IBM

VS FORTRAN Programming Guide

Program Product

```
 IRR(I,J) = 1
 IRI(I,J) = 2
 CONTINUE
 PRINT  20, (
1  I = 1, 3)
 FORMAT (3(1X
 STOP
 END
```

Program Numbers:

5748-FO3 Compiler and Library
5748-LM3 Library Only

Release 4.0

IBM

VS FORTRAN Programming Guide

Program Numbers
5748-FO3 (Compiler and Library)
5748-LM3 (Library Only)
Release 4.0

SC26-4118-0

First Edition (October 1984)

This edition applies to Release 4.0 of VS FORTRAN, Program Products 5748-FO3 (Compiler and Library) and 5748-LM3 (Library only), and to any subsequent releases until otherwise indicated in new editions or technical newsletters. This edition merges *VS FORTRAN Application Programming: Guide*, SC26-3985-4, and relevant parts of *VS FORTRAN Application Programming: System Services Reference Supplement*, SC26-3988-2. Both of these manuals are now obsolete.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Changes specific to Release 4.0 are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

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

Preface

The VS FORTRAN (Virtual Storage FORmula TRANslator) Compiler and Library, Version 1, Release 4.0, program product is sometimes referred to as Level 1.4.0. It is referred to as Release 4 (or 4.0) in this manual. The changes for this edition are summarized under "Summary of Amendments" following this section.

This manual describes how to design, develop, test, and run programs using VS FORTRAN at the 1978 language level. It is designed for two types of users:

- Engineers and scientists who use FORTRAN as a tool in mathematical problem solving
- Application programmers who use the FORTRAN features to code FORTRAN programs

This manual is not intended as a tutorial on the FORTRAN language. It is designed, rather, for the user who has basic knowledge of FORTRAN and now wants to apply that knowledge to coding VS FORTRAN programs.

Organization

The manual is organized in the following manner. Part 1 describes how to write, compile, and execute VS FORTRAN programs; Part 2 tells about specific task implementation when operating in and under certain environments; and the Appendixes contain information that supplements both parts.

Part 1. Using VS FORTRAN in Source Programs
Describes how to develop and run mathematical problem-solving FORTRAN programs. In this part, the steps necessary for the development of FORTRAN application programs are presented in the following order:

- Chapter 1. Introducing VS FORTRAN
- Chapter 2. Referencing Data as Variables, Arrays, and Constants
- Chapter 3. Using Expressions and Assignment Statements
- Chapter 4. Controlling Program Flow
- Chapter 5. Programming Input and Output
- Chapter 6. Subprograms and Shared Data

- Chapter 7. Optimizing Your Program
- Chapter 8. Compiling Your Program and Identifying User Errors
- Chapter 9. Executing Your Program and Fixing Execution-Time Errors
- Chapter 10. Sample Programs and Subroutines

Part 2. Using VS FORTRAN—Environmental Considerations
Describes how to code and/or run with or under various other programs. (You may remove any sections that do not apply to your installation.)

- Chapter 11. Using VS FORTRAN under VM
- Chapter 12. Using VS FORTRAN under MVS
- Chapter 13. Using VS FORTRAN under TSO
- Chapter 14. Using VS FORTRAN under VSE
- Chapter 15. Using VSAM with VS FORTRAN
- Chapter 16. Using VS FORTRAN Interactive Debug with VS FORTRAN
- Chapter 17. Using VS FORTRAN under VM/PC

Appendixes
Contain the following supplementary information:

- Assembler Language Considerations
- Object Module Records
- Differences Between VS FORTRAN and Other IBM FORTRANs
- Internal Limits in VS FORTRAN

Glossary
See *VS FORTRAN Language and Library Reference* for terms specific to VS FORTRAN as well as some general programming definitions.

Industry Standards

The VS FORTRAN Compiler and Library program product is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of May, 1982. The following two standards are technically equivalent:

- 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

In addition, the bit string manipulation functions are defined in ANSI/ISA-S61.1.

In this manual, the following two standards are also technically equivalent:

- American Standard FORTRAN, X3.9-1966 (also known as FORTRAN 66)
- International Organization for Standardization ISO R 1539-1972 Programming Languages-FORTRAN

Both the FORTRAN 77 and the FORTRAN 66 standard languages include IBM extensions.

Related Publications

This manual is designed as a guide to VS FORTRAN. It is not intended as a reference manual. Related information is in the following publications.

VS FORTRAN

- *VS FORTRAN Language and Library Reference*, GC26-4119

  Describes each syntactic element available in FORTRAN 77 and the types of subprograms in the VS FORTRAN library. Also contains library, execution-time, and operator messages. Has information about the method used in the library to compute a mathematical function and the effect of an argument error upon the accuracy of the answer returned.

- *VS FORTRAN Compiler and Library Installation and Customization*, SC26-3988

  Contains information on installation planning, on the compiler and library installation macros, storage requirements, and administrative information about controlling input/output, and how to create and alter the option table.

- *VS FORTRAN Compiler and Library Diagnosis Guide*, SC26-3990

  Describes how to diagnose failures in the VS FORTRAN compiler and library.

- *VS FORTRAN Compiler and Library Reference Summary*, SX26-3731

  A pocket-size booklet containing the syntax of each VS FORTRAN statement, compiler options, and additional information condensed from *VS FORTRAN Language and Library Reference*.

In addition, a binder for VS FORTRAN publications and a combination of binder and publications are available.

- Binder only, SX26-3747
- Binder and the following publications, SBOF-1192

  –VS FORTRAN Programming Guide

  –VS FORTRAN Language and Library Reference

  –VS FORTRAN Compiler and Library Reference Summary

FORTRAN IV

- *IBM System/360 and System/370 FORTRAN IV Language*, GC28-6515

  Describes the source language available in the FORTRAN IV language (VS FORTRAN at the 1966 language level).

- *FORTRAN Coding Form*, GX28-7327

  Aids in coding fixed-form FORTRAN programs.

VS FORTRAN Interactive Debug (IAD)

- *VS FORTRAN Interactive Debug Guide and Reference*, SC26-4116
- *VS FORTRAN Interactive Debug Installation*, SC26-4117
- *VS FORTRAN Interactive Debug Diagnosis*, SY26-3944
- *VS FORTRAN Interactive Debug Reference Summary*, SX26-3742

Time Sharing Option (TSO)

See "MVS" below for TSO publications.

Virtual Storage Access Method (VSAM)

See "MVS" and "VSE" below for VSAM publications.

System and Device Publications

Specific system information and details about block size, track capacity, and so on, of the various input/output devices are **not** included in this manual. See the following publications for this information:

IBM 3800 Printing Subsystem

IBM 3800 Printing Subsystem Programmer's Guide, GC26-3846

IBM Assembler

Assembler H Version 2 Application Programming: Language Reference, GC26-4037

Assembler H Version 2 Application Programming: Guide, SC26-4036

OS/VS–DOS/VSE–VM/370 Assembler Language, GC33-4010

OS/VS–VM/370 Assembler Programmer's Guide, GC33-4021

Guide to the DOS/VSE Assembler, GC33-4024

IBM DASD

Introduction to IBM Direct Access Storage Devices and Organization Methods, GC20-1649

Contains algorithms for direct files.

MVS

OS/VS Linkage Editor and Loader, GC26-3813

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

OS/VS Tape Labels, GC26-3795

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

OS/VS2 MVS Supervisor Services and Macro Instructions, GC28-1114

OS/VS2 MVS Access Method Services, GC26-3841

OS/VS2 MVS JCL, GC28-0692

OS/VS2 MVS Debugging Guide, GT28-0632

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

OS/VS2 MVS TSO Command Language Reference, GC28-0646

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

MVS/Extended Architecture Integrated Catalog Administration: Access Method Services Reference, GC26-4019

MVS/Extended Architecture VSAM Catalog Administration: Access Method Services Reference, GC26-4075

MVS/Extended Architecture JCL, GC28-1148

MVS/Extended Architecture Debugging Handbook Vols. 1-5, GC28-1164 through GC28-1168

MVS/Extended Architecture Data Administration Guide, GC26-4013

MVS/Extended Architecture Supervisor Services and Macro Instructions, GC28-1154

MVS/Extended Architecture Linkage Editor and Loader, GC26-4011

MVS/Extended Architecture VSAM Administration Guide, GC26-4015

MVS/Extended Architecture Magnetic Tape Labels and File Structure Administration, GC26-4003

MVS/Extended Architecture TSO Command Language Reference, GC28-0646, as updated by Supplement SD23-0259

MVS/Extended Architecture TSO Extensions TSO Command Language Reference, SC28-1134

VM/SP and CMS

VM/SP CP Command Reference for General Users, SC19-6211

VM/SP CMS User's Guide, SC19-6210

VM/SP CMS Command and Macro Reference, SC19-6209

VM/SP Terminal Reference, GC19-6206

VSE

VSE/Advanced Functions System Management Guide, SC33-6094

VSE System Data Management Concepts, GC24-5209

VSE/Advanced Functions Serviceability Aids and Debugging Procedures, GC33-6099

VSE/Advanced Functions Tape Labels, SC24-5212

VSE/Advanced Functions DASD Labels, SC24-5213

VSE/Advanced Functions System Control Statements, SC33-6095

VSE/Advanced Functions System Utilities, SC33-6100

VSE/Advanced Functions Macro Reference, SC24-5211

VSE/VSAM Programmer's Reference, SC24-5145

Using VSE/VSAM Commands and Macros, SC24-5144

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

VM/PC

IBM Virtual Machine/Personal Computer User's Guide, SC24-5254

IBM Extension Documentation

This manual describes how to use the VS FORTRAN source language, which is comprised of standard language and IBM language extensions to the standard. The IBM extensions are indicated in the following ways:

IBM Extension

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

End of IBM Extension

In examples and figures, IBM extensions are boxed:

```
DATA TYPE     VALID STORAGE LENGTHS     DEFAULT LENGTH

Integer       | 2 |  or 4                4
```

Syntax Used in This Manual

The following paragraphs define how to interpret the syntax used in this manual.

- Uppercase letters, words, and numbers must be coded in the statement exactly as shown. For example, the word JOB in a format is to be coded as JOB.

- Lowercase letters and words represent variables, for which user-supplied information is substituted. For example, the word "option" in a format can be coded as NODECK.

- Symbols in the following list must be coded exactly as shown:

apostrophe	'
asterisk	*
colon	:
comma	,
equal sign	=
parentheses	()
period	.
slash	/

- Hyphens (-) join lowercase letters and words and symbols to form a single variable name. For example, the word "program-name" in a format could be coded as MYPROG1.

- Square brackets ([ ]) group optional items from which none, one, or more choices can be made. For example, the sequence "option[,option]" in a format could be coded as NODECK or as NODECK,XREF as needed.

- Ellipses (...) specify that the preceding syntactical unit can, optionally, be repeated. A *syntactical unit* is a single syntactical item, or a group of syntactical items enclosed in braces or brackets. For example, the sequence "option[,option]..." in a format could be coded as NODECK or as NODECK,XREF or as NODECK,XREF,MAP as needed.

- OR signs (|) specify that only one of the list of units they separate can be coded.

- Blanks are used to improve readability. In FORTRAN statements, they are not significant. In non-FORTRAN statements, they may be significant. Any non-FORTRAN statement should be coded exactly as shown.

Summary of Amendments

October 1984

Merger of Programming Guide and System Services Manual

The *VS FORTRAN Application Programming: Guide*, SC26-3985, and selected parts of *VS FORTRAN Application Programming: System Services Reference Supplement*, SC26-3988, have been merged into this manual.

Release 4.0 Enhancements

VSAM Key-Sequenced Data Sets

VS FORTRAN programs can now load and access VSAM KSDS files:

- Records can be retrieved, added, replaced, and deleted, using key values (designated fields within the records).
- Both direct and sequential processing (by key value) are allowed.
- Multiple alternate keys, as well as a primary key, can be used.

REWRITE and DELETE statements have been added to the language to process these files, and some existing I/O statements have been expanded.

Reentrant Object Code (MVS and VM)

The compiler can create a reentrant version of the object-code portion of a program. When object code is reentrant (and placed in a reentrant area), multiple end-users can share a single copy, thereby saving execution-time storage.

Execution-Time Loading of Library Routines

The library has been restructured to allow more execution-time loading of library routines. This has multiple benefits:

- Reduces auxiliary storage requirements for load modules
- Speeds execution for users in compile-link-go mode

- In an MVS/XA environment, allows many library routines to reside above 16 megabytes, thus providing virtual-storage constraint relief.

(This new library design will not impact users who have Release 2 or Release 3 load modules that access the old reentrant I/O library (via IFYVRENT), and who do not want to relink. Maintenance is automatically provided, and relinking is only necessary if Release 4 function is desired.)

Automatic Precision Increase

This feature allows a user to selectively boost the precision of floating-point items in an existing program without recoding it. Single precision items can be made double, double can be made extended. Users merely recompile the program with a specified option (AUTODBL).

Faster Character Handling

Character assignment and comparison operations are now handled by in-line code, rather than by calls to the library. This speeds execution time. Error messages previously issued from the library, for conditions such as overlap detection and invalid character length, will no longer appear.

Improved Diagnostic Support

The following enhancements will allow easier program maintenance and debugging.

- MAP and XREF output can be formatted to fit a terminal screen.
- LIST output now gives ISNs, and XREF output now identifies variables referenced but not initialized.
- An explicit SDUMP compiler option is now available (previously, this was available only as an installation-wide default).
- SDUMP tables have been condensed and simplified, decreasing object module size. The symbol table size, however, remains the same.
- Execution-time error messages have been expanded to supply line numbers, ISNs, and offsets.

Improved I/O Support

The following improvements have been made to VS FORTRAN I/O statements:

- For sequential unformatted I/O, you can now use all record formats. Fixed, fixed blocked, undefined, variable, and variable spanned formats are supported.
- You can now use data initialization values in the character and double precision explicit-type statements.
- You can specify a character type unit designator for list-directed READ and WRITE statements. This allows you to do list-directed reads and writes to an internal file.

- The NUM parameter is now a valid control list parameter for the unformatted READ I/O statements for LANGLVL(77). The NUM parameter returns the number of bytes transferred.

- Several extensions have been made to the namelist READ and WRITE statements. You can now use the keywords UNIT and FMT. The unit designator for namelist I/O can be character type, so you can do namelist reads and writes to an internal file. The unit designator can also be an asterisk to represent an installation-dependent unit. You can now use a shortened form for reading and printing at LANGLVL(77).

Release 3.1, March 1984

VS FORTRAN Interactive Debug Support

When a VS FORTRAN program is executed, the user has a choice of two different execution options:

- DEBUG, which activates VS FORTRAN Interactive Debug immediately; and

- NODEBUG, the IBM default, which does not invoke VS FORTRAN Interactive Debug.

Note: The TEST compiler option is not necessary for VS FORTRAN Interactive Debug.

Release 3.0, March 1983

Character Data Type Handling

VS FORTRAN Release 3.0 provides for passing character length arguments in a manner that is not apparent to the user.

In addition:

- Character and noncharacter data types are allowed in the same common block.

- Character and noncharacter data types are allowed in an EQUIVALANCE relationship.

- The CHARLEN compiler option may be specified to set the maximum length of the character data type to a range of 1 through 32767. The default maximum length remains 500 characters, or whatever was set at installation time.

- The SC option has been removed because the character length is now passed in a manner that is not apparent to the user.

Debugging and Diagnostic Aids

- The TRMFLG compiler option may be specified to display a source statement in error on the SYSTERM data set, along with the diagnostic message.
- A symbolic dump of variables at abnormal termination can be obtained for modules *not* compiled with the NOSDUMP compiler option.
- A symbolic dump of variables in a module *not* compiled with the NOSDUMP option can be obtained on request by calling the SDUMP library routine.
- The SYM compiler option may be specified to produce SYM cards along with the object deck.
- The SRCFLG compiler option may be specified to insert diagnostic messages in the printed source listing.

INCLUDE Statement Improvement

- INCLUDE statements can be selectively activated during compilation.
- Blocked file support has been added to the INCLUDE facility.

Miscellaneous Changes

- OPEN, CLOSE, and INQUIRE parameters that are constants are checked at compile time.
- VS FORTRAN continues executing after transmission input/output errors have occurred.
- Formatting for a new direct-access data set has been provided for the OPEN statement.
- For direct-access I/O, the records of a file must be either all formatted or all unformatted, not mixed.
- Various service changes have been made.

Warning: Every program that has been compiled with versions of VS FORTRAN previous to Release 3.0, and that either references or defines a user subprogram that has character-type arguments or is itself of character type, must be recompiled with VS FORTRAN Release 3.0.

Contents

Figures

Part 1. Using VS FORTRAN in Source Programs

Part 1 discusses the following topics:

The sample programs and subroutines at the end of this part illustrate how to use VS FORTRAN to obtain solutions to problems.

Chapter 1. Introducing VS FORTRAN

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

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

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

VS FORTRAN is designed to make use of IBM's virtual storage architecture.

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

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

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

The VS FORTRAN compiler operates under control of one of the following operating systems:

- VM/SP (all releases)
- MVS/SP (all releases, including MVS/XA)
- VSE/Advanced Functions (Release 3 or later)
- VM/PC (Release 1 or later)

In this manual, the VM/SP system is referred to as "VM"; the MVS systems are collectively referred to as "MVS" (although special MVS/XA considerations are discussed in Chapter 12, "Using VS FORTRAN under MVS" on page 257); and

VSE/Advanced Functions is referred to as "VSE." VM/PC is discussed in Chapter 17, "Using VS FORTRAN under VM/PC" on page 381.

You can compile your program under any one of these systems and then link-edit the program and its subroutines to execute under any of the others.

The examples given in the following chapters are included in the sample programs in Chapter 10, "Sample Programs and Subroutines" on page 213.

Note: The VSE/Advanced Functions version of the VS FORTRAN compiler and library is not supported under CMS/DOS.

VS FORTRAN—A Quick Overview

The VS FORTRAN Compiler and Library is an IBM program product that runs under the systems listed above. The Compiler and Library are packaged together as a single program product; the Library is also available separately.

VS FORTRAN implements the most recent FORTRAN standards and maintains older capabilities. The compiler accepts two language levels:

FORTRAN 77—that is, 1978 American National Standard FORTRAN (technically equivalent to ISO FORTRAN 1980) plus IBM extensions

FORTRAN 66—that is, 1966 American National Standard FORTRAN (technically equivalent to ISO FORTRAN 1972) plus IBM extensions

VS FORTRAN has facilities that can make system management easy, reduce programming development effort, and increase programmer productivity.

In addition, the following sections list features that give VS FORTRAN efficient execution-time performance plus ease of use.

VS FORTRAN Features

1978 ANSI
Complies with standards.

1966 ANSI
Complies with standards.

Optimized Object Code
Enables faster execution because four levels of optimization are available.

Reentrant Compiler and I/O Library
Offers virtual storage savings because the compiler and many library modules can be shared concurrently by many problem programs.

Cross-Compilation
Enables flexible system use because you can compile on one of the supported operating systems, link-edit, and execute on another supported system.

Upward Compatibility

Enables valid source programs written in the old language to be link-edited to execute with programs written in current language; old programs can make use of the VS FORTRAN library routines.

Warning: Programs compiled with versions of VS FORTRAN prior to Release 3.0 that reference a subprogram with character-type arguments, must be recompiled with VS FORTRAN Release 3.0 and later. Character subprograms compiled with versions of VS FORTRAN prior to Release 3.0 must be recompiled. Any program compiled with versions of VS FORTRAN prior to Release 3.0 that references the intrinsic character manipulation functions CHAR, LEN, INDEX, LGE, LGT, LLE, or LLT, must be recompiled.

VSAM Capabilities

Let you access key sequenced (KSDS), entry sequenced (ESDS), and relative record (RRDS) files directly from the VS FORTRAN program.

INCLUDE Statements

Let you place prewritten sequences of code inline in your programs. INCLUDE statements can be selectively activated during compilation.

Bit String Manipulation Functions

Provide for interrogation and manipulation of integer data on a bit-by-bit basis.

DC Compiler Option

Defines the names of common blocks to be allocated at execution time. This allows specification of very large common blocks that can reside in the additional storage space available through MVS/XA.

Formatted Traceback

Provides debugging information in sentence form.

DEBUG Execution-Time Option

Allows compiled programs to execute under the control of VS FORTRAN Interactive Debug.

Free-Form Source Option

In VS FORTRAN, both free format and fixed format are available. You can use whichever you prefer when coding new programs, and existing programs can always be recompiled without change to their source format. For more details, see "Using Fixed- and Free-Form Input" on page 7.

Language Flagger

Helps you ensure that VS FORTRAN programs conform to the selected level of the current FORTRAN standard.

Symbolic Dump at Abend

Provides a symbolic dump of variables at abnormal termination; a symbolic dump of variables can also be requested on the object time error unit.

Asynchronous Input/Output Statements and Options
Enable you to transfer unformatted data between external sequential files and arrays in your FORTRAN program and, while the data transfer is taking place, continue other processing (MVS only).

Alternate Mathematical Library
Gives more precision for some mathematical functions.

NAMELIST Statement
Specifies I/O list of data names.

Character and Noncharacter Data Types
Allowed in the same common block and in an equivalence relationship. In addition, you may increase (by compiler option) character data type length from 500 bytes to a maximum of 32767 bytes.

Extended Error Handling
Gives you control over program execution after an error.

FORTRAN Language Level 77

OPEN, CLOSE, and INQUIRE Statements
Let you define and control FORTRAN files.

Internal Files
Let you format records in virtual storage.

IF Statements
Specify alternate paths of execution, through arithmetic, logical, and block IF (including ELSE, ELSE IF, END IF) statements.

Character Data Type
Provides flexible and direct control of string data by means of:

- Concatenation
- Substring
- Internal file

PARAMETER Statement
Defines named constants.

Intrinsic Functions
Include generic functions, arithmetic functions, and character functions.

IMPLICIT Statement
Provides user-specified default data type declaration.

Expressions
Allow valid combinations of arithmetic, character, and logical variables, and other expressions.

DO Statement
Gives a convenient way to program loops; integer, real, and double precision DO variables are allowed; negative incrementation parameter is allowed.

PROGRAM Statement
Names a main program.

GENERIC Statement
Specifies generic function names.

INTRINSIC Statement
Explicitly defines intrinsic functions.

SAVE Statement
Saves values in named common blocks, variables, or arrays after a called program completes executing.

Edit Descriptors
Provide an extensive set of edit descriptors.

List-Directed Formatting
Provides default formatting.

Variable Formats
Let you construct I/O formats at execution time.

FORTRAN Language Level 66

GENERIC Statement
Specifies generic function names.

Ampersand (&)
Provided for use as special character.

DEFINE FILE Statement
Specifies a direct access file.

FIND Statement
Efficiently locates next input record in a direct access file.

PUNCH Statement
Provides installation-dependent WRITE statement.

Using Fixed- and Free-Form Input

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

IBM Extension

If you're using free-form input, you can enter your source program into the file, line by line, according to the rules for free-form source programs.

The maximum line length you can enter is 80 characters; however, your source statements (excluding statement numbers and statement break characters) can be up to 1320 characters long.

You must ensure that sequence numbers do not appear in your free-form source (columns 73 through 80).

End of IBM Extension

For reference documentation about VS FORTRAN fixed- and free-form input, see *VS FORTRAN Language and Library Reference*.

Chapter 2. Referencing Data as Variables, Arrays, and Constants

Kinds of Data and Data Entities

The basic kinds of data available to the programmer in VS FORTRAN are arithmetic, character, and logical data:

> *Arithmetic data* is decimal numbers, with either integer or real value (including complex numbers).
>
> *Character data* is alphameric strings.
>
> *Logical data* is truth values.

This data is made available to a VS FORTRAN program by way of program data entities called variables, arrays, and constants, that you can declare:

> *Variables and arrays* are program areas in which the working data of the program resides while being used.
>
> *Constants* are fixed data values known to the program, provided as necessary in the program context.

Variables, arrays, and constants all have associated data types and lengths corresponding to the associated data values. Variables and arrays are named so that program statements may refer to them; constants may be named if desired.

A variable contains a single value, or datum; an array contains multiple values, or data, of the same type and length. A single value of an array is called an element of the array, and is referred to by its relative position in the array and the name of the array.

Variables or arrays may be declared, for example, to internally store for program use the contents of a record of an external data file as the file is read by the program. The data in successive records referred to by the variable or array would most likely vary in value; hence the name "variable."

An example of a constant is the number 3.1416 appearing in a source statement. This number is an unvarying arithmetic constant that is stated in the program for use in calculations involving pi.

In VS FORTRAN, you may default and/or explicitly declare the data types and lengths of the program variables, arrays, and constants.

VS FORTRAN statements useful for declaring data types and lengths are summarized below:

Statement Name	Statement Function
IMPLICIT	Explicitly specifies the data types and lengths of all variables, arrays, and user-supplied functions whose names begin with particular letters.
Explicit Type	Declares the data type for specific variables, arrays, and functions.
PARAMETER	Names a value and allows programs to subsequently refer to that value by the name.
DIMENSION	Defines an array of up to seven dimensions.
DATA	Initializes variables and arrays to the stated values.
EQUIVALENCE	Controls storage allocation within a program.

The examples in Chapters 2 and 3 are summarized in Chapter 10, "Sample Programs and Subroutines" on page 213.

Declaring Data Types and Lengths

When you write a VS FORTRAN program, you declare the data types and lengths associated with your variable and array names either implicitly or explicitly.

The available data types and their possible and default lengths are shown in Figure 1 on page 11.

DATA TYPE	POSSIBLE STORAGE LENGTHS IN BYTES	DEFAULT LENGTH IN BYTES
INTEGER	2 $\boxed{\text{or 4}}$	4
REAL	4 $\boxed{\text{, 8 or 16}}$	4
DOUBLE PRECISION	8	8
COMPLEX	8 $\boxed{\text{, 16 or 32}}$	8
LOGICAL	$\boxed{\text{1 or}}$ 4	4
CHARACTER	1 through 32767	1

Figure 1. Available Data Types and Lengths for Variables and Arrays

The data types integer, real, double precision, and complex describe arithmetic data:

- The integer data type refers to whole numbers for which fixed-point arithmetic operations are used.

- The real data type refers to decimal numbers for which floating-point arithmetic operations are used.

- The double precision data type refers to the use of two computer words to represent a number in accordance with the required precision.

- The complex data type refers to paired decimal numbers—first the real part and then the imaginary part, both regarded as real numbers.

The logical data type describes truth-value data, which may have only the value true or false. Logical variables are used in logical operations.

The character data type describes alphameric data, that is, the characters in the character code known to the computer system. For VS FORTRAN, these are EBCDIC coded characters that are handled as byte strings.

You define the data type of a variable or array name either through implicit naming conventions or through explicit definitions.

Implied Default Data Type Declaration

In VS FORTRAN, if you don't otherwise declare a name, it is given a data type of real or integer, and a length of 4 bytes, depending on the initial letter:

Names that begin with I through N are given type integer 4 bytes long.

Names that begin with any other letter are given type real 4 bytes long.

IBM Extension

Names that begin with the currency symbol ($) are given type real 4 bytes long.

End of IBM Extension

No other data types have an implied default declaration.

Explicit Data Type Declaration

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

- Implicitly

  In the IMPLICIT statement, the first letter of a name is associated with a particular type. Thus, if you want to declare all names beginning with a certain letter or $ (or range of letters) as a particular type and, optionally, length, use the IMPLICIT statement. For example,

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

  would cause a name, CAT, to be given type character, 15 characters long.

  The IMPLICIT statement declaration takes precedence over the implied default typing.

- Explicitly

  To identify a specific name as a particular type, use an explicit type statement (integer, real, complex, logical, or character). The way to explicitly accomplish the same thing shown in the above example for the variable CAT is as follows:

  ```
  CHARACTER*15 CAT
  ```

  An explicit specification takes precedence over an IMPLICIT declaration.

Typing Groups of Names—IMPLICIT Statement

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

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

your program will treat data items as shown below.

NAMES BEGINNING WITH	HAVE DATA TYPE	HAVE LENGTH
A through C, and F	DOUBLE PRECISION	8
E and L	LOGICAL	4
D, G, and H	CHARACTER	1
I through K, M, N	INTEGER	4 (default)
O through z $\boxed{\text{and \$}}$	REAL	4 (default)

IBM Extension

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

(Y - B)

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

End of IBM Extension

Typing Specific Names—Explicit Type Statements

Explicit type statements declare the data type for specific names in your program. For such names, you can specify the data type and, optionally, the length. You may optionally specify dimension information for array names. You may also optionally assign initial values for names of every data type.

For example, you can specify:

```
DOUBLE PRECISION   MEDNUM
CHARACTER*80     INREC /'ABCD'/
```

IBM Extension

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

As an alternative for MEDNUM, you could specify:

```
REAL*8          MEDNUM
```

End of IBM Extension

These statements declare that:

MEDNUM is a real variable 8 bytes long.

INREC is a character variable 80 bytes long and contains ABCD followed by 76 blanks.

COUNTR is an integer variable 2 bytes long.

BIGNUM is a real variable 16 bytes long.

ARRAY2 is a two-dimensional array, with elements 4 bytes long (specified by *4). There are five elements in each dimension (specified by (5,5)) for a total of 25 elements. Arrays are explained in "Arrays and Subscripts" on page 20.

If you specify an IMPLICIT statement in the same program as these explicit type statements—such as one of the following:

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

the explicit type statements override the IMPLICIT statement specifications, and in your program:

INREC is a character variable 80 bytes long, not an integer variable 4 bytes long.

MEDNUM is a real variable 8 bytes long, not an integer variable 4 bytes long.

But all other names beginning with I or M represent integers that are each 4 bytes long.

BIGNUM is a real variable 16 bytes long.

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

COUNTR is an integer variable 2 bytes long.

But all other names beginning with A, B, or C represent real variables 8 bytes long.

Constants

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

The constants you can use are:

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

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

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

IBM Extension

- **Hollerith**—use Hollerith constants as data only in FORMAT statements and in initialization other than character initialization.

- **Hexadecimal**—use hexadecimal constants to initialize items.

- **Literal** (FORTRAN66 only)—similar in usage to character constants.

End of IBM Extension

Defining Constants by Value

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

```
CIRC =2*PI*RAD

   or

CIRC =2.0*PI*RAD
```

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

You can specify all types of constants in this way:

- **Arithmetic Constants**—integer, real, complex, or double precision

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

    ```
    -12345
     12345
    ```

– **Real Constant**—can take two forms:

1. **Basic Real Constant**—written as an optional sign, followed by an integer part (made up of digits), followed by a decimal point, followed by a fraction part (made up of digits). Either the integer or fraction part can be omitted. In the latter case, a decimal point is not necessary. For example:

```
+123.45
0.12345
```

2. **Basic Real Constant with Real Exponent**—written as a basic real constant followed by a real exponent; the real exponent is written as one of the letters D, E, or Q, followed by a 1- or 2-digit integer constant. Optionally, the exponent can be signed. (See D exponent below for example of double precision constant.) For example:

0.12345E+2	E exponent (which occupies four storage positions and has the value +12.345; the precision is approximately 6 decimal digits)
0.12345D-03	D exponent (which occupies eight storage positions and has the value +0.00012345; the precision is approximately 15 decimal digits)

IBM Extension

-1234.5Q03	Q exponent (which occupies 16 storage positions and has the value -1,234,500; the precision is approximately 32 decimal digits)

End of IBM Extension

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

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

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

IBM Extension

(123.45,-123.45D2)	(has the value +123.45-12345i; the real part is 4 bytes long, the imaginary part is 8 bytes long) (The real part (a real constant) is converted from 4 bytes long to a real constant that is 8 bytes long.)
12345,-123.45Q2)	(has the value +12345-12345i; the real part is 4 bytes long, the imaginary part is 16 bytes long) (The real part (an integer constant) is converted from 4 bytes long to a real constant that is 16 bytes long.)

End of IBM Extension

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

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

IBM Extension

(You can also use T and F as abbreviations in the DATA initialization statement.)

End of IBM Extension

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

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

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

- **Character Constant**—a string of characters, enclosed in apostrophes. The character string can contain any characters in the computer's character set. For example:

```
'PARAMETER = '

'THE ANSWER IS:'

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

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

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

IBM Extension

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

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

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

  VS FORTRAN now implements Hollerith constants for the FORMAT and DATA statements, for noncharacter data type, and for subroutine and function calls. For example:

```
DATA INT2/2HEF/, REAL4/4HABCD/
RESLT = FUNCT1 ( A,B, 4H1234)
```

  However, Hollerith constants in a DATA statement are not recommended.

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

  You can use hexadecimal constants only in a DATA statement to initialize data items of all types and in explicit type statements that allow initialization.

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

- **Literal Constant**—in FORTRAN 66, performs functions similar to the FORTRAN 77 character constant, and the Hollerith constant. (Reference documentation is contained in *IBM System/360 and System/370 FORTRAN IV Language.*)

End of IBM Extension

Defining Constants by Name—PARAMETER Statement

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

There are two advantages in handling constants this way:

- The name for the constant can be a meaningful name—which makes the logic of the program easier to understand by someone doing maintenance updates.

- If the value of the constant must be changed, you can change it once, in the PARAMETER statement, and all references throughout the program are updated.

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

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

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

> C1 has the value "DATE ." The constant is five characters long; the blank following the word DATE is part of the constant.
>
> C2 has the value "TIME ." The constant is five characters long; the blank following the word TIME is part of the constant.
>
> RATE is defined implicitly as a REAL*4 item. Therefore, it's a real constant, four storage positions long, with a value of 2 times 1.414 or 2.828.
>
> You'll note that RATE is defined through the expression 2*1.414; when you define a constant using an expression in this way, the expression you specify must be a constant expression, and can contain no implicit or explicit function references.

In the PARAMETER statement, the value you assign to the constant must be consistent with its data type; that is, C1 and C2 must contain character data, and RATE must contain real data. If any data conversions must be performed, they are made according to the rules for the assignment statement. (To find out what the "assignment statement" is, see "Assigning Values to Variables, Array Elements, and Character Substrings" on page 50.)

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

Variables

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

The first occurrence of a variable name defines the variable; no specific definition statement is needed.

The first letter of the name of a variable determines its data type, as described in "Implied Default Data Type Declaration" on page 12, or you can define its data type explicitly, as described in "Explicit Data Type Declaration" on page 12.

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

Arrays and Subscripts

An array is a named set of consecutive data locations that have the same data type and length.

In FORTRAN, you assign a name to the entire array and then refer to each of the individual locations, called array elements, by specifying its position within the array through one or more subscripts, depending upon the number of dimensions in the array.

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

One-Dimensional Arrays

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

```
DIMENSION ARRAY1(5)
```

In this case, you've defined ARRAY1 as an array containing five elements, each implicitly defined as a real item 4 bytes long.

Subscript References: Program references to ARRAY1 take the form of subscripts:

- As integer constants:

  ```
  ARRAY1(2)
  ```

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

- As integer variables:

  ```
  ARRAY1(NUM)
  ```

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

IBM Extension

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

End of IBM Extension

Multidimensional Arrays

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

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

For example, if you define the 3-dimensional array

```
REAL*4 ARR3(2,2,2)
```

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

Arrays—Implicit Lower Bounds

In the preceding examples, the subscripts are shown as having a range from 1 through the upper bound for each dimension of the array; that is, in ARR3, the implicit lower bound for each dimension is 1, and the explicit upper bound for each dimension is 2.

Arrays—Explicit Lower Bounds

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

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

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

ARR3—Implicit Lower Bounds

1,1,1	2,1,1	1,2,1	2,2,1	1,1,2	2,1,2	1,2,2	2,2,2

ARR3A—Explicit Lower Bounds

4,2,1	5,2,1	4,3,1	5,3,1	4,2,2	5,2,2	4,3,2	5,3,2

Figure 2. Three-Dimensional Array—Implicit and Explicit Lower Bounds

Arrays—Signed Subscripts

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

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

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

the two arrays are laid out in storage as shown in Figure 3:

- Valid array references for ARR2 range from ARR2(1,1) through ARR2(4,2), and there are eight array elements.
- Valid array references for ARR2S range from ARR2S(-2,1) through ARR2S(2,2) (with ARR2S(0,1) and ARR2S(0,2) included), and there are ten array elements.

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

ARR2(4,2)—is arranged in storage like this:

1,1	2,1	3,1	4,1	1,2	2,2	3,2	4,2

ARR2S(-2:2,2)—is arranged in storage like this:

-2,1	-1,1	0,1	1,1	2,1	-2,2	-1,2	0,2	1,2	2,2

Figure 3. Arrays—Effect of Negative Lower Bounds

Arrays—Execution-Time Considerations

When performance is critical, as in inner loops, specify arrays as one-dimensional rather than multidimensional. The fewer the dimensions in an array, the faster your array references execute. Always be sure that subscript values refer to elements within the bounds of the array; if they don't, you may destroy data or instructions.

For large arrays of more than a few thousand elements, the program should endeavor to vary the first subscript most rapidly in order to localize the reference pattern and prevent excessive "paging." See the example under "Processing Multidimensional Arrays—Nested DO Statement" on page 63.

Substrings of Character Data

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

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

For example:

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

  To declare character VAR1, do the following:

  ```
  CHARACTER*10 VAR1
  ```

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

  To declare character ARR1, do the following:

  ```
  CHARACTER*10 ARR1(2)
  ```

You can omit the lower bound of the substring reference if it is equal to 1; that is, ARR1(2)(:4) is exactly equivalent to ARR1(2)(1:4).

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

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

and you specify the following assignment:

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

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

To omit the upper bound of the substring reference to SUVAR in the example above, you would specify:

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

Using Data Efficiently

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

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

Initializing Data—DATA Statement

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

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

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

```
DEG    (REAL constant)         initialized to    10.2
CELS   (CHARACTER constant)    initialized to    DG
CARL   (CHARACTER constant)    initialized to    SURD
AVCH   (REAL constant)         initialized to   .1515
```

You can also use named constants to initialize data items:

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

which initializes the real variable DEG to the value 10.2.

Initializing Arrays—DATA Statement

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

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

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

The array elements are initialized as follows:

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

Initializing Character Array Elements: In a character array, it isn't necessary to specify the constant as the same length as the character array element:

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

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

For example, if you specify the following statements:

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

the CARRAY array is initialized as follows:

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

Initializing Arrays—Implied DO Lists: You can use implied DO lists to initialize parts or all of an array. You use the implied DO list to specify the values the subscripts should assume.

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

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

This DATA statement tells the compiler to:

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

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

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

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

This DATA statement tells the compiler to:

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

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

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

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

Example:

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

Invalid Example:

If more than one implied-DO list shares a list of constants or names of constants (see example below), then no implied-DO variable may be shared by any two of these implied-DO lists:

```
      DIMENSION K(3),L(3),M(2)
      DATA (K(I),I=1,3),(L(I),I=1,3),(M(J),J=1,2)/8*1/
```

where the DO-variable (I) is shared by the first two DO-lists that also share the list of constants in the above DATA statement. VS FORTRAN interprets this DATA statement as a three-level, nested DO-loop:

```
      DO 20 I=1,3
      K(I)=1
      DO 20 I=1,3
      L(I)=1
      DO 20 J=1,2
      M(J)=1
 20   CONTINUE
```

This will cause a compile-time error because DO-variable (I) cannot be redefined within a DO-loop. The following is valid:

```
      DATA (K(I),I=1,3),(M(J),J=1,2)/5*1/,(L(I),I=1,3)/3*1/
```

For more information on DO loops, see "Programming Loops—DO Statement" on page 64.

Managing Data Storage—EQUIVALENCE Statement

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

When your program's logic permits it, you can use this statement to specify that one storage area is to be shared by two or more variables and/or arrays of the same or differing data types.

IBM Extension

Character data can be equivalenced with noncharacter data.

End of IBM Extension

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

The EQUIVALENCE statement is particularly useful with the COMMON statement; this kind of usage is described in "EQUIVALENCE Considerations—COMMON Statement" on page 130.

When you use the EQUIVALENCE statement with array elements, you can implicitly specify the storage sharing of other array elements within the same array; this is because arrays are stored in a predetermined order. For example, if you write an EQUIVALENCE statement referring to ARR3 (illustrated in Figure 2 on page 21), as follows:

IBM Extension

```
DIMENSION ARR3(2,2,2)
CHARACTER *4 CHAR3(4)
EQUIVALENCE (ARR3(2,2,1),CHAR3(1))
```

then the array elements of ARR3 and CHAR3 share storage as shown in Figure 4, with the displacement for the array elements shown in the right-hand column.

ARR3 Storage	CHAR3 Storage	Displacement (in bytes)
ARR3(1,1,1)		0-3
ARR3(2,1,1)		4-7
ARR3(1,2,1)		8-11
ARR3(2,2,1)	CHAR3(1)	12-15
ARR3(1,1,2)	CHAR3(2)	16-19
ARR3(2,1,2)	CHAR3(3)	20-23
ARR3(1,2,2)	CHAR3(4)	24-27
ARR3(2,2,2)		28-31

Figure 4. Sharing Storage between Arrays—EQUIVALENCE Statement

End of IBM Extension

When you use the EQUIVALENCE statement with array elements, no checking is done to verify that the array elements specified are within the range of the array declaration.

Execution-Time Efficiency Using EQUIVALENCE

For efficiency in execution, it is important to ensure that arithmetic and logical items start on word boundaries appropriate to their length. Since an equivalence group is started on a doubleword boundary, it is possible to insure correct boundary alignment of these items by positioning them at a displacement from the beginning of the group that is a multiple of the item's length or, if the item is complex, half its length.

The following EQUIVALENCE statement (where A is REAL*4, I is INTEGER*4, and A2 is DOUBLE PRECISION) causes the items to be laid out in storage in a manner which promotes execution efficiency:

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

As shown in Figure 5, A and A2 begin at a displacement of 24 storage positions from the beginning of the equivalence group.

