request a program structured this way. Figure 69 on page 270 shows the concept of sharing.

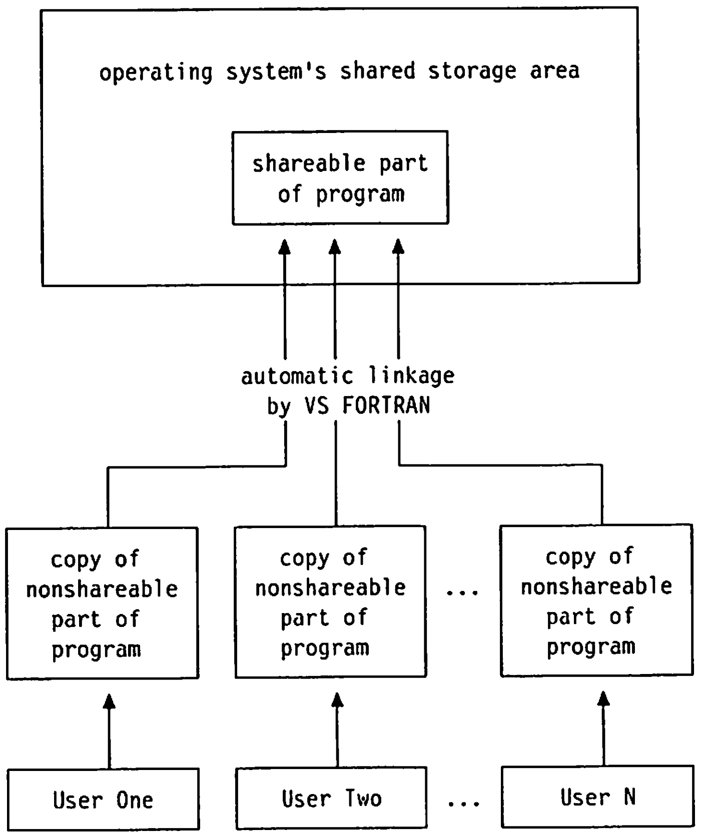

operating system's shared storage area
shareable part of program
automatic linkage by VS FORTRAN
copy of nonshareable part of program
copy of nonshareable part of program
...
copy of nonshareable part of program
User One
User Two
...
User N

Figure 69. 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.

This loading of the shareable part is done automatically by the VS FORTRAN Version 2 run-time library, as is the communication between the program parts, so no special coding or assembler language interface is necessary — only the normal FORTRAN CALL to the subprogram.

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 compiler 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 70 on page 272 shows the input to and output from the separation tool program in the case of a single object module.

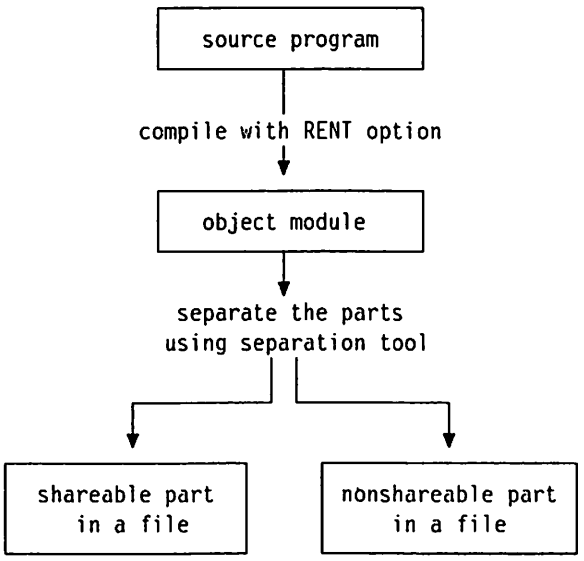


Figure 70. Using The Separation Tool on a Single Program

Figure 70 shows a very simple case. However, your applications will probably involve more than one source program unit, and may also involve nonreentrant program units as well. For such cases, the separation tool can accept multiple object files. Figure 71 on page 273 shows a more complex situation, and provides greater detail about the input to and output from the separation tool.

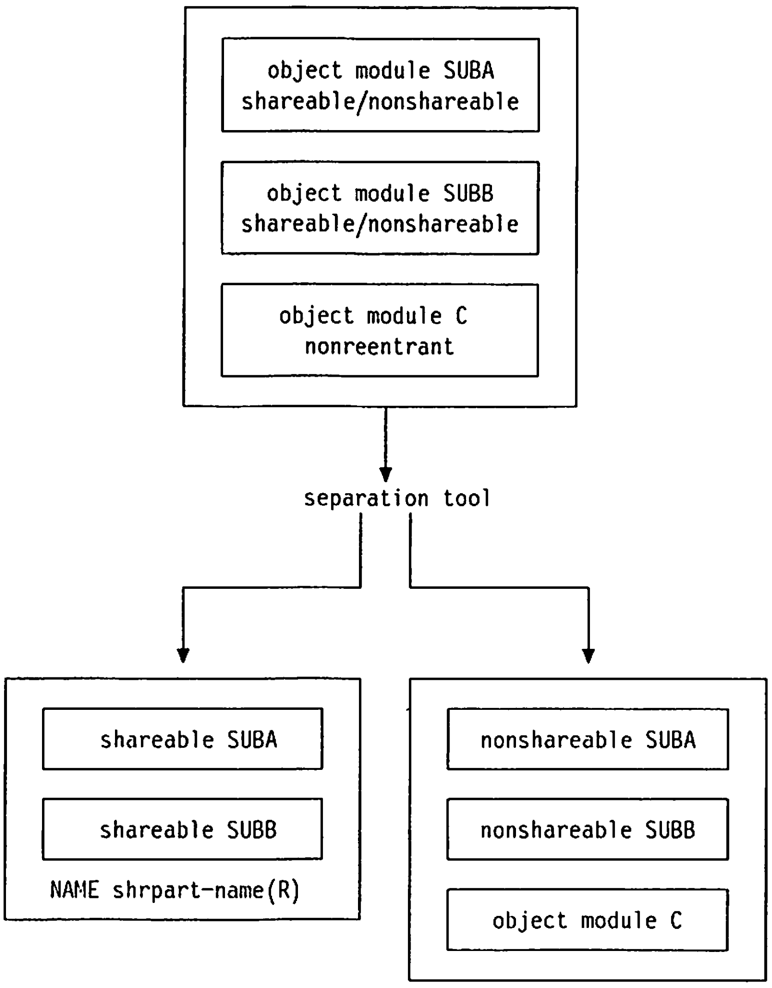


Figure 71. Using The Separation Tool on Multiple Programs with the Assigned Name Form

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 71, 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 71 on page 273 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 72 on page 275 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.

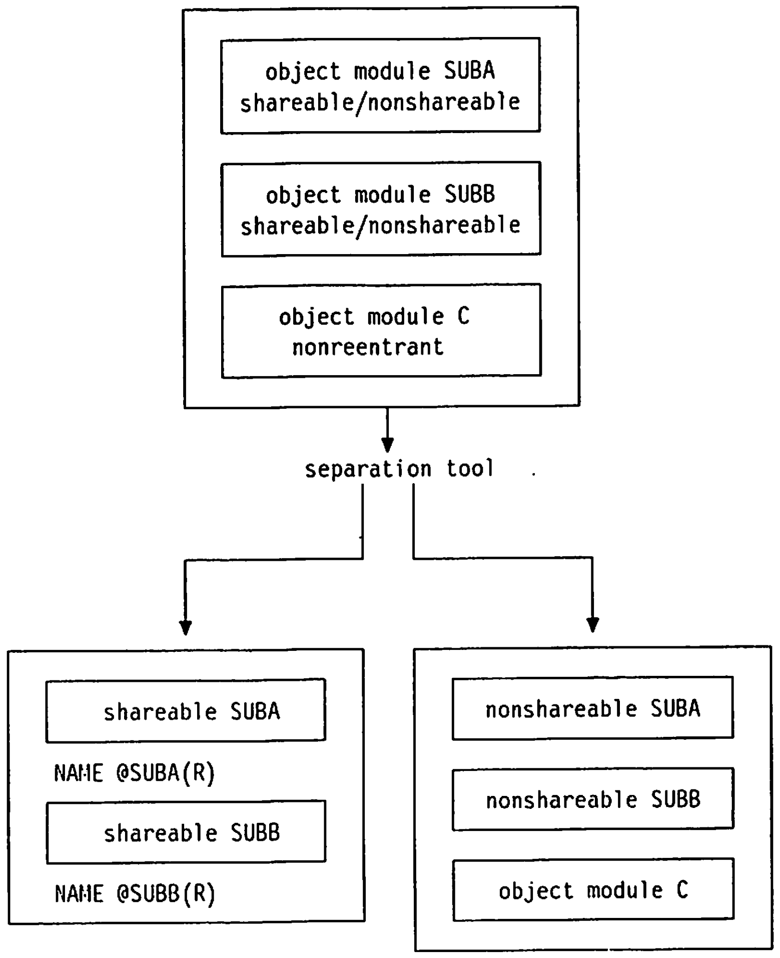


Figure 72. Using the Separation Tool with the Default Name Form

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 discontiguous shared segment (DCSS) for VM, 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.

Summary of Steps to Create and Use a Reentrant Program

To prepare a reentrant program for sharing, complete all the steps below. If you do not want to share the shareable part of the program but want to take advantage of the dynamic loading capability of reentrancy, you can omit step 6.

Detailed information about the procedures for each step under VM is included in "Creating and Using a Reentrant Program under VM" on page 277. Detailed information for MVS is included in "Creating and Using a Reentrant Program under MVS" on page 291.

1. **Design and Code**

   Design and code your program as you would normally. You need do nothing different for a program that will be reentrant.

2. **Compile**

   When you compile your program unit, request a reentrant version of your object module by specifying the RENT compiler option. The compiler then produces an object module composed of two parts: the shareable code and the nonshareable 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 make 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 library subroutines.

3. **Separate the Two Parts**

   This step is not done when preparing a regular, nonreentrant program, but is necessary if you want to share a reentrant program or use the shareable part for dynamic loading.

   Using the separation tool program provided as part of the VS FORTRAN Version 2 product, separate the object module into its nonshareable and shareable parts.

4. **Prepare an Executable Program from the Nonshareable Parts**

   The nonshareable parts of the program need to be prepared as usual to be run. These parts can be regular nonreentrant program units as well as the nonshareable parts of reentrant program units.

   Under MVS, you must link-edit the parts. Under VM, you have two options: putting the text files into a TXTLIB or creating a member in a CMS LOADLIB.

5. **Link-edit the Shareable Parts**

   Under MVS, link-edit the shareable parts into a library. Under VM, use the LKED command to put the shareable parts into one or more members of a

CMS LOADLIB. Only the shareable parts produced by the separation tool can be link-edited in this step.

6. **Install the Shareable Parts in a DCSS or LPA**

   This, too, is an extra step not done when preparing a nonreentrant program. If you do **not** want to share the shareable part of your program, you can skip this step and go on to the next one.

   To be shared, the shareable part of the program must be placed in the operating system's shareable area. Under MVS, this is the LPA. Under VM, this is the DCSS. This is a relatively involved step that requires system programmer assistance.

7. **Run the Program**

   Invoke the program in the normal way by specifying the name of the nonshareable module. From your perspective as a user, there is no difference between running a reentrant program and a regular, nonreentrant one.

Creating and Using a Reentrant Program under VM

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 compiler 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" on page 278

"3b. Invoking the Separation Tool" on page 279

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 become 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.

"CMS EXEC Files to Run the Separation Tool" on page 285 shows examples of two EXECs that can be used to perform this separation step. VSF2RCS compiles a FORTRAN program with the RENT compiler option and then separates the nonshareable and shareable parts. VSF2RSEP separates the nonshareable and shareable parts from a TEXT file that was created previously by a compilation with the RENT compiler option. (The input may also include object modules from nonreentrant program units.)

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," below.

As with any program, you have several options of preparing a program to be run:

Form of the Stored Program	Command to Create an Executable Program
TEXT files or TXTLIB members	`LOAD MYPROG ...`
Nonrelocatable MODULE file	`GENMOD MYPROG`
CMS LOADLIB member	`LKED MYPROG`

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 and Running It" on page 60.

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 is 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 during processing of the program.

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

For the shareable parts of a program to 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: Run the Program" on page 285.

This step consists of three parts:

"6a. Gathering the Necessary Information"
"6b. Defining the DCSS to VM/SP" on page 283
"6c. Storing the Parts in the DCSS" on page 284

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*, SC19-6201, 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*, SC19-6201, and *VM/SP System Programmer's Guide*, SC19-6203.) 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
        VSYSADR=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*, SC24-5237.
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: 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 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.

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:

Action	Procedure	Figure
Compile and separate	VSF2RCS	Figure 73
Separate	VSF2RSEP	Figure 74

VSF2RCS

The VSF2RCS EXEC, shown below, compiles a reentrant program unit and then invokes the separation tool to separate the nonshareable and shareable parts. To run the EXEC, issue the following command.

```
VSF2RCS ftnname shrpart (options ...
```

Figure 73 on page 286 explains all of the parameters and their possible values. The options ASGNNAME or DEFNAME control your choice of assigned name form or default name form, respectively. For example, you might use the following command to compile the program unit MYPROG and separate it using the assigned name form:

```
FOVSRCS MYPROG SHRPART (ASGNNAME
```

```
&CONTROL OFF NOMSG
&GOTO -START
*
*  COMPILE A REENTRANT FORTRAN PROGRAM AND SEPARATE THE NONSHAREABLE
*    AND SHAREABLE PARTS
*
-RULES
&BEGTYPE
This EXEC performs the following functions:
  1. Compiles a VS FORTRAN program using the RENT option.
  2. Separates the object deck from the compilation into its
     nonshareable and shareable parts.

Run it as follows:

&END
&TYPE &EXEC ftnname shrpart (options, ...
&BEGTYPE

 ftnname   is the file name of the FORTRAN source program.  This file
           must have a file type of FORTRAN.

 shrpart   is the file name of the TEXT file into which all of the
           shareable parts are to be placed.  This name must be
           different from "ftnname," above.  If the assigned name
           form of this file is chosen (ASGNNAME option, below), then
           this TEXT file name also becomes the name of the LOADLIB
           member or of the DCSS that will be built later to contain
           the shareable parts.

 options   may be any VS FORTRAN compiler options.  RENT need not be
           given since it is provided automatically by this EXEC.
           In addition, one of the following may be coded to specify
           the form of the sharaable parts file.  This controls the
           names of the LOADLIB members or of the DCSSs that will
           contain the shareable parts.  If neither form is given,
           the assigned name form is assumed.

             ASGNNAME   specifies the assigned name form.  In this
                        case, the shareable parts file will be
                        structured such that the shareable parts of
                        all of the reentrant programs will be combined
                        into one LOADLIB member or one DCSS with a
                        name of "shrpart," above.

             DEFNAME    specifies the default name form.  In this
                        case, the shareable parts file will be
                        structured such that a separate LOADLIB
                        member or DCSS will be created for each
                        reentrant program that is compiled.  These
                        member or DCSS names consist of the program
                        names prefixed 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.

&END
&EXIT  &RCVALUE
-START
```

Figure 73 (Part 1 of 2). VSF2RCS — Compile Reentrant Program and Separate the Parts

```
*
*  CHECK FOR VALID OR OMITTED PARAMETERS
*
&RCVALUE = 0
&SEPPARM  =   ASGNNAME
&IF  .&1     EQ  .?   &GOTO -RULES
&IF  .&1     EQ  .    &GOTO -RULES
&IF  &INDEX  LT  2    &GOTO -BADINP
&IF  &INDEX  EQ  2    &GOTO -ENDOPT
&IF  .&3     NE   &GOTO -BADINP
&IF  &INDEX  EQ  3    &GOTO -ENDOPT
*
*  LOOK FOR THE OPTIONS ASGNNAME OR DEFNAME
*
&P  =  4
-NEXTOPT
&IF  &P   GT  &INDEX     &GOTO -ENDOPT
&IF  &&P  EQ  ASGNNAME   &GOTO -CLRPARM
&IF  &&P  EQ  ASSGNNAM   &GOTO -CLRPARM
&IF  &&P  EQ  ASGNAME    &GOTO -CLRPARM
&IF  &&P  EQ  ASSGNAME   &GOTO -CLRPARM
&IF  &&P  NE  DEFNAME    &GOTO -INCRPAR
-SETPARM
&SEPPARM =  &&P
-CLRPARM
&&P  =
-INCRPAR
&P  =  &P + 1
&GOTO -NEXTOPT
-ENDOPT
*
*  ENSURE THAT THE INPUT SOURCE FILE NAME DIFFERS FROM THE
*  SHAREABLE TEXT FILE NAME
*
&IF  &1  NE  &2  &GOTO -INOUT
&BEGTYPE
The input FORTRAN file name must differ from the output shareable
TEXT file name.
&END
&EXIT 12
-INOUT
*
*  COMPILE THE PROGRAM
*
&F  =  FORTVS2
&R  =  RENT
&F &1 (&R &4 &5 &6 &7 &8 &9 &10 &11 &12 &13 &14 &15 &16 &17 &18 &19 &20 &21 &22
&IF  &RETCODE  GT  4    &EXIT &RETCODE
*
*  SEPARATE THE NONSHAREABLE AND SHAREABLE PARTS
*
EXEC  VSF2RSEP  &1 &2 (&SEPPARM
&EXIT &RETCODE
*
*  INCORRECT INPUT PARAMETERS
*
-BADINP
&BEGTYPE
Invalid input parameter format.

&END
&RCVALUE  =  8
&GOTO -RULES
```

Figure 73 (Part 2 of 2). VSF2RCS — Compile Reentrant Program and Separate the Parts

VSF2RSEP

The VSF2RSEP EXEC 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 inptext shrpart (option
```

Figure 74 explains all of the parameters and their possible values. The options ASGNNAME or DEFNAME control your choice of assigned name form or default name form, respectively. For example, you might use the following command to separate a TEXT file using the assigned name form:

```
VSF2RSEP PROGTEXT SHRPART (ASGNNAME
```

```
&CONTROL OFF NOMSG
&GOTO -START
*
*  SEPARATE THE NONSHAREABLE AND SHAREABLE PARTS THAT ARE IN A
*    TEXT FILE CREATED BY A PREVIOUS VS FORTRAN COMPILATION THAT
*    WAS DONE WITH THE "RENT" OPTION
*
-RULES
&BEGTYPE
This EXEC separates a TEXT file produced by the VS FORTRAN compiler
into two parts:  one with the nonshareable parts and the other with
the shareable parts.  Run it as follows:

&END
&TYPE  &EXEC inptext shrpart (option
&BEGTYPE

 inptext   is the file 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 plus all of the non-
           reentrant programs that are found in the input.

 shrpart   is the file name of the TEXT file into which all of the
           shareable parts are to be placed.  This name must be
           different from "inptext," above.  If the assigned name
           form of this file is chosen (ASGNNAME option, below), then
           this TEXT'file name also becomes the name of the LOADLIB
           member or of the DCSS that will be built later to contain
           the shareable parts.

 option    may be either of the following to specify the form of the
           shareable parts file.  This controls the names of the
           LOADLIB members or of the DCSSs that will contain the
           shareable parts.  If neither form is specified, the
           assigned name form is assumed.

            ASGNNAME   specifies the assigned name form.  In this
                       case, the shareable parts file will be
                       structured such that the shareable parts of
                       all of the reentrant programs found in the
                       input will be combined into one LOADLIB member
                       or one DCSS with a name of "shrpart," above.

            DEFNAME    specifies the default name form.  In this
                       case, the shareable parts file will be
                       structured such that a separate LOADLIB
                       member or DCSS will be created for each
                       reentrant program that is found in the input
                       TEXT file.  These member or DCSS names
                       consist of the program names prefixed by @.
```

Figure 74 (Part 1 of 3). VSF2RSEP — Separate Nonshareable and Shareable Parts

```
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.

&END
&EXIT  &RCVALUE
-START
*
*  CHECK FOR VALID OR OMITTED PARAMETERS
*
&RCVALUE = 0
&SEPPARM  =  ASGNNAME
&IF  .&1     EQ  .?   &GOTO -RULES
&IF  .&1     EQ  .    &GOTO -RULES
&IF  &INDEX  LT  2    &GOTO -BADINP
&IF  &INDEX  EQ  2    &GOTO -ENDOPT
&IF  &3      NE  (    &GOTO -BADINP
&IF  &INDEX  EQ  3    &GOTO -ENDOPT
*
*  LOOK FOR THE OPTIONS ASGNNAME OR DEFNAME
*
&P  =  4
-NEXTOPT
&IF  &P   GT  &INDEX     &GOTO -ENDOPT
&IF  &&P  EQ  ASGNNAME   &GOTO -INCRPAR
&IF  &&P  EQ  ASSGNNAM   &GOTO -INCRPAR
&IF  &&P  EQ  ASGNAME    &GOTO -INCRPAR
&IF  &&P  EQ  ASSGNAME   &GOTO -INCRPAR
&IF  &&P  EQ  DEFNAME    &GOTO -SETPARM
&GOTO -BADINP
-SETPARM
&SEPPARM  =  &&P
-INCRPAR
&P  =  &P + 1
&GOTO -NEXTOPT
-ENDOPT
*
*  ASSURE THAT THE INPUT TEXT FILE EXISTS
*
STATE  &1 TEXT A
&IF  &RETCODE  EQ  0   &GOTO -ITEXTOK
&TYPE The file &1 TEXT A does not exist.
&EXIT 12
-ITEXTOK
*
*  ENSURE THAT THE INPUT TEXT FILE NAME DIFFERS FROM THE SHAREABLE
*  TEXT FILE NAME
*
&IF  &1  NE  &2  &GOTO -INOUT
&BEGTYPE
The input TEXT file name must differ from the output shareable
TEXT file name.
&END
&EXIT 12
-INOUT
```

Figure 74 (Part 2 of 3). VSF2RSEP — Separate Nonshareable and Shareable Parts

```
*
*  DETERMINE WHETHER TO USE THE ASSIGNED NAME
*
&IF  &SEPPARM  NE  ASGNNAME   &GOTO -ENDRPAR
&LMNAME  =  &2
-ENDRPAR
*
*  FILES NEEDED BY THE SEPARATION TOOL PROGRAM
*
FILEDEF  SYSIN     DISK  &1 TEXT     A
FILEDEF  SYSPRINT  DISK  &1 SEPLIST  A
FILEDEF  SYSUT1    DISK  &1 TEXTNSHR A
FILEDEF  SYSUT2    DISK  &2 TEXT     A
FILEDEF  SYSUT3    DISK  &1 TEMP     A
*
*  RUN THE SEPARATION TOOL PROGRAM
*
AFBVRSEP &LMNAME
*
*  ENSURE THAT SEPARATION WAS SUCCESSFUL
*
&IF  &RETCODE  EQ  0   &GOTO -SEPOK
&BEGTYPE
Errors occurred while running the separation tool.
&END
&RCVALUE  =  &RETCODE
ERASE  &1 TEXTNSHR A
ERASE  &2 TEXT     A
ERASE  &1 TEMP     A
&GOTO -FINAL
-SEPOK
*
*  LEAVE REQUIRED OUTPUT FILES IF SEPARATION WAS SUCCESSFUL
*
ERASE  &1 TEMP     A
ERASE  &1 TEXTORIG A
RENAME &1 TEXT     A   &1 TEXTORIG A
RENAME &1 TEXTNSHR A   &1 TEXT     A
*
*  CLEAR FILEDEFs THAT WERE SET AND THEN STOP
*
-FINAL
FILEDEF  SYSIN     CLEAR
FILEDEF  SYSPRINT  CLEAR
FILEDEF  SYSUT1    CLEAR
FILEDEF  SYSUT2    CLEAR
FILEDEF  SYSUT3    CLEAR
&EXIT  &RCVALUE
*
*  INCORRECT INPUT PARAMETERS
*
-BADINP
&BEGTYPE
Invalid input parameter format.