```
Displacement in bytes:

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

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

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

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

Figure 5. An EQUIVALENCE Storage Layout That Promotes Efficiency

The following EQUIVALENCE statement (using the same items as in Figure 5) causes the items to be laid out in storage so as to reduce execution efficiency.

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

As shown in Figure 6, A2 begins at a displacement of 20—not divisible by 8—from the beginning of the equivalence group.

```
Displacement in bytes:

0   4   8   12  16  22  24  28  32  36  40  44  48  52  56  60  64
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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

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

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

Figure 6. An EQUIVALENCE Storage Layout That Reduces Execution Efficiency

Using the Automatic Precision Increase Facility—AUTODBL Option

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 comprises two functions: promotion and padding. 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 doubling the storage size of nonpromoted items. 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, that is, REAL*4 to REAL*8 and COMPLEX*8 to COMPLEX*16.
- Double-precision items to be promoted to extended-precision items, that is, 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 will be 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) will be passed. If there is no function name of higher precision corresponding to the original intrinsic function name, the original intrinsic function name will be used and an informational message will be issued.
- If the AUTODBL option specifies padding only (for a given mode), then the intrinsic function name used as argument will not be changed.

Note that, when a substitution of an intrinsic function is made in order to honor a promotion option, the actual name substituted is an alias; that is, 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 will arise as a result of the substitution of the promoted name.

See Figure 7 on page 39 and Figure 8 on page 41 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 be converted 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.

Format of the AUTODBL Option

The AUTODBL compiler option indicates the form that the conversion will take. VS FORTRAN ensures that, when a single-precision argument is converted to double precision, storage boundaries are changed from word to doubleword alignment.

If AUTODBL is specified, and an error in coding the parameter is detected, the compiler ignores the AUTODBL option.

The AUTODBL option has the following format:

```
AUTODBL(value)
```

where value can be:

NONE
Indicates no conversion is to be performed. This is the default condition.

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 only promotion of single-precision items is to take place.

DBL8
Indicates that only promotion of 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 is to take place; that is, REAL*4 and COMPLEX*8 items are promoted. Items of other types are padded if they share storage space with promoted items.

DBLPAD8
Indicates that promotion of double-precision items is to take place; that is, REAL*8 and COMPLEX16 items are promoted. 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 with a 0.

Each position is coded with a numeric value that specifies how a particular conversion function is to be performed. The leftmost position describes the promotion function; that is, whether promotion is to occur and, if so, which items are to be promoted. The second position describes the padding function; that is, 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 describe whether padding is to occur for particular types (logical, integer, and real/complex, respectively) within the program areas specified in the second position. The values for each position are as follows:

Position 1, the 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, the padding function:

Value	Meaning
0	No padding
1	Pad all COMMON statement variables and all argument list variables.
2	Pad EQUIVALENCE statement variables equivalenced to promoted variables.

3	Pad all COMMON statement variables, pad EQUIVALENCE statement variables equivalenced 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 AUTODBL(nnnnn) settings that correspond to the mnemonic options are:

```
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).

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 or 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 Literal Constants

Care should be exercised when specifying literal 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 will be initialized correctly, but array A will not because padding takes place at the end of each element; therefore, no spill will occur if array A is padded or promoted. 'ABCDEFGH' will initialize 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 to be 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 hexadecimal 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 execute correctly for the AUTODBL option given. In this case, the format specification would be:

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

Effect on Programs Calling 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 execution. The recommended procedure is to compile all such program units with AUTODBL(DBLPAD). If an option other than DBLPAD is selected, care must be taken if the common area variables in one program unit differ from those in another; common area variables that are 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 on FORTRAN Subprograms

If a call to a FORTRAN library subprogram contains promoted arguments, the next higher precision subprograms are substituted for the original ones.

If you supply your own function in place of a FORTRAN-supplied function but neglect to specify this function name in an EXTERNAL statement (that is, neglect to identify this name as user-supplied), the wrong function may be executed.

For example: AUTODBL(DBL4)

```
    REAL*4  X,  Y
3   Y = SIN(X)
    STOP
    END
      .
      .
      .
    FUNCTION SIN(X)
      .
      .
      .
    RETURN
    END
```

In this example, because the compiler cannot recognize SIN as a user-supplied function, it substitutes the (alias) name of the FORTRAN-supplied function DSIN in statement 3. However, the compiler does not change the FUNCTION statement; that name remains SIN. At execution time, the user-supplied function SIN is ignored and the FORTRAN-supplied function DSIN is executed in its place.

You can avoid this confusion by making sure that you indicate that the name SIN is a user-supplied function by placing the name SIN in an EXTERNAL statement (or, for LANGLVL(66), in an EXTERNAL &SIN statement).

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). But, in the above example, 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. That is, in case such an array may be 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 will 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 execution time. (This can only be used 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)
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)

Figure 7 (Part 1 of 3). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(77)

Generic Name	Single Precision Function	Corresponding Double Precision Function	Corresponding Extended Precision Function
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)

Figure 7 (Part 2 of 3). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(77)

Generic Name	**Single Precision Function**	**Corresponding Double Precision Function**	**Corresponding Extended Precision Function**
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 7 (Part 3 of 3). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(77)

Notes for the above table follow the table below because some notes apply to both tables.

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)

Figure 8 (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
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)
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 8 (Part 2 of 2). Promotion of Single and Double Precision Intrinsic Functions for LANGLVL(66)

Notes to Figure 7 and Figure 8 :

1. The extended-precision equivalent of this function does not exist. In promoting REAL*8 to REAL*16, the double-precision function will be used. A warning message will be 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 will be used. A warning message will be 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).

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.

Chapter 3. Using Expressions and Assignment Statements

Defining and Using Expressions

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

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

Arithmetic operators—for arithmetic expressions

Character operators—for character expressions

Relational operators—for relational expressions

Logical operators—for logical expressions

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

For additional information about any of the topics discussed in this chapter, see *VS FORTRAN Language and Library Reference*.

Arithmetic Expressions

You can use arithmetic expressions to specify mathematical relationships of various kinds. The valid arithmetic expressions, and how you can combine them, are shown in Figure 9 on page 46.

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

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

- Operands are evaluated in their order of precedence from highest to lowest.
- Parentheses are evaluated as they are in mathematics; they specify the order in which operations are to be performed. That is, expressions within parentheses

are evaluated before the result is used. For example, the expression ((A-B)+C)*E is evaluated as follows:

1. A-B is evaluated, giving *x*.

2. *x*+C is evaluated, giving *y*.

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

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

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

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

OPERATION	ARITHMETIC OPERATOR	PRECEDENCE
Function Evaluation	(none)	First
Exponentiation	**	Second
Multiplication	*	Third
Division	/	
Addition	+	Fourth
Subtraction	-	

Figure 9. Arithmetic Operators and Operations

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

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

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

AR3 and AR1 (both REAL *4 numbers) are multiplied together, and the REAL *4 result is padded with zeros and placed in RESULT. Thus, while RESULT is the length of a double precision item, it has only the precision obtainable using REAL*4 operands (approximately 6 decimal digits).

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

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

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

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

You can also perform more complex mathematical operations by using the intrinsic functions that VS FORTRAN provides; see "Invoking Function Subprograms" on page 116 for details.

Character Expressions

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

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

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

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

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

```
ABCDEFGHIJKL
```

which is then assigned to the character variable CHAR.

Relational Expressions

You can form relational expressions by combining two arithmetic expressions with a relational operator, or by combining two character expressions with a relational operator. A relational expression can be used in the same places as a logical expression.

The relational operators you can use are listed in Figure 10.

Relational Operator	Meaning
.GT.	greater than
.LT.	less than

Figure 10 (Part 1 of 2). Relational Operators and Their Meanings

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

Figure 10 (Part 2 of 2). Relational Operators and Their Meanings

You can combine expressions as shown in the following examples:

```
A.GE.B
```

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

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

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

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

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

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

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

Relational Expressions—Character Operands

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

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

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

C1.GT.C2 evaluates as "true."

However, if you use the intrinsic functions (LLT, LGT, LLE, and LGE) to compare character operands, the comparison is made using the ISCII/ASCII collating sequence.

The expression

```
LGT(C1,C2)
```

evaluates as "false."

Logical Expressions

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

The logical operators you can use are shown in Figure 11.

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

Figure 11. Logical Operators and Their Meanings

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

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

   This logical expression is "true" if **one** of the following is "true":

   A is greater than B, or

   A is equal to C

   Otherwise, it is "false."

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

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

A is greater than B, and also

A is equal to C

otherwise, it is "false."

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

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

   A is greater than B, and also

   A is not equal to C

   otherwise, it is "false."

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

   This logical expression is evaluated in the following order:

   a. A.GT.B is evaluated, giving a truth value *v*.

   b. A.EQ.C is evaluated, giving a truth value *w*.

   c. B.LT.D is evaluated, giving a truth value *x*.

   d. *w*.AND.*x* is evaluated, giving a truth value *y*.

   The expression is "true" if either *v* or *y* evaluates as "true."

Assigning Values to Variables, Array Elements, and Character Substrings

In a VS FORTRAN program, the assignment statement lets you assign values to variables, array elements, and character substrings. It is distinguished from other FORTRAN statements by use of the equal sign.

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

You can use the assignment statement to assign a value to a variable, an array element, or a character substring.

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

Arithmetic Assignments

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

For example, assuming default naming conventions (see "Implied Default Data Type Declaration" on page 12), if you make the following assignments, you'll get the indicated results:

PI = 3.14159
Assigns the real constant 3.14159 to the 4-byte real variable named PI.

ARRAY3(NUM) = DIFF
Assigns the value currently contained in the 4-byte real variable DIFF to the 4-byte real array element ARRAY3(NUM)

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

DIFF = INTR
The value of INTR is converted to a 4-byte real value and placed in the variable DIFF.

DIFF = INTR+DIFF
The value of INTR is converted to a 4-byte real value and added to the current value of DIFF; the result is the new value of DIFF.

You can use and combine all the arithmetic operators, as shown in Figure 9 on page 46.

Character Assignments

Character assignments are only allowed when you are initializing character items. For example:

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

which assigns the value ABCDEFGHIJ to the character variable SUVAR.

The items in the assignment statement need not be the same length. When you execute the assignment, the item to the left is either padded at the right with blanks or the data is truncated to fit into the item at the left:

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

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

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

```
WHICH DOG BIT    (The word ''BITES'' is truncated to ''BIT.'')
```

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

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

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

There's one restriction upon character assignment statements: The items to the left of the equal sign, and those to the right must not overlap in any way; that is, they must not refer to the same character positions, completely or in part. If they do, you will get unpredictable results.

If you compiled your program with Release 4, library messages IFY193I through IFY197I will not be generated. This is because of the faster character handling now being done in-line, which is described in "Writing Efficient Character Manipulations" on page 146.

Logical Assignments

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

In a logical assignment, the items to the right may be either logical or relational expressions. Within the relational expressions, you can use arithmetic or character operands.

For example, using arithmetic operands:

```
LOGICAL *4 LOGOP
REAL *8 AR1,AR2,AR3,AR4
LOGOP = (AR4.GT.AR1).OR.(AR2.EQ.AR3)
```

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

For example, using character operands:

```
LOGICAL *4 LOGOP
CHARACTER *6 CHAR1, CHAR2, CHAR3
LOGOP = (CHAR2.EQ.CHAR3).AND.(CHAR1.LT.CHAR2)
```

"true" is placed in LOGOP when CHAR2 and CHAR3 are equal and CHAR1 is less than CHAR2; otherwise, LOGOP is evaluated as "false."

Saving Coding Effort with Statement Functions

If your program makes the same complex calculation a number of times, you can simplify your program by defining a statement function that refers to the calculation by name. For example,

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

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

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

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

is equivalent to:

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

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

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

Bit String Functions

The bit string intrinsic functions allow your program to interrogate and manipulate integer data of length 4 on a bit-by-bit basis. You can use the bit string functions to perform logical AND or OR operations, shift operations, and bit test or set operations, as described below.

Logical Intrinsic Functions

In the following functions, j and m are integer expressions of length 4. Operations are performed on all bits which represent the integer values bit-by-bit on corresponding bits to generate the integer result.

Function	Action
IAND(j,m)	Produces the value of the logical AND of the two arguments j and m.
IOR(j,m)	Produces the value of the logical OR of the two arguments j and m.
IEOR(j,m)	Produces the value of the logical exclusive OR of the two arguments j and m.
NOT(j)	Produces the value of the logical complement of the argument j.

Examples:

```
I = IAND(K+3,-1)
J = IOR(J,K)
L = IEOR(5,-1)
M = NOT(J)
```

Explanations:

The variable K and the constant 3 are added; a logical AND is performed of their sum and the constant -1; the result is stored in variable I.

A logical OR is performed of variables J and K; the result is stored in variable J (replacing its previous value).

A logical exclusive OR is performed of constants 5 and -1; the result is stored in variable L.

A logical NOT is performed of variable J; the result is stored is variable M.

Shift Intrinsic Function

In the following function, j and m are integer expressions of length 4. Operations are performed on all bits that represent the integer value to generate the integer result.

Function	**Action**
ISHFT(j,m)	Logically shifts j by the count and direction designated by m as follows: If m is less than 0, the shift is right. If m is greater than 0, the shift is left. If m is equal to 0, there is no shift. The value of m may be in the range -32 through 32. Bits shifted in are zero and bits shifted out are lost, and the final sign bit is not propagated.

Example:

```
K2 = -5
I = ISHFT(J+K,K2)
```

Explanation:

The variables J and K are added; a right shift of 5 bits is performed on their sum; the result is stored in variable I. (Five existing bits were shifted off at the low-order (right) end, and 5 zero bits were shifted in at the high-order (left) end.)

Bit Testing and Setting Intrinsic Functions

In the following functions, j and k are integer expressions of length 4. The value of k indicates the position of bit k relative to the low-order (rightmost) bit of j. Note that the low-order (rightmost) bit is bit 0 (not bit 1) and the count progresses to the left, having a maximum value of 31.

Function	Action
BTEST(j,k)	Tests bit k of integer argument j. If bit k is 1, the result is true.
IBSET(j,k)	Sets bit k of the integer argument j to 1.
IBCLR(j,k)	Sets bit k of the integer argument j to 0. For example: `0 <= k <= 31`

Example:

```
DIMENSION J(5),IAR(10)
J(KL)=IBSET(IBCLR(IAR(10),4),5)
```

Explanation:

The value of the 10th element of array IAR is obtained; the 4th bit of this value is set to 0; then the 5th bit of this value is set to 1; this value is then stored in the element of array J that is identified by the value of variable KL.

Chapter 4. Controlling Program Flow

The statements that enable you to control program flow in a VS FORTRAN program are the DO, GO TO, and IF statements.

For additional information about any of the topics discussed in this chapter, see *VS FORTRAN Language and Library Reference.*

Arithmetic IF Statement

You may want a program to execute some statements and skip around others, depending on the results of some previous evaluation. The arithmetic IF statement provides this function, as shown in the following example:

```
IF (J - 2) 20,21,22
```

which tells VS FORTRAN to:

1. Evaluate the expression (J - 2) (could be any arithmetic expression).
2. If the result is less than zero, transfer control to statement number 20.
3. If the result is equal to zero, transfer control to statement number 21.
4. If the result is greater than zero, transfer control to statement number 22.

Logical IF Statement

The logical IF statement provides similar control, depending on whether some condition is "true" or "false." For example:

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

tells VS FORTRAN to:

1. Evaluate the relational expression

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

   as to whether or not the value in variable TZ is equal to 5.0. If TZ is equal to 5.0, the expression is "true"; if TZ is not equal to 5.0, the expression is "false."

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

2. If the result of the evaluation is "true," transfer control to statement 25.

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

Block IF Statement

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

1. Write all code in control structures.

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

3. Design control structures so that they can be nested.

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

5. Indent the source code to reflect the level of nesting.

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

An IF-THEN-ELSE structure contains one or more blocks of code at the same level. Each of these blocks may contain other structures, and these nested structures have a higher level. However, when the IF-THEN-ELSE structure is executed, one and only one of its blocks may execute. For example:

```
10      IF (J.LT.2) THEN
11         A(J)=1
12      ELSE IF(J.GT.2) THEN
13         A(J)=3
14      ELSE
15         A(J)=2
16      ENDIF
17      ...
```

Here, depending on the value of J, exactly one of statements 11, 13, or 15 is executed, and control passes to statement 17.

An IF-THEN-ELSE structure begins with an IF-THEN statement and ends with an ENDIF statement. It optionally includes ELSE IF and ELSE statements.

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

There are three forms of blocks in an IF-THEN-ELSE structure:

IF block
ELSE-IF block
ELSE block

Each of these blocks begins with the statement following the IF, ELSE IF, or ELSE statement, and ends with the statement preceding the first ELSE IF, ELSE, or ENDIF at the same level of nesting.

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

- Statements 10-16 form an IF-THEN-ELSE structure.
- Statement 11 is an IF-block.
- Statement 13 is an ELSE-IF-block.
- Statement 15 is an ELSE-block.

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

You must not transfer control into an IF-THEN-ELSE structure from outside the structure; nor may you transfer control between blocks of a structure. However, you may transfer control within a block as you would in any valid program.

An IF-THEN-ELSE structure can contain any number of ELSE IF blocks. This is handy when you want to consider several exclusive cases. Even the ELSE block is optional.

This leads to the forms shown in Figure 12 on page 60.

```
Form 1:
  IF (expression) THEN

    (code executed if expression is "true")

  END IF

Form 2:
  IF (expression) THEN

    (code executed if expression is "true")

  ELSE

    (code executed if expression is "false")

  END IF

Form 3:
  IF (expression1) THEN

    (code executed if expression1 is "true")

  ELSE IF (expression2) THEN

    (code executed if expression1 is "false" and
      expression2 is "true")

  ELSE

    (code executed if both expression1
      and expression2 are "false")

  END IF

Form 4:

  IF (expression1) THEN

    (code executed if expression1 is "true")

  ELSE IF (expression2) THEN

    (code executed if expression2 is "true" and
      expression1 is "false")

    ELSE IF (expression3) THEN

      (code executed if expression3 is "true" and
      expression1 and expression2 are "false"

    ELSE IF (expression4) THEN

      (code executed if expression4 is "true" and
        expression1, expression2, and
        expression3 are "false")

  END IF
```

Figure 12. Block IF Statement—Valid Forms

Any one of these forms can be nested to any depth within any of the others, as shown in Figure 13 on page 61.

```
      IF (A.EQ.B) THEN

         (code executed if A.EQ.B)

      ELSE IF (A.GT.B) THEN

         (code executed if A.GT.B)

      ELSE

         (code executed if A.LT.B, including:

        IF (C.LT.D) THEN

           (code executed if C.LT.D)

        ELSE

           (code executed if C.GE.D, including:)

              DO
              .
              .
              .
90            CONTINUE
        END IF
      END IF
```

Figure 13. Nesting Block IF Statements

The following list shows the sequence of execution:

1. The first block IF statement has an IF block that includes the range of statements to the ELSE IF statement.

2. The ELSE-IF-THEN statement has as its block the range to the ELSE statement.

   If your program executes one of these blocks, control is transferred to the statement following the last END IF statement. Otherwise:

3. The first ELSE statement is the alternative condition for the first IF and ELSE-IF-THEN statement.

4. The second IF THEN statement is subordinate to the first ELSE statement. This IF block continues to the second ELSE statement.

5. The second ELSE statement is the alternative condition for the second IF THEN statement.

6. The DO loop is contained in the second ELSE block. (All nested DO loops must be completely contained within one block.)

7. The first END IF statement corresponds to the nested block IF statement.

8. The second END IF statement corresponds to the first IF statement (the one that begins the entire code sequence).

Executing Procedures Repetitively—DO Statement

Another powerful control structure—the DO loop—lets you repetitively execute a whole series of statements for a given number of times and then continue normal sequential processing.

The DO statement is particularly useful when you want to assign values to the elements in an array.

Processing One-Dimensional Arrays—DO Statement

For example, you may want to assign a specific set of values to the odd elements of ARRAYO. The DO statement in the following example sets up the loop to select the specified array elements:

```
      DOUBLE PRECISION ARRAYO
      DIMENSION ARRAYO(8), ARRAY1(8)
          .
          .
          .
      DO 40 INT=1,8,2

         ARRAYO(INT)=ARRAY1(INT)+ARRAY1(INT+1)

40    CONTINUE
          .
          .
          .
```

Notes:

1. *The program is to set INT to the value 1.*

2. *The program is to execute the code sequence while INT is less than or equal to 8.*

3. *Each time the code sequence is executed, the value of INT is to be increased by 2.*

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

Processing Multidimensional Arrays—Nested DO Statement

You can also include other DO loops completely within the range of a DO loop—for example, to initialize multidimensional arrays. The DO statements in the following example show how it could be done:

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

The first DO statement varies the second subscript from its minimum to its maximum value. The second DO statement varies the first subscript from its minimum to its maximum value. When you specify these two DO statements in this order, your program places ascending values in each array element in the sequence in which they are stored.

To initialize a four-dimensional table in storage sequence, you could specify

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

2. Contained in that DO loop would be a second DO loop controlling references to the third subscript.

3. Contained in that DO loop would be a third DO loop controlling references to the second subscript.

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

For more efficient execution, programs with large arrays should vary the first subscript most rapidly. That is,

```
DIMENSION ARRAY2 (1000,1000)
DO 10 J= 1, 1000
   DO 10 I= 1,1000
     ARRAY2 (I,J) = 0
10 CONTINUE
```

Using the CONTINUE Statement

In the previous example, there's a CONTINUE statement that provides a convenient ending point for procedures within the current ELSE block. You'll find the CONTINUE statement particularly useful in this way within DO loops.

Branching into Block IF Statements and DO Loops

When you use the CONTINUE statement within a block IF statement sequence or a DO loop, you must use the CONTINUE only for control transfers within the local code block. If you branch into the block (with GO TO statements, for example), the results are unpredictable, even though you won't get an error message.

Programming Loops—DO Statement

VS FORTRAN provides several ways to execute a range of statements a specific number of times using the DO statement.

- You can nest DO loops; if you nest one DO loop within another, you must include the range of the inner DO loop entirely within the range of the outer DO loop.

  You can use the same terminal statement for both the inner and the outer DO statement ranges.

- If you code a DO loop within a block IF, ELSE IF, or ELSE block, make sure that the range of the DO loop is completely contained within that block.

- If you code an IF-THEN-ELSE structure within a DO loop, ensure that the entire structure, including END IF, is within the range of the DO loop. (You can't use the END IF as the terminal statement for the DO loop.)

- You may not use any of the following as terminal statements:

  - A GO TO statement
  - An IF, IF-THEN, ELSE IF, ELSE, or END IF statement
  - Another DO statement
  - A RETURN, STOP, or END statement
  - An INCLUDE statement

- When you execute a DO statement, the DO loop becomes active. It remains active until one of the following occurs:

  - The loop executes completely.
  - The program executes a RETURN statement within its range.
  - A transfer is made out of its range.
  - Any STOP statement is executed anywhere in the loop.
  - Program execution is terminated because of an error condition.

Warning: A DO loop and an IF block must not overlap. If you branch into a DO block (with GO TO statements, for example), the results are unpredictable, and the condition is not diagnosed by VS FORTRAN.

In VS FORTRAN, you can specify the DO variable as an integer or real variable, and you can specify the initial value, the test value, and the increment as integer or real expressions, positive or negative.

For example, if you code the following DO statement:

```
    DO 20 VAR = START,END,INC
           .
           .
           .
20  CONTINUE
```

how the loop executes depends upon the values you place in START, END, and INC.

You can specify them all as

```
     START = 1.0
     END = 11.0
     INC = 2
```

The starting value (START) for VAR is 1.0, and the ending value (END) is 11.0. Each time the loop is executed, VAR is incremented by 2.0 (INC). (Note that, because VAR is a real item, the integer value in INC is converted to a real value.) After the loop has been executed six times, VAR contains the value 13.0, and DO statement processing is completed.

You can specify a decreasing INC value, with a START value higher than the END value:

```
     START = 11.0
     END = 1.0
     INC = -2
```

Again, the loop is executed six times, after which VAR contains the value -1.0.

You can specify values that cause the statements inside the DO loop not to be executed:

```
     START = 10.0          The START value is higher
     END = 1.0             than the END value and INC;
     INC = 2               the increment is positive.
                            Control passes to the statement
                            following the label 20, and VAR
                            contains the value 10.0.

or

     START = 1.0           The START value is lower
     END = 10.0            than the END value and INC;
     INC = -1              the increment is negative.
                            Control passes to the statement
                            following the label 20, and VAR
                            contains the value 1.0.
```

In either case the loop is not executed at all.

Note: Be careful when you're processing DO loops using real values for the starting or ending values, or for the increment; because floating point numbers are an approximation of actual values, there can be times when the loop is not executed exactly as you expect. In general, numbers that cannot be represented exactly in the computer may give unexpected results. (The numbers in the above examples have exact representations.)

The number of iterations of a loop can always be determined even though the loop variable is approximated. The iteration count is

```
     INT((END-START)/INC)+1
```

Using Program Switches—Assigned GO TO Statement

You can make one GO TO statement transfer control to different statements in the same program unit, depending upon a control variable. You set the control variable by means of an ASSIGN statement:

```
     ASSIGN 20 TO LVAR
     GO TO LVAR
30   (next executable statement)
     ...
20
```

When the GO TO statement is executed, control is transferred to statement label 20.

You can optionally include a list of statement labels in the assigned GO TO statement:

```
     ASSIGN 20 TO LVAR
     GO TO LVAR(10, 20, 50, 100)
30   (next executable statement)
```

When this GO TO statement is executed, control is transferred to statement label 20.

When your program executes either of these assigned GO TO statements, LVAR must have been assigned a valid label for an executable statement by an ASSIGN statement. Observe that LVAR may be assigned integer values in the same program unit, as long as the ASSIGN statement is executed before the assigned GO TO. For example, the following is *invalid* since LVAR is assigned an integer value after the ASSIGN statement:

```
     ASSIGN 20 TO LVAR
     LVAR=LVAR+10
     GO TO LVAR
```

No error message is generated if LVAR does not have a valid value. Therefore, mixing the two types of assignment for LVAR is not recommended.

Using Conditional Transfers—Computed GO TO Statement

You can transfer control conditionally to one of a number of statements, depending on the value contained in a control item:

```
     INT1 = 2
     GO TO (10,20,30,50,100) INT1
```

When this statement is executed, the integer value in INT1 (2) specifies that the second statement label is to be used for the transfer, and control is transferred to statement label 20.

You can use an integer expression as the control item:

```
     INT1 = 20
     INT2 = 16
     GO TO (10,20,30,50,100) INT1-INT2
```

When this statement is executed, the expression INT1-INT2 is evaluated, and the resulting value (4) specifies that the fourth statement label in the list is to be used for the transfer. Control is transferred to statement label 50.

If the value in the control item is either less than one or greater than the number of labels listed, then control passes to the *next* executable statement. Thus, in the previous examples, if the value in INT1 or of INT1-INT2 is greater than five, control is transferred to the next executable statement.

Suspending Execution Temporarily—PAUSE Statement

You can use the PAUSE statement to temporarily suspend program execution pending console operator response:

```
PAUSE
```

You can also include a message. The message can be a numeric string of 5 digits or less:

```
PAUSE 20200
```

where 20200 can have any meaning you want to assign it.

The message can also be a character constant:

```
PAUSE 'MOUNT TEMPORARY TAPE. TO RESUME PRESS ENTER'
```

The character constant you specify must contain no more than 72 characters.

When the program executes either of these PAUSE statements, a message is displayed at the console:

```
IFY001A PAUSE 20200
```

or

```
IFY001A PAUSE MOUNT TEMPORARY TAPE. TO RESUME PRESS ENTER
```

Program execution continues when the console operator presses the ENTER key.

Note: Under CMS or TSO, these messages are displayed at the terminal.

Stopping Programs Permanently—STOP Statement

You use the STOP statement to stop the program permanently:

```
STOP
```

You can also send a message to the console when the program stops through the STOP statement. The message can be a numeric string of 5 digits or less:

```
STOP 21212
```

where 21212 can have any meaning you want to assign it.

If you are running on MVS, the value of the STOP statement is returned to the job as the condition code for the job step being processed. For example,

```
STOP 8
```

sets a condition code of 8.

If you are using CMS in an EXEC, the value of the STOP statement is returned to your EXEC as the contents of variable &RETCODE, for which your EXEC may then test.

The message can also be a character constant:

```
STOP 'PROGRAM BACGAM EXECUTION COMPLETED'
```

(The character constant you specify must contain no more than 72 characters.)

When the program executes either of these STOP statements, the message is displayed at the console. If the message is in the form of a numeric string, the program return code will be set to this number. This can be used by the EXEC or PROC that caused the program to be executed.

Ending Your Program—END Statement

In VS FORTRAN, the last statement in your program must be an END statement, and (unless your program executes a RETURN or STOP statement first) it must be the last statement executed.

You can label the END statement. This lets you ensure that it is executed (if that's what you want), no matter which branch of the program is executed last:

```
110    END
```

Chapter 5. Programming Input and Output

This chapter describes VS FORTRAN I/O system, including access modes, file organization, record formats, and basic usage of I/O statements.

Access Mode and File Organization

Three types of access methods are available to FORTRAN users:

- Sequential
- Direct
- Keyed

These access modes may be employed to process five types of files:

- Non-VSAM physical sequential
- Non-VSAM direct
- VSAM entry sequenced data set (ESDS)
- VSAM key sequenced data set (KSDS)
- VSAM relative record data set (RRDS)

The various file processing possibilities are shown in Figure 14, which is a matrix of access modes and file organization. For example, Figure 14 shows that a VSAM RRDS file can be accessed by VS FORTRAN either sequentially or directly. You specify the access mode in an OPEN statement for the file with the ACCESS parameter, which causes the file to be connected to a FORTRAN I/O unit for the access mode named, when it is executed.

File Organization	Sequential Access	Direct Access	Keyed Access
Physical Sequential Non-VSAM	YES	YES[1]	NO
Direct Non-VSAM	NO[2]	YES	NO
VSAM ESDS	YES	NO	NO
VSAM KSDS	NO	NO	YES
VSAM RRDS	YES	YES	NO

Figure 14. Summary of File Organization and FORTRAN Access

1 Can be used for direct access only if the records are fixed length and unblocked.

2 Can be used for sequential access only if the file were created using a VS FORTRAN program.

The following sections first describe generally the basic characteristics of the various file organizations, as known to VS FORTRAN, and then enumerate the properties of a file when connected under VS FORTRAN for a given access mode. This brief presentation is VS FORTRAN-oriented, and does not address aspects not relevant to FORTRAN. You should refer to other publications if you want details not given here. More specific information on VSAM file processing for the VS FORTRAN user is available in Chapter 15, "Using VSAM with VS FORTRAN" on page 355.

Sequential Access

The types of files for sequential access are:

- Non-VSAM physical sequential files
- VSAM ESDS
- VSAM RRDS
- Non-VSAM direct files (only if created by VS FORTRAN)

Sequential File Organization

Sequential files are those in which the records are arranged and/or can be accessed serially, one after the other from first to last.

Tape files are always sequential files, as are files for terminals, printers, and card readers and punches (also called unit record devices). Your program must work sequentially with each record for such files as it is presented or as it is sent out.

Files on disk devices can be organized sequentially, and, if they are then under VS FORTRAN, they can be accessed sequentially (or directly if the records are fixed length and unblocked). The sequential nature of a disk file can be a result of entry-order sequencing, as for non-VSAM physical sequential files and VSAM ESDS files; or it can be due to relative ordering by keys, as for VSAM RRDS files and FORTRAN direct files.

Sequential File Properties

When a file is connected for sequential access, it has the following properties:

- The order of the records, as accessed, is the serial order in which they appear relative to the beginning of the file. Only records that were written after the file was created can be read.
- The records of the file must be either all formatted or all unformatted, except that the last record of the file may be an endfile record.
- Reading and writing of records is accomplished only by sequential access input/output statements.
- All records of the file have the same length.