&END
&RCVALUE  =  12
&GOTO -RULES
```

Figure 74 (Part 3 of 3). VSF2RSEP — Separate Nonshareable and Shareable Parts

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 10, "Creating Reentrant Programs" on page 269. 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 compiler 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 292

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.

This option is preferable 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 in "MVS Cataloged Procedures for the Separation Tool" on page 295. 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.

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=&&SHRPART,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

If you will be running your reentrant program on an MVS/XA 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."

To enable the shareable parts of your program to be shared, your MVS systems programmer must place 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 systems 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 293, 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.

MVS Cataloged Procedures for the Separation Tool

Figure 75 shows cataloged procedures that you can use to compile, separate, link-edit, and runyour 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.

Action	Procedure	Figure
Compile, separate, and link	VFT2RCL	
Compile, separate, link, and go	VFT2RCLG	
Separate, link, and go	VFT2RLG	

Figure 75. IBM-Supplied Cataloged Procedures for the Separation Tool

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 an 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 295.

Link-Editing and Running a Reentrant Program under 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 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 following CLIST. 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 76. 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 94.

  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 *noshrpgm*, for example:

```
CALL mylib(noshrpgm)
```

Chapter 11. Using VSAM with VS FORTRAN Version 2

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

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 (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.

Processing VSAM Files

VSAM is an access method for files on direct access storage devices. Like non-VSAM files, VSAM can be used in three basic ways:

- To load a data set
- To retrieve a data set
- To update a data set

VSAM processing has these advantages:

- Data protection against unauthorized access, including password protection for VSAM files
- 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.

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. (VS FORTRAN Version 2 does not support all VSAM functions.)

VSAM Terminology

VSAM terminology is different from the terminology used for MVS files, as shown in Figure 77 on page 301, for example.

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

Figure 77. VSAM Terminology

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 sets (clusters). This process includes creating an entry in a VSAM or 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
- For KSDS, 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. You shouldn't try to use VSAM files as "scratch" files, because VSAM files are more difficult to allocate and erase than other files.

The following general programming considerations apply to VSAM files:

- Use VSAM KSDSs for applications in which you want to access data in a number of ways, and want to update or insert records at any point.
- Use VSAM ESDSs for applications in which you 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 RRDSs for work files, or for files in which records must be created and later updated in place.
- 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*, GC26-4135.

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. (However, an ESDS file alternate index cannot be accessed by VS FORTRAN Version 2, although VSAM allows such indexing.)

For example, an employee file might be indexed by personnel number, by name, and also by department number.

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 limits of index record size defined, any number of records can have the same key.

In the example suggested above, 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" on page 306).

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.

How to Build and Use Alternate Index Paths

If you are using alternate indexes, knowledge of how to use them is required at four stages of the programming process, as it is with normal data sets. These stages are:

1. When planning and coding the program
2. When creating the alternate indexes
3. When running the program that accesses the data set through the alternate indexes
4. When deleting the alternate index, if you wish to delete it at a different time from the associated data set

Discussions of what to do at these stages follow.

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 (may use 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.

Loading Your VSAM KSDS

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.)

Using Operating System Data Definition Statements

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 VM system. The name that identifies a particular data definition statement is called a *ddname* in MVS and VM.

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.

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. However, 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 table below indicates the required names.

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.

In the event that 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.

Separate FORTRAN keyed files (that is, those that are opened with different unit numbers) must not involve the same base cluster, either through the primary key or through one of its alternate index keys, if any of the files which are to remain open at the same time were opened with an ACTION parameter having a value other than READ. Violation of this restriction may cause unexpected or undesirable results when file updates are made.

The following table lists the names required to open a single FORTRAN keyed file.

DDNAME No.	VM		MVS	
	FILE=*fn*	No FILE=	FILE=*fn*	No FILE=
1	*fn*	FT*nn*K01	*fn*	FT*nn*K01
i (*i* > 1) Note 1.	*fn* suffixed with *m* Note 2.	FT*nn*K0*i*	*fn* suffixed with *m*	FT*nn*K0*i*

In the table:

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 VM 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.

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 "Job Processing" on page 12.

For VM: To define a VSAM file to VM, 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 78 summarizes the FORTRAN input/output statements you can use with each form of access.

Access Mode and FORTRAN I/O Statements	**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[6]	Yes	Yes[4]	Yes[7]
REWRITE	No	No	No	Yes
DELETE	No	No	No	Yes
READ	Yes	No[2] Yes[3]	Yes	Yes
BACKSPACE	Yes	Yes[5] Yes[3]	No	Yes
REWIND	Yes	Yes[5] Yes[3]	No	Yes
CLOSE	Yes	Yes	Yes	Yes

Figure 78. FORTRAN Statements Valid with VSAM Files

Notes to Figure 78 on page 310:

1 Sequential OPEN
2 Empty file
3 Nonempty file
4 Update or replace
5 For a file that was empty when opened, has the effect of CLOSE
6 Add a new record to the end of the file
7 Add or insert a new record

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, see Chapter 6, "Performing Input/Output Operations" on page 121.

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'
```

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 non-empty 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 6, "Performing Input/Output Operations" on page 121.

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. During file loading, the BACKSPACE statement has the same effect as a CLOSE statement, followed by an OPEN statement, followed by file positioning to the last record written; the last record in the file is then available for retrieval.

Using Direct Access—VSAM Direct Files

You can 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. Then code the WRITE statement, using the preset REC variable.
4. Repeat steps 2 and 3 until you've processed all the records you need to process.

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.

For specific information about VS FORTRAN Version 2 statement usage in processing VSAM KSDS files, see "Input/Output Operations for Keyed Access" on page 181.

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.

Part 4. Appendixes

Appendix A. Assembler Language Considerations

FORTRAN and assembler language can both be used in the same application, either as a FORTRAN main program with assembler subprograms or vice versa.

In FORTRAN programs, you can invoke the assembler subprogram in either of two ways: through CALL statements or through function references in arithmetic expressions. 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.

This appendix describes calling FORTRAN subprograms from assembler programs, invoking a FORTRAN main program, requesting compilation of a FORTRAN program from an assembler program, and retrieving arguments in an assembler program. This appendix also describes the representation of data in VS FORTRAN.

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# in dynamically loaded routines when you do not know if initialization has occurred in the original module.

Before Release 4 of FORTRAN Version 1, the call to initialize the run-time environment was made to a special entry point within VSCOM#. The call to VSCOM# is still supported, but you might significantly reduce the size of the load module by calling VFEIN# or VFEIL# instead.

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 may 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, the module must not be deleted. 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 or not a previous initialization has occurred. Therefore, the use of VFEIL# allows you to perform initialization without having to determine whether or not 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 assembly 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 assembly language calls for VFEIN# and VFEIL# with no run-time options are:

```
SR      1,1
L       15,=V(VFEIN# or VFEIL#)
BALR    14,15
```

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 VS FORTRAN Version 2 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.

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
- CALL EXIT statement
- CALL SYSRCx statement

When the run-time environment is terminated, control returns to the routine that invoked the routine that effected initialization. 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 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.

The only way you can run FORTRAN programs in different MVS subtasks in the same region is to use the VS FORTRAN Multitasking Facility (MTF).

Register Conventions

When you call a FORTRAN subprogram, you must follow the standard linkage conventions. In particular:

- 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 at which control is to 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.

Assuming that register 13 already points to a save area, your call should look similar to this:

```
         LA     1,parmlist
         L      15,=V(fortran-subprogram)
         BALR   14,15
         .
         .
         .
parmlist DC     A(argument1)
         .
         .
         .
         DC     A(argumentn+X'80000000')
```

If the FORTRAN subroutine processes a RETURN statement, register 15 will contain a zero on return to the calling routine. If a "RETURN i" statement is processed, register 15 will contain the value 4*i.

Passing Character Arguments

The linkage convention for passing character arguments between subprograms is different from the linkage convention for passing noncharacter 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 noncharacter 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, enter a sequence of one-word addresses pointing to the length attributes of each argument in the list. 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: In cases with both character and noncharacter arguments, 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.

The following example illustrates the linkage convention of a call to a subprogram with three character arguments.

The following example is given under the assumption 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)
```

Invoking a FORTRAN Main Program

There may be times when you wish to invoke 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 invoking a FORTRAN main program are shown below:

```
         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'execution-options'
```

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. There should be no embedded blanks in the parameter string.

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, load a new copy into virtual storage.

Assembler Subprograms to Be Called from FORTRAN

When you write an assembler language subprogram to be called from FORTRAN, it should expect the same register conventions as previously described in "Register Conventions" on page 321. That is, when the assembler language program is entered:

- Register 13 is pointing to an 18-word save area.
- Register 14 is pointing to the location to which control should be returned when the subprogram has completed.
- Register 15 is pointing to the entry point of the subprogram.
- Register 1 is pointing to an argument address list. If there is no argument list, register 1 contains zero.

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 will call other subprograms.

The usual form of these instructions is as follows:

```
routine SAVE   (14,12),,*
        DROP   15
        LR     12,15
        USING  routine,12
        LA     15,save-area
        ST     15,8(,13)
        ST     13,4(,15)
        LR     13,15
```

A function subprogram must return its function value as indicated under "Returning a Function Value from an Assembler Program" on page 328.

Using FORTRAN Data in Assembler Subprograms

Your assembler language subprograms can use data defined in FORTRAN subprograms, data that is contained either in common areas or in argument lists.

Arrays used in FORTRAN programs should be efficiently aligned. Alignment is described in the section on COMMON and EQUIVALENCE statements. If an inefficiently aligned array is passed to a subroutine as a dummy argument, each time the actual 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 blank and named common areas.

Using Blank Common Data in Assembler Programs: To refer to the blank common area, the assembler language program must also define a blank common area, using the COM assembler instruction. Only one blank common area is generated; 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 Named Common Data in Assembler Programs: To refer to named common areas, your assembler program should use an external A-type address constant:

```
         EXTRN name-of-common-area
comaddr  DC    A(name-of-common-area)
```

Requesting Compilation from an Assembler Program

VS FORTRAN Version 2 can be invoked through the use of the CALL, ATTACH, or LINK macro instructions that are used as part of 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 VM).
- The ddnames of the MVS data sets or VM files to be used during processing by the FORTRAN compiler. These can be any valid ddnames.

Name	Operation	Operand

[*name*]	LINK
	
	ATTACH
		EP=*compiler-name*,
		PARAM=(*optionaddr*[,*ddnameaddr*]),
		VL=1

[*name*]	CALL	FORTVS2, (*optionaddr*[,*ddnameaddr*]), 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 VM)
2	00000000
3	00000000
4	00000000
5	SYSIN (under MVS), FORTRAN (under VM)
6	SYSPRINT (undr MVS), LISTING (under VM)
7	SYSPUNCH
8	00000000
9	00000000
10	SYSTERM
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.

Link Macro Instruction Example:

```
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'MYSYSPRT',CL8'MYSYSPU'
         DC    2XL8'0000000000000000'
         DC    CL8'MYSYST'
         DC    CL8'MYSYSLIB'
COMPILER DCB   DDNAME=VSFORT,DSORG=PO,MACRF=R
         END
```

Retrieving Arguments in an Assembler Program

Retrieving Variables from the Argument List: The argument list contains the address of a variable. The assembler program can retrieve the variable, using the following instructions:

```
L    Q,x(0,1)
MVC  LOC(y),z(Q)
```

where:

Q	is any general register except 0, 1, 13, or the program's base register.
LOC	is the location that will contain the variable.
x	is the displacement of the address of the variable from the start of the argument list.
y	is the length of the variable itself.
z	is 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: 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     Q,x(1)
L     R,disp
L     S,0(Q,R)
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 an 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   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 length pointer)
L   R,0(0,R)     (Get character 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 will contain the address of the character variable and R will contain the length of the character variable.

Returning a Function Value from an Assembler Program: The FUNCTION name must be declared with a type that corresponds to the type of the value returned (for example, INTEGER TIMER). The method of returning the value depends on whether the function is a CHARACTER function or a noncharacter function.

For noncharacter 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" on page 322). The last argument in this list provides the address and length of the storage area in which the function value will be returned.

The assembler 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 length pointer)
L    R,0(0,R)     (Get character 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 string to be returned

Following is an alternative solution to moving the character string (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 string to move)
         LR   3,15              (Length same as receiving--see Note)
         MVCL 14,2
```

where:

x	The displacement of the last entry in the argument list
LOC	The address of the character string to be returned

Note: If the length of the character string 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 string)
         ICM   3,X'8',BLANK  (Inserts pad character)
BLANK DC    C' '   or   X'40'
```

where:

LCLLEN	A fullword containing the length of the local character string.

Returning to Alternative 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.

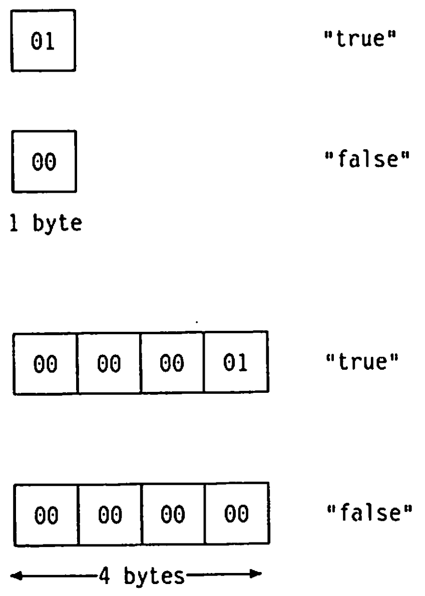
For REAL and COMPLEX items in internal storage, a nonzero floating point number is said to be normalized if the first hexadecimal digit of its fraction is not zero. The normalized representation of a floating point zero has sign, characteristic, and fraction all equal to zero.

The following examples show how VS FORTRAN Version 2 data items appear in internal storage.

Character Items in Internal Storage: Character items are treated internally as one EBCDIC character for each character in the item.

Logical Items in Internal Storage: Logical items are treated internally as items either 1 byte or 4 bytes in length. Their value can be "true" or "false."

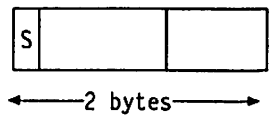
Their internal representation in hexadecimal notation is:



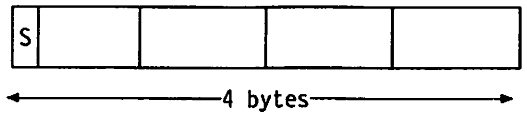
Integer Items in Internal Storage: Integer items are treated internally as two's complement binary fixed-point signed operands, either 2 bytes or 4 bytes in length.

Their internal representation is:

INTEGER *2



INTEGER *4

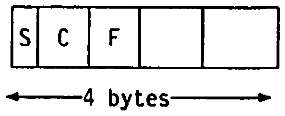


S = the sign bit

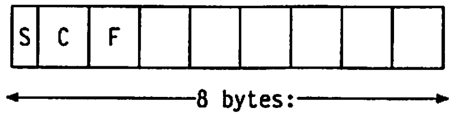
Real Items in Internal Storage: The compiler converts real items into 4-byte, 8-byte, or 16-byte floating-point numbers.

Their internal representation is:

REAL *4



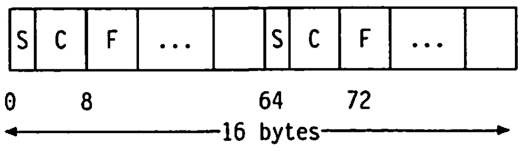
DOUBLE PRECISION (REAL *8)



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)



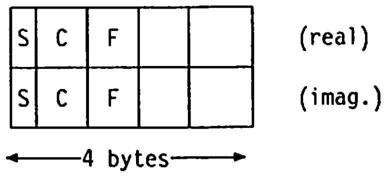
For Extended Precision Items, the codes are:

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.

The internal representations of complex numbers are:

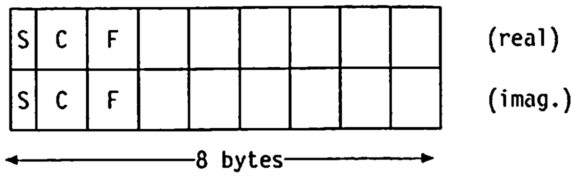
COMPLEX *8



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

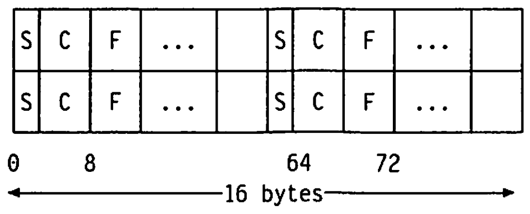
COMPLEX *16



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



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

Appendix B. Object Module Records and Statement Table

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*, SC26-4036.

The statement table is also described.

SYM Record

If you request it with the SYM compiler 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*, SC26-4036.

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	yyyyyyy
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

- yyyyyy 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)

- yyyyyy is the name of the COMMON (for example, X, AAAA, YYYY, FFFF, and so on).

2. Data-type SYM card

Type	Displacement	Name	Variable Portion
xx	wwwwww	yyyyyyy	zzzzzzzzzzzz
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

- wwwwww is the displacement of the variable or array into the module.
- yyyyyy is the name of the variable or array (for example, X, Y, Z, SUMM, FLEX, and so on).
- zzzzzzzzzzzz is the variable portion, which contains the length of the data and the multiplicity (1 for a scalar).
- zzzzzzzzzzzz 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.

  zzzzzzzzzzzz is further divided as follows:

Data Type	Length	Multiplicity
bb	cccc	dddddd
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 *4 (hexadecimal)
X'14' which means INTEGER *2 (halfword)
X'10' which means INTEGER *4 (word)
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:

cccc = X'llll' which means CHARACTER with a length of llll + 1, where 'llll' is the hexadecimal length
X'0000' which means LOGICAL *1 (hexadecimal)
X'0003' which means LOGICAL *4 (hexadecimal)
X'01'which means INTEGER *2 (halfword)
X'03' which means INTEGER *4 (word)
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)

dddddd = 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

Statement Table

The Statement Table (also known as the Program Code Table) contains information on the ISNs in the program unit. The Statement Table is pointed to by the fifteenth word of the PIB (Program Information Block), and the PIB itself is pointed to by the eighth word of the program unit's main entry code.

The Statement Table consists of a variable number of fields, each of which range in length from 1 to 5 bytes. Each user-written executable statement will always have at least one record to indicate the number of halfwords of code generated. The entries in the Statement Table are generated in the order in which the statements appear in memory. The individual fields for a statement, however, may appear in any order.

ISN values are assumed to start with 1 and be incremented by 1 for each statement. When this is not true, as in the case of an intervening FORMAT statement, then one or more special fields are placed in the Statement Table to indicate how many numbers are being skipped (not counting the normal skip of one between statements), except when the count of skipped numbers is greater than 64.

There is a special field to indicate the end of the Statement Table.

The format of each field is as follows (bit locations are origin 0):

bit 0: is 0 if there is another field after this one for the statement; otherwise this bit is 1. Always 1 if this field represents the ending entry for the current statement or the end of the Statement Table.

bits 1 to 7: indicate the type of field. The possible values are:

000xxxx: indicates that this is a 1-byte field where bits 4 to 7 are the count of skipped ISN values. More than one field of this type is required to skip more than 15 ISNs. For example to indicate a skip of 20 numbers requires one field showing a skip of 15 numbers and a second field showing a skip of 5 numbers.

001xxxx: indicates that this is a 2-byte field where bits 4 to 15 contain the label of the statement. This type of field may be used only for statements with labels less than or equal to 4095.

010xxxx: indicates that this is a 3-byte field where bits 4 to 23 contain the label of the statement. This type of field may be used for statements with labels greater than 4095.

011xxxx: indicates that this is a 1-byte field where bits 4 to 7 specify the size of the generated code for the statement in halfwords. This type of field may be used only for statements whose generated code is less than or equal to 15 halfwords (30 bytes).

100xxxx: indicates that this is a 2-byte field where bits 4 to 15 specify the size of the generated code for the statement in halfwords. This type of field may be used only for statements whose generated code is less than or equal to 4095 halfwords (8190 bytes).

101xxxx: indicates that this is a 5-byte field where bits 8 to 39 specify the size of the generated code for the statement in halfwords. This type of field may be used when more than 4095 halfwords are generated.

110xxxx: indicates that this is a 4-byte field where bits 4 to 31 are the sequence number or ISN of the statement. This type of field may be used when the count of skipped ISNs or sequence numbers is greater than 64 or the numbers are out of sequence or for the END statement.

1110000 through 1111110: reserved.

1111111: indicates the end of the Statement Table.

Appendix C. Compatibility Considerations

VS FORTRAN Version 2 produces programs that utilize the IBM 3090 Vector Facility. VS FORTRAN Version 1 programs can be run on the IBM 3090 hardware; however, they cannot utilize the Vector Facility feature of that system. See Chapter 9, "Vectorizing Your Program" on page 227 for a description of this feature.

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 79 shows which libraries contain the various scalar mathematical routines for each version.

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

Figure 79. Libraries Containing Mathematical Routines

See *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 library. 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 upper case. 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 Appendix E, "The Multitasking Facility (MTF)" on page 349 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. Release 3 of VS FORTRAN Version 2 restricts some of this flexibility by ignoring the following compiler options, and providing a level 4 message, if they appear in an @PROCESS statement preceded by comments, EJECT statements, or INCLUDE statements:

- EXCLAM
- 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.

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 Appendix H, "Considerations for Specifying RECFM, LRECL, and BLKSIZE" on page 445.

VS FORTRAN Versions 1 and 2, and Earlier IBM FORTRANs Differences

In VS FORTRAN Versions 1 and 2, logical variables may contain only logical values and should appear only in logical expressions. Logical variables may not contain numeric or character values and may not appear in arithmetic expressions (and an error or serious error message is issued). This is true for both LANGLVL (66) and LANGLVL (77). Under LANGLVL (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.*

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 LANGLVL(77) option only.

In VS FORTRAN Versions 1 and 2, the form of the compiler option to name a program is NAME(nam) under LANGLVL(66).

Arguments are received only by location (or name) in LANGLVL(77). The default in LANGLVL(66) and for FORTRAN H and FORTRAN H Extended is receipt by value with the facility, to allow receipt by name by the use of slashes around the dummy argument in the SUBROUTINE, FUNCTION, or ENTRY statements.

The appearance of an intrinsic function name in a conflicting type statement has no effect in LANGLVL(77), but is considered user-supplied under LANGLVL(66) and FORTRAN H and FORTRAN H Extended.

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 LANGLVL(66). Under LANGLVL(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 LANGLVL(66) and LANGLVL(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 LANGLVL(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 LANGLVL(77), character arguments are passed to a subprogram with both a pointer to the character string and a pointer to the length of the character string. This is required because the receiving program may have declared the dummy character arguments to have inherited length (that is, the length of the dummy argument is the length of the actual argument). The parameter list is therefore longer than for LANGLVL(66), because every character argument generates two items in the parameter list. For LANGLVL(66):

- Literal constants passed as arguments generate only one item in the parameter list.
- Hollerith constants may be passed as subroutine or function arguments.

In LANGLVL(77), a level 8 message is received if Hollerith constants are passed as arguments.

In both languages, only one item is generated in the parameter list for Hollerith arguments.

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.

Appendix D. Internal Limits in VS FORTRAN Version 2

The internal limits of VS FORTRAN Version 2 are given in Figure 80.

Language Item	Limit
Levels of nested DO loops and implied DO loops	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	2000
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

Figure 80. Internal Limits in VS FORTRAN Version 2

Appendix E. The Multitasking Facility (MTF)

Introduction to MTF

What MTF Is

MTF is a VS FORTRAN Version 2 facility that can be used by computationally-intensive application programs to improve turnaround time on tightly-coupled System/370 multiprocessor (MP) and attached-processor (AP) configurations (for example, the 3090-200 or 3090-400). 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.

What MTF Does

MTF takes advantage of the multitasking capabilities of the MVS and MVS/Extended Architecture (MVS/XA) operating systems to allow a single VS FORTRAN Version 2 application program to use more than one processor of a multiprocessing configuration simultaneously. (MTF provides multitasking only on the MVS or MVS/XA operating systems.) MVS operating systems organize all work into units called tasks. These tasks are used by the operating system to assign work to the processors of the multiprocessor configuration.

MTF's facilities allow a single VS FORTRAN Version 2 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 VS FORTRAN Version 2 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 358. They can perform I/O and can share large amounts of data with the main task program by means of dynamic common blocks.

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 VS FORTRAN Version 2 Library creates the MTF environment required to run the separate parts of a program in parallel. 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 assigned to a subtask and is run in parallel with the main task program. *arg1*, *arg2*, ... are arguments passed to the parallel subroutine.

subname can be 8 characters long. If *subname* is a FORTRAN subprogram or has more than 8 characters, you must provide the 7-character name 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.

The main task program can schedule multiple instances of a parallel subroutine for parallel 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.

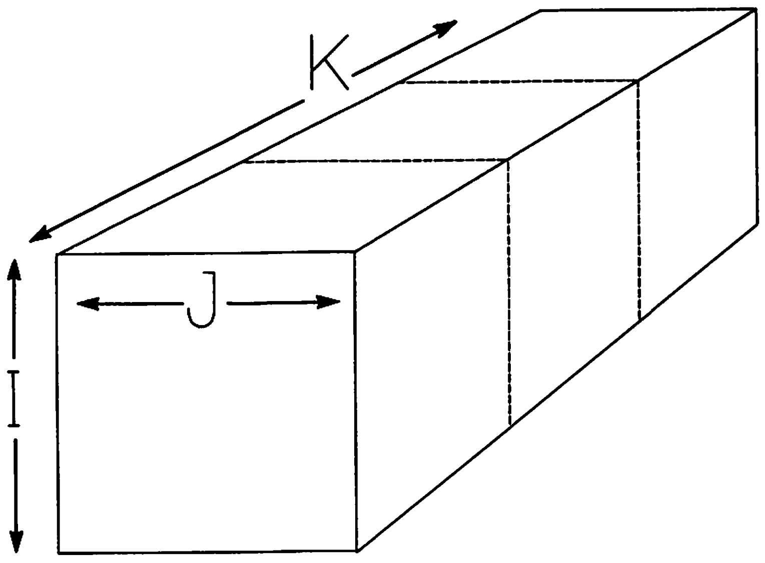
```
CALL SYNCRO
```

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.

In the figure below, you see 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.

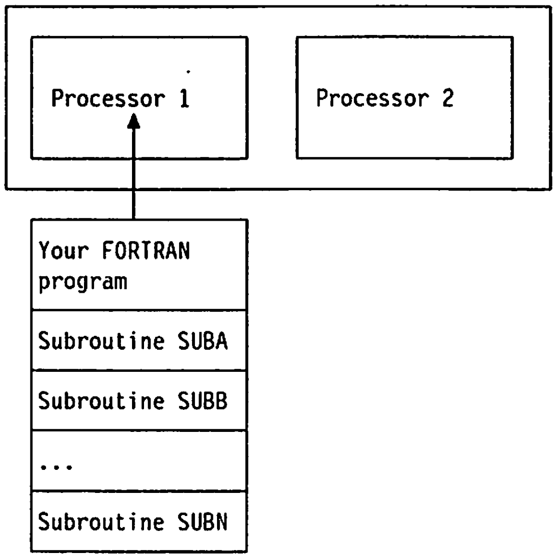
It is also possible to have computational independence that is not based on sections of the same array, but rather on separate arrays (perhaps with completely different types of data), the values of which do not depend on each other. In this case, separate parallel subroutines could be scheduled, with 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.

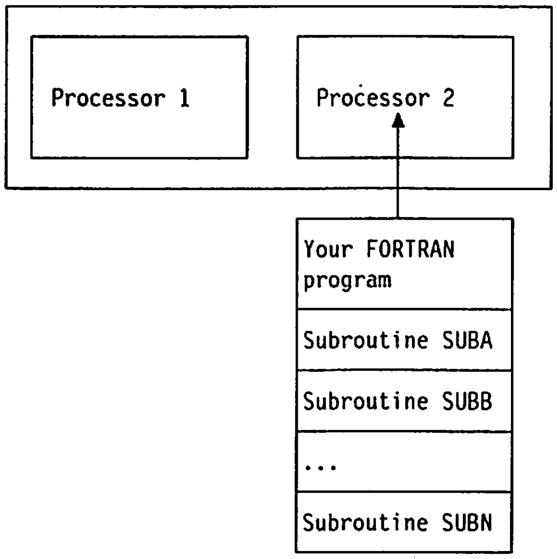
Running a VS FORTRAN Version 2 Program without MTF

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 VS FORTRAN Version 2 program with MTF. These examples show programs using:

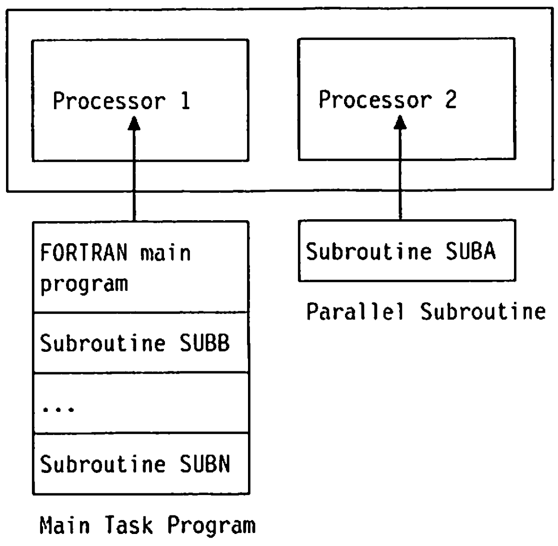
- One parallel subroutine
- Two different subroutines
- Two or more instances of the same subroutine

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 Only One Parallel Subroutine

If your FORTRAN program uses MTF, the main task program and a computationally-independent parallel subroutine can run concurrently.

Processor Use

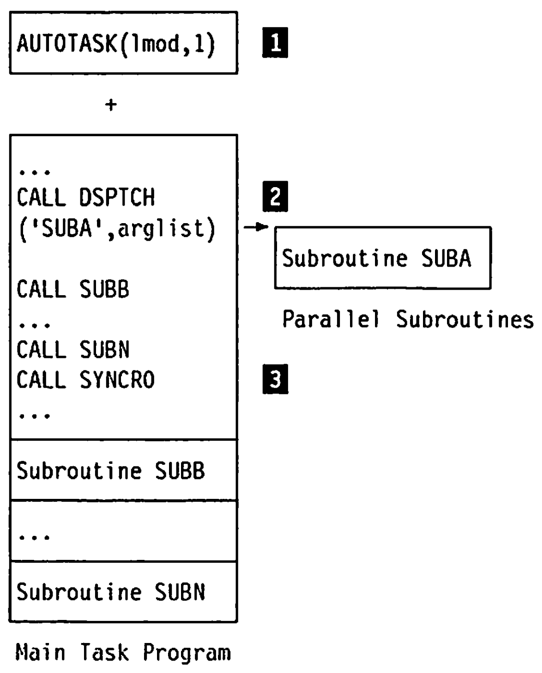


In the drawing to the left, only subroutine SUBA has computations that can be done independently of the main task program, which includes the FORTRAN main program plus its subroutines.

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.

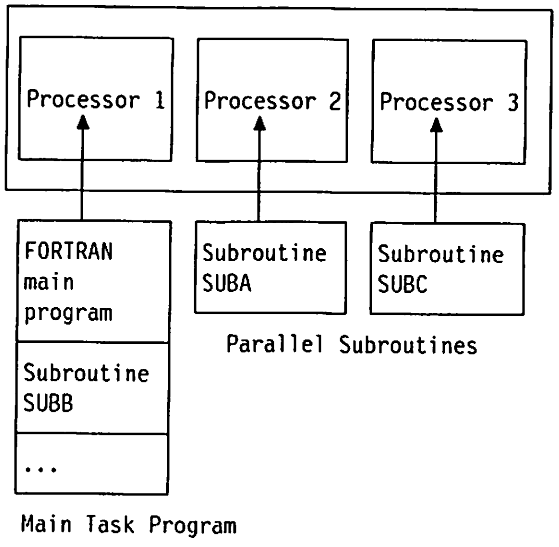
3 SYNCRO makes the main task program wait until SUBA finishes before the main task program continues.

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 concurrently.

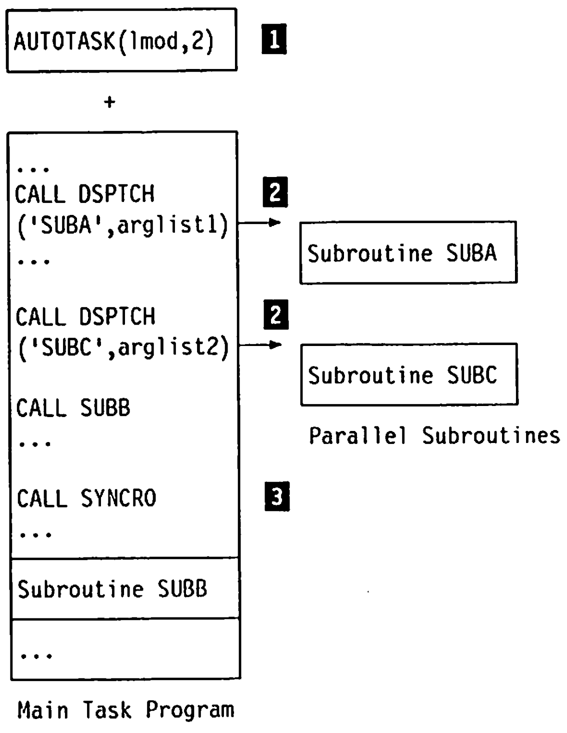
Processor Use



In the drawing to the left, subroutines SUBA and SUBC are independent of each other and of the main task program.

As with "Running with Only One Parallel Subroutine" on page 353, 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.

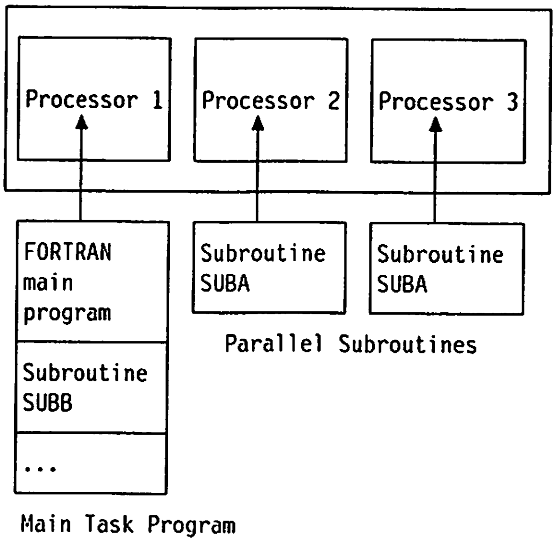
2 Each call to DSPTCH schedules one of the parallel subroutines, passing different data to each for processing. SUBA and SUBC are computationally-independent parallel subroutines.

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

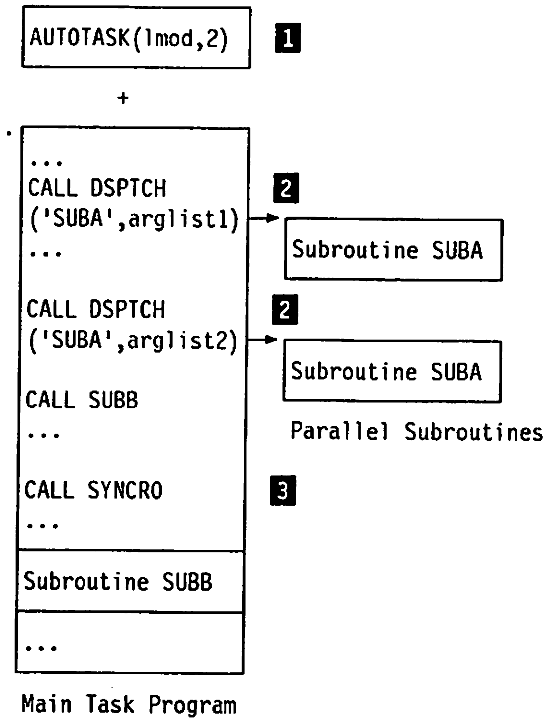
If your FORTRAN program uses MTF, the main task program and multiple instances of the same parallel subroutine can run concurrently.

Processor Use



In the drawing 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

The following steps may be used when preparing a VS FORTRAN Version 2 application to work with MTF:

1. Identify Computationally-Independent Code
2. Create Parallel Subroutines
3. Insert Calls to Parallel Subroutines

New programs can be designed to use MTF, and existing programs can be reconstructed.

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 in parallel. In order to produce correct results, the computations that are done in parallel must be computationally independent. Computational independence is explained under "The Concept of Computational Independence" on page 351.

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 may actually be coded as a series of subroutines which 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 may 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. Should the value be 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 upon 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 may be used to indicate whether a file is to be updated.

A named file must be connected within each subroutine that uses it. The connection does not remain after the subroutine finishes running— when a subroutine 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 parallel computation with a call to DSPTCH which schedules the corresponding parallel subroutine. If parallel operation is to be achieved by scheduling the same subroutine multiple times with different data, the CALL statement may 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.

An Example of Changing an Application to Use MTF

Identify Computationally-Independent Code

Figure 81 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,KH
      VS(1,1,K) = 0.0
      DO 10 J=2,JH
         DO 10 I=2,IH
            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 81. 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 VS FORTRAN Version 2 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 81 now looks like Figure 82.