Direct Access

The types of files for direct access are:

- Non-VSAM physical sequential (only if records are fixed length and unblocked on disk)
- Non-VSAM direct
- VSAM RRDS

Direct File Organization

You can store direct files only upon direct access devices. Direct access storage devices hold many records at a time. The arrangement of the records in the file is significant; it determines the ways your program can process the data. Because of this, your program's use of direct access storage can be more varied than its use of the sequential-only devices.

Direct files are those in which all the records are arranged in the file according to the relative addresses of their keys. Each record is the same size, and each occupies a predefined position in the file, depending upon its relative record number.

In a direct file, the first record has relative record number 1, the tenth record has relative record number 10, the fiftieth record has relative record number 50. You can think of the file as a series of slots, each of which may or may not actually contain a record. That is, record 50 may hold an actual record, and be identified as record number 50, even though records 24, 38, and 42 are vacant slots.

You can process the records by supplying the relative record number or key of the record you want with each READ or WRITE statement.

Direct File Properties

When a file is connected for direct access, it has the following properties:

- The order of the records is the order of their record numbers or keys. The records can be read or written in any order.
- The records of the file are either all formatted or all unformatted. If sequential access is also allowed for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If sequential access is not allowed for the file, the file must not contain an endfile record.
- Reading and writing records is accomplished only by direct access input/output statements.
- All records of the file have the same length.
- Each record of the file is uniquely identified by a positive integer called the **record number**. The record number of a record is specified when the record is written. After it is established, the record number of a record can never be changed. Note that a record may not be deleted; however, a record may be rewritten.
- Records need not be read or written in the order of their record numbers. Any record can be written into the file while it is connected to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record was written since the file was created.

IBM Extension

Keyed Access

The only possible type of file for keyed access is VSAM KSDS.

Keyed File Organization

Each record in a keyed file contains a field whose contents form its key. The position of this key field is the same in each record.

The index component of the file provides the logical arrangement of the main file ordered by the key. The actual physical arrangement of the records in the main file is not significant to your VS FORTRAN program.

A keyed file can also make use of alternate indexes—keys that let you access the file using a different logical arrangement of the records.

The existence of the index as an entity separate from the file itself, incorporating reference by primary key and by optional alternate keys, gives the VSAM KSDS file considerable flexibility of use not available in sequential and direct files.

Keyed File Properties

When a file is connected for keyed access, it has the following properties:

- Each record contains a **primary key** whose value uniquely identifies the record. In every record, the primary key consists of the same number of contiguous characters and is at the same position relative to the beginning of the record.

- Each record can contain one or more **alternate index keys** whose values identify the record. In every record, a given alternate index key consists of the same number of contiguous characters or processor-dependent units, and is at the same position relative to the beginning of the record. Unlike the primary key, the same value of an alternate index key may be duplicated in more than one record.

- Records may vary in length, but no record can be so short that it does not include the primary and all of the alternate index keys that are defined for it.

- The **key of reference** for any input/output statement is the single key (that is, either the primary key or one of the alternate index keys) that is used for the particular I/O statement. The key of reference may be changed during the time that the file is connected.

- The records of the file are either all formatted or all unformatted.

- The records must not be read or written using list-directed formatting.

- For formatted records, the order of the values of a key 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 values of a key is the order defined by the EBCDIC collating sequence for the data that forms the key on the external medium. For the purpose of comparing key values, that data, regardless of whether it was transferred 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 noncharacter data is used 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.

- The order of the records is the order of the values of their key of reference. If the key of reference changes, the order of the records changes accordingly. In the event that the key of reference is an alternate index key that has duplicate values, then the order of the records having the same key value is the order in which the records were entered into the alternate index.

- Reading and writing of records is accomplished only by keyed access input/output statements.

- During the execution of a single keyed access input/output statement, no more than one record may be read or written.

- The records in the file may be read or written either sequentially or directly, depending on whether a keyed sequential input/output statement or a keyed direct input/output statement is used. In sequential processing, records are read or written in increasing sequence of the value of the key of reference. In direct processing, any record may be read or written regardless of the previous value of the key of reference. While a file is connected, the type of processing can be changed depending upon the statement used to access the file.

- The **processing intent** specified when the file is connected for keyed access defines the types of input/output statements that can be used with the file. There are three possible processing intents: **WRITE, READ ONLY**, and **READ/WRITE**.

End of IBM Extension

Record Format and Length

The format and size of your records are determined by two things: the requirements of your application and whatever limitations the input/output devices have. If your program will be using preexisting files, the record formats will, of course, already have been decided, and you will fit your program to them.

Figure 15 shows the possible file organizations and record formats for non-VSAM files.

Note: The following discussion applies primarily to non-VSAM files. For similar information on VSAM files, refer to Chapter 15, "Using VSAM with VS FORTRAN" on page 355.

File Organization	**Fixed Length**	**Variable Length**	**Variable Spanned**	**Undefined**
Physical Sequential Non-VSAM	YES	YES	YES[1]	YES
Direct Non-VSAM	YES[2]	NO	NO	NO

Figure 15. Summary of Non-VSAM File Organization and Record Formats

Notes to Figure 15:

1 Unformatted only

2 Unblocked only

Fixed-Length Records

Fixed-length records are Format F. In an unblocked format-F file, the logical record is the same as the block. In a blocked format-F file, the number of logical records within a block (the blocking factor) is constant for every block in the file, except the last block, which may be shorter.

Figure 16 shows an unblocked record:

For MVS:

```
DCB=(RECFM=F,BLKSIZE=80)
```

For VM:

```
RECFM F BLOCK 80 specified as options in FILEDEF command
```

For VSE:

For sequential formatted I/O the RECFM is always set to F, so there is no need to specify anything. The default BLKSIZE is 260 bytes; you could use CALL OPSYS to change it.

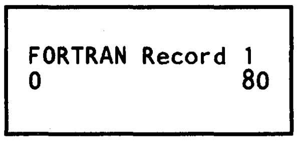


Figure 16. Fixed-Length Records—Unblocked

Figure 17 shows a blocked record (VS FORTRAN does not support fixed-length blocked records in VSE):

For MVS:

```
DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
```

For VM:

```
FILEDEF ... ( RECFM FB LRECL 80 BLOCK 400
```

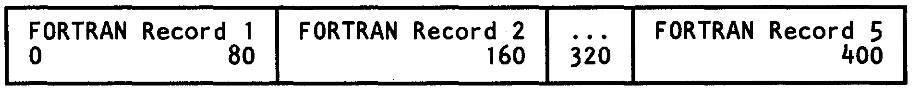


Figure 17. Fixed-Length Records—Blocked

Variable-Length Records (Format V or D)

Variable-length records can be format V or format D. Format-D records are variable-length records on ISCII/ASCII tape files. Format-D records are processed the same as format-V records.

Format-V records have control fields preceding your data. The control fields are shown in Figure 18 on page 76 and Figure 19 on page 76.

4 bytes	4 bytes	Variable bytes
LL BB	ll bb	Data
BDW	RDW	

Figure 18. Variable-Length Records—Unblocked

4 bytes	4 bytes	Variable bytes	4 bytes	Variable bytes
LL BB	ll bb	Data	ll bb	Data
BDW	RDW			

Figure 19. Variable-Length Records—Blocked

where

BDW—block descriptor word
The first four bytes of each block contain control information.

LL
Represents two bytes designating the length of the block (including the BDW field)

BB
Represents two bytes reserved for system use

RDW—record descriptor word
The first four bytes of each logical record contain control information.

ll
Represents two bytes designating the logical record length (including the RDW field)

bb
Represents two bytes reserved for system use

For **unblocked** format-V records, the block is composed of:

```
BDW + RDW + the data portion of one record
```

For **blocked** format-V records, the block is composed of:

```
BDW + the RDW of each record + the data portion of each record
```

The operating system provides the control bytes when the file is written. Although they don't appear in the description of the logical record you provide, FORTRAN allocates input and output buffers large enough to accommodate them. When variable-length records are written on unit record devices, control bytes are neither printed nor punched. They appear, however, on other external storage devices as well as in buffer areas of storage.

Variable-length record format can be specified for non-VSAM physical sequential files only. VS FORTRAN does not support Format V in VSE.

Variable Spanned Records

Spanned records are subclassification S of format V. A spanned record is a logical record that can be contained in one or more physical blocks. When you are creating files containing spanned records, if a logical record is larger than the remaining space in a block, a segment of the record is written to fill the block, and the rest of the record is written in the next block or blocks, depending on its length.

When you are retrieving files with spanned records, your program can only retrieve complete logical records.

Spanned records are preceded by control fields, as shown in Figure 20.

4 bytes	4 bytes	Variable bytes
LL BB	ll bb	Data Record or Segment
BDW	SDW	

Figure 20. Spanned Variable-Length Records

Each block is preceded by a block descriptor word (BDW). There is only one block descriptor word at the beginning of each physical block.

Each segment of a record in a block, even if the segment is the entire record, is preceded by a segment descriptor word (SDW). There is one segment descriptor word for each record segment within the block. The segment descriptor word also indicates whether the segment is the first, the last, or an intermediate segment. These words are not available for you to use in your program.

Spanned Blocked File: A spanned blocked file is a file composed of physical blocks that you define. The logical records can all be the same size or they can vary in length; their size may be smaller, equal to, or larger than the physical block size. There are no required relationships between logical records and physical block sizes. See Figure 21 on page 78.

Blocked Records. DCB=(RECFM=VBS,LRECL=130,BLKSIZE=200); assume three FORTRAN records, the first 130 characters in length, the second and third 100 characters in length:

```
<-----------block 1---------------> <------------block2---------------->

|B|S|                   |S|FORTRAN   |  |B|S|FORTRAN    |S|                 |
|D|D|FORTRAN Record 1|D|Record 2  |  |D|D|Record 2   |D|FORTRAN Record 3 |
|W|W|(130 chars)      |W|Segment 1 |  |W|W|Segment 2  |W|(100 chars)      |
| | |                 | |(58 chars)|  | | |(42 chars)| |                   |
0 4 8               138 142      200 0 4 8         50 54              154
```

Figure 21. EBCDIC Sequential Data Sets—Structure of Variable-Spanned Blocked Unformatted Records

Note: In VSE, VS FORTRAN pads the last block to the length specified by the block size for variable-spanned format.

Spanned Unblocked File: A spanned unblocked file is a file made up of physical blocks, each containing one logical record or one segment of a logical record. The logical records can all be the same size or they can vary in length. When the physical block contains one logical record, the length of the block is determined by the logical record size. The block size is controlled by the DCB BLKSIZE parameter for unformatted I/O. See Figure 22.

If record format hasn't been specified by means of JCL (the DCB parameter on a DD statement in MVS) or the RECFM option in a FILEDEF (in VM), format VS is the default for non–VSAM, physical, sequential, unformatted files. Under VSE, sequential unformatted records are always written in variable spanned format.

Unblocked Records. DCB=(RECFM=VS,BLKSIZE=68); assume two FORTRAN records, one 50 characters in length, the other 130 characters:

```
<--block 1-->  <---block 2--->  <---block 3--->  <---block 4---->

|B|S|        |  |B|S|FORTRAN   |  |B|S|FORTRAN   |  |B|S|FORTRAN   |
|D|D|FORTRAN |  |D|D|Record 2  |  |D|D|Record 2  |  |D|D|Record 2  |
|W|W|Record 1|  |W|W|Segment 1 |  |W|W|Segment 2 |  |W|W|Segment 3 |
| | |        |  | | |(60 chars)|  | | |(60 chars)|  | | |(10 chars)|
|0  8      58 0 4 8          68 0 4 8          68 0 4 8          18
```

Figure 22. EBCDIC Sequential Data Sets—Structure of Variable-Spanned Unblocked Unformatted Records

Note: In VSE, VS FORTRAN pads the last block to the length specified by the block size for variable-spanned format.

When you define files with spanned records, you can make the most efficient use of external storage and still organize your files with logical record lengths that best suit your needs:

- You can specify block lengths to make most efficient use of track capacities on direct access devices.

- You are not required to adjust the logical record lengths to device-dependent physical block lengths. One logical record can span across two or more physical blocks.

- You have greater flexibility when you want to transfer logical records between direct access storage types.

- You will, however, have additional overhead in processing spanned files.

You can specify only one type of variable spanned file: Non-VSAM physical sequential unformatted. You can only read or write variable spanned records with unformatted READ or WRITE statements.

Undefined Records

Undefined records are format U. Format-U records have undefined or unspecified characteristics. With format-U, you can process blocks that don't meet format-F or format-V specifications.

Each block on external storage is treated as a logical record; there are no record-length or block-length fields.

Format-U records are shown in Figure 23. Format-U records can be unblocked only:

```
DCB=(RECFM=U,BLKSIZE=80)
```

In this example, a record may be any size up to 80 bytes.

```
FILEDEF ... (RECFM U BLOCK 80
```

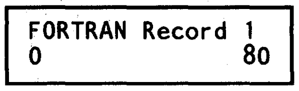


Figure 23. Undefined-Length Records

Only two types of file organization are possible with undefined records: Non-VSAM physical sequential unformatted and formatted.

If the record format hasn't been specified, format U is the default for non–VSAM, physical, sequential, formatted files.

Note: VS FORTRAN does not support format U in VSE.

Block Sizes

You establish the size of a physical block with the externally defined block size. If the record format hasn't been specified by means of JCL (the DCB parameter on a DD statement in MVS) or the RECFM option in a FILEDEF (in VM), the library assumes the records are not blocked. Blocking FORTRAN files on disk can enhance processing speed and decrease storage requirements.

Note: To specify the block size in VSE, use CALL OPSYS.

If the block size has already been specified, it must not be greater than the maximum block size for the device.

The block size specified for a format-F file must be an integral multiple of the record length.

If your program uses files on tape, use a physical block size of at least 12 to 18 bytes. Otherwise, the block will be treated as noise and skipped over when a parity check occurs while:

- Reading a block of records of less than 12 bytes
- Writing a block of records of less than 18 bytes

Block Size for ISCII/ASCII Files: If you specify the block size for an ISCII/ASCII sequential file that has a block prefix, be sure to include the length of the block prefix in the block size you specify.

Block Size and the DCB RECFM Subparameter: You can specify the S or T option in the DCB RECFM subparameter:

- You can use the S (standard) option in the DCB RECFM subparameter for a format-F record with only standard blocks. (A standard block has no truncated blocks or unfilled tracks within the file, except for the last block of the last track.) Use of the standard block option results in significant input/output performance improvements for direct access devices.
- You can use the T (track overflow) option only with MVS non-VSAM files.

Using VS FORTRAN Input/Output Statements

A list of the VS FORTRAN input/output source statements follows:

OPEN statement—connects a file to an I/O unit in a FORTRAN program.

WRITE statement—transmits a record to an external or internal unit (see "Specifying the Input/Output UNIT Parameter" on page 81 for meaning of "external or internal unit").

PRINT statement—formats and transmits a record to an external unit.

READ statement—retrieves a record from an external or internal unit.

ENDFILE statement—writes an end-of-file record on an external sequential file.

BACKSPACE statement—backspaces a sequential file by one record, or a keyed file to the first record with a specific key.

REWIND statement—positions a sequential file so that the next READ or WRITE statement processes the first record in the file, and positions a keyed file to the beginning of the first record with the lowest key in the file.

CLOSE statement—disconnects a file from a FORTRAN I/O unit.

INQUIRE statement—requests information about a file; can be used to return the record number to user for direct access I/O, or the key of the record just retrieved with keyed access I/O.

IBM Extension

REWRITE statement—for a keyed file, replaces a record just read.

DELETE statement—for a keyed file, erases a record just read.

WAIT statement—completes an asynchronous input/output transmission.

End of IBM Extension

For reference information about VS FORTRAN input/output statements, see *VS FORTRAN Language and Library Reference*.

Specifying the Input/Output UNIT Parameter

The UNIT parameter is required in most I/O statements. It has three forms:

- Character expression – for internal file identifier
- Integer expression – any valid unit number
- Asterisk – for the system default

The UNIT parameter is specified as follows:

```
[UNIT =] u
```

where *u* identifies an internal or an external file.

An internal file identifier is the name of a character variable, character array, character array element, or character substring.

An external unit identifier is one of the following:

- An integer expression whose value must be zero or positive (and that must represent a valid unit number).

- An asterisk, identifying a particular processor-determined external unit that is preconnected for formatted sequential access.

When you want to use one input/output statement to process more than one file, you can specify an expression. Between one execution of the statement and the next, you can change the value of the expression in the UNIT parameter and thus change the file to which the statement refers.

Monitoring Input/Output Errors—IOSTAT and ERR Parameters

Two parameters available with most I/O statements that allow for error checking are:

IOSTAT—I/O Status
which, after the input/output operation is completed, gives you the result:

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

ERR—Error Label
which you can use to specify special processing after an error occurs during execution of the I/O statement.

This lets you branch to a special procedure in the same programming unit that's executed when an error occurs. The procedure can obtain information about the last record processed. For example, the procedure 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 errors will set the IOSTAT field, nor will they all branch to the ERR label. See the figure entitled "IOSTAT and ERR Parameters Honored for I/O Errors" in *VS FORTRAN Language and Library Reference* for a complete list.

Extended error handling is also available. See "Extended Error Handling" on page 197 for a description of this aid.

Connecting to an External File—OPEN Statement

An OPEN statement connects a FORTRAN I/O unit number with an external file and may state other file characteristics. Direct files and VSAM files must be opened with an OPEN statement. For other types of files, the OPEN statement is optional.

The parameters in the OPEN statement are:

UNIT

which is an integer expression that specifies the I/O unit to be connected. For more information, see "Specifying the Input/Output UNIT Parameter" on page 81.

FILE

which specifies the name of the data definition statement (*ddname* in MVS and CMS; *filename* on the DLBL statement in VSE) to be represented internally by the unit number.

Note: On all systems, the name of the file must be specified if the default name is not desired. The name can contain up to seven alphameric characters, the first of which must be alphabetic.

If a file status of SCRATCH is specified, the name must not be specified. The FILE parameter cannot be specified if the default name is desired.

Example (VM):

```
    OPEN (UNIT=15, FILE='myfile1',...)
    .
    .
    .
FILEDEF MYFILE1 DISK SAMPLE DATA A
```

If you do not use a FILEDEF command, the filename is always FILE, and the filetype is taken from the OPEN statement—in this case, *myfile1*.

If, on the other hand, you do not use a FILEDEF command and FILE is not specified in the OPEN statement, the default filename, filetype, and filemode are:

```
FILE FTxxF001 A1
```

where:

xx is the unit number to which you are putting out the data.

For VSAM keyed files, the default is FTxxKnn, where:

xx is the unit number (00 through 99), and
nn is the key sequence number (01 through 09).

For VM, the filename specified in the OPEN statement is the ddname.

Example (MVS):

```
//MYFILE1 DD DSN=SAMPLE,...
        .
        .
        .
     OPEN (UNIT=15, FILE='myfile1',...)
```

The default name of the file is FTxxF001, where xx is the unit number to which you are putting out the data.

For VSAM keyed files, the default is FTxxKnn, where:

xx is the unit number (00 through 99), and
nn is the key sequence number (01 through 09).

For MVS, the filename specified in the OPEN statement is the ddname.

Example (VSE):

```
//DLBL myfile1,...
      .
      .
      .
  OPEN (UNIT=15, FILE='myfile1',...)
```

For VSE, the filename specified in the OPEN statement is the filename specified in the DLBL statement. Your installation may have set up default DLBLs specifying default filenames. If OPEN is specified without a filename, a filename must have been supplied on a DLBL statement; for example, with default filenames such as IJSYS00 through IJSYS04.

For VSAM sequential and direct files, the default is IJSYSxx; for VSAM keyed files, the default is FTxxKnn:

where

xx is the unit number (00 through 99), and
nn is the key sequence number (01 through 09)

STATUS
which lets you specify the file status of this file, as follows:

NEW	for a file that you're creating for the first time
OLD	for a file that already exists
SCRATCH	for a temporary file to be used during this job and then erased at the end of the job
UNKNOWN	for a file whose status is not currently known to the program; it may or may not currently exist. If it does not exist, it will be created.

ACCESS
which lets you specify the access method for this file: SEQUENTIAL or DIRECT.

IBM Extension

Or KEYED.

End of IBM Extension

Sequential files are always assumed; therefore, for this type of file, specifying the ACCESS parameter is not required. However, specifying it is useful for documentation.

For direct and keyed files, you must specify the access method.

BLANK
which lets you specify how blanks in numeric fields of a formatted input record are to be treated:

NULL blanks are ignored

ZERO any blanks that aren't leading blanks are treated as zeros

FORM
which lets you specify whether this file is connected for formatted or unformatted input/output. For sequential files, the default is FORMATTED; for direct or keyed files, the default is UNFORMATTED.

RECL
which lets you specify the logical record length of a direct file. Must not be used for sequential or keyed files.

IBM Extension

ACTION
which indicates the type of file processing to be done.

WRITE to open a VSAM file for loading of records

READ to open a non-empty VSAM file for retrieval operations

READWRITE to open a VSAM file to allow possible update as well as retrieval operations.

The following parameters can only be used if ACCESS=KEYED.

KEYS
which is a list of one or more different primary or alternate index keys which may be used to access the KSDS.

PASSWORD
which is the password that is required to access the VSAM KSDS.

End of IBM Extension

To see the formats of the above parameters, see *VS FORTRAN Language and Library Reference.*

Handling Multiple Files

In MVS and VM, you can refer to multiple files using the same unit number—but to do so, you must use the default name for the file:

```
FTxxFyyy
```

where

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

In MVS and VM, the file sequence number identifies the data set sequence number.

The file sequence number for a given unit number is incremented when an END= branch is taken for a READ statement, or when an ENDFILE statement is executed. Execution of a REWIND or CLOSE resets the sequence number to 001.

In MVS and VM, each file sequence number refers to a separate physical sequential file. Each of these files must be described by a separate DD statement (MVS) or a CMS FILEDEF command (VM). Refer to Chapter 12, "Using VS FORTRAN under MVS" on page 257, or Chapter 11, "Using VS FORTRAN under VM" on page 227, for a discussion of *ddname* requirements.

Formatting New, Direct, Non-VSAM Data Sets—OPEN Statement

Under MVS and VM, you can cause the primary allocation of a direct access data set to be formatted. (Under VSE, you should still use the CLEAR DISK system utility to preformat the data set because it provides device independence and performance in that environment.)

Under MVS, in order to format the data set, you specify DISP=NEW in the DD statement for the file. Under VM, the file must not exist. In addition, you specify the STATUS= parameter in the FORTRAN OPEN statement as NEW, UNKNOWN, or SCRATCH; and the ACCESS= parameter as DIRECT.

This causes the primary allocation of the data set to be filled with dummy records during the formatting operation. A dummy record is identified by the constant X'FF' in the first byte of the record. Although such dummy records are automatically inserted into the data set when it is created, they are not ignored when the data is read. They are replaced with valid data by a FORTRAN direct access WRITE statement.

Creating File Records—WRITE Statement

You can use the WRITE statement for two different purposes:

- To transfer data items from internal storage to a record in an external file
- For internal files, to transfer a number of data items (each of which may have a different data type) into one character item

The form of WRITE statement you specify depends upon the access you're using. See the descriptions, later in this chapter, of each type of access.

With the WRITE statement, you can specify any of the common processing options previously described.

Retrieving File Records—READ Statement

You can use the READ statement for two different purposes:

- To transfer a record from an external file to data items in internal storage
- For internal files, to transfer one character item into a number of data items, each of which may have a different data type

The form of READ statement you specify depends upon the access you're using. See the descriptions, later in this chapter, of each type of access.

For terminal files, a null entry in response to a READ is taken to be an end-of-file.

Obtaining File Information—INQUIRE Statement

You can use the INQUIRE statement to gather information about an external sequential, direct, or keyed file, or about a particular external unit. Your program can then take alternative actions, depending upon the information provided.

The INQUIRE statement is never required in a program, and you can execute it whether or not the file or unit is currently connected with your program.

You can ask for information about either a file or a unit number; and you can specify the I/O Status and Error Routine options, described under "Monitoring Input/Output Errors—IOSTAT and ERR Parameters" on page 82.

In addition, you can request the following information:

- Whether the file or unit exists
- Whether the file or unit is connected
- The unit number of the file or unit
- Whether the file has a name
- The access to this file—sequential, direct, or keyed
- Whether the file can be connected for sequential I/O
- Whether the file can be connected for direct I/O
- Whether the file is connected for formatted or for unformatted I/O
- Whether you can connect the file for formatted I/O

- Whether you can connect the file for unformatted I/O
- The record length, if this is a direct access file
- The number of the next record in the file, if this is a direct access file
- Whether input blanks are treated as zeros or as nulls

IBM Extension

- What type of access is desired: sequential, direct, or keyed
- The various properties of a keyed file as follows:
  - The value of the key of the last record read or written.
  - Whether you can connect the file for keyed access input/output
  - What type of action is desired: write, read, or readwrite
  - Whether the keyed file is to be opened for loading of records into a file, retrieval of records only, or update as well as retrieval.
  - Where file positioning is to take place for loading, retrieval, or update operations

End of IBM Extension

For more details about these properties, see *VS FORTRAN Language and Library Reference*.

Disconnecting a File—CLOSE Statement

You can use the CLOSE statement to disconnect an external file and a FORTRAN I/O unit.

The CLOSE statement is never required, but it lets you specify special processing when the connection is ended.

In addition to the common processing options previously described, you can also specify whether or not the file still exists (internally to the FORTRAN program) after the CLOSE statement is executed.

If you don't explicitly specify that the file is to be deleted, the file still exists after you've closed it. In this case, you can subsequently open the file for updating or retrieval.

Note: For MVS and VSE, the CLOSE statement does not override what you specify in the job control statements for the file.

Types of I/O

The following section discusses the various types of I/O:

- Unformatted I/O for sequential, direct, and keyed access
- Formatted I/O for sequential, direct, and keyed access
- Internal I/O for data conversions to and from the character type

IBM Extension

- NAMELIST I/O for sequentially accessed files
- List-directed I/O for sequentially accessed files
- Asynchronous I/O for high-speed sequential input/output

End of IBM Extension

Using Unformatted and Formatted I/O

For sequential, direct, and keyed files, you can specify one of two forms of READ and WRITE statements: unformatted or formatted.

Labels and record formats are of importance for every file your program uses. See Chapter 12, "Using VS FORTRAN under MVS" on page 257, and Chapter 14, "Using VS FORTRAN under VSE" on page 331, for specific system requirements for labels and record formats.

For further details of DCB considerations, see "Defining Records" on page 307 or the appropriate *Data Management Services Guide.*

Formatted and Unformatted Records

A **formatted record** consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters, and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero.

Formatted records may be read or written only by formatted input/output statements.

An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain both character and noncharacter data—or may contain no data. The length of an unformatted record is measured in processor-dependent units and depends on the output list used when it is written, as well as on the processor and the external medium.

Unformatted records may be read or written only by unformatted input/output statements.

Unformatted I/O

Unformatted I/O lets you use a list of FORTRAN data items to control the transfer of data; the length of the FORTRAN items in storage controls the amount of data transferred.

You can specify unformatted I/O using a READ, WRITE, or REWRITE statement. Each unformatted READ, WRITE, or REWRITE statement processes a record at a time, transferring the data items without conversion. (This means the transfer is quicker than when the program must convert each item as it is processed.)

Unformatted I/O allows only one logical record for each READ, WRITE, or REWRITE statement. The records of a file must be either all unformatted or formatted, not mixed. You can specify the NUM= parameter on any unformatted I/O statement to get the exact number of bytes read or written.

Following is an example of unformatted I/O:

```
CHARACTER*3 CH
CH = 'ABC'
I = 5
A = 2.0
WRITE (10) I , CH , A
```

where

CH	is a character variable of length 3, and has the internal hexadecimal representation of C1C2C3.
I	is an integer variable of length 4 and has the internal hexadecimal representation of 00000005.
A	is a real variable of length 4 and has the internal hexadecimal representation of 41200000.

After the WRITE statement has been executed, the record contains 11 bytes of data as shown here in hexadecimal representation:

```
| 00000005 C1C2C3 41200000 |
     I        CH      A
```

Formatted I/O

Formatted I/O lets you control input/output by specifying the format of the FORTRAN records and the form of data fields within the records.

This form of I/O also lets you convert items from internal representation to a character format that is readable on a listing. (The conversions cause data transfer to be considerably slower than with unformatted input/output.) The records of a file must be either all formatted or unformatted, not mixed.

In this form of input/output, you specify a FORMAT statement to be used in conjunction with the READ and WRITE statements. The FORMAT statement specifies the format of the FORTRAN records—the receiving field in a WRITE statement and the sending field in a READ statement.

Assume variables CH, I, and A have the same initial values, as defined in the preceding example:

```
   WRITE (10,15) I, CH, A
15 FORMAT ( I2,1X,A3,2X,F5.1)
```

After the above statements are executed, the record contains 13 characters of data as shown here in EBCDIC representation:

```
| 5 ABC   2.0|
```

Formatting FORTRAN Records—FORMAT Statement

When you're using formatted I/O, the FORMAT statement lets you specify the format of the FORTRAN records in READ, WRITE, or REWRITE statements. You can place FORMAT statements anywhere between the first and last statements in your program unit; you must specify a statement label.

You can use the FORMAT statement with both external and internal files.

When you use it for external files, be sure that the size of the FORMAT record doesn't exceed the size of the input/output medium; for example, if you're sending the record to a printer, it must not be longer than the printer line length. See "Preface" on page iii for a list of publications containing device information.

When you use formatted output, be sure that the representation of the number does not exceed the width of the output field. If the width is exceeded, asterisks are written instead of data. For example, F7.3 cannot be used for any number that is less than -99.999 or greater than 999.999.

Each field in the FORTRAN record is described with a FORMAT code specifying the data type for the field. The order in which you specify the codes is their order in the record. Some of the codes available with VS FORTRAN are shown in Figure 24 on page 92 (for a comprehensive list, see *VS FORTRAN Language and Library Reference*).

FORMAT Code	*Meaning*
	Data Field Codes:
Aw	Character data field (optional length specification)
aAw	Character data field (optional repeat count)
pP,aEw.dEe	Real data field (optional exponent (Ee))
aIw.m	Integer data field (with minimum number of digits to be displayed (.m))
	IBM Extension
aGw.dEe	Integer, real, or logical data field (optional exponent (Ee))
	End of IBM Extension
	Edit Codes:
:	End of format control, but only if I/O list is completely processed
/	End of record
BN	Nonleading blanks in a numeric field are ignored on input
BZ	Blanks treated as zeros on input
S	Specifies display of an optional plus sign

Figure 24 (Part 1 of 2). Some Codes Used with the FORMAT Statement

SP	Specifies plus sign must be produced on output
SS	Specifies plus sign is not to be produced on output
TLr	Data transfer starts r characters to left
TRr	Data transfer starts r characters to right

The lowercase letters have the following meanings:

a	An optional repeat count, less than 256
d	The number of decimal places to be carried
e	The number of digits in the exponent field
m	The minimum number of digits to be displayed
p	The number of digits for the scale factor
r	A character displacement in a record
w	The total number of characters in a field

Figure 24 (Part 2 of 2). Some Codes Used with the FORMAT Statement

Example:

If you want to define the format of an output record, you could specify your FORMAT statement as follows:

```
200   FORMAT (SP,2A10,I6.4,2E14.5E2)
```

which specifies that the output line is to be formatted as follows:

SP specifies that if the value of any of the numeric fields is positive, a plus sign is to be displayed. (If the value is negative, a minus sign is always displayed.)

2A10 specifies that the first and second items are character items of length 10.

I6.4 specifies that the third item is an integer item of total length 6, and that when the line is produced the display is:

```
(blank)(+ or -)(4 digits)
```

2E14.5E2 specifies that the fourth and fifth items are real items of total length 14; the display for each is shown in Figure 25 on page 94.

```
(2 blanks)(sign)(0)(decimal point)(5 digits)(E)(sign)(2 digits)
          |_______________________________||________________|
                     Numeric Field              Exponent

     (The sign is displayed as a + or a -.)
```

Figure 25. Display for FORMAT E14.5E2

Notes to Figure 25 :

1. The total width you can specify for this field is 14: 2 characters for leading blanks, 7 characters for the numeric field (including the leading sign and the decimal point), and 4 characters for the exponent (including the E and the sign).

2. The length of the record you've defined is 54 characters (bytes).

A formatted WRITE statement uses the statement label for this FORMAT statement and writes a record in this format.

Group FORMAT Specifications

VS FORTRAN lets you specify group specifications nested within the overall FORMAT specification, by specifying the group within parentheses. The group can contain a combination of format codes and groups, each separated by commas, slashes, or colons.

For example, you could specify an input record as follows:

```
100   FORMAT (BZ,A4,(A8,(I4,E8.4,(E4.0,E8.2))))
```

which specifies that the input receiving fields are structured as follows:

- BZ specifies that blanks in the input are treated as zeros.

- The first field in the record, A4, is a character field of length 4.

- It is followed by a group field, (A8,(I4,E8.4,(E4.0,E8.2))), consisting of the following:

  - An 8-character field, A8, followed by

  - A nested group field, (I4,E8.4,(E4.0,E8.2)).

- This nested group field contains yet another nested group field, (E4.0,E8.2).

Note: Up to 50 levels for nesting group fields can be specified in the VS FORTRAN FORMAT statement.

Using Specifications Repeatedly—FORMAT Control

Your FORMAT statements need not contain a format specification for each field in the READ or WRITE I/O list. If the end of the FORMAT specification list is reached before the last item in the I/O list is processed, control is returned to the rightmost left parenthesis in the format list; if there aren't any embedded parentheses, then control is returned to the first item in the format list:

```
10    FORMAT (A4,2(I2,I4),3(I4,I4),E8.2)
```

In this example, control returns to the repeat count in 3(I4,I4). A new record is processed each time control returns to the repeat count.

You can take advantage of this to reduce your coding effort, but be sure that the items repeated are the items you want repeated.

Using One FORMAT Statement with Variable Formats

You can specify variable FORMAT statements by placing a format specification into a character variable or an array during execution. You could read the specification in from external storage, or you could initialize the area using a DATA statement or an explicit assignment statement. You can then use the array as the format specification in READ or WRITE statements.

Using this feature, you can refer to a different array element each time you execute the READ or WRITE statement, and thus change the format.

Internal I/O

Internal I/O statements let you move data from one internal storage area to another while converting it from one format to another. This gives you a convenient and **standard** method of making such conversions.

If your external file is coded in EBCDIC character data, you can execute an unformatted READ statement to bring it into a character item in storage:

You can then execute a formatted internal READ statement to convert individual items in the record from EBCDIC to their internal formats.

For internal READ and WRITE statements, you specify the UNIT identifier as an internal data item—a character variable or a substring, or as a character array or an array element.

Using the READ Statement—Internal Files

A READ statement referring to an internal unit converts the data from character format to the internal format(s) of the receiving item(s):

```
     CHARACTER*13 BUF
     CHARACTER*3 CH
     BUF = ' 5 ABC    2.0'
     READ ( UNIT= BUF, FMT=40) I , CH, A
 40  FORMAT (I2, 1X, A3, 2X, F5.0)
```

Execution of the READ statement in this case is equivalent to executing the following assignment statements:

```
     I = 5
     CH = 'ABC'
and  A = 2.0
```

Using the WRITE Statement—Internal Files

A WRITE statement referring to an internal unit converts the data transferred from internal format to character format:

```
     CHARACTER *13 BUF
     CHARACTER * 3 CH
     I = 5
     A = 2.0
     CH = 'ABC'
     WRITE ( UNIT=BUF, FMT=40) I, CH, A
 40  FORMAT (I2, 1X, A3, 2X, F5.1)
```

Execution of the WRITE statement in this case is equivalent to executing the following assignment statement:

```
BUF = ' 5 ABC    2.0'
```

NAMELIST I/O

The NAMELIST statement specifies one or more lists of names of variables or arrays for use in READ and WRITE statements.

You can also use a READ statement to transfer data from an external I/O device into storage, or from one area of internal storage to one or more other areas of internal storage. In both cases, you must specify the NAMELIST statement to associate the name given to the data in the FORTRAN program with the data itself.

For more details and examples, see *VS FORTRAN Language and Library Reference.*

IBM Extension

Using List-Directed Input/Output

List-directed input/output statements—READ and WRITE—simplify your data entry for sequential files. They let you use formatted input/output—that is, input/output statements that perform data conversions as the data is transferred between internal and external storage—without the restrictions of a FORMAT statement. You can enter the data to be transferred without regard for column, line, or card boundaries.

You can specify list-directed I/O by specifying FMT=* on READ and WRITE statements. You can also specify a character type as the unit on I/O statements. This allows you to do list-directed I/O to an internal file.

For default format specifications for each data type, see *VS FORTRAN Language and Library Reference.*

This makes list-directed READ and WRITE statements particularly useful for terminal input and output, and for developing program test data.

For TSO considerations, see Chapter 13, "Using VS FORTRAN under TSO" on page 317.

Input Data—List-Directed I/O

You enter list-directed input data as a series of FORTRAN constants and separators:

- Each constant can be any valid FORTRAN constant. (Enter character constants within apostrophes.)

  Each constant you specify must agree in type with its corresponding item in the I/O list.

  The numeric part(s) of a noncharacter constant, including the optional sign, may not contain embedded blanks.

- You can specify a repetition factor for any constant or null item. For example:

  ```
  3*2.6
  ```

  specifies that the real constant 2.6 is to appear three times in the data input stream.

- Each separator can be:

  - one or more blanks
  - a comma
  - a line advance (for terminal input)
  - an end of card (on card devices)
  - a slash (/)

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, an error is detected.
- If there are more items in the record than in the data list, the excess items are ignored.

- A null item is represented by two successive commas.

List-Directed READ Statement

The list-directed READ statement retrieves a record from an input file, with conversions to internal forms of data:

```
READ (FMT=*,UNIT=5,IOSTAT=INT1) A,E,I,O,U
```

This READ statement specifies:

FMT=*
specifies that this is a list-directed READ statement.

UNIT=5
specifies that 5 is the unit from which the data is to be retrieved.

IOSTAT=INT1
defines INT1 as the FORTRAN integer variable into which information about the last operation is placed; by testing INT1 you can specify special programming actions for special conditions.

INT1 contains a positive value when an error occurs.

INT1 contains a negative value at end-of-file.

INT1 contains a zero value when neither condition occurs.

A,E,I,O,U
are the data items in which you want the data placed.

When the READ statement is executed, the data placed in A, E, O, and U is converted to real data of length 4; the data placed in I is converted to integer data of length 4.

Note: This is true only if there have been no explicit-type statements to the contrary.

List-Directed WRITE Statement

The list-directed WRITE statement writes a record on an output file, with conversions from internal forms of data:

```
WRITE (UNIT=6,IOSTAT=INT1,FMT=*) A,E,I,O,U
```

This WRITE statement specifies:

UNIT=6
specifies that 6 is the unit on which the data is to be written.

IOSTAT=INT1
defines INT1 as the FORTRAN integer variable into which information about the last operation is placed; by testing INT1 you can specify special programming actions for special conditions.

INT1 contains a positive value when an error occurs.

INT1 contains a negative value at end-of-file.

INT1 contains a zero value when neither condition occurs.

FMT=*
specifies that this is a list-directed WRITE statement

A,E,I,O,U
are the source data items for the data transfer.

When the WRITE statement is executed, the data is converted from the internal formats to external EBCDIC format.

Asynchronous I/O for MVS Only

Asynchronous input/output statements let you transfer unformatted data quickly between external sequential files and arrays in your FORTRAN program, and, while the data transfer is taking place, continue other processing within your FORTRAN program.

Always specify variable-spanned record format for asynchronous I/O.

For more information concerning asynchronous input/output, which is only available on MVS, see "Using Asynchronous Input/Output" on page 301.

End of IBM Extension

Sequential Access I/O Statements

The FORTRAN statements you can use with sequential file processing are the OPEN, INQUIRE, WRITE, READ, ENDFILE, BACKSPACE, REWIND, and CLOSE statements. The following sections tell how to use these statements with sequential access.

For system considerations, see "Sequential Files—System Considerations" on page 303 (MVS), and "Sequential Files—System Considerations" on page 349 (VSE).

Using the OPEN Statement—Sequential Access

It's never necessary to specify an OPEN statement with sequential files unless the file is a VSAM sequential file. However, the OPEN statement lets you take advantage of the special processing it makes available. You can, for example, specify the status of the file—NEW, OLD, SCRATCH, or UNKNOWN— as well as any special processing to be performed if the OPEN statement fails.

If your OPEN statement doesn't specify the formatting, formatted is assumed.

| If you don't specify an OPEN statement, the first READ or WRITE statement for the file establishes the file connection.

For descriptions of the options you can use, see "Specifying the Input/Output UNIT Parameter" on page 81, "Monitoring Input/Output Errors—IOSTAT and ERR Parameters" on page 82, and "Connecting to an External File—OPEN Statement" on page 82.

Using the WRITE Statement—Sequential Access

You can use either an unformatted or formatted WRITE statement with a sequential file; for example:

```
WRITE (UNIT=10,FMT=40,ERR=300,IOSTAT=INT) A,E,I,O,U
```

where, in this example:

10	is the unit number of the external file.
40	is the statement label of the FORMAT statement (used only with the formatted WRITE statement).
300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.
INT	is the name of an integer variable or an array element into which is placed a positive or a zero value indicating failure or success of the WRITE operation.

A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists to be included in the output record. Within main storage, these items need not be contiguous.

See "Specifying the Input/Output UNIT Parameter" and "Monitoring Input/Output Errors—IOSTAT and ERR Parameters" on page 82 for a description of the options you can use.

Using the READ Statement—Sequential Access

You can use either an unformatted or a formatted READ statement with a sequential file; an example of an unformatted READ is:

```
READ (UNIT=11,ERR=300,IOSTAT=INT,END=200) A,E,I,O,U
```

where, in this example:

11	is the unit number of the external file.
300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs
INT	is the name of an integer variable or array element into which is placed a positive or a zero value, indicating failure or success of the READ operation
200	is the statement label of the FORTRAN statement to which control is transferred when end-of-file is reached.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists into which the input record is transferred. Within main storage, these items need not be contiguous.

For a description of the options you can use, see "Specifying the Input/Output UNIT Parameter" and "Monitoring Input/Output Errors—IOSTAT and ERR Parameters" on page 82.

For an unformatted READ statement:

- If an external record contains more data than the items in the list, the excess external data is skipped.
- If an external record contains less data than the items in the list, an error occurs and processing continues. However, the NUM parameter in an unformatted I/O statement overrides this. NUM allows the data to be transferred without any indication of an error. The integer variable or array element specified by the NUM parameter is set to the number of bytes transferred.

Using the ENDFILE Statement—Sequential Access

You can use the ENDFILE statement to write an end-of-file record on an external file. You could also read a file as input and issue an ENDFILE. The file must be connected when you issue the statement.

You can use the ENDFILE statement when you need to write an end-of-file record for an output file. For example, the following ENDFILE statement:

```
ENDFILE (UNIT=10,IOSTAT=INT,ERR=300)
```

performs the following actions:

- Writes an end-of-file record on unit number 10.
- Returns a positive or zero value in INT to indicate failure or success.
- Transfers control to statement label 300 if an error occurs.

Using the REWIND Statement—Sequential Access

You use the REWIND statement to reposition a sequentially accessed file to its beginning point. The file must be connected when you execute the statement.

For example, the following REWIND statement:

```
REWIND (UNIT=11,IOSTAT=INT,ERR=300)
```

performs the following actions:

- Positions the file on unit number 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.

Using the BACKSPACE Statement—Sequential Access

You use the BACKSPACE statement to reposition a sequentially accessed file to the beginning of the record last processed. The file must be connected when you execute the statement.

Before your program issues a BACKSPACE statement, it must issue a READ, WRITE, or REWIND statement, or the BACKSPACE statement is ignored. A BACKSPACE statement for a SYSIN file (for VSE) is also ignored.

The following example shows how to use the BACKSPACE statement to reprocess a record that was just written.

```
WRITE ...          (writes the record to the file)
```

BACKSPACE ...	(positions the file at the beginning of the record just written)
READ ...	(retrieves the record for reprocessing)

You can use the BACKSPACE statement to replace a record in a magnetic tape file or a sequential direct access file:

READ ...	(retrieves the record to be replaced)
BACKSPACE ...	(positions the file at the beginning of the record just retrieved)
WRITE ...	(writes the new record)

After execution of this WRITE, no records exist in the file following this record. Any records that did exist are lost.

Using the CLOSE Statement—Sequential Access

You use the CLOSE statement to terminate the connection between the external file and the unit.

For sequential files, the CLOSE statement is optional; however, you can use it to specify specific processing actions when you disconnect from the external file.

For a description of options you can specify, see "Specifying the Input/Output UNIT Parameter" and "Monitoring Input/Output Errors—IOSTAT and ERR Parameters" on page 82.

Direct Access I/O statements

The FORTRAN statements you can use to process direct files are the OPEN, INQUIRE, WRITE, READ, and CLOSE statements.

For system considerations for direct files, see "Direct Files—System Considerations" on page 303 (MVS), and "Direct Files—System Considerations" on page 349 (VSE).

Using the OPEN Statement—Direct Access

You must specify an OPEN statement with a direct file; you must specify the following options:

ACCESS—to specify the file as direct

RECL—to specify record length

The record length you specify when you create the file is the record length you must specify when you retrieve records from the file.

If you don't specify the formatting, unformatted is assumed.

For descriptions of other options you can use, see "Specifying the Input/Output UNIT Parameter," "Monitoring Input/Output Errors—IOSTAT and ERR Parameters," and "Connecting to an External File—OPEN Statement" on page 82.

Using the WRITE Statement—Direct Access

You can use either an unformatted or formatted WRITE statement with a direct file; an example of an unformatted WRITE is:

```
WRITE (UNIT=15,REC=KEY,ERR=300,IOSTAT=INT)A,E,I,O,U
```

where, in this example:

15 is the unit number of the external file. It must identify a direct file.

KEY is an integer variable into which you place the relative record number for the record you're writing.

You can specify the variable as an integer item of length 4.

IBM Extension

You can also specify the variable as an integer item of length 2.

End of IBM Extension

300 is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.

INT is the name of an integer variable or array element into which is placed a positive or a zero value indicating failure or success of the WRITE operation.

A,E,I,O,U are the names of variables, arrays, array elements, character substrings, or implied DO lists to be included in the output record.

Within main storage, these items need not be contiguous.

Using the READ Statement—Direct Access

You can use either an unformatted or a formatted READ statement with a direct file; an example of a formatted READ is:

```
READ (UNIT=11,FMT=40,REC=KEY,ERR=300,IOSTAT=INT) A,E,I,O,U
```

where, in this example:

11 is the unit number of the external file. It must identify a direct file.

40	is the statement label of the FORMAT statement (used only with the formatted READ statement).
KEY	is an integer variable into which you place the relative record number of the record you want to retrieve.
	You can specify the variable as an integer item of length 4.

IBM Extension

You can also specify the variable as an integer item of length 2.

End of IBM Extension

300	is the statement label of the FORTRAN statement to which control is to be transferred if an error occurs.
INT	is the name of an integer variable or array element into which is placed a positive or a zero value, indicating failure or success of the READ operation.
A,E,I,O,U	are the names of variables, arrays, array elements, character substrings, or implied DO lists into which the input record is transferred.
	Within main storage, these items need not be contiguous.

For an unformatted READ statement:

- If an external record contains more data than the items in the list, the excess external data is skipped.
- If an external record contains less data than the items in the list, an error occurs and processing continues. However, the NUM parameter in an unformatted I/O statement overrides this. NUM allows the data to be transferred without any indication of an error. The integer variable or array element specified by the NUM parameter is set to the number of bytes transferred.

Using the CLOSE Statement—Direct Access

You use the CLOSE statement to terminate the connection between the external file and the unit. The CLOSE statement is never required for direct files; however, you can use it to specify special processing to occur when you disconnect from the external file.

For other options you can specify, see "Specifying the Input/Output UNIT Parameter" and "Monitoring Input/Output Errors—IOSTAT and ERR Parameters" on page 82.

Keyed Access I/O Statements

The FORTRAN statements you can use with keyed file processing are the OPEN, READ, WRITE, REWRITE, CLOSE, DELETE, INQUIRE, REWIND, and BACKSPACE statements. Keyed access is available only with VSAM. Both formatted and unformatted I/O can be performed.

Using the OPEN Statement—Keyed Access

When your program processes a keyed file, you must open that file with the OPEN statement. The OPEN statement option to use is

ACCESS='KEYED'

A keyed file can be opened with an ACTION of:

'READ'	which allows the retrieval (READ) and positioning (BACKSPACE and REWIND) operations to be performed on non-empty keyed files. In order to minimize the contention for the file between different users, ACTION should always be specified or allowed to default to READ when only retrieval and positioning operations are to be done.
'READWRITE'	which allows the update operations (REWRITE, DELETE, and WRITE) to be performed in addition to the retrieval and positioning operations. For this value of the ACTION parameter, records may be added to the file in any order.
'WRITE'	which allows new records to be loaded into an empty keyed file or at the end of a nonempty keyed file using the WRITE statement. The records must be written in increasing sequence of their primary key values.

When a previously loaded file is opened for retrieval and update (ACTION='READWRITE'), one or more of the primary or alternate index keys may be specified in the KEYS parameter of the OPEN statement. (Only those keys by which access to the file is desired need to be listed.) If more than one key is specified, then the READ statements which retrieve a record by key may indicate which one of these keys is to be used in the search. The specific key that is used for a given 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 it is explicitly changed.

The operations that may be performed for a file opened with ACTION='READWRITE' allow for updating the file by:

- Replacing an existing record, using a READ statement followed by a REWRITE statement.

- Deleting an existing record, using a READ statement followed by a DELETE statement.

- Adding a new record, using a WRITE statement.

An example of an OPEN statement, using all the parameters discussed, is:

```
OPEN (UNIT=8,ACCESS='KEYED',ACTION='READWRITE',KEYS=(16:19,1:3)
```

This OPEN connects an existing VSAM KSDS to FORTRAN I/O unit 8, in an access mode of keyed (the only mode allowed for a KSDS), for retrieval and update of records, using a primary key located in positions 16 through 19 of a record (key 1), and an alternate key located in positions 1 through 3 of a record (key 2).

Using the READ Statement—Keyed Access

The READ statement can be used for direct or sequential retrieval from a keyed file. (To use keyed retrieval, you must have opened the file with ACTION='READ' to perform retrievel operations, or ACTION='READWRITE' to perform both update and retrieval operations, and ACCESS='KEYED'.

In the event that the key of reference is an alternate index key rather than the primary key, then there could be more than one record in the file with the same key. When duplicate keys exist, a direct READ retrieves the first record, and sequential READ operations may be used to obtain the additional records. (The records with the same key are retrieved in the sequence that they were placed into the alternate index.)

Direct Retrieval

A *direct retrieval* statement is defined to be a READ statement with a KEY, KEYGE, or KEYGT parameter, which provides the value of the key that is to be used as the search argument for locating the desired record in the file. Only one of the three key argument parameters may be used in a single READ statement.

- KEY=key specifies that the record having the identical key be retrieved.

- KEYGE=key specifies that the record having a key either equal to the key or greater than and closest to the key be retrieved.

- KEYGT=key specifies that the record having a key greater than and closest to the key be retrieved.

The optional KEYID parameter of the direct retrieval statement indicates which key (first, second, and so on, from the KEYS parameter of the OPEN statement) is to be used as the key of reference for a direct retrieval and for all subsequent operations, until the key of reference is changed with another KEYID parameter.

Any of the key arguments (KEY, KEYGE, or KEYGT) used in a direct retrieval may specify a key which is shorter than the keys defined for the file. Such a search argument is called a *generic key*. This type of search argument causes a file search in which only the partial key supplied is compared with the leading portion of the keys in the file records. When KEY is used, for example, the record retrieved from the file is the first one in which the beginning of the key is identical to the partial key specified. The first record in key sequence that meets this criterion is returned to the caller; if there is no such record, the NOTFOUND condition occurs. When a generic key is provided with the KEYGE parameter, then the record retrieved is the first one in which the leading portion of its key is either equal to or greater than the partial key given. Similarly, the use of a generic key with the KEYGT parameter indicates that the desired record is the first one having the leading portion of its key greater than the partial key supplied. Note that a single READ statement retrieves only a single record even though there may be additional records in which the leading portions of the keys are identical.

Sequential Retrieval

A *sequential retrieval* statement is defined to be a READ statement without a KEY, KEYGE, or KEYGT parameter. The key of the record previously read or updated is used as the starting point and the next record in increasing key sequence is obtained. The key of reference from the previous direct retrieval statement remains the key of reference for a sequential retrieval. (In order to initiate sequential retrieval using a different key of reference, a direct retrieval statement must be issued first with a KEYID parameter to change the key of reference.) In the event that the file was just opened, 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.

A common scenario involving the use of both direct and sequential operations is to retrieve a single record directly (for example, with a KEY parameter specifying a generic key) and then continue reading with a series of sequential retrievals, until there are no more records in which the leading portion of the key matches the generic key specified in the first retrieval. The NOTFOUND parameter coded on the sequential READ statement causes control to pass to the specified statement when there are no additional records in which the leading portion of the key is identical to the generic key. If the NOTFOUND parameter is not used, the logic of the program must determine when to stop reading more records; otherwise, successive sequential retrieval operations would continue to the end of the file (that is, to the record with the highest key). Control would then pass to the statement indicated by the END parameter.

An example of this scenario is:

```
   I=1
   READ (UNIT=8,FMT=99,KEY='D5',KEYID=2,NOTFOUND=100) DEPTNO(I),
     NAME(I),EMPNO(I)
30 READ (UNIT=8,FMT=99,NOTFOUND=300) DEPTNO(I),NAME(I),EMPNO(I)
   I=I+1
   GOTO 30
```

The initial READ statement directly retrieves the first record having a generic key 2 of D5. Then the loop READ sequentially retrieves all following records having the same generic key. Fields in the records are placed in successive elements of the arrays DEPTNO, NAME, and EMPNO. The NOTFOUND exits allow for the possibility that no records meeting the key qualifications exist in the file.

Using the WRITE Statement—Keyed Access

The WRITE statement is used to add one record with a new primary key to the file.

Files are opened with ACTION=WRITE for initial loading of records into an empty file, or for adding additional records at the end of the file.

An example of the loading is:

```
10 READ (UNIT=5,FMT=20,END=50) DEPTN,NAM,EMPN
   WRITE (UNIT=8,FMT=30) DEPTN,NAM,EMPN
   GOTO 10
20 FORMAT (A3,2X,A12,2X,I4)
30 FORMAT (A3,A12,A4)
```

The keyed WRITE statement places records sequentially into the file connected to FORTRAN I/O unit 8, opened for action WRITE (not shown); after reading the records sequentially, from FORTRAN I/O unit 5 (physical sequential file assumed); until end-of-file on unit 5 is reached, when the END= branch is taken.

Notice that the field represented by the variable EMPN is read in using format I4 which converts it to fixed-point internal format, but is written out to the keyed file using format A4 in order to retain the fixed-point representation, because this is desired in the use of this field as a key.

When records are being written to or loaded into a file that was opened with an ACTION of WRITE, the records must be presented in increasing primary key sequence. (If they are not, an error condition exists.)

If the file was opened with an ACTION of READWRITE for updating, records may be added to the file in any order. For this write operation to complete successfully, the primary key of the record being written must be unique. And, if an alternate index key exists for the file (and is defined to have unique key values), the alternate key of the record being written must also be unique.

If the primary key of the record being written, or possibly the alternate key, is not unique, a *duplicate key* condition exists: the new record is not written, the record in the file is not modified, and control passes to the statement indicated by the DUPKEY parameter in the WRITE statement. If no DUPKEY parameter exists, control passes to the statement indicated by the ERR parameter, and the variable specified by the IOSTAT parameter, if any, is given the "duplicate key" value. If the ERR parameter isn't given either, execution terminates with an appropriate error message unless the option table entry for the message specifies a user exit which takes corrective action. Note that the "duplicate key" condition can arise even if the key value being duplicated is not for one of the keys being used in the program (as specified in the KEYS parameter of the OPEN statement).

An example of an update WRITE statement is:

```
WRITE (UNIT=8,FMT=98,DUPKEY=40) DPTN,NAM,EMPN,'NEW EMPLOYEE'
```

This statement adds a record to the file referenced by FORTRAN I/O unit 8, using FORMAT statement 98. The data placed in the record is taken from variables DPTN,NAM,EMPN, and, literally, the constant 'NEW EMPLOYEE'. For this update to complete successfully, DPTN and EMPN, which contain key values, must be unique; if they are not, the DUPKEY statement 40 is exited to and the write operation does not occur. Also, the file must have been opened for update with ACTION='READWRITE'.

Using the REWRITE Statement—Keyed Access

For keyed files, you can use the REWRITE statement to replace a record that was successfully retrieved by the immediately preceding sequential or direct READ operation. No other I/O operations, such as BACKSPACE or WRITE, may be issued for the same file between the READ and REWRITE statements. Any data in the record just read may be changed except for the value of the key of reference. (If the key of reference is an alternate index key, then neither the value of that alternate index key nor the value of the primary key may be changed.) The I/O list in the REWRITE statement contains each item, changed or unchanged, which is to appear in the record.

For instance, the following statements demonstrate updating several records by means of a READ, REWRITE sequence.

```
   READ (UNIT=8,FMT=99,KEY='F10',KEYID=2) DEPTN,NAM,EMPN
40 REWRITE (UNIT=8,FMT=98) DEPTN,NAM,EMPN, 'MOVING TO BLDG. 10'
   READ (8,99,NOTFOUND=120) DEPTN,NAM,EMPN
   GOTO 40
```

The first direct retrieval READ processes the initial record for the key 2 value F10, and this record is then rewritten with the added comment. The second sequential retrieval READ then processes the next record for the key, and loops back to replace it, continuing until no more records of that key 2 value are found, at which point the NOTFOUND exit is taken.

Using the DELETE Statement—Keyed Access

For keyed files, you can use the DELETE statement to 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.

Using the INQUIRE Statement—Keyed Access

The INQUIRE statement may be used to determine various properties of a keyed file. See "Obtaining File Information—INQUIRE Statement" on page 87 for a list of the properties related to keyed files.

Using the REWIND Statement—Keyed Access

The position in the keyed file for a subsequent sequential retrieval operation may be controlled by the REWIND statement. The REWIND statement positions the file to the record having the lowest value for the key of reference; a sequential READ statement then retrieves that record. Another way of putting this: The REWIND positions the file to the first record in the file for the file arrangement associated with the current key of reference.

Using the BACKSPACE Statement—Keyed Access

You can use one or more BACKSPACE statements to reestablish the position of a keyed file to a point prior to the current file position. You can then use a sequential retrieval statement to read the record to which the file was just positioned.

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 determines the next lower key value and positions the file to the first record with that 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, if a sequence of records in file 8 were:

Record	*Key 2 Value*
n	D47
n+1	D47
n+2	F10
n+3	F10
n+4	F10

Assume you have just read record n+4 with key 2 as the key of reference. Then, two consecutive backspaces would position the file as follows:

Record	*Key 2 Value*	
n+2	F10	(first backspace)
n	D47	(second backspace)

Because key 2 is nonunique, 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 issuing a sequential retrieval statement with the END=*stn* parameter after having already read the last record in the file. Control will pass to *stn* in this case.

- By issuing a direct retrieval statement with a KEYGE or KEYGT parameter which specifies a search argument so large that no record in the file satisfies the search criterion. Control passes to the statement indicated by the NOTFOUND=*stn* parameter in this case.

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.) A sequential retrieval may then be used 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). It is not permitted if the previous retrieval or update operation failed for any reason other than reaching the end of the file.

End of IBM Extension

Chapter 6. Subprograms and Shared Data

You may need to write programs that require a specific operation to be performed again and again, with different data for each repetition; for example, a complex mathematical operation.

You can simplify the writing of such programs if you write the statements that perform a repetitive operation as a separate subprogram. You can then simply refer to the subprogram throughout the program at the points where you need the operation done.

The program that requires the services of the subprogram is the calling program. The subprogram itself is the called program.

Two kinds of subprograms that VS FORTRAN supplies are:

- **Subroutine Subprograms**—which are invoked in the calling program through the CALL statement. For example:

  ```
  CALL CROOT (RNBR)
  ```

  This CALL statement makes the value in RNBR available to subprogram CROOT and transfers control to the first executable statement in subprogram CROOT. When CROOT has completed execution, control is transferred back to the calling program.

- **Function Subprograms**—which are invoked in the calling program through function references. For example:

  ```
  ANS = LNGTH * CROOT1(RNBR)
  ```

  When this statement is executed, the main program makes the value in RNBR available to function subprogram CROOT1 and transfers control to the first executable statement in it. As soon as CROOT1 finishes processing, control is returned to the main program together with a value that is multiplied by LNGTH to give ANS.

  Calling and called programs can share data by means of common data areas and passed data items.

The FORTRAN language statements that comprise the functions needed for calling and called programs and shared data are summarized below:

Statement Name	Statement Function
PROGRAM	Names a main program.
CALL	Invokes a subroutine subprogram and passes actual arguments.
INTRINSIC	Specifies FORTRAN instrinsic function names that are passed as arguments.
FUNCTION	Designates a subprogram as a function subprogram and names its dummy arguments.
SUBROUTINE	Designates a subprogram as a subroutine subprogram and names its dummy arguments.
ENTRY	Specifies an alternative entry point in a subprogram.
RETURN	Specifies alternative return points in a subprogram; may also specify variable return points in the calling program.
END	Terminates a subprogram and has the effect of a RETURN statement.
SAVE	Retains values that would otherwise become undefined after a RETURN or END is executed; is accepted but has no effect in VS FORTRAN because the values remain available.
COMMON	Specifies a shared data storage area and the names of the variables and arrays it contains.
EQUIVALENCE	Specifies program data names that share storage with common data names.
BLOCK DATA	Names a block of data that initializes items in a named common area.

Program Hierarchy

Calling and called programs make up a hierarchy of programs:

- The first program unit to invoke others is the *main program*; one and only one main program is required in every program hierarchy. The main program invokes subprograms; however, subprograms cannot invoke the main program.

- Subprograms can invoke other subprograms to any depth. However, a subprogram cannot invoke itself, and it cannot invoke any subprogram in the hierarchy that invoked it.

An example of a program hierarchy is shown in Figure 26. In the example,

```
A, the main program, invokes subprograms B and E.

B invokes C and F.

C invokes D.
```

Notes:

1. *Subprogram D could not invoke subprogram B or C; it could, however, invoke subprogram E or F.*

2. *Subprogram C could invoke F or E but not B.*

3. *Subprogram B could invoke E.*

4. *Main program A could invoke C, D, or F.*

5. *None of the invoked subprograms can invoke the main program A.*

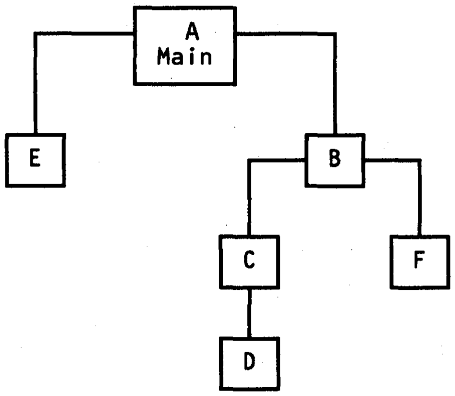


Figure 26. Calling and Called Program Hierarchy

If you do not designate a program as a subprogram, it is by default a main program. A main program is always named MAIN when compiled unless you use the PROGRAM statement to give it another name, probably a description name to document it.

Invoking Subprograms

Invoking Function Subprograms

When the name of a function, followed by a list of its arguments, appears in any FORTRAN expression, it refers to the function and causes the computations to be performed as indicated by the function definition. The resulting quantity (the function value) replaces the function reference in the expression and assumes the type of the function. The type of the name used for the reference must agree with the type of the name used in the definition.

For instance, you can invoke a function subprogram you've named BCALC, with the following statement in your invoking program:

```
R = BCALC(A+B) * 3.1
```

Your program calculates the sum of A and B and passes that value to the function subprogram BCALC; when BCALC completes executing, it returns the value to the invoking program, which then multiplies it by 3.1 to give the value of R.

Function Subprograms and the EXTERNAL Statement: You use the EXTERNAL statement with your function subprograms in two different ways:

- If you name the program with an IBM-supplied function name, you must list the name in an EXTERNAL statement.

  For example, if you write your own square root routine, and you name it SQRT, you must specify it in an EXTERNAL statement:

  ```
  EXTERNAL SQRT
  ```

  which tells the compiler that you want any SQRT references in your program to invoke your own SQRT routine rather than the IBM-supplied SQRT routine.

- If you want to pass a function subprogram as an argument, you must specify the name of the subprogram in an EXTERNAL statement.

When you specify an EXTERNAL statement, it must precede all the statement function definitions and executable statements in your program. The names you specify in an EXTERNAL statement can be names of external procedures, dummy procedures, or block data subprograms.

You can't use the same name in both an EXTERNAL statement and an INTRINSIC statement.

Invoking Subroutine Subprograms—CALL Statement

To execute a subroutine subprogram at a certain point in a program, issue a CALL statement at that point in the invoking program. The CALL statement can, optionally, pass actual arguments to replace the dummy arguments in the called subroutine subprogram:

```
CALL OUT                           (0 actual arguments passed)
CALL SUB1(X+Y*5,ABDF(IND),SINE)    (3 actual arguments passed)
```

When it's executed, the CALL statement transfers control to the subroutine subprogram, and associates the dummy variables in the subroutine subprogram with the actual arguments that appear in the CALL statement, as shown in Figure 27.

```
Calling Program             Subroutine Subprogram

DIMENSION X(90),Y(90)
   .                           SUBROUTINE COPY(A,B,N)
   .                           DIMENSION A(N),B(N)
   .                           DO 10 I = 1,N
CALL COPY (X,Y,90)          10 B(I) = A(I)
   .                           RETURN
   .                           END
   .
```

Figure 27. CALL Statement Execution

When the CALL COPY statement is executed:

The addresses of the actual arguments, array X and array Y, become the addresses of the dummy arguments, array A and array B, in the subprogram.

The variable N in the subprogram is associated with the value 90.

Thus a call to subprogram COPY, in this instance, results in the 90 elements of array X being copied into the 90 elements of array Y.

Invoking FORTRAN-Supplied Functions

There are a number of FORTRAN-supplied functions (intrinsic functions) you'll find useful, including mathematic functions to derive trigonometric values, logarithms, exponential values, maximum and minimum values, sign conversions, absolute values, error functions, and functions to manipulate character operands.

Invoke a FORTRAN-supplied function by referring to the function name within an arithmetic statement; the function name is replaced by the value returned from the invoked function, after the invoked function has completed its calculations.

For instance, you can invoke the FORTRAN-supplied function subprogram that returns the square root of a number, with statements in your program such as:

```
Y = SQRT(X+Z) * 3.1
```

If the sum of X and Z is 9.0, then the square root of X+Y is 3.0, and the value assigned to Y would be 3.0 times 3.1, or 9.3.

In VS FORTRAN, you can use the generic function name, and the compiler will select the function your program actually should use,

```
DOUBLE PRECISION  X, Z, Y
Y = SQRT(X+Z) * 3.1
```

In this case, you've specified SQRT, the generic function name; however, during compilation the compiler selects the DSQRT function, which gives a double precision result.

When you are using the current language level and specify an intrinsic function name in an explicit type statement, the intrinsic function is not removed from its status as an intrinsic function; this is true whether you specify the predefined function type or whether you respecify it as another type. When the intrinsic function is executed, the mode used is the mode predefined to the compiler.

However, when you are using the old language level and specify an intrinsic function name in an explicit but conflicting type statement, you remove it from its intrinsic status and it becomes the name of a user-supplied external function.

The names ARSIN, ARCOS, DARSIN, and DARCOS are intrinsic function names when using the option LANGLVL(66). The corresponding names for LANGLVL(77) are ASIN, ACOS, DASIN, and DACOS. The extended precision names for LANGLVL(66) and LANGLVL(77) are QARSIN and QARCOS.

If you use the names ARSIN, ARCOS, DARSIN, or DARCOS under LANGLVL(77), the corresponding functions are considered to be external; that is, it is assumed you are supplying these functions in your library or as one of your subprograms. However, if you fail to supply your own and are using the MVS or VM system, the corresponding FORTRAN functions supplied for the LANGLVL(66) names will be obtained from the library and used. In this case, no indication is given to the non-DOS user that the resolution was made to the FORTRAN library routine instead of to your own routine.

However, if you are using VSE, this resolution is not made and you must supply your own function.

Coding Subprograms

This section describes:

- How to code function subprograms
- How to code subroutine subprograms
- How to establish alternative entry points
- How to specify alternative and variable return points
- How to retain subprogram values

Coding Function Subprograms

The first statement in a function subprogram (excluding debugging statements) is a FUNCTION statement, identifying the program. For example:

```
FUNCTION TRIG (DELTA, THETA, ABSVAL)
```

This statement identifies the subprogram named TRIG as a function subprogram, with dummy arguments DELTA, THETA, and ABSVAL.

The data type of the function is real of length 4—derived from the predefined naming conventions. (The data type of the function determines the data type of the value it returns to the invoking program.)

You can also explicitly specify the data type of the function:

```
DOUBLE PRECISION FUNCTION TRIG (DELTA,THETA,ABSVAL)
```

which specifies that the data type of TRIG is real of length 8.

IBM Extension

If you want TRIG to be a real function of length 8, you can alternatively specify:

REAL FUNCTION TRIG*8(DELTA,THETA,ABSVAL)

End of IBM Extension

You can also specify a function subprogram as being of character type:

```
CHARACTER*10 FUNCTION TEXT1 (WORD1,WORD2)
```

which defines TEXT1 as a character function which returns a character value of length 10, using dummy arguments WORD1 and WORD2.

In a function subprogram, you can use any FORTRAN statements, except PROGRAM (which would define it as a main program), SUBROUTINE (which would define it as a subroutine subprogram), or BLOCK DATA (which would define it as a block data subprogram).

The last statement in a function subprogram must be an END statement.

You can also specify any number of RETURN statements.

Both the END and RETURN statements return control to the statement making the function reference in the calling program.

Coding Subroutine Subprograms

The first statement in a subroutine subprogram (excluding debugging statements) is a SUBROUTINE statement, identifying the program:

```
SUBROUTINE TRIG (DELTA, THETA, ABSVAL)
```

This statement identifies the subprogram named TRIG as a subroutine subprogram, with dummy arguments DELTA, THETA, and ABSVAL.

In a subroutine subprogram you can use any FORTRAN statements, except PROGRAM (which would define it as a main program), FUNCTION (which would define it as a function subprogram), or BLOCK DATA (which would define it as a block data subprogram).

The last statement in a subroutine subprogram must be an END statement. You can also specify any number of RETURN statements. Both of these statements return control to the statement following the CALL statement in the calling program, except when you specify an alternate return.

Specifying Alternative Entry Points—ENTRY Statement

When you're developing either function or subroutine subprograms, you can specify alternative entry points within the program, using the ENTRY statement.

For example, subprogram TRIG could have an alternative entry point, depending on the data type of the values you wanted returned.

Alternative Entry Points in Function Subprograms

If function TRIG had an alternative entry point, the sequence of statements would look something like this:

```
FUNCTION TRIG (DELTA,THETA,ABSVAL)
DOUBLE PRECISION BETA,ZETA,ABVAL1,TRIG8
      .
      .
      .
RETURN
ENTRY TRIG8 (BETA,ZETA,ABVAL1)
      .
      .
      .
END
```

This function subprogram can be executed in either of two ways:

1. When the calling program uses TRIG in a function reference, the function subprogram is entered at the first executable statement, and the value returned is a real value of length 4.

   The RETURN statement returns control to the calling program.

2. When the calling program uses TRIG8 in a function reference, the function subprogram is entered at the first executable statement following TRIG8, which is defined as a double precision function; therefore, the value returned is a real value of length 8.

   The END statement returns control to the calling program.

Alternative Entry Points in Subroutine Subprograms

If subroutine TRIG had an alternative entry point, the sequence of statements would look something like this:

```
SUBROUTINE TRIG (DELTA,THETA,ABSVAL)
DOUBLE PRECISION BETA,ZETA,ABVAL1
      .
      .
      .
RETURN
ENTRY TRIG8 (BETA,ZETA,ABVAL1)
      .
      .
      .
END
```

This subroutine subprogram can be executed in either of two ways:

1. When the calling program uses TRIG in a CALL statement, the subroutine subprogram is entered at the first executable statement, and the subprogram uses the arguments DELTA, THETA, and ABSVAL, which are real items of length 4.

   The RETURN statement returns control to the calling program.

2. When the calling program uses TRIG8 in a CALL statement, the subroutine subprogram is entered at the first executable statement following TRIG8, and the subprogram uses the arguments BETA, ZETA, and ABVAL1, which are real items of length 8.

   The END statement returns control to the calling program.

Specifying Alternative and Variable Return Points—RETURN Statement

When you're developing subroutine subprograms, you can specify alternative return points within the calling program, using the RETURN statement.

The RETURN statement, with no operands, can serve as one alternative return point, as the previous examples illustrate.

You can also code the RETURN statement with an integer variable operand; this allows you to specify variable return points. That is, you can return control to any labeled statement in the calling program.