```
    SUBROUTINE SUB (KLIH1,KLIH2,VX,VT,VS,VR,RHO,P,IH,JH)
    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=KLIH1,KLIH2
     VS(1,1,K)=0.0
     DO 10 J=2,JH
        DO 10 I=2,IH
           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 82. 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 81 on page 361, 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 83 for an example of how two instances of a parallel subroutine can be scheduled.

```
    .
    .
    .
C  SCHEDULE 2 INSTANCES OF PARALLEL SUBROUTINE SUB
      KH=KH/2
      KHS=KH+1
      CALL DSPTCH('SUB',1,KH,VX,VT,VS,VR,RHO,P,IM,JM)
      CALL DSPTCH('SUB',KHS,KH,VX,VT,VS,VR,RHO,P,IM,JM)

C  WAIT FOR BOTH INSTANCES OF SUB TO FINISH RUNNING
      CALL SYNCRO
    .
    .
    .
```

Figure 83. 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 84. (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)=KH
         DO 25 I=1,NT-1
         KUB(I)=KH*I/NT
25       KLB(I+1)=KUB(I)+1
      .
      .
      .
      END
```

Figure 84. 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 85 on page 363.

```
 1            3333 3334         6666 6667          KM=10000
|-----------------|-----------------|-----------------|

   KLB(1)=1           KLB(2)=3334      KLB(3)=6667
   KUB(1)=3333        KUB(2)=6666      KUB(3)=10000
```

Figure 85. Lower and Upper Bounds for the K Loop

Based on the computation of the bounds in Figure 84 on page 362, 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 86.

```
     .
     .
     .
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 86. 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 reassigned, 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 87 on page 364.

```
  .
  .
  .
C RUN NT INSTANCES OF PARALLEL SUBROUTINE XYZ
    DO 10 I=1,NT
     IX(I)=I
     CALL DSPTCH('XYZ',IX(I),...)
10  CONTINUE
  .
  .
  .
```

Figure 87. Passing the Value of the DO Variable to a Parallel Subroutine

Another Example of Changing an Application to use MTF

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 88 on page 365 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 initilize 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 88. 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 88 are shown in Figure 89 on page 366.

```
@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 89. 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 routine 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 88 on page 365 and in Figure 89, the code that schedules the two parallel subroutines SUBA and SUBB and waits for them to finish running is shown in Figure 90 on page 367.

```
@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 initilize 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 90. 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 taskprogram.