For example, subroutine subprogram TRIG could have a variable return point, depending on the data values it develops:

```
Calling Program                Called Program
  .                            SUBROUTINE TRIG (X,Y,Z,*,*)
  .                              .
  .                              .
10 CALL TRIG (A,B,C,*30,*40)     .
20 Y = A + B                   100 IF (M) 200,300,400
  .                            200 RETURN
30 Y = A + C                   300 RETURN 1
  .                            400 RETURN 2
40 Y= B - C                    END
```

When statement 10 of the calling program is executed, control is transferred to the first executable statement in TRIG. In TRIG, when statement 100 is executed, the value of M in the arithmetic IF statement determines which RETURN statement is executed:

If M is less than zero, control is transferred to statement 200, and the RETURN statement returns control to statement 20 in the calling program. (This is the statement following the CALL statement.)

If M is equal to zero, control is transferred to statement 300, and the RETURN 1 statement returns control to the first statement number in the calling program's actual argument list (*30). (Thus, control is returned to statement 30 in the calling program.)

If M is greater than zero, control is transferred to statement 400, and the RETURN 2 statement returns control to the second statement number in the calling program's actual argument list (*40). (Thus, control is returned to statement 40 in the calling program.)

Execution then continues in the calling program.

Retaining Subprogram Values—SAVE Statement

The current FORTRAN standard states that, when a RETURN or an END statement in a subprogram is executed, all variables become undefined except for those in blank common, those in the argument list, and those specified in a SAVE statement. Thus, you would use the SAVE statement to retain such undefined values.

For program portability, you can use the SAVE statement to ensure, if the program is recompiled on some other FORTRAN compiler, that values in specific named common blocks, variables, or arrays are saved when a RETURN or an END statement is executed.

In VS FORTRAN, these values are still available after a RETURN or an END statement is executed; however, the compiler accepts the SAVE statement and treats it as documentation. If a name other than a common block name is enclosed in slashes, no error message is generated.

Sharing Data as Arguments or in Common Areas

Calling and called programs can share data between them, as we have seen in the previous examples.

In FORTRAN, there are two ways to share data: by passing arguments (data names identifying data items) between the programs, or by using common data areas (areas that can be shared by more than one program).

- Passing Arguments – 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.

Passing Arguments to Subprograms

Passing Arguments to a Function Subprogram

You can use actual and dummy arguments when you're invoking a function subprogram. If your calling program contains the following function reference:

```
G = B * ZCALC(INUM,X,Y)
```

the actual arguments in function ZCALC are INUM, X, and Y; they contain the actual values you want to make available to the function subprogram.

In the ZCALC function subprogram, you define the dummy arguments:

```
FUNCTION ZCALC(M,X,ZZ)
```

The dummy arguments of function subprogram ZCALC are M, X, and ZZ.

When the calling program executes the statement containing the function reference, the values in the actual arguments are made available to the dummy arguments:

The Value of:	Is Made Available in:
INUM	M
X	X
Y	ZZ

again, according to their positions in the argument lists.

M, X, and ZZ can then be used in operations within the function subprogram.

When control returns to the calling program, a value is returned to the calling program; then the assignment statement is executed, using the value returned.

Passing Arguments to a Subroutine Subprogram

You can use actual and dummy arguments to pass data between a calling program and a subroutine subprogram. For example, if the calling program contains the statement:

```
CALL MAXNUM(PI,FOURV,XYZ,BIGM,HH)
```

PI, FOURV, XYZ, BIGM, and HH are actual arguments; they contain values you want to make available to the subroutine subprogram.

The MAXNUM subprogram, in order to make the values available, must contain a matching list of dummy arguments:

```
SUBROUTINE MAXNUM(A,B,C,D,E)
```

The dummy arguments of subroutine subprogram MAXNUM are A, B, C, D, and E.

When the CALL statement is executed, the addresses of the actual arguments are used as the addresses of the matching dummy arguments:

The Address of:	Becomes the Address of:
PI	A
FOURV	B
XYZ	C
BIGM	D
HH	E

When MAXNUM is executed, the newly assigned values of A, B, C, D, and E can be used in operations.

When control returns to the calling program, the current values in A, B, C, D, and E are also the current values of PI, FOURV, XYZ, BIGM, and HH in the calling program.

General Rules for Arguments

You must define dummy arguments to correspond in number, order, and type with the actual arguments. For example, if you define an actual argument as an integer constant of length 4, you must define the corresponding dummy argument as an integer of length 4.

Actual arguments are passed by name; if you alter the value of an argument in the subroutine or function subprogram, you're altering the value in the calling program as well.

If you define an actual argument as an array, then the size of your paired dummy array must not exceed the size of the actual array.

If you define a dummy argument as an array, you must define the corresponding actual argument as an array or an array element.

If you define the actual argument as an array element, your paired dummy array must not be larger than the part of the actual array which follows and includes the actual array element you specify.

If your subprogram assigns a value to a dummy argument, you must ensure that its paired actual argument is a variable, an array element, or an array. Never specify a constant or expression as an actual argument, unless you are certain that the corresponding dummy argument is not assigned a value in the subprogram.

Your subprograms should not assign new values to dummy arguments that are associated with other dummy arguments in the subprogram, or with variables in the common area. You may get unexpected results, but the compiler cannot give you a warning message.

For example, if you define the subprogram DERIV as:

```
SUBROUTINE DERIV (X,Y,Z)
COMMON W
```

and if you include the following elements in the calling program:

```
COMMON B
   .
   .
   .
CALL DERIV (A, B, A)
```

the DERIV subprogram should not assign new values to X, Y, Z, and W:

X and Z because they are both associated with the same argument, A.

Y because it is associated with argument B, which is in the common area.

W because it is also associated with B.

Using Common Areas—COMMON Statement

Some kinds of usages for common data storage are:

- To implicitly transfer arguments among program units
- To contain data that form a logical whole, that is, referenced by multiple program units; for example, a logical record of a data file
- To conserve storage by establishing only one area used by several program units

A COMMON statement for a particular shared area must appear in every program unit that shows the area. The names, types, and lengths of the variables and arrays contained in a common area can be either the same or different in each sharing program unit.

If the programs merely use a common area independently as a work area, then each program can describe variables and arrays in the area to meet only its requirements, because no data sharing occurs in the area.

When data stored in the common area is shared, however, the referencing programs must take into account the types, lengths, and order of the data to assure appropriate usage. Often, in this case, the common area variables and arrays are specified identically in all programs that share the area, in order to prevent usage errors.

Passing Arguments in Common

In order to pass arguments using the COMMON statement, you should define the items that are to share common storage with the same type and length, and in the same order.

Arguments passed in a common area are subject to the same rules as arguments passed in a subroutine subprogram argument list (see "General Rules for Arguments" on page 124).

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 28. Each column of variables starts at the beginning of the common area. Variables on the same line share the same storage locations.

```
Main                                                                  Displacement
Program        Subprogram 1       Subprogram 2       Subprogram 3     (Bytes)

-----          -----              -----              -----            0
                                    Q      <--->       V
  A    <--->     D                -----              -----            4
                                    R      <--->       W
-----          -----              -----              -----            8
                                    S      <--->       X
  B    <--->     E                -----              -----            12
                                    T      <--->       Y
-----          -----              -----              -----            16
  C    <--->     F      <--->       U      <--->       Z
-----          -----              -----              -----            20
```

Figure 28. 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. The specific names of the common area can be unique to each program, or they can be the same.

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 124.

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 executing 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 two real 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 executing program, see "Internal Representation of VS FORTRAN Data" on page 398.

Efficient Arrangement of Variables—COMMON Statement

Your programs lose some object-time efficiency unless you ensure that all of the common variables have proper boundary alignment. (However, it isn't necessary for you to align complex, integer, logical, or real variables; your programs will still execute correctly.)

You can ensure proper alignment either by arranging the noncharacter type variables in a fixed descending order according to length, or by defining the block so that dummy variables force proper alignment.

Fixed Order of Variables—COMMON Statement

If you use the fixed order, noncharacter type variables must appear in the following order:

Length Type

IBM Extension

Length	Type
32	COMPLEX
16	COMPLEX or REAL
8	REAL

End of IBM Extension

Length	Type
8	COMPLEX or DOUBLE PRECISION
4	REAL, INTEGER, or LOGICAL

IBM Extension

2	INTEGER
1	LOGICAL

End of IBM Extension

Using Dummy Variables—COMMON Statement

If you don't use the fixed order, you can ensure proper alignment by constructing the block so that the displacement of each variable can be evenly divided by the reference length associated with the variable. (Displacement is the number of storage locations, or bytes, from the beginning of the block to the first storage location of the variable.) The reference length in bytes for each type of variable is as follows:

Type Specification	Length Specification	Reference Length (Bytes)
LOGICAL	4	4
INTEGER	4	4
REAL	4	4
DOUBLE PRECISION	8	8
COMPLEX	8	8

IBM Extension

LOGICAL	1	1
INTEGER	2	2
REAL	8	8
REAL	16	8
COMPLEX	16	8
COMPLEX	32	8

End of IBM Extension

The first variable in every common block is positioned as though its length specification were 8. Therefore, you can assign a variable of any length as the 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:

```
                 Displacement (Bytes)
Variable         in the Common Area


-----            0
  A
-----            4
  K
-----            8
CMPLX

-----            16
```

The displacements of K and CMPLX are evenly divisible by their reference numbers.

IBM Extension

However, if you define K as an integer of length 2, then CMPLX is no longer properly aligned (its displacement of 6 is not evenly divisible by its reference length of 8). In this case, you can ensure proper alignment by inserting a dummy variable (DV) of length 2 either between A and K or between K and CMPLX.

```
                 Displacement (Bytes)
Variable         in the Common Area


-----            0
  A
-----            2
  DV
-----            4
  K

-----            8
CMPLX


-----            16
```

End of IBM Extension

EQUIVALENCE Considerations—COMMON Statement

IBM Extension

VS FORTRAN allows you to equivalence character and noncharacter data.

End of IBM Extension

When you use the EQUIVALENCE statement together with the COMMON statement, there are additional complications resulting from storage allocations. The following examples illustrate programming considerations you must take into account.

Your program contains the following items:

REAL R4A, R4B, R4M(3,5), R4N(7)
DOUBLE PRECISION R8A, R8B, R8M(2)

IBM Extension

LOGICAL*1 L1A

End of IBM Extension

LOGICAL L4A

which are defined in 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)

IBM Extension

L1A	20	Word

End of IBM Extension

R8A	21	Byte (should be doubleword)
L4A	29	Byte (should be word)
R4M	33	Byte (should be word)

Now add an EQUIVALENCE statement to this inefficient COMMON statement:

1. First Example (valid but inefficient):

   ```
   EQUIVALENCE (R4M(1,1), R4B)
   EQUIVALENCE (R4B, R8B)
   ```

   This results in the following additional inefficiencies:

Name	Displacement	Boundary
R4B	33	Byte (same as R4M(1,1))
R8B	33	Byte (same as R4M(1,1) and R4B)

   which means that both R4B and R8B are now also inefficiently aligned.

2. Second Example (illegal):

   ```
   EQUIVALENCE (R8A, R4N(7))
   ```

This is illegal because the seventh element of R4N has the same displacement as R8A, or 21.

This means that the *first* element of R4N is located 24 bytes (4*6) before this, at displacement -3. It is illegal to extend a common area to the left in this way.

3. Third Example (valid but inefficient):

```
EQUIVALENCE (R8A, R4N(2))
EQUIVALENCE (R4M, R4N(5))
```

This has the following results:

Name	**Displacement**	**Boundary**
R4N(2).	21	Byte
R4N(3)	25	Byte
R4N(4)	29	Byte
R4N(5)	33	Byte
R4M	33	Byte (same position as R4N(5)

This is valid because the EQUIVALENCE statement places R4M at displacement 33, the same displacement as that specified in the COMMON statement. However, it is inefficient because both R4N and R4M begin at 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)

This is valid because the EQUIVALENCE statement

This is illegal, because the EQUIVALENCE statement (which places R4M at displacement 29) contradicts the COMMON statement (which places R4M at displacement 33). The COMMON statement controls the displacement of R4M, not the EQUIVALENCE statement.

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 no name is specified for the storage area.

- Named Common – You can name common storage areas (or blocks of storage)—known as *named common*. Blocks given the same name occupy the same space.

IBM Extension

- Dynamic Common – A *named common* with the DC compiler option specifying the *named common* area to be allocated at execution time. Note that this type of common cannot be initialized at compile time.

Note: VS FORTRAN allows you to place character and noncharacter variables and/or arrays in the same common area.

For more information, see "Using Dynamic Common above the 16-Megabyte Line" on page 314.

End of IBM Extension

There are different FORTRAN rules for blank and named common areas, which may cause you to choose one type over the other, depending on what you want your program to do.

The differences are:

- You can define only one blank common block in an executable program, although you can specify more than one COMMON statement defining items in blank common; you cannot assign blank common a name. You can define many named common blocks, each with its own name.

- You can define blank common as having different lengths in different program units. You must define a given named common block with the same length in every program unit that uses it.

- You can't assign initial values to variables and array elements in blank common.

- In named common, you can assign initial values to variables and array elements, through a block data subprogram that contains DATA statements or explicit specification statements.

In a COMMON statement, you specify a common block name by enclosing it in slashes. The following example defines a named common block, PAYROL, that contains the variables FICA, MANHRS, SICKDA:

```
COMMON/PAYROL/FICA,MANHRS,SICKDA
```

If the common name is omitted, a blank common is assumed. Or you can define blank common and named common in a single COMMON statement by omitting a name, and defining the blank common area first:

```
COMMON A,C,G/PAYROL/FICA,MANHRS,SICKDA
```

and the variables A, C, and G are placed in blank common.

You can also specify blank common items after named common items, by placing two consecutive slashes before the list of blank common variables:

For example, in the following statement:

```
COMMON A, C, G /PAYROL/FICA,MANHRS,SICKDA// JJ, VMN, LP7
```

you've defined the variables A, C, G, JJ, VMN, and LP7 in blank common, and in that order. You've defined PAYROL as named common, containing FICA, MANHRS, and SICKDA, in that order.

If you specify more than one COMMON statement in a program, the definitions are cumulative through the program. For example, if you specify the following two COMMON statements:

```
COMMON A, B, C /R/ D, E /S/ F
COMMON G, H /S/ I, J /R/P // W
```

they have the same effect as if you specified the single statement:

```
COMMON A,B,C,G,H,W /R/ D, E, P /S/ F, I, J
```

The name of a named common cannot be used in PROGRAM, SUBROUTINE, FUNCTION, ENTRY, or BLOCK DATA statements. It cannot be an intrinsic function name that is referenced in the same program unit. In particular, you cannot use a name that is an entry name in an intrinsic function library module that is referenced. For example, if you use the ICHAR intrinsic function, you cannot use 'CHAR' or 'LEN' as the name of your common block because both of these names are entry point names in the same library routine which also has ICHAR as an entry point. (For details, see "Module Names" in *VS FORTRAN Language and Library Reference.*)

You can, however, use the generic name of a function as the name of a common block if the intrinsic function use will reference a specific name different from the generic name. For example, you can call the name of your common block SIN, and use SIN as a non-REAL*4 intrinsic function.

A named common can be specified as a dynamic common by use of the dynamic common (DC) option, described under "Using the Compiler Options" on page 157. A dynamic common is allocated just before the program containing it is executed.

Dynamic common is useful in the MVS/XA environment for utilizing the expanded address capability. Also, the size of a load module is reduced when dynamic common is used, since 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 executed.

Initializing Named Common—Block Data Subprograms

Block data subprograms let you initialize data items in named common. (You can't initialize data items in blank common or in dynamic named common.)

The first statement you specify in a block data subprogram must be the BLOCK DATA statement. For example:

```
BLOCK DATA
```

or

```
BLOCK DATA COMDAT
```

where COMDAT is the name (optional) you've given the block data subprogram. If you specify a name, it must not be the same as the name of any other program, of an alternate entry point, of a common block, or of any data item within this block data subprogram.

The only statements you can specify in a block data subprogram are:

- BLOCK DATA (first statement in program)
- IMPLICIT (if used, must immediately follow BLOCK DATA statement)
- PARAMETER
- SAVE
- DIMENSION
- COMMON (must specify named common areas, each defined once)
- EQUIVALENCE
- Type statements
- DATA (must follow data item definitions in named COMMON statements, and must specify only data items in named common)
- END (must be last statement in subprogram)

The presence of a block data subprogram initializes named common data values in main programs or subprograms that refer to the named common blocks. Therefore, your programs must not contain CALL statements or function references to block data subprograms.

The following example shows how a block data subprogram might be coded:

```
BLOCK DATA
COMMON /ELJ/JC,A,B/DAL/Z,Y
REAL B(4)/1.0,0.9,2*1.3/,Z*8(3)/3*5.42311849D0/
INTEGER*2 JC(2)/74,77/
END
```

This program initializes items in two named common areas, ELJ and DAL:

- The REAL type statement assigns the type of and initializes array B in ELJ and array Z in DAL.
- The INTEGER type statement initializes array JC in ELJ.
- Because they're not included in either type statement or in a DATA statement, item A in ELJ and item Y in DAL are assigned default types and are not initialized.

Using the Execution-Time Library

The VS FORTRAN execution library include four categories of subroutines that you may access directly by coding subprogram call statements or function references. These are discussed briefly below, with references provided to the publications that contain more detailed information.

1. Mathematical and Character Functions
2. Alternative Mathematical Library Subroutines
3. Error Handling Subroutines
4. Service and Utility Subroutines

Mathematical and Character Functions

These routines provide standard intrinsic functions you can use in mathematical and character operations. The several categories of mathematical and character functions are:

- Logarithmic and exponential routines
- Trigonometric routines
- Hyperbolic function routines
- Miscellaneous mathematical routines
- Character manipulation routines

See *VS FORTRAN Language and Library Reference* for descriptive information on the mathematical and character functions.

Alternative Mathematical Library Subroutines

For very large absolute values, these routines provide slightly more accurate results than the corresponding VS FORTRAN library routines. They are:

- Exponential routines for single and double precision values
- Trigonometric routines for double precision, sine, cosin, tangent, and cotangent values
- Notational routines for single and double precision powers

These routines may be available to you as a separate library (VALTLIB), or they may have been put permanently into the standard library as replacements. Check with your system administrator to find out how these routines are installed. If they are in the standard library, you need not take any special action to access them. If they are in a separate library, you need to make that library known to the system ahead of the standard library when loading or link-editing your program.

For descriptive information about the alternative mathematical library subroutines, see *VS FORTRAN Language and Library Reference.*

Error Handling Subroutines

These routines enable you to provide user error exits for library errors and to modify the conditions associated with an error that control how the library handles the error. These changes are dynamic changes that are put into effect while your program executes.

For reference and usage information on the error handling routines, see *VS FORTRAN Language and Library Reference* and "Extended Error Handling" on page 197.

Service and Utility Subroutines

These routines provide four types of function:

- Control over certain mathematical exceptions
- Selective formatted or symbolic displays of program variables and arrays
- Immediate exit from execution
- In VSE only, multiphase job execution and block size and buffer changes.

For more information about the service and utility subroutines, see *VS FORTRAN Language and Library Reference.*

Chapter 7. Optimizing Your Program

When you use the OPTIMIZE(1/2/3) options, you can get faster program execution. However, when you use these options, you should be aware of programming practices that can help or hinder optimization.

Some of the suggestions are obvious, some are not. However, it's easy to forget even the obvious ones when you're developing or revising programs over a period of time.

The following paragraphs suggest ways you can make your programs execute faster and use the OPTIMIZE features to best advantage.

OPTIMIZE Compiler Option

The OPTIMIZE compiler option allows for the selection of no optimization or one of three optimization levels. Optimization is done at the cost of compile time but results in greatly reduced execution time. The VS FORTRAN optimizer is an efficient optimizer and can produce object code that is very close to that produced by an efficient assembler language programmer. The optimization techniques refined in the development of earlier IBM FORTRAN compilers have been used with VS FORTRAN.

The decision as to which optimization level should be used can usually be made based on the number of times that the compiled program will be executed. If a program is to be executed more than a few times, then the highest optimization level practical with this program should be selected. Generally, you should choose either OPTIMIZE(2) or OPTIMIZE(3) for most programs.

The optimization feature is particularly helpful within loops that contain subscripts. The address calculations implicit in an array subscript are not under the control of the FORTRAN programmer. Even a simple assignment statement with array subscripts can result in addressing expressions that have common expressions that can be eliminated by optimization or loop invariant expressions that can be moved out of the loop by optimization.

Four optimization levels are available:

OPTIMIZE(0): This level, with no optimization at all, provides the fastest compile time but with the least efficient execution time. No optimization is valuable when a program is being debugged or for compiles that are intended only to check for correct syntax.

OPTIMIZE(1): This is register optimization and improved branch optimization. Local register assignment is improved by retaining variables in registers where possible. Time is saved because the number of loads and stores is reduced. Branching is improved by the use of RX format branch instructions. This provides a moderate level of optimization for programs that do not have nested loops. Loop structure is not considered.

OPTIMIZE(2): This is full text and register assignment. It is identical to OPTIMIZE(3) but with safe code analysis. The basic rule in safe code optimization is that it is not permissible to cause an error to occur if it *might not* occur. An example of such an error would be a possible divide by zero within a loop. If the divide were to be moved out of the loop by the optimizer, it could possibly be moved before an IF that checks for the zero condition. As a practical matter, this condition occurs very infrequently and most programs can safely be compiled at OPTIMIZE(3).

OPTIMIZE(3): This is the highest level of optimization. Control flow and data flow analysis are done for the entire program. This analysis allows optimizations such as common expression elimination, strength reduction, code motion, and global register assignment to be done. Particular attention is paid to innermost loops.

Optimization Techniques

A number of different techniques are used by the optimizer. Some are special cases and will not be discussed here. The more general techniques used by OPTIMIZE(2) and OPTIMIZE(3) are:

Subscript collecting. Subscript collection merely arranges the sequence of calculations in a subscript expression into an order that results in more candidates for common expression elimination.

Common expression elimination. A common expression is one in which the same value is calculated as was done in a previous expression. The duplicate expression can be eliminated by using the value previously calculated. This is an important optimization feature because the use of subscripts for arrays frequently results in common expressions.

Constant propagation and constant folding. This is done by combining constants used in an expression and generating new ones. In addition, some mode conversions are done as well as evaluation of some intrinsic functions.

Strength reduction. This is done by replacing less efficient instructions such as a multiply, implicit in an array addressing expression, with an addition. The primary value is in a loop containing array subscript expressions.

Code motion. If both of the variables used in a computation within a loop are not altered within the loop, then it may be possible to move the calculation outside of the loop and simply use the result of the calculation within the loop. The other optimization techniques frequently result in code sequences that can be moved out of the loop.

Global register assignment. The variables and constants most frequently used within a loop can frequently be assigned to registers. The registers are initialized prior to the loop, and if necessary, stored on exit from the loop.

Section oriented branching. The number of program address registers required is reduced by dividing the executable code in a very large program into sections. There are a limited number of registers available and this technique frees more of them for other uses.

Programming Considerations When Using Optimization

In the majority of cases, it is sufficient to allow the compiler to handle optimization. FORTRAN programmers can concentrate on writing and debugging their programs.

It is generally true that most of program execution time is isolated in less than 10% of the code. Changes to the algorithms in the critical 10% frequently have dramatic results. The optimizer will generate efficient code but cannot fully compensate for an inefficient algorithm.

It is seldom useful to look for improvements such as elimination of redundant expressions or movement of expressions out of loops. The compiler is already very effective in doing this. More is to be gained by considering the attributes of variables which may limit optimization. Variables that are in common or used as arguments are difficult to optimize. For example, the optimizer must assume that any variable in a COMMON statement can be altered by a called subroutine. Frequently, better object code will result if global variables are temporarily assigned to variables that are used only locally.

An area that is frequently overlooked is I/O programming. Merely blocking a file can result in significant improvements. Use of unformatted I/O in place of formatted I/O is much faster. Avoid the use of implied DO loops in I/O statements and transfer the entire array (or equivalenced section). Passing variable dimension arrays to a subroutine that will write the entire array can be used to avoid an implied DO.

If an optimized program calls a subprogram, any variables that will be referenced in the subprogram must be in the common area or in the parameter list. If, for example, a program calls a subprogram to initialize a variable's address and then tries to reference that variable after a subsequent call, the value in that location may, or may not, be the value of the variable. See "Debugging Optimized Programs" below for more details.

Example 1:

The following example, using OPT(0), shows a subroutine called, has all addresses resolved, and upon subsequent invocation references the values expected.

```
@PROCESS OPT(0)
C234567
      A=1.0
      B=2.0
      C=3.0
      CALL SUB1(A,B,C)
      WRITE(6,*) 'SHOULD PRINT 1.0'
      CALL PRTX
      WRITE(6,*) 'SHOULD PRINT 2.0'
      CALL PRTY
      WRITE(6,*) 'SHOULD PRINT 3.0'
      CALL PRTZ
      STOP 'VALID VALUES'
      END
@PROCESS
      SUBROUTINE SUB1(X,Y,Z)
      RETURN
      ENTRY PRTX
      WRITE(6,*) X
      RETURN
      ENTRY PRTY
      WRITE(6,*) Y
      RETURN
      ENTRY PRTZ
      WRITE(6,*) Z
      RETURN
      END
```

Output for Example 1:

```
SHOULD PRINT 1.0
   1.00000000
SHOULD PRINT 2.0
   2.00000000
SHOULD PRINT 3.0
   3.00000000
IFY002I STOP  VALID VALUES
```

Example 2:

The following example demonstrates the pitfall of assuming that correct values will be available when OPT(2) is specified. Variable A is assigned the value of 4.0, but when OPT(2) is specified, the value of 4.0 is placed in a register and not in storage. When the call to PRTX occurs, A is printed and a value of 4.0 is expected but the value printed is 1.0.

```
@PROCESS OPT(2)
C234567
      A=1.0
      B=2.0
      C=3.0
      CALL SUB1(A,B,C)
C
C     BECAUSE OF OPT(2), THE FOLLOWING ASSIGNMENT
C     IS NOT REFLECTED IN STORAGE
C
      A=4.0
      WRITE(6,*)
     1'SHOULD PRINT 4.0 BUT WILL PRINT STORAGE VALUE OF 1.0'
      CALL PRTX
      WRITE(6,*) 'SHOULD PRINT 2.0'
      CALL PRTY
      WRITE(6,*) 'SHOULD PRINT 3.0'
      CALL PRTZ
      STOP 'INVALID VALUES RETURNED'
      END
@PROCESS
      SUBROUTINE SUB1(X,Y,Z)
      RETURN
      ENTRY PRTX
      WRITE(6,*) X
      RETURN
      ENTRY PRTY
      WRITE(6,*) Y
      RETURN
      ENTRY PRTZ
      WRITE(6,*) Z
      RETURN
      END
```

Output for Example 2:

```
EXECUTION BEGINS...
SHOULD PRINT 4.0 BUT WILL PRINT STORAGE VALUE OF 1.0
   1.00000000
SHOULD PRINT 2.0
   2.00000000
SHOULD PRINT 3.0
   3.00000000
IFY002I STOP  INVALID VALUES RETURNED
```

Debugging Optimized Programs

Debugging a program that has been optimized can be more difficult than debugging one that has not been optimized. Variables that have been temporarily assigned to registers may not have been saved in a storage location at the time that an abend dump occurs. A common expression evaluation may have been deleted or moved. This is not to say that the program has not been compiled correctly, but the very fact that optimization has made changes can add confusion. Understanding a bit about the optimizer can help to reduce the confusion.

Debugging techniques that rely on examining values in storage should be used with caution if the program has been optimized. The variable may be in a register, not yet stored, when storage is examined or the abend dump occurs.

A FORTRAN program that appears to work properly when compiled with OPT(0) may fail when compiled at OPT(3). This is generally caused by program variables that have not been initialized prior to being used. The OPT(0) option stores all variables. An uninitialized variable in storage generally contains zero. When assigned to a register by the optimizer, it usually contains a large number. If a program that worked at OPT(0) fails when compiled at OPT(1) or OPT(2), it is a good idea to look at the cross-reference listing for variables that are fetched but never set, and for program logic that can allow a variable to be used prior to being set.

If VS FORTRAN Interactive Debug (5668-903) is installed on your system, see *VS FORTRAN Interactive Debug Guide and Reference* for more information on debugging optimized code.

Selecting the Higher Optimization Levels

In general, you should select the highest level of optimization. The higher the optimization level you select, the longer the compilation time and the more storage required; however, both object program storage and execution time are reduced.

Very few iterations through most subroutines can cause optimization savings to exceed optimized compilation cost.

You should use NOOPTIMIZE or OPTIMIZE(1) only for testing purposes. For example, they're useful if you want to check the syntax of the program without executing it, or to debug a subroutine that doesn't iterate correctly.

Whenever you can, you should choose either OPTIMIZE(2) or OPTIMIZE(3) for most programs. However, in order to compile some very large programs, you may have to use OPTIMIZE(1) because very large program may fail to compile at other optimization levels.

Writing Programs of Efficient Size

For efficient optimization, programs can be either too large or too small.

Keep programs smaller than 8192 bytes in size. Programs larger than this cause the compiler to use a register as a program address register that would otherwise be used for optimization.

You should exercise caution when designing a program in a top-down (modular) fashion. It is possible that the implicit cost of the subroutine or function call overhead may exceed the value of coding in this manner. However, you shouldn't be too hasty in sacrificing clarity for speed. After identifying the "most-called" subroutines and functions, you should consider moving the code into the main program. This allows the compiler to optimize the combined code and thereby execute faster.

Using Unformatted I/O

Unformatted I/O takes less processing time and uses less storage than formatted I/O. Unformatted I/O also exactly maintains the precision of the data items being processed.

With formatted I/O, each data element is converted between internal and external format. This takes time and storage. In addition, rounding errors can accumulate during conversion.

Implied-DO I/O Statements

When processing I/O statements, the compiler is able to recognize certain combinations of implied-DOs and combine them in what is known as a partial short-list. The effect of this may be seen in the generated code, where instead of the usual call to the I/O library surrounded by a conventional DO-loop, there is just the call to the I/O entry point. This call represents those "nests" in the implied-DO that qualify as partial short-lists. These calls may, in turn, be surrounded by DO-loops representing those levels not qualifying as partial short-lists. 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 will generate 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. Examples of this are:

```
DIMENSION A(100)
WRITE(3) (A(I), I=1, 100)
```

This has the effect of writing 100 elements of array A, starting with element A(1).

The following example:

```
READ(5,10) (A(J), J=N, M)
```

would have the effect of reading (M-N+1) elements into array A, starting with element A(N).

When coding a block of I/O statements, you should try to place as many list items on one READ or WRITE statement as is practical. The compiler will "bundle" together up to 20 such items, and make just one call to the I/O library, instead of making one call per item as was previously done.

Writing Efficient Character Manipulations

An efficient code sequence is generated for character move and comparison under the following conditions:

- The character length for both operands is constant, is less than or equal to 256, and is greater than 0.

- For character move, the character length of target operand is less than or equal to that of the source operand. For character comparison, the character length for both operands is the same.

In the following example, an efficient code sequence (including MVC or CLC) will be generated for the first three statements (1 through 3), while a less efficient code sequence (including MVCL or CLCL) will be 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'
```

Using Logical Variables of Length 4

Logical variables of length 4 can be accessed directly without clearing a register.

IBM Extension

Every reference to a LOGICAL*1 variable causes a register to be cleared before the variable is accessed. In some situations, the compiler allocates a register throughout an entire loop for this purpose; if not, it must at least generate an extra instruction.

End of IBM Extension

Using Integer Variables of Length 4

Integer variables of length 4 are optimized by strength reduction; they're also generated into branch-on-index instructions. You should always use integer variables of length 4 for DO loop indexes.

IBM Extension

INTEGER*2 variables are not optimized by strength reduction, and they aren't generated into branch-on-index instructions. Therefore, they're less efficient than INTEGER*4 variables.

End of IBM Extension

Eliminating EQUIVALENCE Statements

Equivalenced variables cannot be optimized.

Initializing Large Arrays during Execution

If you initialize large arrays using a DO loop, you get faster overall execution and use less storage than if you initialize using a DATA statement.

For example, the following statements:

```
DOUBLE PRECISION A(5000)
DATA A(5000)/5000*0.0/
```

generate 40000 bytes of object module information—more than 500 TXT cards. The 40000 zeros must be placed in the object module by the compiler, placed in the load module by the linkage editor, and fetched into storage when you execute the program.

Using Common Blocks Efficiently

Each reference to a variable in common requires that the address of the common block be in a register. The following recommendations are based on this fact.

1. The number of common blocks should be minimized. The following example shows why:

   Three Registers Required: / **One Register Required:**

   ```
   COMMON /X/ A          COMMON /Q/ A,B,C
   COMMON/Y/ B           A=B+C
   COMMON /Z/ C
   A=B+C
   ```

   As the example shows, you should group concurrently referenced variables into the same common block.

2. Place scalar variables before arrays in a given common block. The following example shows why:

   Two Registers Required: / **One Register Required:**

   ```
   COMMON /Z/ X(5000),Y        COMMON /Z/ Y, X(5000)
   X(1)=Y                      X(1)=Y
   ```

   In the same way, you should place small arrays before large ones. All the scalar variables and the first few arrays can then be addressed through one address constant. The subsequent larger arrays will probably each need a separate address constant.

3. If a scalar variable in a common block is referred to frequently, assign it from the common block into a local variable. References to the local variable will not then require that the common block address be in a register.

If you do this, always remember that changing the local variable does *not* change the common variable.

Passing Subroutine Arguments in Common Blocks

If you 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.

Avoiding Adjustable Dimensioned Arrays

Subscripting of adjustable dimensioned arrays requires additional indexing computations. In addition, if you use an adjustable dimensioned array as a subroutine parameter, there are additional calculations that must be performed on each entrance into the subroutine.

You can lessen the amount of extra processing, however, in the following ways:

1. If the location and size of the array do not change during repeated calls to the subroutine, you can insert an initialization call to the subroutine to define the array; subsequent execution calls need not then refer to the array, as the following example shows:

```
Dimensions Calculated              Dimensions Calculated
Once:                              At Each Entrance:

Main Program                       Main Program
      CALL INIT(A,I,J)
      DO 1 N=1,10                        DO 1 N=1,10
1     CALL EXEC                    1     CALL EXEC(A,I,J)

Subprogram                         Subprogram

      SUBROUTINE INIT(A,I,J)       SUBROUTINE EXEC(A,I,J)
      REAL*8 A(I,J)                REAL *8 A(I,J)
      RETURN
      ENTRY EXEC
```

2. If the 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 the following example shows:

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`

Writing Critical Loops Inline

If your program has a short heavily-referenced DO loop, it's probably worth the effort to remove the loop and expand the code inline in the program. Each loop iteration will execute faster.

Ensuring Recognition of Duplicate Computations

If components of a computation are duplicates, make sure you code the duplicate elements in one of the following ways:

- At the left end of the computation
- Within parentheses

The compiler must follow the left-to-right FORTRAN rules, and this order of computations follows those rules.

The following examples illustrate this concept:

Duplicates Recognized:	**No Duplicates Recognized:**
`A=B*(X*Y*Z)`	`A=B*X*Y*Z`
`C=X*Y*Z*D`	`C=X*Y*Z*D`
`E=F+(X+Y)`	`E=F+X+Y`
`G=X+Y+H`	`G=X+Y+H`

In the pair of examples at the left, the compiler can recognize X*Y*Z and X+Y as duplicates because they're either coded in parentheses or coded at the left end of the computation. In the pair of examples at the right, these rules are not followed, and the compiler cannot, therefore, recognize these duplicates.

Ensuring Recognition of Constant Computations

In a loop, when several components of a computation are constant, ensure that they can be recognized by following one of these coding rules:

1. Move all the constant computations to the left end of the computation.

2. Group constant computations within parentheses.

The compiler follows the left-to-right FORTRAN rules, and this order of computations allows the compiler to recognize the constant portions of the computations.

If C, D, and E are constant and V, W, and X are variable, the following examples show the difference in evaluation:

Constant Computations Recognized	Constant Computations Not Recognized
`V*W*X*(C*D*E)`	`V*W*X*C*D*E`
`C+D+E+V+W+X`	`V+W+X+C+D+E`

Ensuring Recognition of Constant Operands

The compiler can recognize only local variables as having a constant value. (It must always assume that operands in common or in a parameter list can change, and therefore cannot optimize them.)

Therefore, for such items you should define constant operands as local variables.

Eliminating Scaling Computations

If your program performs calculations representing physical values of some kind, you can save computation time by using factoring, as the following simple example shows:

```
Not Using Factoring              Using Factoring

  SUM=0.0                          SUM=0.0
  DO 1 I=1,9                       DO 1 I=1,9
1 SUM=SUM+FAC*ARR(I)             1 SUM=SUM+ARR(I)
                                   SUM=SUM*FAC
```

In many programs, you can use factoring much more extensively than this simple example shows.

Defining Arrays with Identical Dimensions

If all your arrays have the same shape, then the compiler can use a subscript calculated for one array to subscript the others.

In some cases, therefore, you should consider expanding some smaller arrays to match the dimensions of the other arrays with which they're involved. The compiler can then maintain only one index for all the arrays defined as having the same dimensions.

Defining Arrays with Identical Element Specifications

If you define arrays as having the same dimensions *and* the same element specifications, the compiler can compute a subscript for one array and then use it without change for the others.

In some cases, therefore, you should consider expanding smaller arrays to match the elements in the others. You should always do this for arrays with integer or logical operands.

Using Critical Variables Carefully

Certain variables cannot be optimized in certain circumstances:

- Control variables for direct access input/output data sets cannot be optimized at all.
- Variables in input/output statements and in argument lists cannot be optimized by register optimization in the loops that contain the statements.
- Variables in COMMON blocks cannot be optimized across subroutine calls.

You shouldn't use DO loop indexes for any of these purposes.

Avoiding Unneeded Fixed/Float Conversions

Avoid forcing the compiler to convert numbers between the integer and the floating-point internal representations; each such conversion requires several instructions, including some double-precision floating-point arithmetic.

The following example shows one method of avoiding such unnecessary conversions:

One Conversion Needed:

```
      X=1.0
      DO 1 I=1,9
      A(I)=A(I)*X
1     X=X+1.0
```

Multiple Conversions Needed:

```
      DO 1 I=1,9
1     A(I)=A(I)*I
```

When you can't avoid using mixed-mode arithmetic, then code the fixed-point and floating-point arithmetic as much as possible in separate computations.

Minimizing Conversions between Single and Double Precision

Two, or even three, instructions are required to convert data between single and double precision.

Using Scalar Variables as Accumulators

When you're accumulating intermediate summations, keep the result in a scalar variable rather than in an array. Array accumulators require load and store instructions; scalar variable accumulators can be maintained in a register.

Using Efficient Arithmetic Constructions

In subtraction operations, if only the negative is required, change the subtraction operations into additions, as follows:

Efficient: **Inefficient:**

```
      Z=-2.0
      DO 1 I=1,9                     DO 1 I=1,9
1     A(I)=A(I)+Z*B(I)         1     A(I)=A(I)-2.0*B(I)
```

In division operations, do the following:

- For constants, use one of the following constructions:

  ```
  X*(1.0/2.0)
  0.5*X
  ```

  rather than the construction X/2.0.

- For a variable used as a denominator in several places, use the same technique.

Using IF Statements Efficiently

In general, use a block or logical IF statement rather than the arithmetic IF statement.

If you must use an arithmetic IF statement, try to make the next succeeding statement one of the branch destinations.

For multiple branches, either use the computed GO TO statement, or, if the branch can be initialized so that it remains invariant, use an assigned GO TO statement.

In logical IF statements, if your tests involve a series of AND and/or OR operators, try to do the following:

- Put the simplest conditions tested in the leftmost positions.
- Also, put the tests most likely to be decisive in the leftmost positions.
- Put the more complex conditions (such as tests involving function references) in the rightmost positions.

If the first part of the expression causes the logical condition to test as true, then the rest of the expression need not be evaluated, saving execution time.

Using the Object Program Listing

Use the object program listing, which you obtain through the LIST compiler option, to find out what machine instructions the compiler has generated for your program. You can often tell whether the program has been well or poorly optimized.

The essential work for most FORTRAN programs is to compute floating-point numbers (rather than subscripts or DO loop indexes). Take a quick look at the inner loops for such programs; if they contain essentially no fixed-point instructions, the program is efficiently optimized.

Similarly, you can tell from a FORTRAN source program which additions and multiplications and other operations are necessary and which ought to disappear under optimization. You can examine the object program to discover whether there's a reasonable correlation between the generated program and your expectation.

Using the object code listing in this way, is the best way you can study the efficiency of source program optimization.

Neither of these examinations requires detailed knowledge of assembler language.

Source Considerations with OPTIMIZE(3)

When you're using the OPTIMIZE(3) compiler option, there are additional coding considerations you should be aware of.

Common Expression Elimination

OPTIMIZE(3) evaluates expressions and eliminates those that are common to more than one statement. That is, if an expression occurs more than once and the path of execution always executes the first expression and then the second, with no change in the expression value, the first value is saved and used instead of the second expression. OPTIMIZE(3) does this even for intermediate expressions within expressions. For example, if your program contains the following statements:

```
10    A=C+D
       .
       .
       .
20    F=C+D+E
```

the common expression C+D is saved from its first evaluation at 10, and is used at 20 in determining the value of F.

Computational Reordering

OPTIMIZE(3) may move an expression outside of a loop when the operands of the expression are not defined as part of the loop. This can cause execution differences from nonoptimized code.

For example, when an IF statement controls the execution of a computation within a loop, and the computation is moved outside the loop, program execution results may change:

```
      DO 11 I=1,10
      DO 12 J=1,10
 9    IF (B(I).LT.0) GO TO 11
12    C(J)=SQRT(B(I))
11    CONTINUE
```

OPTIMIZE(3) moves the library function call to precede statement 9, which causes the square root computation to be made before the test for zero.

To avoid this unwanted code movement, use the OPTIMIZE(2) option.

You can also get unexpected results when you use CALL OVRFL or CALL DVCHK, because the computations causing overflow, underflow, or divide check conditions could be moved out of the loop in which the test occurs.

Instruction Elimination

If your program defines nonsubscripted variables, and their values are not used between two definitions within one block, or have not been used before the exit from the block, the compiler may eliminate any intermediate storing of the variables.

Chapter 8. Compiling Your Program and Identifying User Errors

The next step is to request the VS FORTRAN compiler program to translate your FORTRAN source statements into an "object module"—a machine code translation of your source program.

Compiling Your Program

Using whatever method your installation requires, you must enter the source program as a file with 80-character records; each record must follow VS FORTRAN formatting rules. Your output from the compiler depends upon the job control options and compiler options you specify.

Requesting Compilation

When you request compilation, the VS FORTRAN compiler reads and analyzes your source program and translates it into machine code.

You can compile, link-edit, and execute all at once, or invoke each step separately. For early debugging, however, it's usually better to request a compile-only run. That way, the compiler can find syntax errors in your program that would prevent a successful execution, and waste machine time.

For information on how to compile your program, see the appropriate chapter (11 through 14).

Automatic Cross-Compilation

Cross-compilation of VS FORTRAN programs is automatic. That is, you can compile your source program under any supported operating system; you can then link-edit the resulting object module to execute under any of the other supported systems.

Compiling Programs for Interactive Debug Under CMS and TSO

If you are using CMS or TSO with the VS FORTRAN Interactive Debug product, see Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377, for general information; and "Specifying CMS Line Numbers When Debugging" on page 230, or "Specifying TSO Line Numbers When Debugging" on page 322, for examples of how to specify line numbers for debugging.

Printing on the IBM 3800 Printing Subsystem

The EBCDIC assignment for the characters in a specific Character Arrangement Table can be found in *IBM 3800 Printing Subsystem Programmer's Guide.*

You may select the type style to be used by including the TRC as the second character to be output by the FORMAT statement for the line. For example:

```
100 FORMAT (' ','n',...
```

The TRC, n, must be 0, 1, 2, or 3.

It is possible to combine more than one type style on a single line by use of the print-without-spacing control character, '+'. For example:

```
100 FORMAT (' ','n',...
200 FORMAT ('+','m',...
```

If you issue a WRITE to the 3800 with FORMAT statement 100, using TRC n, followed by a WRITE with FORMAT statement 200, using TRC m, the output from the two WRITE statements will be combined into one line.

The 3800 will not overprint characters (with the exception of the underscore), so care must be used when setting up the lines to insure that blanks are provided where additional characters are to be placed.

If you want to mix lines of different pitch, note the following:

- When blanks of different pitches are merged, the resulting blank has the pitch of the first one.
- When a printable graphic character is merged with a blank, the resulting character has the pitch of the printable character.

It is possible to underscore characters by either of two methods:

- Five underscored character sets are provided.
- The 3800 has a built-in underscore capability. When it detects that an underscore and another printable character are to be printed in the same character position in a line, it generates an underscored character.

Certain parameters are required to support the IBM 3800 Printing Subsystem. These parameters and sample FORTRAN programs using the 3800 are shown in "Printing on the IBM 3800 Printing Subsystem under MVS" on page 271, "Printing on the IBM 3800 Printing Subsystem under CMS" on page 231, and "Printing on the IBM 3800 Printing Subsystem under VSE" on page 335.

IBM Extension

Using the VS FORTRAN INCLUDE and Conditional INCLUDE Statements

If your source program uses the INCLUDE statement or the conditional INCLUDE statement and CI option, you must create a library member(s) containing the source code to be included. This library must be identified to the system during your compile.

For more information, see the correct section for your operating system: "Using the FORTRAN INCLUDE Statement" on page 230 (VM), "Using the FORTRAN INCLUDE Statement" on page 273 (MVS), and "Using the FORTRAN INCLUDE Statement" on page 336 (VSE).

Conditional INCLUDE

You may selectively activate INCLUDE statements within the FORTRAN source during compilation by using a conditional INCLUDE with the following format:

```
INCLUDE (name) [n]
```

where name is the name of a file to be included, and n, the identifying number of this INCLUDE, can be any number from 1 through 255. To activate this INCLUDE, it is necessary to use the CI (conditional include) compiler option. See "Using the Compiler Options" for more information on the CI option.

End of IBM Extension

Using the Compiler Options

VS FORTRAN compiler options let you specify details about the input source program and request specific forms of compilation output. See Chapter 11, "Using VS FORTRAN under VM" on page 227, Chapter 12, "Using VS FORTRAN under MVS" on page 257, Chapter 13, "Using VS FORTRAN under TSO" on page 317, or Chapter 14, "Using VS FORTRAN under VSE" on page 331, for system considerations in specifying the options.

Figure 29 lists the compiler options and their abbreviations and IBM-supplied defaults. Your system administrator may have changed these defaults for your installation; you may want to note any change in the figure.

Option	Abbreviation	IBM-Supplied Default	Installation Default
AUTODBL(value)	AD	AUTODBL(NONE)	
CHARLEN(number)	CL	500	

Figure 29 (Part 1 of 2). VS FORTRAN Compiler Options

Option	Abbreviation	IBM-Supplied Default	Installation Default
CI(number1,number2,...)	None	None	
DC(name1,name2,...)	None	None	
DECK | NODECK	D | NOD	NODECK	
FIPS(S | F) | NOFIPS	None	NOFIPS	
FLAG(I | W | E | S)	None	FLAG(I)	
FREE | FIXED	None	FIXED	
GOSTMT | NOGOSTMT	GS | NOGS	NOGOSTMT	
LANGLVL (66 | 77)	LVL	LANGLVL(77)	
LINECOUNT (number)	LC	60	
LIST | NOLIST	L | NOL	NOLIST	
MAP | NOMAP	None	NOMAP	
NAME(name)	None	MAIN	
OBJECT | NOOBJECT	OBJ | NOOBJ	OBJECT	
OPTIMIZE(0 | 1 | 2 | 3) | NOOPTIMIZE	OPT | NOOPT	NOOPTIMIZE	
RENT | NORENT	None	NORENT	
SDUMP | NOSDUMP	SD | NOSD	SDUMP	
SOURCE | NOSOURCE	S | NOS	SOURCE	
SRCFLG | NOSRCFLG	SF | NOSF	SRCFLG	
SXM | NOSXM	None	NOSXM	
SYM | NOSYM	None	NOSYM	
TERMINAL | NOTERMINAL	TERM | NOTERM	TERMINAL	
TEST | NOTEST	None	NOTEST	
TRMFLG | NOTRMFLG	TF | NOTF	TRMFLG	
XREF | NOXREF	X | NOX	NOXREF	

Figure 29 (Part 2 of 2). VS FORTRAN Compiler Options

AUTODBL(value)
Provides an automatic means of converting single-precision, floating-point calculations to double precision, and double-precision calculations to extended precision. For more information concerning (value), see "Using the Automatic Precision Increase Facility—AUTODBL Option" on page 29.

The IBM-supplied default is AUTODBL(NONE).

CHARLEN(number)
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.

The IBM-supplied default is a value of 500.

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

Specifies the identification numbers of the INCLUDEs to be processed (where number is any number less than 256).

IBM does not supply any default CI values.

DC(name1,name2,...)

Defines the names of common blocks that are to be allocated at execution time. This option allows the specification of very large common blocks that can reside in the additional storage space available through MVS/XA. This option can be repeated; the lists of names are combined. No blanks are allowed in the list.

On an @PROCESS statement, multiple names can be supplied as parameters to the DC option or on invocation of the compiler (EXECUTE options). In VM, if you specify DC on the FORTVS command, only the 8 characters following the left parenthesis are passed to VS FORTRAN. No error message is generated if any truncation occurs.

No checking is done to see if the names specified are valid names of common blocks.

IBM does not supply any default DC values.

DECK | NODECK

Specifies whether or not the object module in card image format is to be produced.

The IBM-supplied default is NODECK.

FIPS (S | F) | NOFIPS

Specifies whether or not standard language flagging is to be performed, and, if it is, the standard language flagging level: subset or full.

Items not defined in the current American National Standard are flagged. Flagging is valuable only if you want to write a program that conforms to the American National Standard for FORTRAN implemented in LANGLVL(77). If you specify LANGLVL(66) and FIPS flagging at either level, the FIPS option is ignored.

The IBM-supplied default is NOFIPS.

FLAG (I | W | E | S)

Specifies the level of diagnostic messages to be written: I (information) or higher, W (warning) or higher, E (error) or higher, or S (severe) or higher. FLAG allows you to suppress messages that are below the level desired. Thus, if you want to suppress all messages that are warning or informational, specify FLAG(E).

The IBM-supplied default is FLAG(I).

FREE | FIXED

Indicates whether the input source program is to be in free format or in fixed format. These formats are described in more details under "Using Fixed- and Free-Form Input" on page 7.

The IBM-supplied default is FIXED.

GOSTMT | NOGOSTMT

Specifies whether or not internal sequence 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. GOSTMT is useful if you have an error condition in a subprogram and want to know the source of the call. Specification of this option costs only 4 bytes of overhead per call to a subprogram or to a run-time library routine. Specifying GOSTMT for subprograms is recommended.

The IBM-supplied default is NOGOSTMT.

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 manuals only describe the LANGLVL(77) processing.

The IBM-supplied default is LANGLVL(77).

LINECOUNT (number)

Specifies the maximum number of lines on each page of the printed source listing. The number may be in the range 5 to 32765. 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.

The IBM-supplied default is 60 lines per page.

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 204.

The IBM-supplied default is NOLIST.

MAP | NOMAP

Specifies whether or not a table of source program variable names and statement labels is to be written. MAP output is helpful in debugging your program. A complete description of the output is given under "Source Program Map—MAP Option" on page 175.

The IBM-supplied default is NOMAP.

NAME(name)

Can only be specified when LANGLVL(66) is specified. It specifies the name that is generated on the output and the name of the CSECT generated in the object module. It only applies to main programs.

When NAME is omitted, the name is MAIN.

OBJECT | NOOBJECT

Specifies whether or not the object module is to be produced. An object module is required to execute your program.

The IBM-supplied default is OBJECT.

OPTIMIZE (0 | 1 | 2 | 3) | NOOPTIMIZE

Specifies the optimizing level to be used during compilation:

OPTIMIZE (0) OR NOOPTIMIZE specifies no optimization.

OPTIMIZE (1) specifies register and branch optimization.

OPTIMIZE (2) specifies partial code-movement optimization. OPTIMIZE(2) will not relocate any code when it has been determined that relocating the code under consideration would cause unplanned or unexpected interrupts.

OPTIMIZE (3) specifies full code-movement optimization.

If you are debugging your program, it is advisable to use NOOPTIMIZE. To create more efficient code and, therefore, a shorter execution time with (usually) a longer compile time, use OPTIMIZE(2) or (3). The different levels of optimization are described under Chapter 7, "Optimizing Your Program" on page 139.

The IBM-supplied default is NOOPTIMIZE.

RENT | NORENT

Allows a program compiled as RENT to be generated as a reentrant object module that can be invoked as a main program or subprogram. If you are not planning on running your program in a reentrant area, specify NORENT. Otherwise, see "VS FORTRAN Separation Tool (for Both VM and MVS)" on page 189.

If the DEBUG statement (see "Static Debug Statements" on page 201) is specified on the same compilation as RENT, a warning message will be issued and RENT ignored.

The IBM-supplied default is NORENT.

SDUMP | NOSDUMP

Specifies that symbolic dump information is to be generated. This information is used to produce symbolic listings of your program data at abnormal termination or in response to a program call to the SDUMP service subroutine. For details on calling SDUMP, see "Requesting Symbolic Dumps—CALL Statement" on page 211. SDUMP listings are shown in *VS FORTRAN Language and Library Reference*.

The NOSDUMP option makes the object module smaller. Execution time is the same, whether or not NOSDUMP is specified.

Note: If you want to use CMS or TSO line numbers during your interactive debug sessions, both NOSDUMP and TEST must be specified.

For more information, see Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377.

The IBM-supplied default is SDUMP.

SOURCE | NOSOURCE
Specifies whether or not the source listing is to be produced. By using the NOSOURCE option, you can decrease the size of your listing. If SRCFLG is specified, NOSOURCE is overridden.

The IBM-supplied default is SOURCE.

SRCFLG | NOSRCFLG
Controls the inserting of error messages in the source listing. The SRCFLG option allows you to view the error message after the line which created the error instead of at the end of the listing. If SRCFLG is specified, NOSOURCE is overridden.

The IBM-supplied default is SRCFLG.

SXM | NOSXM
Formats XREF or MAP listing output for a 72-character-wide terminal screen. The NOSXM option formats listing output for a printer. For more details, see "Using the SXM Option" on page 174.

The IBM-supplied default is NOSXM.

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 404.

The IBM-supplied default is NOSYM.

TERMINAL | NOTERMINAL
Specifies whether or not error messages and compiler diagnostics are to be written on the SYSTERM output data set and whether or not a summary of error messages is to be printed.

Specify the NOTERMINAL option if you are running batch jobs on MVS or VSE and do not want output to a SYSTERM data set.

The IBM-supplied default is TERMINAL.

TEST | NOTEST
TEST overrides any optimization level above OPTIMIZE(0), and adds execution-time overhead.

VS FORTRAN Interactive Debug (5668-903) does not require programs to be compiled with this option. See the table under "Compiling a VS

FORTRAN Program" on page 377 for information about programs compiled with TEST, with or without SDUMP. See also the appropriate VS FORTRAN Interactive Debug manual listed in "Related Publications" on page v.

The IBM-supplied default is NOTEST.

TRMFLG | NOTRMFLG
Causes the FORTRAN source statement in error, if applicable, and its associated error messages (formatted for the terminal being used) to be displayed at the terminal; all other information will be suppressed. Specify the NOTRMFLG option if you are running batch jobs on MVS or VSE and do not want output to a SYSTERM data set.

The IBM-supplied default is TRMFLG.

XREF | NOXREF
Specifies whether or not a source cross-reference listing is to be produced. For a description of cross-reference output, see "Source Program Cross-Reference Dictionary—XREF Option" on page 178.

The IBM-supplied default is NOXREF.

Conflicting Compiler Options

The following table lists conflicting compiler options that will create an error message if both are used. The table also reflects those options that will be assumed when conflicting compiler options are specified.

Conflicting Compiler Options		Options Assumed	
FIPS	FLAG¬ =I	FIPS	FLAG=I
FIPS	LANGLVL(66)	NOFIPS	LANGLVL(66)
LANGLVL(77)	NAME	LANGLVL(77)	Ignore NAME
NOSOURCE	SRCFLG	SOURCE	SRCFLG
SYM	NODECK and NOOBJ	NOSYM	NODECK and NOOBJ
TEST	NAME¬ =MAIN	TEST	NAME=MAIN
TEST	NOOBJ	TEST	OBJ
TEST	OPT>0	TEST	OPT=0

Modifying Compilation Options—@PROCESS Statement

The options specified when the compiler is invoked remain in force for all source programs you're compiling, unless you override them with the @PROCESS statement.

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

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

  All compiler options except OBJECT, DECK, DISK, PRINT, and NOPRINT are permissible (the latter three are available only to CMS users; for details, see "Using the VS FORTRAN Compiler Options" on page 229).
- If NODECK or OBJ has been specified on the EXEC statement, you cannot specify DECK or NOOBJ, respectively, on the @PROCESS statement.
- If both TEST and OPT(0) have been specified and NAME is not MAIN, the name becomes MAIN.
- TERMINAL and TRMFLG cannot be specified on the @PROCESS statement if TERMINAL was not specified on the EXEC statement or in the system defaults.

Using VS FORTRAN Interactive Debug with VS FORTRAN

If you want to use VS FORTRAN Interactive Debug (5668-903) with VS FORTRAN, you should not specify the combination of TEST and NOSDUMP options unless you also want to use TSO or CMS line numbers during your debugging session. If you want to use TSO or CMS line numbers, both TEST and NOSDUMP must be specified.

For more information, see Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377.

Compiler Output

The output the compiler gives you depends upon whether you've accepted the default compiler options in force for your organization, or whether you've modified the defaults using explicit compiler options.

Compiler Output with Default Options

If you use the default compiler options, you should get output printed in the following order (unless your installation has changed the defaults):

- 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.

Output with Explicit Compiler Options

In addition to that listed above, you can cause each compilation to produce the following output:

- A listing of the object module (LIST option) in pseudo-assembler language (that is, the assembler instructions that would have been generated for the object module if the compiler had translated it into assembler before producing the machine code)
- A copy of the object module (OBJECT option) in card image format
- A table of names and statement labels (MAP option) defined in the source program, and their location in the module
- A cross-reference listing (XREF option) of variables and labels used in the source program, and informational flags for each
- Messages flagging statements that do not conform to the language standard level you've chosen (FIPS option)

Depending on the options you've chosen, the output you'll get is shown in Figure 30 on page 166. (Options that produce output at compile time are shown in the order in which they are printed in the output listing.)

For information on how to use the SOURCE, FLAG, MAP, XREF, SRCFLG, TRMFLG, and FIPS options, see "Identifying User Errors" on page 167.

For information on how to use the DECK and OBJECT options, see "Object Module as Link-Edit Data Set" on page 183.

For information on how to use the LIST option, see Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185.

Compiler Option	Produces the Following Output
---	Compilation Headings: Compiler Name and Release Level, Source Program Name, Compilation Date, Listing Page Number
SOURCE	A listing of the source program
SRCFLG	Error messages displayed in the source listing immediately following the source statement in error during the syntax scan of the statement Error messages occurring after the syntax scan displayed at the end of the listing
SXM	XREF and MAP listing output formatted to a 72-character-wide terminal screen
FIPS	Error messages resulting from non-ANS standard usages
XREF	Cross-reference information, showing each symbol and its type and usage, as well as where each symbol and label is defined and used in the program, including variables referenced but not initialized
LIST	A listing of the object module, containing the ISN, if available, the relative location of each generated constant or statement, the name of the source item used in the instruction, plus section headings for: constants and data addresses, common areas, equivalenced variables, address constants, external references, parameter lists, the save area, and generated instructions; the program information block, the entry table, and the compiler properties table
MAP	A map of the source program, showing the program name and size, name usage and type, common block information, and statement label usage and location

Figure 30 (Part 1 of 2). Compiler Output Using Explicit Options

FLAG	A listing of error messages at the FLAG level you've chosen: I requests a listing of all messages produced W requests a listing of warning, error, severe error, and abnormal termination messages (return code 4 or higher) E requests a listing of error, severe error, and abnormal termination messages (return code 8 or higher) S requests a listing of severe error and abnormal termination messages (return code 12 or higher) For an explanation of these message codes, see "Diagnostic Message Listing—FLAG Option" on page 172.
OBJECT	The object module in machine language form
TERMINAL	Statistics and messages directed to the SYSTERM data set; also produces an indexed summary of statistics and messages for each compilation at the end of all compilations
TRMFLG	Error messages displayed on your SYSTERM data set
DECK	The object module in machine language form to be produced as an output data set for punching
---	Compilation statistics: source program name, number of statements compiled, generated object module size (in bytes), the number and severity of error messages produced

Figure 30 (Part 2 of 2). Compiler Output Using Explicit Options

Identifying User Errors

Many common coding errors can cause problems in your compilation; for example:

1. Misspelling a VS FORTRAN language element.
2. Omitting required punctuation, or inserting unneeded punctuation.
3. Ignoring VS FORTRAN formatting rules.
4. Forgetting to assign values to variables and arrays before you use them.
5. Moving data into an item that's too small for it. (This causes truncation.)

   In particular, beware of assigning data to more array elements than the array contains; you can inadvertently destroy subsequent data and instructions.

6. Specifying subscript values that are not within the bounds of an array.

7. Inadvertently changing types defined in an IMPLICIT statement by explicit type statements.

8. Making invalid data references to equivalenced items of differing types (for example, integer and real).

9. Transferring control from outside a DO loop into an intermediate point in a DO loop.

   Note: The extended range of a DO-loop is not part of the current American National Standard for FORTRAN. It is a valid construction under the LANGLVL(66) compiler option, but under the default option LANGLVL(77), its use is not diagnosed and no diagnostic message is generated.

10. Using arithmetic items for intermediate calculations that are too small to give the precision you need in the result. For example, if you want to show more than six decimal digits, you should perform the intermediate calculations in double precision.

11. Failing to inspect code movements in programs compiled with the OPTIMIZE(3) option. For example, if an IF statement controls execution of a computation within a loop, the computation may be moved outside the loop and give you results you don't expect. See Chapter 7, "Optimizing Your Program" on page 139.

12. Attempting to process input records after a file has been used for output. See Chapter 5, "Programming Input and Output" on page 69.

13. Writing main programs or subprograms that directly or indirectly invoke themselves. See Chapter 6, "Subprograms and Shared Data" on page 113.

14. Writing a series of subprograms without a required main program.

15. Defining dummy arguments and actual arguments that do not agree in type and/or length; for example, arrays that do not have the same dimensions, integer actual arguments with real dummy arguments or vice versa.

16. In a subprogram, assigning new values to arguments associated with variables in common.

17. Failing to initialize VFEIN# when your main program is not a FORTRAN program. See Appendix A, "Assembler Language Considerations" on page 389.

18. Referring to a statement function with other than the defined number of arguments.

19. Supplying a printer carriage control character as the first character of a line to be printed, but not specifying in a data definition statement that carriage control characters are present.

20. The VS FORTRAN compiler uses the OS FORTRAN H Extended architecture for rounding infinite binary expansions. The OS FORTRAN G1 compiler also rounds, but the DOS FORTRAN F compiler truncates. If you recompile programs originally written for the DOS FORTRAN F compiler, you may see different results from execution.

21. If you compile a program under OPT(0) that has too many branch addresses, you will receive the message:

    ```
    IFX4001I--YOU HAVE EXCEEDED THE COMPILER LIMIT ON
              ADDRESS CONSTANTS.
    ```

    To avoid this problem, compile the program at OPT(1), OPT(2) or OPT(3). If OPT(2) or OPT(3) do not compile the program properly, use OPT(1). If the problem persists, divide the program into smaller subprograms.

22. Concatenating character strings in such a way that overlap can occur.

23. Failing to DIMENSION an array before use. When this happens, the array name is assumed to be a function call, and a severe program failure occurs.

24. Not using the lowest value of CHARLEN possible, and causing routines to be larger than necessary.

Using the Compiler Output Listing

The compiler output listing is designed to help you pinpoint any errors you've made in your source program. The listing is described in the following sections, together with hints on how to use it effectively.

Compilation Identification

This portion of the listing helps you identify each compilation, even among several done on the same day.

The heading on each page of the output listing gives the name of the compiler and its release level, the name of the source program, and the date and time of the run in the format:

```
DATE: month day, year     TIME: hour.minute.second
```

For example:

```
DATE: MAY 1, 1981         TIME: 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: Your installation may 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, except for the addition of internal sequence numbers (ISN) as a prefix. Figure 31 on page 171 is an example of the source program listing.

Use the source listing for desk checking to make sure that your source statements conform to VS FORTRAN syntax. The internal sequence numbers help you identify the statements causing diagnostic messages.

```
REQUESTED OPTIONS (EXECUTE):  NOTERM NOTRMFLG SXM
REQUESTED OPTIONS (PROCESS):  LIST XREF MAP
OPTIONS IN EFFECT: LIST MAP XREF NOGOSTMT NODECK SOURCE NOTERM OBJECT FIXED
                   NOTEST NOTRMFLG SRCFLG NOSYM NORENT SDUMP AUTODBL(NONE)
                   SXM OPT(0) LANGLVL(77) NOFIPS FLAG(I) NAME(MAIN  )
                   LINECOUNT(60) CHARLEN(500)
             *....*...1.......2.......3.......4.......5.......6.......7.*......8

                C SAMPLE PROGRAM TO DEMONSTRATE VS FORTRAN
 ISN        1         REAL*8 PI
 ISN        2         PARAMETER (PI = .314159265D1)

 ISN        3         COMPLEX*8    C8V, C8A, C8B
 ISN        4         COMPLEX*32   C32
 ISN        5         LOGICAL*1    L1
 ISN        6         REAL*8       R8A, R8V, R8VNU
 ISN        7         CHARACTER*15 CHAR15

 ISN        8         EQUIVALENCE  (R4A(4), R8A(2), C32(1,1))
 ISN        9         EQUIVALENCE  (I2(3), L1)

 ISN       10         DIMENSION    R8A(7), C32(4,5), R4A(11), I2(3)

 ISN       11         COMMON /COM1/   R4A, C8A
 ISN       12         COMMON /COM2/   L1,  C8B

 ISN       13    111  FORMAT('1OUTPUT FOR ', A14, 14F10.7, E15.7, 2(F20.16))

 ISN       14         DATA A2/3.14159/
 ISN       15         DATA CHAR15/'SAMPLE PROGRAM '/

 ISN       16         ASSIGN 111 TO J

 ISN       17         A1 = (A3 + R8A(2))*3

 ISN       18         IF (A1 .EQ. 0) GOTO 200

 ISN       19         IF (A2 .EQ. 0) GOTO 200

 ISN       20         DO 100 I = 1,7
 ISN       21            IF (L1)
                     1       R8V =  R8V + R8A(I) + (.0007, .0021) + FLOAT(I)
 ISN       23    100  CONTINUE

 ISN       24         CALL CXSUB(*300,R8V,A1,PI)

 ISN       25         R8A = 1.0002 + R8A(1)

***ERROR 1027(S)***   NON-SUBSCRIPTED ARRAY NAME APPEARS AS LEFT-OF-EQUAL
                          SIGN VARIABLE. SPECIFY A SUBSCRIPTED ARRAY NAME
                          OR A VARIABLE NAME.

 ISN       26    200  WRITE(6,J) CHAR15, R8A, A1, C32

 ISN       27         DATA C8V/(2.540005, 2.781828)/

 ISN       28    300  PAUSE 'THE END'
 ISN       29         END
```

Figure 31. Source Program Listing Example—SOURCE and SRCFLG Options

Note: When this program is executed, the diagnostic messages shown in Figure 32 on page 172 are produced.

Source Program Listing—SRCFLG Option

The SRCFLG option enables you to obtain error messages immediately following the statement in error. Figure 31 on page 171 shows an example of an error that occurred at ISN 25.

Note: If SRCFLG is specified, the statement in error will be printed even if NOSOURCE is specified.

Diagnostic Message Listing—FLAG Option

If the severity level of the message is greater than or equal to that you've specified in the FLAG option, and there are syntax errors in your VS FORTRAN source program, the compiler detects them and gives you a message. The messages are self-explanatory, making it easy to correct your syntax errors before recompiling your program. Examples of VS FORTRAN messages are shown in Figure 32.

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

IFX1027I   RPLC   12(S)    25    NON-SUBSCRIPTED ARRAY NAME APPEARS AS LEFT-OF-
                                 EQUAL SIGN VARIABLE. SPECIFY A SUBSCRIPTED
                                 ARRAY NAME OR A VARIABLE NAME.

IFX2323I   COMN    4(W)          VARIABLE "R8A" IN COMMON "COM1" IS INEFFICIENTLY
                                 ALIGNED. VARIABLES SHOULD BE ALIGNED ON
                                 APPROPRIATE BYTE BOUNDARIES.

IFX2323I   COMN    4(W)          VARIABLE "C32" IN COMMON "COM1" IS INEFFICIENTLY
                                 ALIGNED. VARIABLES SHOULD BE ALIGNED ON
                                 APPROPRIATE BYTE BOUNDARIES.

IFX2332I   COMN   12(S)          THE VARIABLE "I2" WILL CAUSE COMMON "COM2" TO
                                 EXTEND TO THE LEFT BECAUSE OF ITS POSITION IN
                                 EQUIVALENCE STATEMENT AT ISN "9". CHECK VARIABLE
                                 PLACEMENT.

IFX2323I   COMN    4(W)          VARIABLE "C8B" IN COMMON "COM2" IS INEFFICIENTLY
                                 ALIGNED. VARIABLES SHOULD BE ALIGNED ON
                                 APPROPRIATE BYTE BOUNDARIES.

*STATISTICS*  SOURCE STATEMENTS = 28, PROGRAM SIZE = 1544 BYTES, PROGRAM
NAME = MAIN   PAGE:  1.

*STATISTICS* 5 DIAGNOSTICS GENERATED. HIGHEST SEVERITY CODE IS 12.

**MAIN** END OF COMPILATION 1 ******

******* SUMMARY STATISTICS ******* 5 DIAGNOSTICS GENERATED. HIGHEST
SEVERITY CODE IS 12.
```

Figure 32. Examples of Compiler Messages—FLAG Option

All VS FORTRAN compiler messages are in the following format:

```
     IFXnnnnI mmmm level [isn] 'message text'
```

where the areas have the following meanings:

IFX	is the message prefix identifying all VS FORTRAN compiler messages
nnnnI	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 merely 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 usually tells you that a minor error, which does not violate VS FORTRAN syntax, was detected.

The severity level is 4.

If no messages are produced that exceed this level, you can safely link-edit and execute the compiled object module.

8(E) Is an error message; usually, a VS FORTRAN syntax error was detected, but the compiler makes a corrective assumption and completes the compilation.

The severity level is 8.

It is still possible that the program will execute correctly.

12(S) Is a serious error message; an error was detected which violates VS FORTRAN syntax, and for which the compiler could make no corrective assumption.

The severity level is 12.

You should not attempt execution, except possibly for debugging purposes. During compilation, if the compiler detects an S-level error, it inserts a call for a library function instead of generating the code for the statement. During execution, if and when this statement in the program is reached, an error message that includes the internal sequence number of the statement in error is produced, and the program is terminated.

16(U) Is an abnormal termination message; an error was detected which stopped the compilation before it could be completed.

The severity level is 16.

[isn] gives the internal sequence number of the statement in which the error occurred, if the internal sequence number can be determined.

'message-text' explains the 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 SXM is specified, 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 fixing compile-time and execution-time errors.

A storage map and a cross-reference dictionary show the use made of each variable, statement function, subprogram, or intrinsic function within a program; this can help you figure out obscure syntax or logic errors that aren't immediately apparent from the source listing.

A cross-reference dictionary shows the names and statement labels in the source program, together with the internal statement numbers in which they appear. The cross-reference dictionary can also be of great assistance during the debugging of execution errors.

If SXM is also in effect, MAP and XREF listing output at a terminal will be formatted to 72-character width.

You can use the storage map and cross-reference dictionary 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.)
- 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 33 on page 176.

The first line of a storage map gives the name and size of the source program; the size is given in hexadecimal format.

```
STORAGE MAP

TAG: SET(S)    REFERENCED(F)       USED AS ARGUMENT(A)   EQUIVALENCED (E)
ASSIGNED(G)    INITIAL VALUE (I)   NAMED CONSTANT (K)
IN COMMON(C)   SUBPROGRAM(X)       STATEMENT FUNCTION (T)

PROGRAM NAME: MAIN.   SIZE OF PROGRAM:     608 HEX BYTES.

NAME  MODE  TAG  ADDR.      NAME  MODE  TAG  ADDR.

A1     R*4   SFA   000260       A2     R*4   FI    000264
A3     R*4   F     000268       CHAR15 CHAR  FAI   0002D4
CXSUB        FX    0002F0       C32    C*32  FCEA  00000C
C8A    C*8   FCE   00002C       C8B    C*8   FCE   000001
C8V    C*8   I     UNREFD       FLOAT  R*4   X     UNREFD
I      I*4   SF    000258       I2     I*4   CE    UNREFD
J      I*4   SFAG  00025C       LI     L*1   FCE   000000
PI     R*8   FK    0001D8       R4A    R*4   FCE   000000
R8A    R*8   FCEA  000004       R8V    R*8   SFA   000250
R8VNU  R*8         UNREFD       VFEE#        FX    0002F4
VFEIM#       FX    0002FC       VFEP#        FX    0002F8
VFFXF#       FX    000300       VFIXF#       FX    000304
VFWSF#       FX    000308       VSERH#       FX    00030C

COMMON INFORMATION

  NAME:   COM1.   SIZE:      28C HEX BYTES.   (E) - EQUIVALENCED

  NAME    MODE   DISPL.    NAME     MODE   DISPL.