The procedures that you normally use to compile and link-edit a VS FORTRAN Version 2 program can be used to create the main task program load module. For example, the following JCL sequence (see Figure 91) 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 91. 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 following JCL sequence 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 92. Sample JCL to Compile and Link Parallel Subroutines

Under MVS/XA, 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, then 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 VS FORTRAN Version 2 library become available and you relink either load module, then you have to be sure to 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 VS FORTRAN Version 2 programs, plus a few additional JCL statements that are required for MTF to run. This additional JCL is shown in Figure 93. 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=USERPGH.LOAD,DISP=SHR
//FTERR001  DD    ...
//FTERR002  DD    ...
//FTERRO..  DD    ...
//FTERROnn  DD    ...
```

Figure 93. 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 in parallel. 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 may 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 time, you need not supply DD statements for them unless you wish to override the default device type or other file characterstics. The default device type is a terminal in TSO or SYSOUT=A in batch.

If you do supply DD statements, use the following ddnames:

- FTERR*sss* for files containing diagnostic messages and possibly output from PRINT or WRITE statements
- FTPRT*sss* 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 94. 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 94. 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.

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

VS FORTRAN Version 2 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 Libraries in Load Mode" on page 74.

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 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 noticable.

Appendix F. Vector Report Diagnostic Messages

The messages in this appendix are those that may appear in the vector report listing when either the VECTOR(REPORT(XLIST)) or the VECTOR(REPORT(SLIST)) 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.

The messages in this appendix 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 ISN's) that might be inserted into the long form of the message. Where appropriate, there may also be examples of situations that cause the message along with suggestions for rewriting programs to improve vectorization.

The messages are grouped into six categories. The first four groups are divided according to the vectorization status flags that mark the loops printed on the program listing portion of the vector report. The fifth group consists of special messages that clarify certain ambiguities in the vector report program listing. The last group consists of messages that describe the effects of vector directives on the vectorization process.

- Unanalyzable Messages (UNAN flag)
- Recurrence Detection Messages (RECR flag)
- Unsupported Operation Messages (UNSP flag)
- Vectorizable 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 the section "Printing Reports" on page 238.

Message for Unanalyzable Loops (UNAN)

These messages appear whenever a DO loop contains some construct that is considered to be unanalyzable. In general, these are control structures and language constructs that make it difficult or impossible for the compiler to gather the information needed to perform correct vectorization.

The vectorization status flag used for these messages is UNAN.

- **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 UNANALYZABLE
      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,*) A(I,J)
 10   CONTINUE
```

  In this case, the inner loop is unanalyzable because it contains an I/O statement. 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 UNANALYZABLE
      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,*) A
```

► **ILX0104I**

Short Form: **I/O OPERATION**

Long Form: **ONE OR MORE I/O STATEMENTS OCCUR AT ISN(S) <ilist>.**

Explanation: Indicates that a loop contains one or more I/O statements.

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
        A(I) = B(I) * 3.3
        WRITE(6,*) A(I)
 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
        A(I) = B(I) * 3.3
 10   CONTINUE
      DO 11 I = 1,100
        WRITE(6,*) A(I)
 11   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: **STOP, RETURN, OR EXIT BRANCH**

Long Form: **STOP OR RETURN STATEMENT(S) OR LOOP EXIT BRANCH(ES) HAVE BEEN USED AT ISN(S) <ilist>.**

Explanation: Indicates that a loop was rejected because of the presence of a STOP or RETURN statement or because there is a branch out of the loop. (The reason that these three separate situations are identified by a single message is that they all result in the abnormal termination of a loop. The compiler does not distinguish between the specific causes of the abnormal termination when deciding whether a loop is eligible for vectorization analysis.)

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 STOP, RETURN, OR EXIT BRANCH
      REAL*4 A(20),B(20),C(20)

      DO 10 I = 1,20
        IF (C(I) .EQ. 0.0) STOP
        A(I) = A(I) ** 2.1
        B(I) = B(I) ** 2.1
 10   CONTINUE
```

Possible Response: Insert a new loop to determine the number of iterations that will be processed before the original loop terminates. Then rewrite the original loop to remove the test that causes the loop to terminate and change the upper bound to the number of iterations that has just been computed. After the loop, add a new test to determine whether the STOP (or RETURN or exit branch) should be used.

Modified Example:

```
C POSSIBLE RESPONSE
C STOP, RETURN, OR EXIT BRANCH
      REAL*4 A(20),B(20),C(20)

      DO  8 I = 1,20
        IF (C(I) .EQ. 0.0) GOTO 9
  8   CONTINUE
  9   ISTOP=I
      DO 10 I = 1,MIN(ISTOP,20)
        A(I) = A(I) ** 2.1
        B(I) = B(I) ** 2.1
 10   CONTINUE
      IF (ISTOP.LE.20) STOP
```

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 by the loop. 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: **EXIT BRANCH**

Long Form: **ONE OR MORE EXIT BRANCHES ORIGINATE AT ISN(S) <ilist>.**

Explanation: Indicates that a loop was rejected because it 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 EXIT BRANCH
      REAL*4 A(20),B(20),C(20)

      DO 10 I = 1,20
        IF (C(I) .EQ. 0.0) GOTO 500
        A(I) = A(I) ** 2.1
        B(I) = B(I) ** 2.1
 10   CONTINUE
        - - -
500   CONTINUE
```

Possible Response: Insert a new loop to determine the number of iterations that will be processed before the original loop terminates. Then rewrite the original loop to remove the test that causes the loop to terminate and change the upper bound to the number of iterations that has just been computed. After the loop has completed, add a new test to determine whether the branch should be taken.

Modified Example:

```
C POSSIBLE RESPONSE
C EXIT BRANCH
      REAL*4 A(20),B(20),C(20)

      DO  8 I = 1,20
        IF (C(I) .EQ. 0.0) GOTO 9
  8   CONTINUE
  9   IEXIT=I

      DO 10 I = 1,MIN(IEXIT,20)
        A(I) = A(I) ** 2.1
        B(I) = B(I) ** 2.1
 10   CONTINUE
      I=IEXIT
      IF (IEXIT.LE.20) GOTO 500
        - - -
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 by the loop. 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 invoked within the loop in such a way that its value could be changed by the 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 non-vectorizable 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 (CHAR or LEN, for example) 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 service subprogram (DSPAN#, DSPN2#, DSPN4#, or DYCMN#). These subprograms 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 service subprograms 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.

Messages about Recurrences (RECR)

These messages appear with statements that are found to be in a recurrence. Some of these messages point out situations where the compiler has had to presume dependences between statements that reference a particular array because the subscript computations were too complex to be analyzed in detail. In these cases, it may be possible to rewrite the statements so that the compiler can gather more precise information. There is no guarantee that the removal of the indicated dependences will allow the statements to be vectorized.

The vectorization status flag used for these messages is RECR.

- **ILX0117I**

Short Form: **EQUIVALENCE USED**

Long Form: **THE VARIABLE(S) <vlist> ARE EQUIVALENCED OR ARE IN A COMMON BLOCK THAT CONTAINS AN EQUIVALENCED VARIABLE. THE COMPILER HAS ASSUMED THAT THESE ARRAYS CARRY DEPENDENCES IN LOOP(S) AT NESTING LEVEL(S) <levlist>.**

Explanation: Whenever there is a potential for overlap between variables contained in an equivalence group, the compiler presumes the existence of recurrent dependences between statements that reference those variables.

Note that the analysis the compiler does to determine whether there is a potential overlap is not very detailed. This means that in these cases the compiler may assume a dependence exists, even though it is obvious to you that this is not the case.

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 vector report. They do not necessarily correspond to the nesting level indications that appear on the source listing.

Example:

```
C EXAMPLE
C EQUIVALENCE USED
         COMMON // A(200)
         REAL*4 B(100)
         EQUIVALENCE (A(101),B(1))

         DO 10 I = 1,20
          A(I) = A(I) * B(I) ** 2.1
 10      CONTINUE
```

Possible Response 1: It is sometimes possible to avoid these dependences by rewriting all references to variables in a given equivalence group in terms of a single variable in that group.

Modified Example 1:

```
C POSSIBLE RESPONSE 1
C EQUIVALENCE USED
         COMMON // A(200)
         REAL*4 B(100)
         EQUIVALENCE (A(101),B(1))

         DO 10 I = 1,20
          A(I) = A(I) * A(I+100) ** 2.1
 10      CONTINUE
```

Possible Response 2: It is also possible to eliminate this dependence by using the IGNORE EQUDEPS vector directive. 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 USED
         COMMON // A(200)
         REAL*4 B(100)
         EQUIVALENCE (A(101),B(1))

*DIR     IGNORE EQUDEPS
         DO 10 I = 1,20
          A(I) = A(I) * B(I) ** 2.1
 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 vector 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

Vector Directives" on page 253 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 vector 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 Vector Directives" on page 253 for details on how to correctly specify and verify this directive.

- **ILX0121I**

Short Form: **COMMON OR EQUIVALENCE USED**

Long Form: **THE SCALAR VARIABLE(S) <vlist> ARE NOT ELIGIBLE FOR EXPANSION BECAUSE THEY ARE IN COMMON OR ARE IN AN EQUIVALENCE GROUP.**

Explanation: Variables that are in COMMON or in an EQUIVALENCE group are not eligible for scalar expansion because the compiler cannot tell whether they are local to the loops in which they are used. (A scalar variable is local to a given loop if the values that it holds while the loop is running could not have been set before the loop began to run and will never be used after the loop terminates. Scalar expansion can only be performed on variables which the compiler knows to be local.)

Supplemental Data:

<vlist> is a list of the names of the scalar variables that are ineligible for expansion.

Example 1:

```
C EXAMPLE 1
C SCALAR EXPANSION - COMMON USED
        COMMON // T
        REAL*4 A(50),B(50),T

        DO 10 I = 1,50
         T = B(I)
         B(I) = A(I)
         A(I) = T
 10     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 1:

```
C POSSIBLE RESPONSE 1
C SCALAR EXPANSION - COMMON USED
        COMMON // T
        REAL*4 A(50),B(50),T,T1

        DO 10 I = 1,50
         T1 = B(I)
         B(I) = A(I)
         A(I) = T1
 10     CONTINUE
        T = A(50)
```

Example 2:

```
C EXAMPLE 2
C SCALAR EXPANSION - EQUIVALENCE USED
        REAL*4 X(50),Y(50),Z(50),D(50),S
        EQUIVALENCE (S,D(50))

        D(50) = 1.1
        DO 20 I = 1,50
         Z(I) = S
         S = X(I) + Y(I)
 20     CONTINUE
```

Possible Response: Another possible action is 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 array to the value held by the scalar.
- Prior to the first statement that defines the scalar within the loop, replace all references to the scalar with references to the element of the 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 array.

Modified Example 2:

```
C POSSIBLE RESPONSE 2
C SCALAR EXPANSION - EQUIVALENCE USED
         REAL*4 X(50),Y(50),Z(50),D(50),S
         EQUIVALENCE (S,D(50))
         REAL*4 SS(0:50)

         D(50) = 1.1
         SS(0) = S
         DO 20 I = 1,50
          Z(I) = SS(I-1)
          SS(I) = X(I) + Y(I)
 20      CONTINUE
         S = SS(20)
```

Note that this transformation is only valid if the first assignment to the original scalar variable is not a conditionally processed statement. Also note that you should use these transformations with care since they may not always increase the vectorizability of your program and may result in increased scalar run time.

► **ILX0123I**

Short Form: **INTERCHANGE PREVENTING DEP**

Long Form: **THE ARRAY(S) <vlist> 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:

<vlist> 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 vector 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 vector 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 vector 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, non-vectorizable, 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 non-vectorizable 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) <vlist> 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(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(IVAR)," where "IVAR" is incremented explicitly within the DO loop by some statement of the form "IVAR=IVAR+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:

<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 vector 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 non-inductive subscript expression and copy them into a new array.

- Replace the non-inductive 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 non-inductive 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 none of the previous transformations is appropriate, an IGNORE RECRDEPS directive may be used to cause the compiler to assume that no dependence exits.

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 Vector Directives" on page 253 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: In order to vectorize a scalar variable that is modified within a loop, the compiler must use a process known as scalar expansion. This involves replacing the scalar variable with a vector register.

Scalar expansion can only be performed on variables which the compiler knows to be local. A variable is local to a given loop if the values that it holds while the loop is running could not have been set before the loop began to run and will never be used after the loop terminates.

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),B(20),C(20),T

         T = 1.1
         DO 11 I = 1,20
          C(I) = T
          T = A(I) + B(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),B(20),C(20),T
         REAL*4 TT(0:20)

         T = 1.1
         TT(0) = T
         DO 11 I = 1,20
          C(I) = TT(I-1)
          TT(I) = A(I) + B(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: In order to vectorize a scalar variable that is modified within a loop, the compiler must use a process known as scalar expansion. This involves replacing the scalar variable with a vector register.

Scalar expansion can only be performed on variables which the compiler knows to be local. A variable is local to a given loop if the values that it holds while the loop runs could not have been set before the loop began to run and will never be used after the loop terminates.

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),T

         DO 22 I = 1,20
          T = A(I)
          A(I) = B(I)
          B(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),T
         REAL*4 T_LOCAL

         DO 22 I = 1,20
          T_LOCAL = A(I)
          A(I) = B(I)
          B(I) = T_LOCAL
 22      CONTINUE
         T=B(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),T
         REAL*4 TT(20)

         DO 22 I = 1,20
          TT(I) = A(I)
          A(I) = B(I)
          B(I) = TT(I)
 22      CONTINUE
         T = TT(20)
          - - -
         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 vector report, this message may appear with each copy, even though it may not be applicable in all cases.

Example 1:

```
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 vector 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 non-linear 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) <vlist> 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:

<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 vector 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(-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 Vector Directives" on page 253 for details on how to correctly specify and verify this directive.