  R8A(E)    R*8     000004    C32(E)   C*32    00000C
  R4A(E)    R*4     000000    C8A(E)   C*8     00002C

  NAME:   COM2.   SIZE:        9 HEX BYTES.   (E) - EQUIVALENCED

  NAME     MODE   DISPL.    NAME    MODE   DISPL.

  I2(E)      I*4     FFFFF8    L1(E)    L*1     000000
  C8B(E)     C*8     000001

LABEL INFORMATION

  LABEL   DEFINED   ADDR.    LABEL   DEFINED   ADDR.

    100       23      00045A     111      13      0000C8
    200       26      000496     300      28      0004B2
```

Figure 33. Example of a Storage Map—MAP Option

NAME Column

The first column is headed NAME—it shows the name of each variable, statement function, subprogram, or implicit function in the program.

MODE Column

The second column is headed MODE—it gives the type and (except for character items) length of each name, in the format:

```
    type*length
```

where the *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. The use codes are:

A	for a variable used as an actual argument in a parameter list, including variables used as arguments to calls that do not appear explicitly in the source but are generated by the compiler; for example, character manipulation functions and I/O statements.
C	for a variable in a common block.
E	for a variable in an equivalenced block.
F	for a variable whose value was referred to during some operation.
G	for a variable referenced in an ASSIGN statement.
I	for a variable specified with an initial value.
K	for a named constant.
S	for a variable whose value was set during some operation.
T	for a statement function.
X	for an external function.

Address Column

The fourth column is headed ADDR—it gives the relative address assigned to a name.

Note that, 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 numbers used in the program. The label map shows the following forms of statement numbers:

- Source statement labels—as entered
- FORMAT statement labels—as entered

It also gives you the internal sequence number (ISN) for the statement in which the label is defined and the address assigned to the label.

Source Program Cross-Reference Dictionary—XREF Option

Figure 34 on page 179 shows a cross-reference listing of the XREF option.

Data Item Dictionary—XREF Option

The symbols used in the TAG column are defined at the top of the cross-reference dictionary.

From left to right, the subsequent columns give you the following information:

NAME Column: Names are listed in alphabetic order.

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

```
SYMBOL CROSS-REFERENCE DICTIONARY

PROGRAM NAME: MAIN.

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

NAME    MODE  TAG   DECLARED  REFERENCES

A1      R*4                   17    18    24    26
A2      R*4   V               14    17    19
CHAR14  CHAR  VT          7   15    26
CXSUB         X               24
C32     C*32  ACET        4    8    26
C8A     C*8   CT          3   11
C8B     C*8   CT          3   12
C8V     C*8   VT          3   27
FLOAT   R*4   I               22
I       I*4                   20    22    22
I2      I*4   ACE         9   UNREFERENCED
J       I*4   S               16    26
L1      L*1   CET         5    9    12    21
PI      R*8   KT          1    2    24
R4A     R*4   ACE         8   11
R8A     R*8   ACET        6    8    17    22    25    26
R8V     R*8   T           6   22    22    24
R8VNU   R*8   T           6   UNREFERENCED

VARIABLES REFERENCED BUT NOT SET  (*possibly set as argument)

   C32      R8A

LABEL CROSS-REFERENCE DICTIONARY

TAG: FORMAT(F)       OBJECT OF BRANCH(B)            USED AS ARGUMENT(A)
NON-EXECUTABLE(N)    USED IN ASSIGN STATEMENT(S)

LABEL        TAG  DEFINED  REFERENCES

         100          23     20
         111   NFS    13     16
         200   B      26     18    19
         300   AB     28     24
```

Figure 34. Example of a Cross-Reference Dictionary—XREF Option

TAG Column: The type for each name shows its usage, which can be:

A	an array
C	an item in common
D	a dummy argument
E	an equivalenced item
F	a statement function
G	a generic name
I	an intrinsic function
K	a named constant
N	an entry name
P	a promoted item
Q	a padded item
S	a variable referenced in an ASSIGN statement
T	an item defined in an explicit type statement
V	an initial value specified
X	an external subprogram
Y	an item in a dynamic common

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

REFERENCES Column: The REFERENCES column gives the internal sequence number(s) of each statement in the source program where the data item is referenced.

If there are no references within the program, this column contains the word UNREFERENCED.

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

Statement Label Dictionary—XREF Option

In the statement label dictionary, the following columns are defined:

LABEL Column: Statement labels are listed in ascending order.

TAG Column: Each LABEL is listed by its status, which can be:

A	used as an argument
B	an object of a branch
F	label for a FORMAT statement
N	label for a nonexecutable statement
S	label used in an ASSIGN statement

If a label is used in more than one way, all tags that apply are printed.

DEFINED Column: This column displays the internal sequence number (ISN) of the statement in which the label is defined.

REFERENCES Column: This column displays the internal sequence number (ISN) of all statements in which there are references to the label.

If there are no program references to the label, the word UNREFERENCED is printed.

End of Compilation Message

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

MAINEND OF COMPILATION n ******

where MAIN is the program name and n is the number identifying this program's sequence in a batch compilation.

Using the Terminal Output Display

Through the TRMFLG option, you can present a source statement and its diagnostic together on your terminal. Figure 35 on page 182 shows output generated by the TRMFLG option.

```
VS FORTRAN COMPILER ENTERED.  09:04:07

ISN   25:      RSA = 1.0002 + R8A(1)
NON-SUBSCRIPTED ARRAY NAME APPEARS AS LEFT-OF-EQUAL SIGN
VARIABLE. SPECIFY A SUBSCRIPTED ARRAY NAME OR A VARIABLE
NAME.

VARIABLE "R8A" IN COMMON "COM1" IS INEFFICIENTLY ALIGNED.
VARIABLES SHOULD BE ALIGNED ON APPROPRIATE BYTE BOUNDARIES.

VARIABLE "C32" IN COMMON "COM1" IS INEFFICIENTLY ALIGNED.
VARIABLES SHOULD BE ALIGNED ON APPROPRIATE BYTE BOUNDARIES.

THE VARIABLE "I2" WILL CAUSE COMMON "COM2" TO EXTEND TO THE
LEFT BECAUSE OF ITS POSITION IN EQUIVALENCE STATEMENT AT
ISN "9". CHECK VARIABLE PLACEMENT.

VARIABLE "C8B" IN COMMON "COM2" IS INEFFICIENTLY ALIGNED.
VARIABLES SHOULD BE ALIGNED ON APPROPRIATE BYTE BOUNDARIES.

**MAIN** END OF COMPILATION 1 ******
VS FORTRAN COMPILER EXITED.   09:04:09

SEVERE ERROR MESSAGES ISSUED.
R(00012);
```

Figure 35. Example of Compile-Time Messages—TRMFLG Option

Note: See TERMINAL and TRMFLG options under "Using the Compiler Options" on page 157 for descriptions of output each presents. If both are specified, you get all the output described for both.

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 172, and are all at the 4(W) (warning) level.

Object Module as Link-Edit Data Set

Your input to the linkage editor can be the object module in machine-language format (which you request through the OBJECT compiler option), or as a machine-language input data set (which you request through the DECK compiler option).

You request the OBJECT option when you want either to load and execute your program immediately, or to load the program and keep the load module for execution at some later time. This enables you to combine the link-edit task with the compilation task. You can then catalog and/or execute the load module produced.

You use the DECK option when you want to catalog the object module and save it for future link-edit runs. The deck produced is an 80-character, fixed-format card deck. The deck is a copy of the object module, which consists of dictionaries, text, and an end-of-module indicator.

Chapter 9. Executing Your Program and Fixing Execution-Time Errors

Executing Your Program

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

When you execute a load module, you may need many different files. For information about these files, see the appropriate chapter (11 through 14) that tells about considerations for the operating system at your installation. In particular, see "MVS/XA Considerations" on page 311 for information about execution under MVS/XA.

Reentrant vs. Nonreentrant Programs

With Release 4, you have the option of compiling your program in a reentrant or nonreentrant fashion. This decision has many implications on compiler performance, the code generated, the link-editing, and the actual running of your program.

Reentrant Programs

A reentrant program has several characteristics:

- It must not store any data within its range, but can use a remote work area.
- It cannot be linked with a nonreentrant program and the combination considered reentrant.
- It is sharable among two or more users.
- It is not location sensitive.

Nonreentrant Programs

A nonreentrant program has the following characteristics:

- It may store within its range, and may also use a remote work area.
- It may be linked with another nonreentrant program.

- It is not sharable (each user must have a copy).
- It may be location sensitive.

Considerations for Using Reentrant Programs

You can use reentrant code to get the advantages of sharing code, storage relief, and performance improvement. You need not have a personal copy to run a program, but may share a program with other users. The more users sharing a program, the less storage is used. This is because with one shared copy, the same storage is used by multiple users, so less paging to auxiliary storage takes place. This reduces the amount of time needed to access a program.

There are some disadvantages to using reentrant code. They are:

- Space to store a reentrant program may not be available.
- The link pack area (LPA) or discontiguous shared segment (DCSS) space may not be available.
- Small programs get no observable advantage.
- To install a module in a link pack area or a DCSS, you must reload (IPL) your operating system.

The general procedure for implementing a program that runs reentrant code generated by the VS FORTRAN compiler is as follows:

For VM:

Note: The first four steps in the list below are generally done by a system programmer; the remaining steps can be done by an application programmer, but steps 7 and 8 may require a privileged class E user.

1. Update and reassemble the System Name Table.
2. Reserve storage for the shared segment.
3. Generate and load an updated nucleus.
4. ReIPL VM/SP.
5. Compile your program successfully with the RENT compiler option.
6. Use IFYVSFST to separate your compiler object file into a reentrant part and a nonreentrant part.
7. Load reentrant part.
8. Issue a "SAVESYS reentrant-part-name" command.
9. Run your program, which involves adding any other needed routines, the desired library routines, and file definition statements, and starting up.

For MVS:

Note: The first and fourth steps in the list below are generally done by a system programmer; the remaining steps can be done by an application programmer.

1. Add entries to a SYS1.PARMLIB member (IEALPA09, for example) to indicate the library and program name(s) to load at IPL time.

2. Compile your program successfully with the RENT compiler option.

3. Use IFYVSFST to separate your compiler object file into a reentrant part and a nonreentrant part.

4. Add the reentrant part(s) to the operating system. This involves adding the modules to the previously indicated load module library. MVS requires a reIPL operation to make the modules load into the shared LPA.

5. Run your program, which involves adding any other needed routines and the desired library routines, job control statements, and starting up.

Scenarios for using the RENT Compiler Option

On CMS

The following scenario assumes that the shared segment space, RENTFT, is allocated and available.

1. Compile your program with the RENT option:

   ```
   FORTVS IFYSMPFT (RENT
   ```

2. Use the separation program to process the TEXT compiler output:

   ```
   FI SYSIN DISK IFYSMPFT TEXT
   FI SYSPRINT DISK IFYSMPFT TOOLLIST
   FI SYSUT1 DISK SMPFT TEXT
   FI SYSUT2 DISK RENTFT TEXT
   FI SYSUT3 DISK IFYSMPFT TEMP
   GLOBAL TXTLIB VFORTLIB CMSLIB
   LOAD IFYVSFST IFYVSFIO (CLEAR
   START * RENTFT
   ERASE IFYSMPFT TEMP
   ```

3. Install the reentrant part in a sharable area:

   ```
   LOAD RENTFT (CLEAR ORIGIN 400000
   SAVESYS RENTFT
   ```

4. Run the program:

   ```
   GLOBAL TXTLIB VLNKMLIB VFORTLIB CMSLIB
   LOAD SMPFT (CLEAR NOAUTO
   START
   ```

On MVS

1. Compile your program with the RENT option:

```
//  EXEC PGM=FORTVS,PARM='RENT'
    .
    .
    .
//SYSLIN DD DSN=&TEMP,DISP=(NEW,PASS),...
```

2. Use the separation program to process the object output:

```
//   EXEC  PGM=IFYVSFST,PARM='RENTFT'
//STEPLIB  DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSUT1   DD DSN=&NRENT,DISP=(NEW,PASS),...
//SYSUT2   DD DSN=&RENTP,DISP=(NEW,PASS),...
//SYSUT3   DD DSN=&VTEMP,DISP=(NEW,DELETE),...
```

3. Install the reentrant part in a sharable area:

```
//LKEDR    EXEC PGM=IEWL,PARM='RENT,PRUS,XREF,LET'
    .
    .
    .
//SYSLMOD DD DSN=SYS1.TESTLIB,DISP=OLD
//SYSLIN  DD DSN=&RENTP,DISP=(OLD,PASS)
    .
    .
    .
```

4. Create SYS1.PARMLIB member IEALPAFT with entry:

```
SYS1.TESTLIB RENTFT
```

5. ReIPL MVS to make RENTFT available:

```
r 00,CLPA,MLPA=FT
```

6. Install the nonreentrant part in a "permanent" library:

```
//LKEDN   EXEC PGM=IEWL,PARM='XREF,LET,LIST'
    .
    .
    .
//SYSLIB  DD DSN=SYS1.VFORTLIB,DISP=SHR  (if load mode)
```

or

```
//SYSLIB  DD DSN=SYS1.VLNKMLIB,DISP=SHR  (if link mode)
//        DD DSN=SYS1.VFORTLIB,DISP=SHR
    .
    .
    .
//SYSLMOD DD DSN=mylib(myprog),DISP=OLD,...
//SYSLIN  DD DSN=&NRENT,DISP=(OLD,PASS)
    .
    .
    .
```

7. Run the program (any number of users running concurrently):

```
//GO       EXEC PGM=myprog
//STEPLIB  DD DSN=mylib,DISP=SHR
//         DD DSN=SYS1.VFORTLIB,DISP=SHR   (if load mode used)
//FT06F001 DD SYSOUT=A
//FT05F001 DD *
      data
/*
//FT07F001 DD SYSOUT=B
```

VS FORTRAN Separation Tool (for Both VM and MVS)

The separation tool is a VS FORTRAN library utility to aid in separating the nonreentrant portion of the compiler object output from the reentrant portion if you have compiled with the RENT compiler option.

If you are running on VM, see "Using the VS FORTRAN Separation Tool" on page 235 or, if you are running on MVS, see "Using the VS FORTRAN Separation Tool" on page 290 for more information.

Fixing Execution-Time Errors

Some errors are detected by the compiler and an error message is issued following compilation. Others, however, will not be apparent until you actually run your program. As you test your program, there are several ways in which you might become aware of a possible problem; for example:

Prolonged execution time (loops and waits)
If your program appears to be taking an unusually long time to complete its execution, it may be caught in a loop or wait. Sometimes repetitious output (or the absence of expected output) will provide an early clue. A loop in this sense is a series of instructions continuously repeated because erroneous circular program logic allows for no exit.

Incorrect execution results
Your program may run to its conclusion without interruption, but examination of its output may show that something is wrong. Even though you didn't get any error messages, the program is not producing the expected results.

Premature termination of your program
Either the VS FORTRAN run-time library or the operating system may detect an error condition while your program is running. In either case, you will always get an error message from the VS FORTRAN library, identified by the prefix IFY, and you may get messages from the operating system also.

VS FORTRAN has a number of features that help you find errors. One feature, VS FORTRAN Interactive Debug, is described in Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377 , and in *VS FORTRAN Interactive Debug Guide and Reference*. Other debugging aids described in the following sections are:

- Execution-time messages

- Extended error handling
- Using the execution-time options
- Static debug statements
- Object module listing
- Formatted dumps

Execution-Time Messages

Execution-time messages are issued by the execution-time library. There are three types of messages issued:

Library Diagnostic Messages—contain information about errors occurring when the library subroutines are executed—for example, input/output or mathematical subroutine errors.

Program Interrupt Messages—contain information about errors that occur when system rules are violated.

Operator Messages—communicate with the operator when program execution makes it necessary (for example, when a PAUSE statement is executed).

Library Diagnostic Messages

All VS FORTRAN library diagnostic messages are documented in *VS FORTRAN Language and Library Reference*, and are in the following format:

```
IFYnnnI mmmmm 'message text'
```

where:

IFY	is the message prefix identifying all VS FORTRAN library messages.
nnn	is the unique number identifying this message.
I	is the character that represents an informational message.
mmmmm	are the five rightmost characters of the library module name that originated the message.
'message-text'	explains the execution-time condition that was detected.

Except for operator and informational messages, all VS FORTRAN Library messages are followed by additional information that identifies the name of the last-executed FORTRAN program and the location of the last-executed statement in that program unit. The additional information is indicated in one of three formats based on how the FORTRAN program unit was compiled:

- Program unit compiled with NOSDUMP and NOTEST:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name
  (OFFSET oooooooo).
  ```

- Program unit compiled with TEST and NOSDUMP:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name AT
  SEQ. NO. ssssss (OFFSET oooooooo).
  ```

- Program unit compiled with SDUMP or, for some errors, GOSTMT:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name AT
  ISN iiiiii (OFFSEST oooooooo).
  ```

where:

name	is the name of the failing program unit (reentrant CSECT name if compiled with RENT)
oooooooo	is the hexadecimal offset from the beginning of the program to the last-executed statement.
ssssss	is the sequence number of source statement in failing program unit.
iiiii	is the compiler-generated internal sequence number (ISN).

This additional information is invaluable in determining the source of the error. It should be noted, however, that if the last executed FORTRAN program unit called an assembler routine which invoked the VS FORTRAN Library routine that caused the error, the source of the error may be the user-coded assembler routine.

The additional information identifying the source of the error is not produced if no VS FORTRAN program units are encountered in the active chain of program units that caused issuance of the error message. For further information, see "Extended Error Handling" on page 197.

Using the Optional Traceback Map

Whenever you get a library diagnostic message, you can, optionally, get a traceback map. Your organization 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 CALL ERRSET subroutine.

You can also get a traceback map at any point in your source program by using the CALL ERRTRA subroutine.

For more information on these subroutines, see "Controlled Extended Error Handling—CALL Statements" on page 198.

To cause ISNs to appear in a traceback map, you must have compiled with the GOSTMT or the SDUMP compiler option (see "Using the Compiler Options" on page 157).

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 executed, and ends with the origin of the program.

The sample traceback map in Figure 36 lists the names of called routines, internal sequence 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 36. 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 36, the final routine that executed 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 routine or program that called this routine. The starting address of the calling routine follows the routine name. In Figure 36, 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 sequence number (ISN) of the calling statement in the CALLED BY routine. ISN information is only available if a program unit was compiled as explained under "Library Diagnostic Messages" on page 190.

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

  ```
  'cccc'
  ```

- hexadecimal

  ```
  00000000
  ```

The control program executes 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, execution continues.

The summary of errors printed at the end of the listing can help you determine how many times an error was encountered. If your source program contains many input/output statements, locating an error can become a formidable task. By pinpointing the exact FORTRAN statement involved, the traceback map makes it much easier for you to locate execution errors.

Traceback Map Procedure: To use the traceback map for error detection:

1. Look at the message text in the first line of the IFY message. This text will give you a clue to the type of error that caused the problem.

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 sequence number (ISN) calling each routine. For an example using ISNs, see Figure 31 on page 171. Using the ISN, you can locate the source statement within the calling module.

Program Interrupt Messages

During program execution, messages are generated whenever the program is interrupted because of the following:

- Operation exception
- Privileged operation exception
- Execute exception
- Protection exception
- Addressing exception
- Specification exception
- Data exception
- Fixed-point-overflow exception
- Fixed-point-divide exception
- Decimal-overflow exception
- Decimal-divide exception
- Exponent-overflow exception
- Exponent-underflow exception
- Significance exception
- Floating-point-divide exception

Program interrupt messages are written in the output data set. They guide you in determining the cause of the error, indicating what system rule was violated.

The standard corrective action for each type of interrupt is described in *VS FORTRAN Language and Library Reference.*

The program interrupt message from the VS FORTRAN library indicates the true exception that caused the termination; however, the completion code from the system always indicates that job termination is due to a specification or operation exception.

When a program interrupt occurs in a program unit that was compiled with the SDUMP or TEST option, symbolic dumps of program data are automatically provided for your use in determining the cause of the interrupt. For more information on symbolic dumps, see "Requesting Symbolic Dumps—CALL Statement" on page 211.

Exception codes themselves appear in the eighth digit of the PSW and indicate the reason for the interruption. Their meanings are as follows:

Code	**Meaning**
1	**Operation exception**, that is, the operation is not one that is defined to the operating system.
2	**Privileged-operation exception**, that is, the processor encounters a privileged instruction in the problem state.
3	**Execute exception**, that is, the subject instruction of EXECUTE is another EXECUTE.
4	**Protection exception**, that is, an illegal reference is made to an area of storage protected by a key.
5	**Addressing exception**, that is, a reference is made to a storage location outside the range of storage available to the job.
6	**Specification exception**, for example, a unit of information does not begin on its proper boundary.
7	**Data exception**, that is, the arithmetic sign or the digits in a number are incorrect for the operation being performed.
8	**Fixed-point-overflow exception**, that is, a carry occurs out of the high-order bit position in fixed-point arithmetic operations, or high-order significant bits are lost during the algebraic left-shift operations.
9	**Fixed-point-divide exception**, that is, an attempt is made to divide by zero.
A	**Decimal-overflow exception**, that is, one or more significant high-order digits are lost because the destination field in a decimal operation is too small to contain the result.
B	**Decimal-divide exception**, that is, when in decimal division the divisor is zero or the quotient exceeds the specified data field size.
C	**Exponent-overflow exception**, that is, a floating-point arithmetic operation produces a positive number mathematically too large to be contained in a register (the mathematically largest number that can be contained is 16^{63} or approximately 7.2 x 10^{75}). Exponent-overflow generates the additional message:

```
REGISTER CONTAINED number
```

where number is the floating-point number in hexadecimal format. (When extended-precision is involved, the message prints out the contents of two registers.) A standard corrective action is taken and execution continues.

D **Exponent-underflow exception**, that is, a floating-point arithmetic operation generates a number with a negative exponent mathematically too small to be contained in a register (mathematically smaller than 16^{-65} or approximately 5.4×10^{-79}). Exponent-underflow also generates the message:

```
REGISTER CONTAINED number
```

where number is the number generated. (When extended-precision is involved, the message prints out the contents of two registers.) A standard corrective action is taken and execution continues.

E **Significance exception**, that is, the result fraction in floating-point addition or subtraction is zero.

F **Floating-point-divide exception**, that is, an attempt is being made to divide by zero in a floating-point operation. Floating-point divide also generates the message:

```
REGISTER CONTAINED number
```

(When extended-precision is in use, the message prints out the contents of two registers.) A standard corrective action is taken and execution continues.

Notes:

1. *Operation, protection, addressing, and data exceptions (codes 1, 4, 5, and 7) ordinarily cause abnormal termination without any corresponding message.*

2. *Protection and addressing exceptions (codes 4 and 5) generate a message only if a specification exception (code 6), or an operation exception (code 1), has also been detected.*

3. *A data exception (code 7) generates a message only if a specification exception has also been detected. When a message is generated for code 4, 5, or 7, the job will terminate.*

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 290 (MVS), or "Requesting an Abnormal Termination Dump" on page 348 (VSE).

Information on interpreting dumps is found in the appropriate debugging guide, as listed under "Preface" on page iii.

Operator Messages

Operator messages are generated when your program executes a PAUSE or STOP n statement. Operator messages are written on the system device specified for operator communication, usually the console. The message can guide you in determining how far your FORTRAN program has executed.

The operator message may take the following form:

```
yy n | 'message'
```

Character	**Meaning**
yy	message identification number assigned by the system.
n	string of 1 through 5 decimal digits you specified in the PAUSE or STOP statement. For the STOP statement, this number is placed in register 15.
'message'	character constant you specified in the PAUSE or STOP statement.
0	printed when a PAUSE statement containing no characters is executed. (Nothing is printed for a similar STOP statement.)

A PAUSE message causes program execution to halt pending operator response. 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 be either by default or you can control it, using predefined CALL statements.

Extended Error Handling By Default

Your installation has a default value preset in the error option table for the following execution-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 execution-time message is printed.
- Whether or not a traceback map is to be printed with the message.
- Whether or not a user (or installation) 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 organization.

Notes:

1. *VS FORTRAN Interactive Debug causes the run-time library to ignore the above settings, to specify its own error exit routine, to allow unlimited occurrences of errors and messages, and to specify that no traceback map is to accompany any of the messages.*

2. *If a program is being debugged with VS FORTRAN Interactive Debug and it calls ERRSET to set a user exit routine, the program's user exit routine overrides the Interactive Debug error exit routine.*

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 execution-time messages, see "Execution-Time Messages" on page 190.

   - 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 execution.

   - 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 execution.

Controlled Extended Error Handling—CALL Statements

To make changes to the option table dynamically at load module execution time, you can use the predefined CALL subroutines, summarized here.

For each error condition detected, you have dynamic control over:

- The number of times the error is allowed to occur before program termination

- The maximum number of times each message may be printed

- Whether or not the traceback map is to be printed with the message

- Whether or not a user-written, error exit routine is called

The action that takes place is governed by information stored in your copy of the error option table, which is present with your program as it executes. (The permanent copy of the default option table resides in the VS FORTRAN library.)

The predefined CALL routines 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 during execution of your program:

- **CALL ERRMON**—executes the error monitor.
- **CALL ERRTRA**—executes the traceback routines. (See "Using the Optional Traceback Map" on page 191.)
- **CALL ERRSAV**—copies an option table entry into an area accessible to your program.
- **CALL ERRSTR**—stores an entry into the option table from your program.
- **CALL ERRSET**—changes up to 5 values associated with an entry in the option table, including a user-exit address.

For detailed reference documentation about the error option table and the predefined CALL routines, see *VS FORTRAN 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 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 will not be known to the optimizer, since the user error handling routine will be 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) 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 will not be changed between the READ and the GOTO 1 in the above example. It is kept in a register which causes an incorrect result. If you specify IOSTAT in the READ statement, as follows:

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

the optimizer will be able to optimize it correctly.

5. Under VSE at link-edit time, you must INCLUDE the library modules for the predefined user error handling routines, if your program calls them.

Effects of VS FORTRAN Interactive Debug on Error Handling

If you are executing with VS FORTRAN Interactive Debug, error handling is modified as follows:

- Unlimited error occurrences are allowed for all errors and error counts are not maintained.
- 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.

Using the Execution-Time Options

The following execution-time options are available: DEBUG, NODEBUG, XUFLOW, and NOXUFLOW.

DEBUG and NODEBUG Execution-Time Options

If you want to use VS FORTRAN Interactive Debug (5668-903), you do so by means of the execution-time option DEBUG. This option can be specified by CMS and TSO users at execution time.

The options are:

DEBUG	makes VS FORTRAN Interactive Debug available.
NODEBUG	does not access VS FORTRAN Interactive Debug. This is the default.

For more information, see Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377.

XUFLOW and NOXUFLOW Execution-Time Options

If you want to specify to VS FORTRAN that an exponent underflow is not to cause a program interrupt, you can do so with the execution-time option NOXUFLOW. There are two options which control this:

XUFLOW	allows an exponent underflow to cause a program interrupt, followed by a message from VS FORTRAN, followed by standard fixup. This is the default.
NOXUFLOW	suppresses the program interrupt caused by an exponent underflow. The hardware provides the fixup.

For more information, see "Specifying Execution-Time Options under CMS" on page 250, "Specifying Execution-Time Options" on page 288 , "Specifying Execution-Time Options" on page 327, or "Specifying Execution-Time Options" on page 344.

IBM Extension

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), or OPTIMIZE(3), the compiler changes the optimization parameter to NOOPTIMIZE.

Figure 37 on page 202 shows how you can use VS FORTRAN 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 executed)
     .
     .
 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 number 10.

(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 number 40.

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 37. Using Static Debug Statements

In debug packets you can use the following statements:

DEBUG
specifies the debug options you want performed, which can be:

- Check the validity of array subscripts.
- Trace the order of execution of all or part of the program.
- Display array or variable values each time they change during program execution.

AT
specifies the statement number before which the debug packet is to be executed.

The AT statement begins each new debug packet in the program and ends the previous one.

An IF block must be contained within a single AT packet. If it is not contained within a single AT packet, it is not diagnosed, and unpredictable results can occur.

TRACE ON | TRACE OFF
begin or end program execution tracing.

DISPLAY
writes a list of variable or array values that you specify.

The output from the DISPLAY statement is as follows:

The first line written is the name of the NAMELIST created by the compiler for the DISPLAY statement preceded by the ampersand (&) character.

```
&NM.Lnn
```

where nn is the 2-digit decimal value assigned to the DISPLAY statement. This value begins at 00 for the first DISPLAY statement in the source program and increases by 1 for each subsequent DISPLAY statement in the order found in the DISPLAY statements. As many as 100 displays are valid for any routine. If there are more than 100 DISPLAY statements, only the last 100 valid displays are processed.

The name is followed by the DISPLAY list, in NAMELIST format. The output is terminated with the line:

```
&END
```

END DEBUG
ends the last debug packet specified.

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 during execution.

End of IBM Extension

Object Module Listing—LIST Option

You request an object module listing by specifying the LIST option. The object module listing shows you (in pseudo-assembler format) the machine code the compiler generated from your source statements. A careful examination of this can often give you an idea of what's wrong with your program source.

The object module listing is especially useful when you're compiling and executing using one of the OPTIMIZE compiler options. Further details are contained in Chapter 7, "Optimizing Your Program" on page 139.

A sample object module listing is shown in Figure 38 on page 206. (Some of the information in the listing has been realigned in order to fit the dimensions of the page.)

If the SDUMP or TEST option has been specified, in addition to the LIST option, ISN numbers are printed on the left-hand side of the pseudo-assembler listing.

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

- The letters ISN followed by the internal sequence number. This shows the relationship between the FORTRAN statement and the machine code which is generated.
- A 6- or 8-hexadecimal digit shows the relative address of the instruction or data item.
- 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.

The object module listing, when the reentrant feature has not been invoked, contains the following sections:

1. Entry code
2. Entry table
3. Program information block

4. Compiler properties table

5. Format statements

6. IAD work area

7. Save area

8. Register 12 address constant

9. Temporary storage for FIX/FLOAT or NOT

10. Variables in common areas

11. Constants

12. Arithmetic and logical variables

13. Character variables

14. Address constants for assigned FORMAT statements, common areas, and external references

15. Program code

16. Prologue code

17. Address constants for the prologue, the save area, branch tables, and parameter lists

18. Temporary storage areas and generated constants

19. Address constants for block labels

20. Program code table

21. Symbol dictionary

22. Program information block

23. Entry table list

24. Compiler properties table

Figure 38 on page 206 shows each of the numbered items above.

The object module listing with the reentrant feature contains the same sections as those shown in Figure 38 on page 206; in addition, however, one table can appear in the reentrant listing that does not for nonreentrant: ADCONS (address constants) FOR REENTRANT RELOCATION.

```
ENTRY CODE
   000000  47F0 F020           MAIN  BC     15,32(0,15)
   000004  17                        DC     XL1'17'
   000005  D4C1C9D54040404040        DC     CL9'MAIN     '
   00000E  F8F44BF1F0F9              DC     CL6'84.109'
   000014  F0F84BF2F94BF3F6          DC     CL8'08.29.36'
   00001C                            DS     A(PIB)
   000020  90EC D00C                 STM    14,12,12(13)
   000024  9823 F034                 LM     2,3,52(15)
   000028  5030 D008                 ST     3,8(,13)
   00002C  50D0 3004                 ST     13,4(,3)
   000030  07F2                      BR     2
   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

FORMAT STATEMENTS
   0000C8  021A0CF1D6E4E3D7     111  DC     CL8'   1OUTP'
   0000D0  E4E340C6D6D94030          DC     CL8'UT FOR  '
   0000D8  000E060E0A0A070C          DC     CL8'        '
   0000E0  0F0704020A14101C          DC     CL8'        '
   0000E8  22                        DC     CL1' '

IAD WORK AREA
   0000EC                            DS     4F

SAVE AREA
   0000FC                            DS     18F

REGISTER 12 ADCON
   000144                            DS     A(REG12)

IAD WORK AREA
   000148                            DS     25F

TEMPORARY FOR FIX/FLOAT OR NOT
   0001B0  0000000000000000          DC     XL8'0000000000000000'
   0001B8  4E00000000000000          DC     XL8'4E00000000000000'

VARIABLES IN 'COM1  ' COMMON.
   000000  NO INITIAL DATA     R4A   DS     11F           (ARRAY)
   000004  NO INITIAL DATA     R8A   DS     7D            (ARRAY)
   00000C  NO INITIAL DATA     C32   DS     80D           (ARRAY)
   00002C  NO INITIAL DATA     C8A   DS     F             (REAL)
   000030  NO INITIAL DATA           DS     F             (IMAG)
```

Figure 38 (Part 1 of 4). Object Module Listing Example—LIST Compiler Option

```
VARIABLES IN 'COM2  ' COMMON.
   FFFFF8  NO INITIAL DATA     I2     DS    3F             (ARRAY)
   000000  NO INITIAL DATA     L1     DS    X
   000001  NO INITIAL DATA     C8B    DS    F              (REAL)
   000005  NO INITIAL DATA            DS    F              (IMAG)

CONSTANTS
   0001C8  4F08000000000000           DC    XL8'4F08000000000000'
   0001D0  4E00000080000000           DC    XL8'4E00000080000000'
   0001D8  413243F6A791A9E1    PI     DC    XL8'413243F6A791A9E1'
   0001E0  4130000000000000           DC    XL8'4130000000000000'
   .
   .
   .

ARITHMETIC AND LOGICAL VARIABLES
   000250  NO INITIAL DATA     R8V    DS    D
   000258  NO INITIAL DATA     I      DS    F
   00025C  NO INITIAL DATA     J      DS    F
   000260  NO INITIAL DATA     A1     DS    F
   000264  413243F4            A2     DC    XL4'413243F4'
   000268  NO INITIAL DATA     A3     DS    F

CHARACTER VARIABLES
   0002D4  E2C1D4D7D3C540D7   CHAR15 DC    CL8'SAMPLE P'
   0002DC  D9D6C7D9C1D440             DC    CL7'ROGRAM '

ADCONS FOR ASSIGNED FORMAT STATEMENTS
   0002E4  000000C8                   DC    AL4(X'000000C8')

ADCONS FOR COMMONS
   0002E8  FFFFFF6C                   DC    AL4(X'FFFFFF6C') COM1
   0002EC  FFFFFFF4                   DC    AL4(X'FFFFFFF4') COM2

ADCONS FOR EXTERNAL REFERENCES
   0002F0  00000000                   DC    AL4(00000000)    CXSUB   (SUBR)
   0002F4  00000000                   DC    AL4(00000000)    VFEE#   (SUBR)
   0002F8  00000000                   DC    AL4(00000000)    VFEP#   (SUBR)
   0002FC  00000000                   DC    AL4(00000000)    VFEIM#  (SUBR)
   000300  00000000                   DC    AL4(00000000)    VFFXF#  (SUBR)
   000304  00000000                   DC    AL4(00000000)    VFIXF#  (SUBR)
   000308  00000000                   DC    AL4(00000000)    VFWSF#  (SUBR)
   00030C  00000000                   DC    AL4(00000000)    VSERH#  (SUBR)
```

Figure 38 (Part 2 of 4). Object Module Listing Example—LIST Compiler Option

```
PROGRAM CODE
   ISN   16  000358   5800 D1E8   01 000 00001  L     0,488(0,13)    111
             00035C   5000 D160                 ST    0,352(0,13)    J
   ISN   17  000360   5870 D1EC                 L     7,492(0,13)
             000364   6800 70A0                 LD    0,160(0,7)     R8A
             000368   6000 D214                 STD   0,532(0,13)    .S00
             00036C   6800 D0CC                 LD    0,204(0,13)
             000370   7800 D16C                 LE    0,364(0,13)    A3
             000374   6000 D21C                 STD   0,540(0,13)    .S02
             .
             .
             .

PROLOGUE CODE
   0004C4   1B11                      SR    1,1
   0004C6   18D3                      LR    13,3
   0004C8   58F0 3200                 L     15,512(0,3)      VFEIM#
   0004CC   05EF                      BALR  14,15
   0004CE   58F0 D234                 L     15,564(0,13)
   0004D2   07FF                      BCR   15,15

ADCON FOR PROLOGUE
   000034   000004C4                  DC    XL4'000004C4'

ADCON FOR SAVE AREA
   000038   000000FC                  DC    XL4'000000FC'

ADCONS FOR BRANCH TABLES
   0001C0   0000048C                  DC    XL4'0000048C'
   0001C4   000004B2                  DC    XL4'000004B2'

ADCONS FOR PARAMETER LISTS
   00026C   00000250                  DC    AL4(X'00000250') R8V
   000270   00000260                  DC    AL4(X'00000260') A1
   000274   800001D8                  DC    AL4(X'800001D8') 413243F6A791A9E1
   000278   80000228                  DC    AL4(X'80000228') 25
   .
   .
   .

TEMPORARIES AND GENERATED CONSTANTS
   000310   00000000                  DC    XL4'00000000'
   000314   00000000                  DC    XL4'00000000'
   000318   00000000                  DC    XL4'00000000'
   .
   .
   .
```

Figure 38 (Part 3 of 4). Object Module Listing Example—LIST Compiler Option

```
ADCONS FOR B BLOCK LABELS
   000330  00000358                  DC     XL4'00000358'
   000334  0000039E                  DC     XL4'0000039E'
   000338  000003AC                  DC     XL4'000003AC'
   .
   .
   .

PROGRAM CODE TABLE
   0004D4  0F                        DC     XL1'0F'
   0004D5  B4                        DC     XL1'B4'
   0004D6  C018                      DC     XL2'C018'
   0004D8  B7                        DC     XL1'B7'
   0004D9  B7                        DC     XL1'B7'
   .
   .
   .

SYMBOL DICTIONARY
   0004F0  40407BE2E8D4E5E2          DC     CL8'  #SYMVS'
   0004F8  40404040404040C9          DC     CL8'       I'
   000500  00000258                  DC     XL4'00000258'
   000504  80050004                  DC     XL4'80050004'
   .
   .
   .

PROGRAM INFORMATION BLOCK
   000060  7BD7C9C2E5F0F17B          DC     CL8'#PIBTAB#'
   000068  0060                      DC     XL2'0060'
   00006A  8000                      DC     XL2'8000'
   .
   .
   .

ENTRY TABLE LIST
   00003C  00000000                  DC     XL4'00000000'
   000040  D4C1C9D540404040          DC     CL8'MAIN    '
   000048  000004C4                  DC     XL4'000004C4'
   .
   .
   .

COMPILER PROPERTIES TABLE
   0000C0  1000000000000000          DC     XL8'1000000000000000'
```

Figure 38 (Part 4 of 4). Object Module Listing Example—LIST Compiler Option

Formatted Dumps

You can request various dumps during program execution using the VS FORTRAN dump subprograms: PDUMP, DUMP, CPDUMP, CDUMP, and SDUMP.