Messages for Unsupportable Constructs (UNSP)

These messages indicate usages of data types and constructs for which no support (either in the compiler or in the vector hardware) currently exists.

The vectorization status flag used for these messages is UNSP.

- **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 (FQXPQ#, PQXP2#, or FCQXI#). 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
```

  In this example, the library subroutine FQXPQ# is used to calculate the exponentiation of an extended precision REAL value.

- **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 unsupportable 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 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, 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 unsupportable part of the loop with equivalent supportable constructs, do so.

Otherwise, try to separate the vectorizable statements 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*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 VARIABLE(S) <vlist> IN CONDITIONALLY EXECUTED CODE CANNOT BE VECTORIZED.**

Explanation: Indicates the presence of INTEGER*2 data in conditionally processed code.

Note: Some statements that affect the flow of control within a loop may not be reproduced in the vector 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 unsupportable 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 unsupportable 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) <vlist> ARE USED IN CONDITIONALLY EXECUTED CODE AND HAVE NON-INDUCTIVE SUBSCRIPT EXPRESSIONS.**

Explanation: Indicates the use of non-inductive subscript expressions that occur in conditionally processed code. A non-inductive 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 vector 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 arrays that use non-inductive 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 non-inductive subscript expression and copy them into a new array.

- Replace the non-inductive 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 non-inductive 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) <vlist> HAVE LOGICAL*1 OR INTEGER*2 DATA TYPES AND HAVE NON-INDUCTIVE SUBSCRIPT EXPRESSIONS.**

Explanation: Indicates the use of non-inductive subscript expressions that occur in arrays with data types of INTEGER*2 and LOGICAL*1. A non-inductive 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:

<vlist> is a list of the names of the arrays that use non-inductive 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 non-inductive subscript expression and copy them into a new array.
- Replace the non-inductive 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 non-inductive 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.

► **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.

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.

Messages for Statements which Can Be Vectorized (ELIG or VECT)

The vectorization status flag used for these messages is either ELIG or VECT.

► **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 an 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 non-vectorized 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 non-vectorized 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 VRXPR#, the entry name for 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
```

In this example, the compiler generates a call to the library subroutine VRXPR# to perform the exponentiation operation.

Possible Response: If there is concern for the consistency of program results between vectorized and non-vectorized 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.

Listing Clarification Messages (SCAL or VECT)

The vectorization status flag used for these messages is either SCAL or VECT.

- **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 vector 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 vector report output produced by the VECTOR(REPORT(XLIST)) option.

Vector Directive Messages (VDIR)

These messages are used to describe the effects of vector directives on the vectorization process. They identify statements and DO loops to which directives have been applied and also indicate whether or not the directives affected the outcome of vectorization.

The vectorization status flag used for these messages is VDIR.

- **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 1:

```
@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 vector 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 vector 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
```

► **ILX0174I**

Short Form: **"IGNORE EQUDEPS" USED**

Long Form: **AN "IGNORE EQUDEPS" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP.**

Explanation: This message identifies loops to which IGNORE EQUDEPS 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 1:

```
@PROCESS DIRECTIVE('DIR')
C EXAMPLE
C IGNORE EQUDEPS USED
         EQUIVALENCE (A,B)
         REAL*4 A(128),B(128)

*DIR     IGNORE EQUDEPS
         DO 10 I = 1,128
           A(I) = B(I) ** 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 occurs, that is, determine which potential dependences have been ignored because of the directive. (These dependences are identified by other vector report messages that appear with the statements within a loop.)

Study the mapping of the variables within the equivalence group and the subscript calculations to determine whether two different variables may reference a single storage location when the loop is processed. If so, the directive should be removed.

► **ILX0175I**

Short Form:	**"IGNORE RECRDEPS EQUDEPS"**
Long Form:	**AN "IGNORE RECRDEPS EQUDEPS" DIRECTIVE HAS BEEN SPECIFIED FOR THIS LOOP.**

Explanation: This message identifies loops to which IGNORE RECRDEPS EQUDEPS 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 EQUDEPS USED
        COMMON // N
        EQUIVALENCE (A,B)
        REAL*4 A(128),B(128),C(128)

*DIR    IGNORE RECRDEPS(C) EQUDEPS
        DO 10 I = 64,128
          A(I) = B(I) ** 2.1
          C(I) = C(I-N)
 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 occurs, that is, determine which potential dependences have been ignored because of the directive. (These dependences are identified by other vector report messages that appear with the statements within a loop.)

For equivalence dependences, study the mapping of the variables within the equivalence group and the subscript calculations to determine whether two different variables may reference a single storage location when the loop is processed. If so, the directive should be removed or modified.

For other dependences, 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 EQUDEPS USED
        COMMON // N
        EQUIVALENCE (A,B)
        REAL*4 A(128),B(128),C(128)

        IF (N.GE.1 .AND. N.LE.64) THEN
          PRINT *,'INCORRECT IGNORE DIRECTIVE'
          STOP
        ENDIF
*DIR    IGNORE RECRDEPS(C) EQUDEPS
        DO 10 I = 64,128
          A(I) = B(I) ** 2.1
          C(I) = C(I-N)
 10     CONTINUE
```

► **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 vector 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
```

► **ILX0178I**

Short Form:	**POTENTIAL EQUDEP ELIMINATED**
Long Form:	**POTENTIAL DEPENDENCE(S) INVOLVING THE EQUIVALENCED ARRAY(S) <alist> HAVE BEEN IGNORED BECAUSE OF AN "IGNORE EQUDEPS" DIRECTIVE APPLIED TO THE LOOP(S) AT NESTING LEVEL(S) <levlist>.**

Explanation: This message identifies the statements where an IGNORE EQUDEPS directive has caused the compiler to ignore some dependence that would otherwise have been assumed to exist between arrays in the same equivalence group. 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 vector 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 EQUDEP ELIMINATED
        EQUIVALENCE (A,B)
        REAL*4 A(128),B(128)

*DIR    IGNORE EQUDEPS
        DO 10 I = 1,128
          A(I) = B(I) ** 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, analyze the storage mapping and the subscript calculations used in the specified arrays to make sure that references to the different variables do not involve the same storage locations. If they do, the directive should be removed.

► **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 vector 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 vector 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 vector 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 vector 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.

Appendix G. What Determines File Existence

This appendix discusses the conditions that VS FORTRAN uses to determine the existence of files. File existence, in turn, is used 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 Chapter 6, "Performing Input/Output Operations" on page 121. For detailed information about the syntax of these statements, see *VS FORTRAN Version 2 Language and Library Reference*.

In general, any 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, what type of device the file resides on, and, for some device types, whether the file contains any data. In this appendix you will find a series of decision tables, one for each different device type, that show whether a particular file exists at any point 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 that was created by your program but has no data records in it will be seen as existing within that the 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.

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.

Conditions that Apply to All Files

For all files under both MVS and VM, the following conditions and results apply:

- If a file does not have a file definition in effect, it does not exist.

  Note that under VM, a file definition is always in effect because a default file definition is supplied if you don't supply one. The form of the default file definition is:

  FILEDEF *ddname* DISK FILE *ddname* A

- If a file is internally open, it exists.

  Internally open means that a file either has been connected by the OPEN statement; or, in the case of a preconnected file, a READ, WRITE, PRINT, or ENDFILE statement for that file has been successfully run.

- If a file cannot be accessed because VS FORTRAN detects an access restriction, an error is detected. *An exception to this is for DASD files under VM, as shown in Figure 104 on page 441 .*

  An *access restriction* is any condition that prevents the use of the file in your program, such as RACF protection or the inability to mount a volume.

  Access restriction is checked only for the following:

  - DASD files (including PDS members and VSAM files)
  - Tape files

Conditions that Apply to All Unnamed Files

When you code the INQUIRE statement to check the existence of an unnamed file, the unit number must be within the range of unit numbers allowed for your installation. If the unit number is beyond the range, the value returned for EXIST will be false, even if a file definition is in effect.

Additional Conditions Specific to Certain Files

The following tables show additional conditions that VS FORTRAN checks to determine file existence. Because VS FORTRAN checks different conditions for different file types, the tables are divided by file type.

The tables are divided according to whether the device or file type is on MVS (see "MVS File Existence Tables") or VM (see "VM File Existence Tables" on page 441).

Basic Conditions

All tables refer to files that meet these basic conditions:

- **They have a file definition in effect.**
- **They are not internally open.**
- **They do not have restricted file access, as detected by VS FORTRAN.**

  An exception to this last condition is for DASD files under VM, as shown in Figure 104 on page 441 .

In the tables, "—" indicates that this condition does not affect file existence in this instance.

MVS File Existence Tables

These types of MVS files are referenced:

DASD files, page 436
Reusable VSAM files, page 437
Non-reusable VSAM files, page 437
PDS members, page 438
Labeled tape files, page 438
Unlabeled tape files, page 439
In-stream (DD * or DD DATA) data sets, page 439
System output (sysout) data sets, page 439
Terminals, page 439
Unit record input devices, page 440
Unit record output devices, page 440
Files whose file definitions specify DUMMY, page 440
Files on other devices, including subsystems, page 441

DASD File Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File On Volume[1]	Volume Sequence Number[2] >1	File Concatenated[3]	File Contains Data	File Ever Internally Opened	File Deleted by CLOSE	*Then the file:*
No	—	—	—	—	—	**doesn't exist**
Yes	Yes	—	—	—	—	**exists**
Yes	No	Yes	—	—	—	**exists**
Yes	No	No	Yes	—	—	**exists**
Yes	No	No	No	No	—	**doesn't exist**
Yes	No	No	No	Yes	No	**exists**
Yes	No	No	No	Yes	Yes	**doesn't exist**

Figure 95. File Existence Table for DASD (MVS)

Notes to Figure 95:

1. For DASD, *file 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 JCL*.
3. *Concatenated* refers to a JCL concatenation of data sets having a single ddname, not a set of subfiles having different ddnames.

Reusable VSAM File Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

Subfile Sequence Number > 1	File Contains Data	File Ever Internally Opened	File Deleted by CLOSE	*Then the file:*
Yes	—	—	—	**doesn't exist**
No	Yes	—	—	**exists**
No	No	No	—	**doesn't exist**
No	No	Yes	No	**exists**
No	No	Yes	Yes	**doesn't exist**

Figure 96. File Existence Table for Reusable VSAM (MVS)

Non-Reusable VSAM File Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

Subfile Sequence Number >1	File Contains Data	File Ever Internally Opened	*Then the file:*
Yes	—	—	**doesn't exist**
No	Yes	—	**exists**
No	No	No	**doesn't exist**
No	No	Yes	**exists**

Figure 97. File Existence Table for Non-Reusable VSAM (MVS)

PDS Member Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

Member Exists	*Then the file:*
No	**doesn't exist**
Yes	**exists**

Figure 98. File Existence Table for PDS Member (MVS)

Labeled Tape File Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File On Volume[1]	**Block and Record Lengths Are 0 in Header Label**	*Then the file:*
No	—	**doesn't exist**
Yes	Yes	**doesn't exist**
Yes	No	**exists**

Figure 99. File Existence Table for Labeled Tape File (MVS)

Note to Figure 99:

1. For a labeled tape, *file 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" on page 435, these conditions are in effect:

File On Volume[1]	*Then the file:*
No	**doesn't exist**
Yes	**exists**

Figure 100. File Existence Table for Unlabeled Tape File (MVS)

Note to Figure 100:

1. For an unlabeled tape, *file on volume* means that the volume can be positioned to the data set.

In-stream (DD * or DD DATA) Data Set Existence Table (MVS)

If the basic conditions listed under "Basic Conditions" on page 435 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" on page 435, these conditions are in effect:

File Ever Internally Opened	*Then the file:*
No	**doesn't exist**
Yes	**exists**

Figure 101. File Existence Table for Sysout Data Sets (MVS)

Terminal Existence Table (MVS)

If the basic conditions listed under "Basic Conditions" on page 435 are in effect, the file exists:

Unit Record Input Device Existence Table (MVS)

If the basic conditions listed under "Basic Conditions" on page 435 are in effect, the file exists:

Unit Record Output Device Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File Ever Internally Opened	*Then the file:*
No	doesn't exist
Yes	exists

Figure 102. File Existence Table for Unit Record Output Devices (MVS)

Files Whose File Definitions Specify DUMMY Existence Table (MVS)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File Concatenated[1]	File Ever Internally Opened	File Deleted by CLOSE	*Then the file:*
Yes	—	—	exists
No	Yes	No	exists
No	Yes	Yes	doesn't exist
No	No	—	exists

Figure 103. File Existence Table for Files Whose File Definitions Specify DUMMY (MVS)

Note to Figure 103:

1. *Concatenated* refers to a JCL concatenation of data sets having a single ddname, not a set of subfiles having different ddnames.

File Existence Table for Files on Other Devices, Including Subsystems (MVS)

If the basic conditions listed under "Basic Conditions" on page 435 are in effect, the file exists.

VM File Existence Tables

These types of VM files are referenced:

DASD files, page 441
Reusable VSAM files, page 442
Non-reusable VSAM files, page 442
Library members, page 443
Tape files, page 443
Terminals, page 443
Unit record input devices, page 444
Unit record output devices, page 444
Files whose file definitions specify DUMMY, page 444
Files on other devices, page 444

DASD Device File Existence Table (VM)

If the file is not currently internally open and the following conditions are in effect:

Access Restricted	Explicit FILEDEF	File On Minidisk	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	No	—	doesn't exist
No	—	No	Yes	No	exists
No	—	No	Yes	Yes	doesn't exist

Figure 104. File Existence Table for DASD Device (VM)

Reusable VSAM File Existence Table (VM)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

Subfile Sequence Number > 1	File Contains Data	File Ever Internally Opened	File Deleted by CLOSE	*Then the file:*
Yes	—	—	—	doesn't exist
No	Yes	—	—	exists
No	No	No	—	doesn't exist
No	No	Yes	No	exists
No	No	Yes	Yes	doesn't exist

Figure 105. File Existence Table for Reusable VSAM (VM)

Non-Reusable VSAM File Existence Table (VM)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

Subfile Sequence Number >1	File Contains Data	File Ever Internally Opened Opened	*Then the file:*
Yes	—	—	doesn't exist
No	Yes	—	exists
No	No	No	doesn't exist
No	No	Yes	exists

Figure 106. File Existence Table for Non-Reusable VSAM (VM)

Library Member File Existence Table (VM)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

Library On Minidisk[1]	Member Exists	*Then the file:*
No	—	**doesn't exist**
Yes	Yes	**exists**
Yes	No	**doesn't exist**

Figure 107. File Existence Table for Library Member (VM)

Note to Figure 107:

1. A *library* is a CMS TXTLIB, MACLIB, or LOADLIB.

Tape File Existence Table (VM)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File On Volume	*Then the file:*
Yes	**exists**
No	**doesn't exist**

Figure 108. File Existence Table for Tape File (VM)

Terminal Existence Table (VM)

If the basic conditions listed under "Basic Conditions" on page 435 are in effect, the file exists.

Unit Record Input Device Existence Table (VM)

If the basic conditions listed under "Basic Conditions" on page 435 are in effect, the file exists.

Unit Record Output Device Existence Table (VM)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File Ever Internally Opened	*Then the file:*
No	**doesn't exist**
Yes	**exists**

Figure 109. File Existence Table for Unit Record Output Devices (VM)

Files Whose File Definitions Specify DUMMY Existence Table (VM)

If, in addition to the basic conditions listed under "Basic Conditions" on page 435, these conditions are in effect:

File Ever Internally Opened	**File Deleted by CLOSE**	*Then the file:*
No	—	**exists**
Yes	No	**exists**
Yes	Yes	**doesn't exist**

Figure 110. File Existence Table for Files Whose File Definitions Specify DUMMY (VM)

File Existence Table for Files on Other Devices (VM)

If the basic conditions listed under "Basic Conditions" on page 435 are in effect, the file exists.

Appendix H. Considerations for Specifying RECFM, LRECL, and BLKSIZE

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 old data sets, and defaults.

The following sections discuss the priority of processing these values and list the defaults.

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 old data set if it is available.
2. The values are obtained from the old 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 "MVS and CMS Default Values" on page 446.

For a file connected with an OPEN statement, the record format, record length, and block size information of an old 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 old data set, or I/O errors may occur.

For a preconnected file, the record format, record length, and block size information of an old 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.

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 old 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 old file when the file is written over. For initial READ operations, the old file information is used if available according to the rules under "MVS and CMS Default Values."

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.

MVS and CMS Default Values

This section describes the VS FORTRAN process for assigning default values for RECFM, LRECL, and BLKSIZE for programs under either MVS or CMS.

For programs under CMS, keep in mind when reading this section that information from an old 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 448) 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 old 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 old 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 old 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 old data set, if available. If it is not available, 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 old data set, if available. If it is not available, 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 old data set, if available. If it is not available, 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. Installation defaults are used for unavailable values. 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. Installation defaults are used for unavailable values. 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. Installation defaults are used for unavailable values.

Installation Defaults

The IBM-supplied installation defaults for RECFM, LRECL, and BLKSIZE are given in Figure 111 on page 449. 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.

Characteristic	Unit 5	Unit 6	Unit 7	All Other Units
RECFM				
formatted, sequential access	F	UA	F	MVS: U, VM: F (see note 1)
unformatted, sequential access	F	see note 2	F	VS
LRECL				
formatted, sequential access	80	133	80	MVS: 800, VM: 80 (see note 1)
unformatted, sequential access	80	see note 2	80	unlimited
BLKSIZE				
formatted, sequential access	80	133	80	MVS: 800, VM: 80 (see note 1)
unformatted, sequential access	80	see note 2	80	800

Figure 111. IBM-Supplied Installation Defaults for File Characteristics

Notes to Figure 111:

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.

Appendix I. Sample Compiler Listing with Double-Byte Characters

Figure 112 shows a sample compiler listing with Kanji double-byte characters.

```
LEVEL 2.3.0 (MAR  1988)       VS FORTRAN           DEC 07, 1987  13:27:26                                  PAGE:    1

REQUESTED OPTIONS (PROCESS):  DBCS XREF

OPTIONS IN EFFECT: NOLIST NOMAP XREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NOSYM NORENT
                   SDUMP(ISN) NOSXM NOVECTOR IL(DIM) NOTEST NODC NOICA NODIRECTIVE DBCS NOSAA
                     OPT(3) LANGLVL(77) NOFIPS   FLAG(I)  AUTODBL(NONE)  NAME(MAIN)   LINECOUNT(60)      CHARLEN(500)

 IF DO   ISN   *....*...1.........2.........3.........4.........5.........6.........7.*.......8

               C**********************************************************************C
               C* IN THIS LISTING DBCS AND EBCDIC CHARACTERS APPEAR AS FOLLOWS        *C
               C* ( < AND > REPRESENT THE SHIFT-OUT AND SHIFT-IN CHARACTERS):         *C
               C*                                                                     *C
               C*     <ＤＢＣＳ> - DBCS CHARACTERS                                     *C
               C*     EBCDIC     - EBCDIC CHARACTERS                                  *C
               C**********************************************************************C
               C* DBCS VARIABLE NAMES                                                 *C
               C**********************************************************************C
           1         CHARACTER*47 <ＤＢＣＳＶＡＲ一>, <ＤＢＣＳＶＡＲ二>, <ＤＢＣＳＶ>
                    *<ＡＲ三>, INVAR, OUTVAR
           2         CHARACTER <ＣＨＡＲ一>*4, <ＤＢＣＳＶＡＲ四>*103
           3         INTEGER <ＩＮＴ一>, RCX4, RSX4
           4         REAL    <ＲＥＡＬ一>
               C**********************************************************************C
               C* LINE CONTINUED CHARACTER CONSTANT CONTAINING DBCS                   *C
               C**********************************************************************C
           5         DATA  <ＤＢＣＳＶＡＲ三>/'<ＤＢＣＳ　ＣＨＡＲＡＣＴＥＲ　ＳＴＲＩ>
                    *<ＮＧ>'/
           6         <ＤＢＣＳＶＡＲ四> = '<ＤＯＵＢＬＥ> & SINGLE BYTE <ＣＨＡＲＡＣ>
                    *<ＴＥＲ> CONSTANT CONTINUED <ＡＣＲＯＳＳ　２　ＬＩＮＥＳ>'
               C**********************************************************************C
               C* INPUT/OUTPUT OF DBCS CHARACTERS                                     *C
               C**********************************************************************C
           7   10    FORMAT (A32,I3,F5.2,A36)
           8   20    FORMAT ('<ＤＢＣＳ>',3X,A32,3X,I3,3X,F5.2,3X,A36)
           9         OPEN (UNIT=5,ACCESS='SEQUENTIAL',FORM='FORMATTED',CHAR='DBCS')
          10         OPEN (UNIT=6,CHAR='DBCS')
          11         READ(5,10) <ＤＢＣＳＶＡＲ一>,<ＩＮＴ一>,<ＲＥＡＬ一>,
                    *<ＤＢＣＳＶＡＲ二>
          12         WRITE (6,20) <ＤＢＣＳＶＡＲ一>,<ＩＮＴ一>,<ＲＥＡＬ一>,
                    *<ＤＢＣＳＶＡＲ二>
          13         INQUIRE (6,CHAR=<ＣＨＡＲ一>)
          14         IF (<ＣＨＡＲ一> .EQ. 'DBCS') THEN
      1   15           WRITE (6,'(A19)') '<ＣＨＡＲ一> = DBCS'
      1   16         ELSE
      1   17           WRITE (6,'(A22)') '<ＣＨＡＲ一> NOT = DBCS'
      1   18         ENDIF
               C**********************************************************************C
               C* ASSIGNM SUBROUTINE                                                  *C
               C**********************************************************************C
          19         INVAR = '<ＤＢＣＳ　ＣＨＡＲＡＣＴＥＲＳ>'
          20         CALL ASSIGNM (INVAR,OUTVAR,RCX4,RSX4)
          21         WRITE (6,'(A32,3X,I4,3X,I4)') OUTVAR,RCX4,RSX4
          22         END
```

Figure 112 (Part 1 of 2). Sample Compiler Listing with Double-Byte Characters

```
LEVEL 2.3.0 (MAR  1988)        VS FORTRAN           DEC 07, 1987  13:27:26  NAME:MAIN                                 PAGE:    2

SYMBOL CROSS REFERENCE DICTIONARY

PROGRAM NAME: MAIN

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

NAME                TYPE  TAG   DECLARED REFS  (F:REFD  S:SET  B:REFD/MAY BE SET)

<C H A R-->         CHAR  T            2   13     14B
<D B C S V A R一>   CHAR  T            1   11S    12F
<D B C S V A R二>   CHAR  T            1   11S    12F
<D B C S V A R三>   CHAR  VT           1    5
<D B C S V A R四>   CHAR  T            2   6B
<I N T-->            I*4  T            3   11S    12F
<R E A L-->          R*4  T            4   11S    12F
ASSIGNM                   X                20F
INVAR               CHAR  T            1   19B    20B
OUTVAR              CHAR  T            1   20B    21F
RCX4                 I*4  T            3   20B    21F
RSX4                 I*4  T            3   20B    21F


 VARIABLES REFERENCED BUT NOT SET.  (* POSSIBLY SET AS ARGUMENT.)

 <C H A R --> OUTVAR* RCX4* RSX4*


 LABEL CROSS REFERENCE DICTIONARY

 TAGS:
  A-USED AS ARGUMENT  F-FORMAT          S-USED IN ASSIGN STATEMENT
  B-OBJECT OF BRANCH  N-NON-EXECUTABLE

     LABEL    TAG  DEFINED  REFERENCED

        10    NF        7     11


        20    NF        8     12

 *STATISTICS*   SOURCE STATEMENTS = 22, PROGRAM SIZE = 1888 BYTES, PROGRAM NAME = MAIN    PAGE:    1.

 *STATISTICS*   NO DIAGNOSTICS GENERATED.

 **MAIN** END OF COMPILATION 1 ******                                                      TIME STAMP: 87.34113.27.26
```

Figure 112 (Part 2 of 2). Sample Compiler Listing with Double-Byte Characters

Index

VS FORTRAN Version 2
Programming Guide

SC26-4222-3

Reader's Comment Form

If you have applied any technical newsletters (TNLs) to this book, please list them here: ____________

Chapter/Section ____________

____________ Page No. ____________

Comments:

If you want a reply, please complete the following information.

Name ____________ Phone No. (____) ____________

Company ____________

Address ____________

____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

SC26-4222-3

Reader's Comment Form

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, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

VS FORTRAN Version 2
Programming Guide

SC26-4222-3

Reader's Comment Form

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: ____________________

Chapter/Section ____________________

____________________ Page No. ____________________

Comments:

If you want a reply, please complete the following information.

Name ____________________ Phone No. (______) ____________________

Company ____________________

Address ____________________

____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

SC26-4222-3

Reader's Comment Form

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, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

IBM®

Program Number
5668-805
5668-806

File Number
S370-40

The VS FORTRAN Version 2 Library

Diagnosis Guide	LY27-9516
General Information	GC26-4219
Installation and Customization for MVS	SC26-4340
Installation and Customization for VM	SC26-4339
Interactive Debug Guide and Reference	SC26-4223
Language and Library Reference	SC26-4221
Licensed Program Specifications	GC26-4225
Programming Guide	SC26-4222
Reference Summary	SX26-3751

SC26-4222-3

Printed in U.S.A.