Under VSE only, when you're requesting one of these dumps, you must also specify its library name with a linkage editor INCLUDE statement. Figure 63 on page 348 contains the names to specify.

Requesting Selective Dumps—CALL Statement

Four VS FORTRAN predefined CALL routines let you request a selective dump of storage during program execution:

- **CALL PDUMP**—dumps the requested areas and allows processing to continue
- **CALL DUMP**—dumps the requested areas and terminates processing
- **CALL CPDUMP**—dumps the requested character storage areas and allows processing to continue
- **CALL CDUMP**—dumps the requested character storage areas and terminates processing

When you use the DUMP and PDUMP subprograms, you specify as parameters:

- The variables delimiting the area to be dumped
- A code specifying the format in which the items are to be dumped

For example, if you wanted to dump one item and continue processing, you could specify:

```
CALL PDUMP (A,A,5)
```

which would dump the variable A in real format (the code 5 specifies real format).

You can also dump an entire range of items in storage:

```
CALL DUMP (A,M,0)
```

which would dump every item in storage, beginning with variable A and continuing through variable M, in hexadecimal format (the code 0 specifies hexadecimal format). Execution would then be terminated.

When you're using CDUMP and CPDUMP, the output is always in character format. Therefore, you specify only the delimiting variables in the CALL statement. For example, to dump a range of character variables from character variable C1 to character variable C19, specify:

```
CALL CDUMP (C1,C19)     (and execution terminates)
```

or

```
CALL CPDUMP (C1,C19)    (and execution continues)
```

For reference documentation about these routines, see *VS FORTRAN Language and Library Reference*.

Requesting Symbolic Dumps—CALL Statement

A VS FORTRAN predefined CALL routine provides you with a symbolic dump that is displayed in a format dictated by variable type. Variable and array values are dumped on the error message unit. Variable and array values are dumped automatically upon abnormal termination, or are dumped by program request, on a program unit basis, using CALL SDUMP. Items displayed are:

- All referenced, local, named variables and arrays in their FORTRAN-defined data representation
- All variable and array values contained in a blank common, named common, or a dynamic common area in their FORTRAN-defined data representation
- Nonzero or nonblank character array elements only
- Array elements with their correct indexes

Symbolic Dumps at Abend Time: If a task ends with a nonrecoverable failure (storage access, for example) in a FORTRAN program unit, all variable and array values in that program unit are automatically dumped if it was compiled with either SDUMP or TEST. Additionally, all variable and array values in any FORTRAN program unit in the save area traceback chain compiled with either SDUMP or TEST are also dumped. Variable and array values occurring in common areas are dumped at each occurrence because the variable and array definitions in each program unit may be different.

If the program unit was not compiled with either SDUMP or TEST, it is bypassed and processing continues with the next program unit.

The dump output follows the IFY240I messages and the called program's traceback chain messages.

Note: If the object error unit is directed to a terminal, the post-abend dump is skipped. The object error unit must be directed to a file.

Symbolic Dump Upon Request: Except for user fix-up routines, which are for I/O failures, program-requested dumping of variable and array values can be done by calling the SDUMP utility program from any program unit.

Note: User fix-up routines cannot be dumped because you cannot initiate a new I/O operation while one is in progress.

- A call to the SDUMP utility program from within a FORTRAN program (either a main program, a subroutine subprogram, or a function subprogram) compiled with either SDUMP or TEST will only dump variable and array values from within that program by specifying the call without parameters as follows:

  ```
  CALL SDUMP
  ```

- Variable and array values can be dumped from one or more FORTRAN programs, that have had either SDUMP or TEST specified at compile time, by means of a call to SDUMP and specifying the entry name of the routines in a parameter list.

The syntax for the call is as follows:

```
CALL SDUMP [ (subroutine1,subroutine2,...) ]
```

where *subroutine1,subroutine2,...* are names of other FORTRAN program units and may be omitted when the default is exercised—that is, if only dumping variable and array values from the calling routine. The names must also be specified in a FORTRAN EXTERNAL statement in the calling program (excluding the calling program name).

The following invocation requests symbolic dumps of variables in *subroutine1* and *subroutine2*:

```
EXTERNAL subroutine1,subroutine2
         .
         .
         .
CALL SDUMP(subroutine1,subroutine2)
```

This invocation requests a symbolic dump of variable and array values in the calling program only:

```
CALL SDUMP
```

For reference documentation about this routine, see *VS FORTRAN Language and Library Reference.*

Programming Considerations for SDUMP

- Compilation must be done with either SDUMP or TEST in order to gain symbolic dump information and location of error information.
- SDUMP for routines not entered has unpredictable results.
- SDUMP for the routine in which the CALL statement is located is done without parameters:

  ```
  CALL SDUMP
  ```

- An EXTERNAL statement must be used to identify the names being passed to SDUMP as external routine names and not local variables.
- The user must not have a routine with the name SDUMP.
- At higher levels of optimization (1-3), some variables may not have their true value because of compiler optimization techniques.

Chapter 10. Sample Programs and Subroutines

Sample Program 1

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

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

Sample Program 2

The following sample program demonstrates use of various I/O statements.

```
@PROCESS LIST
      PROGRAM VSF046
C
C  DECLARATIONS OF VARIABLES
C
      IMPLICIT COMPLEX (C), COMPLEX*16 (Y), COMPLEX*32 (Z),
     1         REAL    (R), REAL*8     (D), REAL*16    (X),
     2         INTEGER (I), INTEGER*2  (H),
     3         LOGICAL (L), LOGICAL*1  (B)
      CHARACTER*32 EXPECT(9),ACTUAL(9)
      CHARACTER*29 SEVEN7
      CHARACTER*27 EIGHT8
      CHARACTER*25 FIFTY5
      CHARACTER*5  KNOWN
      CHARACTER*2  UN
      CHARACTER*9  FORMAT
      CHARACTER*10 SEQUEN
      CHARACTER*12 DDNAME,NEW,DIRECT,ZERO,DELETE
      CHARACTER*12 FN,SEQ,DIR,FMT,UNF,ACC,FM,BLNK
      DIMENSION CACT(9),YACT(9),ZACT(9),LACT(9),BACT(9),
     1          RACT(9),DACT(9),XACT(9),IACT(9),HACT(9)
      DIMENSION CEXP(9),YEXP(9),ZEXP(9),LEXP(9),BEXP(9),
     1          REXP(9),DEXP(9),XEXP(9),IEXP(9),HEXP(9)
      NAMELIST /NAMEIO/CACTL,RACTL,ACTUAL,IACTL,LSAME
   55 FORMAT (' ERROR EXIT FROM SEGMENT',I5)
   66 FORMAT (' TEST CASE FAILED IN SEGMENT',I5,
     1    :,/,' EXPECTED VALUE: ',Z64,
     2    :,/,' COMPUTED VALUE: ',Z64)
   77 FORMAT (' TEST CASE VSFOR046 COMPLETED')
   88 FORMAT (' TEST CASE VSFOR046 STARTED')
      IN = 5
      IO = 6
      FIFTY5 = ' ERROR EXIT FROM SEGMENT '
      SEVEN7 = ' TEST CASE VSFOR046 COMPLETED'
      EIGHT8 = ' TEST CASE VSFOR046 STARTED'
      PRINT *, EIGHT8
      I10 = 10
      DDNAME = 'DDNAME'
      NEW = 'NEW   '
      KNOWN = 'KNOWN'
      UN = 'UN'
      FORMAT = 'FORMATTED'
      SEQUEN = 'SEQUENTIAL'
      DIRECT = 'DIRECT'
      ZERO = 'ZERO'
      DELETE = 'DELETE'
```

```
C
C  WRITE STATEMNT.
C
   99 N = 0
 3000 N = 30
      ISEG = 30
      ASSIGN 3200 TO ILAB
      WRITE (IOSTAT=IOS,ERR=99999,FMT='(A9)',UNIT=2*IN-10) '1 2  3   '
      IF( IOS .NE. 0 ) WRITE (IO,66) 31,0,IOS
C
C  ENDFILE STATEMENT.
C
 3200 ISEG = 32
      ASSIGN 3400 TO ILAB
      ENDFILE (IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
      IF( IOS .NE. 0 ) WRITE (IO,66) 33,0,IOS
C
C  REWIND STATEMENT.
C
 3400 ISEG = 34
      ASSIGN 4000 TO ILAB
      REWIND (IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
      IF( IOS .NE. 0 ) WRITE (IO,66) 35,0,IOS
C
C  READ STATEMENT.
C
      IACT(8) = 99
      READ (IOSTAT=IOS,UNIT=2*IN-10,FMT='(I9)',END=3600) (IACT(J),J=7,8)
 3600 IF( IOS .GE. 0 ) WRITE (IO,66) 36,0,IOS
      IEXP(7) = 102003000
      IEXP(8) = 99
 3700 DO 3890 ISEG=37,38
      IF( IACT(ISEG-N) .NE. IEXP(ISEG-N) )
     1      WRITE (IO,66) ISEG,IEXP(ISEG-N),IACT(ISEG-N)
 3890 CONTINUE
C
C  BACKSPACE, ENDFILE AND SEQUENTIAL OPEN STATEMENTS.
C
 4000 ISEG = 40
      BACKSPACE (IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
      IF( IOS .NE. 0 ) WRITE (IO,66) 41,0,IOS
      WRITE (IN+IN-10,'(A9)',IOSTAT=IACT(1)) '4 5  6   '
      IF( IACT(1) .NE. 0 ) WRITE (IO,66) 43,0,IACT(1)
      ENDFILE (IN+IN-10,IOSTAT=IACT(1))
      IF( IACT(1) .NE. 0 ) WRITE (IO,66) 45,0,IACT(1)
      OPEN (UNIT=2*IN-10,FILE='DDNAMX',STATUS=NEW,
     1     ACCESS='SEQU'//'ENTIAL  ',FORM=FORMAT,BLANK=ZERO)
      WRITE (0,'(A8)') UN//SEQUEN(IN:2*IN)
C
C  OPEN, REWIND AND CLOSE STATEMENTS.
C
 4600 ISEG = 46
      ENDFILE (0,ERR=99999)
      OPEN (UNIT=2*IN-10,FILE=DDNAME,STATUS='OLD',
     1     ACCESS='SEQU'//'ENTIAL  ',FORM=FORMAT,BLANK=ZERO)
      REWIND (IN+IN-10,IOSTAT=IACT(1))
      IF( IACT(1) .NE. 0 ) WRITE (IO,66) 47,0,IACT(1)
 4900 N = 50
      ISEG = 49
      CLOSE (IOSTAT=IACT(1),ERR=99999,STATUS='KE'//'EP    ',UNIT=0)
C
C  INQUIRE STATEMENT.
C
 5000 IF( IACT(1) .NE. 0 ) WRITE (IO,66) 50,0,IACT(1)
      INQUIRE (FILE='DD'//'NAME ',IOSTAT=IACT(1),
     1        EXIST=LACT(9),OPENED=LACT(8),NAMED=LACT(7),
     2        NAME=ACTUAL(1),SEQUENTIAL=ACTUAL(2),DIRECT=ACTUAL(3),
     3        FORMATTED=ACTUAL(4),UNFORMATTED=ACTUAL(5),
     4        ACCESS=ACTUAL(6),FORM=ACTUAL(7),NUMBER=IACT(2),
     5        RECL=IACT(3),NEXTREC=IACT(4),BLANK=ACTUAL(8))
 5600 IF( IACT(1) .NE. 0 ) WRITE (IO,66) 56,0,IACT(1)
      EXPECT(1) = 'DDNAME'
      EXPECT(2) = 'YES'
      EXPECT(3) = 'NO'
      EXPECT(4) = 'YES'
      EXPECT(5) = 'NO'
      LEXP(7) = .TRUE.
      LEXP(8) = .FALSE.
      LEXP(9) = .TRUE.
 5100 DO 5590 ISEG=51,55
      IF( ACTUAL(ISEG-N) .NE. EXPECT(ISEG-N) )
     1      WRITE (IO,66) ISEG,EXPECT(ISEG-N),ACTUAL(ISEG-N)
 5590 CONTINUE
 5700 DO 5990 ISEG=57,59
      IF( LACT(ISEG-N) .NEQV. LEXP(ISEG-N) )
     1      WRITE (IO,66) ISEG,LEXP(ISEG-N),LACT(ISEG-N)
 5990 CONTINUE
88888 WRITE (*,*) SEVEN7
      STOP
99999 WRITE (FMT=*,UNIT=*) FIFTY5,ISEG
      GOTO ILAB
      END
```

Sample Subroutines

The following sample subroutines demonstrate coding schemes for various VS FORTRAN functions discussed in previous chapters.

```
@PROCESS CI(10)
      SUBROUTINE INCLUD
C
C   THIS SUBROUTINE DEMONSTRATES CONDITIONAL INCLUDE.
C
      INCLUDE (J) 10
      INCLUDE (J) 11
      RETURN
      END

      SUBROUTINE DECL
C
C   THIS SUBROUTINE DEMONSTRATES DECLARING DATA.
C
      IMPLICIT DOUBLE PRECISION(A-C, F),
     1         LOGICAL (E,L),CHARACTER(D,G,H)
C
      DOUBLE PRECISION MEDNUM
      CHARACTER*80    INREC
      INTEGER*2       COUNTR
      REAL*16         BIGNUM,ARRAY2*4(5,5)
C
      CHARACTER*15 CAT
C
      RETURN
      END

      SUBROUTINE DEFVAL
C
C   THIS SUBROUTINE DEMONSTRATES DEFINING CONSTANTS BY VALUE.
C
      LOGICAL         COMP
C
      CHARACTER*15 CHAR15 /'PARAMETER = '/
      CHARACTER*18 CHAR18 /'THE ANSWER IS:'/
      CHARACTER*40 CHAR40 /'''TWAS BRILLIG AND THE SLITHY TOVES'/
C
100   FORMAT(I3,11H = THE NORM)
200   FORMAT(2D8.6, 18H ARE THE 2 ANSWERS)
C
      REAL* 4 TEMP
      DATA TEMP/ZC1C2C3C4/
C
      RAD=10.
      PI=3.1415
      CIRC = 2*PI*RAD

C
      COMP = .FALSE.
      RETURN
      END
      SUBROUTINE DEFNAM
C
C   THIS SUBROUTINE DEMONSTRATES DEFINING CONSTANTS
C     BY NAME.
C
C
      CHARACTER*5 C1,C2
      PARAMETER (C1='DATE ',C2='TIME ',RATE=2*1.414)
      RETURN
      END
      SUBROUTINE ONEDIM
C
C   THIS SUBROUTINE DEMONSTRATES ONE-DIMENSIONAL ARRAYS.
C
C
      DIMENSION ARRAY1(5)
      ARRAY1(2) = 3.
      NUM = 1
      R = ARRAY1(NUM)
      RETURN
      END

      SUBROUTINE MULDIM
C
C   THIS SUBROUTINE DEMONSTRATES MULTIDIMENSIONAL ARRAYS.
C
      REAL*4 ARR3(2,2,2)
C
      REAL*4 ARRAY2(1000,1000)
      DO 10 J=1,1000
         DO 10 I = 1,1000
             ARRAY2 (I,J) = 0
 10   CONTINUE
C
      RETURN
      END
      SUBROUTINE ARREXP
C
C    THIS SUBROUTINE DEMONSTRATES ARRAYS -- EXPLICIT LOWER BOUNDS.
C
      DIMENSION ARR3A(4:5,2:3,1:2)
      RETURN
      END

      SUBROUTINE ARRSUB
C
C    THIS SUBROUTINE DEMONSTRATES ARRAYS -- SIGNED SUBSCRIPTS
C
      DIMENSION ARR2(4,2),ARR2S(-2:2,2)
      RETURN
      END

      SUBROUTINE ARRCHR
C
C    THIS SUBROUTINE DEMONSTRATES SUBSTRINGS OF CHARACTER ITEMS
C
      CHARACTER*10 SUVAR,SUARR(3)
      SUVAR='ABCDEFGHIJ'
      SUARR(2)(:5)=SUVAR(6:10)
      RETURN
      END

      SUBROUTINE INITD1
C
C    THIS SUBROUTINE DEMONSTRATES INTIALIZING DATA  (FIRST)
C
      CHARACTER*4 CARL,CELS*2
      DATA DEG,CELS,CARL/10.2,'DG','SURD'/,AVCH/.1515/
      RETURN
      END
      SUBROUTINE INITD2
C
C    THIS SUBROUTINE DEMONSTRATES INTIALIZING DATA  (SECOND)
C
      PARAMETER (DEGI=10.2)
      DATA DEG/DEGI/
      RETURN
      END
      SUBROUTINE INITAR
C
C   THIS SUBROUTINE DEMONSTRATES INITIALIZING ARRAYS
C
      DIMENSION A(10)
      CHARACTER*4 CARRAY(4)
      DIMENSION ARRAYE(10,10)
      DIMENSION ARRAYI(10,10)
      DIMENSION ARRAYD(10,10)

      DATA A(1),A(2),A(4),A(5)/1.0,2.0,4.0,5.0/
      DATA CARRAY(1),CARRAY(4)/'ABC','EFGH'/
      DATA ((ARRAYE(I,J),I=1,10),J=1,10)/100*0.0/
      DATA ((ARRAYI(I,J),I=1,J-1),J=2,10)/45*0.0/,
     1     ((ARRAYI(I,J),J=1,I-1),I=2,10)/45*0.0/,
     2     (ARRAYI(I,I),I=1,10)/10*1.0/
      DATA ((ARRAYD(I,J),I=1,10),J=1,10) /100*0.0/
      DO 30 I=1,10,1
      ARRAYD(I,I)=1.0
   30 CONTINUE

      RETURN
      END

      SUBROUTINE MANDAT
C
C   THIS SUBROUTINE DEMONSTRATES MANAGING DATA STORAGE -- EQUIVALENCE
C     STATEMENT
C
      DIMENSION ARR3(2,2,2)
      CHARACTER*4 CHAR3(4)
      EQUIVALENCE (ARR3(2,2,1),CHAR3(1))
      RETURN
      END

      SUBROUTINE EXECEF
C
C   THIS SUBROUTINE DEMONSTRATES EXECUTION-TIME EFFICIENCY USING
C     EQUIVALENCE
C
      DIMENSION A(10),I(16),A2(5)
      EQUIVALENCE (A(1),I(6),A2(1))
      RETURN
      END
      SUBROUTINE ARTHEX
C
C   THIS SUBROUTINE DEMONSTRATES ARITHMETIC EXPRESSIONS
C
      DOUBLE PRECISION RESULT
      DATA I1/10/,I2/15/

      RESULT = AR3*AR1
      RESULT = I2/I1
      RETURN
      END
      SUBROUTINE CHARX1
C
C   THIS SUBROUTINE DEMONSTRATES CHARACTER EXPRESSIONS (FIRST)
C
      CHARACTER*12 CHAR
      CHARACTER*6 CHAR1,CHAR2
      DATA CHAR1/'ABCDEF'/,CHAR2/'GHIJKL'/
      CHAR = CHAR1//CHAR2
      RETURN
      END

      SUBROUTINE CHARX2
C
C   THIS SUBROUTINE DEMONSTRATES CHARACTER EXPRESSIONS (SECOND)
C
      CHARACTER*10 CHAR10
      CHARACTER*5 CHAR5,CHAR2*2
      EQUIVALENCE (CHAR5(2:5),CHAR10)
      CHAR10 = 'ABCDEFGHIJ'
      CHAR5 = 'MNOPQ'
      CHAR2 = 'XY'
      CHAR10 = (CHAR5//CHAR2)
      PRINT *, ' CHAR10 =', CHAR10, 'CHAR5 = ',
     1 CHAR5, 'CHAR2= ',CHAR2
      RETURN
      END

      SUBROUTINE RELEX1
C
C   THIS SUBROUTINE DEMONSTRATES RELATIONAL EXPRESSIONS (FIRST)
C
      LOGICAL L
      L = A.GE.B
      L = (A+B).LT.(C-B)
      RETURN
      END
      SUBROUTINE RELEX2
C
C   THIS SUBROUTINE DEMONSTRATES RELATIONAL EXPRESSIONS (SECOND)
C
      COMPLEX CMPLX1,CMPLX2
      LOGICAL L
      L = (CMPLX1-2).EQ.(CMPLX2+2)
      RETURN
      END

      SUBROUTINE RELEX3
C
C   THIS SUBROUTINE DEMONSTRATES RELATIONAL EXPRESSIONS (THIRD)
C
      CHARACTER*5 CHAR4,CHAR5,CHAR6*8
      LOGICAL L
      L = (CHAR4//CHAR5).GT.CHAR6
      RETURN
      END
      SUBROUTINE RELCHR
C
C   THIS SUBROUTINE DEMONSTRATES RELATIONAL EXPRESSIONS -- CHARACTER
C     OPERANDS
C
      CHARACTER*3 C1/'3AB'/,C2/'XYZ'/
      LOGICAL L
      L = C1.GT.C2
      RETURN
      END

      SUBROUTINE LOGEXP
C
C   THIS SUBROUTINE DEMONSTRATES LOGICAL EXPRESSIONS
C
      LOGICAL L1,L2,L3,L4
      L1 = A.GT.B.OR.A.EQ.C
      L2 = A.GT.B.AND.A.EQ.C
      L3 = A.GT.B.AND..NOT.A.EQ.C
      L4 = A.GT.B.OR.A.EQ.C.AND.B.LT.D
      RETURN
      END

      SUBROUTINE ARTHAS
C
C   THIS SUBROUTINE DEMONSTRATES ARITHMETIC ASSIGNMENTS
C
      DIMENSION ARRAY3(10)
      PI = 3.14159
      ARRAY3(NUM) = DIFF
      INTR = DIFF
      DIFF = INTR
      DIFF = INTR+DIFF
      RETURN
      END

      SUBROUTINE CHARA1
C
C   THIS SUBROUTINE DEMONSTRATES CHARACTER ASSIGNMENTS (FIRST)
C
      CHARACTER*10 SUVAR
      SUVAR = 'ABCDEFGHIJ'
      RETURN
      END

      SUBROUTINE CHARA2
C
C   THIS SUBROUTINE DEMONSTRATES CHARACTER ASSIGNMENTS (SECOND)
C
      CHARACTER*5 A,B,C,E *13
      DATA A/'WHICH'/,B/' DOG '/,C/'BITES'/
      E = A//B//C
      A(4:5) = C(3:4)
      RETURN
      END

      SUBROUTINE LOGL1
C
C   THIS SUBROUTINE DEMONSTRATES LOGICAL ASSIGNMENTS (FIRST)
C
      LOGICAL*4 LOGOP
      REAL * 8 AR1/1.1/,AR2/2.2/,AR3/3.3/,AR4
      LOGOP = (AR4.GT.AR1).OR.(AR2.EQ.AR3)
      RETURN
      END
      SUBROUTINE LOGL2
C
C   THIS SUBROUTINE DEMONSTRATES LOGICAL ASSIGNMENTS (SECOND)
C
      LOGICAL*4 LOGOP
      CHARACTER*6 CHAR1, CHAR2, CHAR3
      DATA CHAR1/'ABCDEF'/ CHAR2/'GHIJKL'/
      LOGOP = (CHAR2.EQ.CHAR3).AND.(CHAR1.LT.CHAR2)
      RETURN
      END

      SUBROUTINE STMTF
C
C   THIS SUBROUTINE DEMONSTRATES STATEMENT FUNCTIONS
C
      WORK(A,B,C,D,E) = 3.274*A + 7.477*B - C/D + (X+Y+Z)/E
      W = WORK(GAS,OIL,TIRES,BRAKES,PLUGS) - V
      RETURN
      END
```

Part 2. Using VS FORTRAN—Environmental Considerations

Part 2 discusses the following topics:

Chapter 11. Using VS FORTRAN under VM

You can compile and execute your programs in a CMS virtual machine running under VM/SP. You can also compile your programs in this environment and run them in either MVS or VSE.

Execution of your program in a CMS virtual machine is done in CMS's OS simulation mode; that is, the VS FORTRAN execution-time library routines use the MVS services that are simulated by CMS. Because of this, you must observe the following restrictions:

1. You cannot compile your programs nor can you execute them 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 compile or execute your VS FORTRAN programs.

2. VS FORTRAN programs that use asynchronous I/O cannot run in a CMS virtual machine.

3. The VSE version of the VS FORTRAN compiler and library does not run in a CMS virtual machine, even in the CMS/DOS environment.

CP and CMS Commands

The CP and CMS commands help you create and edit your source programs, create executable programs, and execute your programs.

The CMS commands you'll use most frequently are shown in Figure 39 on page 229. For a description of all the options that can be specified and for additional information on the use of all the CMS commands, see *VM/SP CMS Command and Macro Reference.*

You can use all the CP and CMS commands to develop, test, and run your VS FORTRAN programs during terminal sessions. See *VM/SP Terminal Reference* for documentation on terminal usage.

Creating Your Source Program—CMS System Product Editor

Your first step is to enter your source program into the system. For the method your organization uses, see your system administrator.

To create a source program file, you can use an editor. You may use the XEDIT command (or the appropriate command for whatever editor you want to use) to create a new file and also to change an existing one.

To create a FORTRAN source program file, you must specify the filetype of your source program file as FORTRAN. You can now enter your source program into the file, line by line, according to the rules for fixed- or free-form source programs.

See *VS FORTRAN Language and Library Reference* for formatting considerations.

CMS Command	Usage
ACCESS	Activates a virtual disk for your use.
EXEC	Executes a file that consists of one or more CMS commands.
FILEDEF	Defines a file and its input/output devices.
FORTVS	Invokes the VS FORTRAN compiler.
GLOBAL	Specifies text libraries to be searched to resolve external references in a program being loaded. Specifies load libraries from which routines are loaded during execution.
INCLUDE	Specifies additional TEXT files for use during program execution.
LISTFILE	Displays a list of your files.
LOAD	Places a TEXT file in storage and establishes the linkages for execution.
PRINT	Prints a file on the spooled virtual printer.
PUNCH	Punches a card file on the virtual card punch.
RENAME	Changes the filetype, filename, and/or filemode of a file.
START	Begins execution of a previously loaded file.
TYPE	Types all or part of a file.
XEDIT	Puts you in XEDIT mode to create and edit source program and data files.

Figure 39. CMS Commands Often Used with VS FORTRAN

Using the VS FORTRAN Compiler Options

The VS FORTRAN compiler options let you specify details about the input source program and request specific forms of compilation output. You specify the compiler options as options of the FORTVS command or the @PROCESS command (see "Modifying Compilation Options—@PROCESS Statement" on page 164).

The VS FORTRAN compiler options are shown under "Using the Compiler Options" on page 157.

Note: Under VM, if you specify DC on the FORTVS command, only the 8 characters following the left parenthesis are passed to VS FORTRAN. No error message is generated if any truncation occurs. Check requested options on listing output.

Additionally, an option for the disposition of the listing file is available to VM users. This may only be specified on the FORTVS 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.

Abbreviation: NOPRI

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

Abbreviation: PRI

Specifying CMS Line Numbers When Debugging

If you want to use CMS line numbers as breakpoints when using VS FORTRAN Interactive Debug, you must specify the NOSDUMP and TEST compiler options.

For more details, see Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377.

IBM Extension

Using the FORTRAN INCLUDE Statement

If your source program uses the INCLUDE statement, you must create a library containing the INCLUDE source code.

1. Create one or more members with a filetype of COPY.

```
XEDIT member1 COPY A
INPUT
     COMMON/COM1/A1,A2,A3,A4
     COMMON/COM2/B1,B2,B3,B4

FILE
```

Note: The included files may be fixed blocked or fixed unblocked by specifying RECFM FB or RECFM F when you create the members (member1 in this example).

2. Create a FORTRAN source program.

```
XEDIT myprog FORTRAN A
INPUT
     INCLUDE (member1)
     Z = A1 * B1
          .
          .
          .
     END
FILE
```

You can selectively activate INCLUDE statements within the FORTRAN source program by specifying the identification numbers of the INCLUDEs to be processed. For example:

```
INCLUDE (name) [n]
```

where *n* is a number that appears in the CI list, and ranges from 1 through 255.

See "Using the Compiler Options" on page 157 for a description of the CI compiler option.

3. Create a CMS macro library.

```
MAC GEN libname member1
```

End of IBM Extension

Printing on the IBM 3800 Printing Subsystem under CMS

Additional run-time parameters are required to support the IBM 3800 printing subsystem. For example, the SETPRT command is used to specify the names of the character arrangement tables.

SETPRT CHARS(cat0[cat1 ...]), ...
specifies the names of from one to four character arrangement tables to be loaded into the virtual 3800. These names must be from one to four alphameric characters. The character arrangement tables must exist as 'XTB1catn TEXT' files on an accessed CMS disk.

PRINT fn ft ... (TRC
specifies that the record contains a Table Reference Character byte.

Sample FORTRAN Program Using the 3800

```
C   SAMPLE PROGRAM FOR THE IBM 3800 PRINTING SUBSYSTEM
C
 100  FORMAT ('12','    W66666666666666666666666666X'
    1    / ' 2','     7                           7'
    2    / '+1','       TABULATION OF THE FUNCTION  '
    3    / ' 2','     7                           7'
    4    / ' 2','     7                           7'
    5    / '+0','               sin',A1,'(x)       '
    6    / ' 2','     Z66666666666666666666666666Y'
    7    / ' 2','     W66666666666616666666666666X'
    8    / ' 2','     7            7              7'
    9    / '+0','         x           sin',A1,'(x) '
    A    / ' 2','     36666666666665666666666666664')

 200  FORMAT (' 2','  7            7              7'
    1    / '+0','     ',I3,A1,I2,'''',I2,'"',5X,F9.6)

      CHARACTER*1 DEG, U3
      DATA DEG/ZA1/, U3/ZB3/

      WRITE (6,100) U3, U3

      WRITE (6,200) 0, DEG, 0, 0, 0.

      END
      .
      .
      .
   SETPRT CHARS(TN GS10 FM10), ...
   PRINT  fn ft  (TRC ...
```

The sample program above produces the following 3800 output:

```
┌───────────────────────────────┐
│ TABULATION OF THE FUNCTION    │
│          sin³(x)              │
└───────────────────────────────┘
┌──────────────┬────────────────┐
│      x       │    sin³(x)     │
├──────────────┼────────────────┤
│  0°  0'  0"  │   0.000000     │
```

Compiling Your Program

If you want to compile *myprog* with the defaults your installation uses, you specify:

```
FORTVS myprog
```

If, however, you want to compile *myprog* using nondefault compiler options, specify, for example:

```
FORTVS myprog (FREE FLAG(E) DECK MAP)
```

which tells the compiler that your source program is in free form, and which gives you a compilation with only E level messages or higher flagged (FLAG(E)), a copy of the object deck sent to your virtual punch (DECK), and a map of names and labels (MAP); all the options you don't specify retain the default values.

You may have FORTRAN source files on tape or punched cards. To make these files known to the system, you must issue a FILEDEF command whose ddname is FORTRAN and which specifies the appropriate device type.

Examples:

To use 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
```

To invoke the compiler using the READER or TAP*n* as input, issue:

```
FORTVS dummy (options
```

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

If you have used an INCLUDE statement in your source program, you need to take **one** of the following steps (if both steps are taken, unpredictable results will occur).

- Define SYSLIB for use by the compiler:

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

- Specify the macro library in a GLOBAL statement:

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

Compiler Output

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

- The source program listing—as you entered it, but with compiler-generated internal sequence numbers prefixed at the left; the sequence numbers identify the line numbers referred to in compile-time messages.

- An object module—a translation of your program in machine code.

- Messages about the results of the compilation.

- Other listings helpful in debugging.

Depending on your organization's compile-time defaults and/or the options you select in your FORTVS command, you may get a LISTING file and/or a TEXT file as output, placed in your mini-disk storage for easy reference.

These listings are described in "Identifying User Errors" on page 167 and Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185; examples of output for each feature are also given there.

If your compilation caused error messages, you may have to correct your source, as described in "Identifying User Errors" on page 167.

LISTING File

The LISTING file contains the compiler output listing; see "Identifying User Errors" on page 167 for an explanation of what the compiler output listing contains and how to use it. It has the filename of your source program, and the filetype LISTING. For example, the file for *myprog* is *myprog* LISTING.

You can display the LISTING file at your terminal, using an editor.

You can print a copy of the LISTING file by means of your virtual printer, using the PRINT command:

```
PRINT myprog LISTING
```

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. The contents of the TEXT file are explained in "Object Module as Link-Edit Data Set" on page 183.

If the OBJECT compiler option is specified, the file is placed in your storage with the filename of your source program and a filetype of TEXT. For example, the file for *myprog* is *myprog* TEXT.

You can link-edit the TEXT file under any of the systems that VS FORTRAN supports to get a load module (or phase).

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
```

For information on how to fix errors that occur during compilation, see "Identifying User Errors" on page 167.

Automatic Cross-System Support

In VS FORTRAN, you can compile your source program under any supported operating system. You can then link-edit the resulting object module under the same system, or under any other supported system.

For example, you could request compilation under CMS and then link-edit the resulting object module for execution under VSE.

You don't have to request anything special during compilation to do this; VS FORTRAN uses the execution-time library for all system interfaces, so the operating system under which you link-edit determines the system under which you execute.

Using the VS FORTRAN Separation Tool

The VS FORTRAN separation tool divides a program compiled with the RENT option into reentrant and nonreentrant CSECTS. For general information on the tool, see "VS FORTRAN Separation Tool (for Both VM and MVS)" on page 189. The following information is specific for CMS.

The separation tool is located in VFORTLIB TXTLIB, and consists of two CSECTS, IFYVSFST and IFYVSFIO. The following command sequence invokes the separation tool:

```
GLOBAL TXTLIB VFORTLIB CMSLIB
LOAD IFYVSFST IFYVSFIO (CLEAR
START * thisname
```

The following file setup must be in effect when the separation tool is invoked:

```
FILEDEF * CLEAR
RENAME myprog TEXT A myprog TEXTIN A
FILEDEF SYSIN DISK myprog TEXTIN A
FILEDEF SYSPRINT DISK myprog TOOLLIST A
FILEDEF SYSUT1 DISK myprog TEXT A
FILEDEF SYSUT2 DISK thisname TEXT A
FILEDEF SYSUT3 DISK myprog TEMP A
```

The input file is SYSIN and the output files are SYSUT1 and SYSUT2, nonreentrant and reentrant files, respectively. After the separation tool has completed, the work file, SYSUT3, may be erased.

To use the text files created in the above example, issue the following commands:

```
GLOBAL TXTLIB VFORTLIB CMSLIB
GLOBAL LOADLIB VFLODLIB
LOAD myprog (NOAUTO CLEAR
START
```

When needed, the VS FORTRAN library will load the text file, *thisname* TEXT, and provide the reentrant modules to the program.

A Simple Scenario That Might Occur

The tool expects the text file to contain entries in a certain order: the reentrant CSECT followed by the corresponding nonreentrant CSECT. If this does not occur because of a routine's not being compiled for reentrancy, the tool will accommodate the difference.

A simple scenario that might occur is: Given a simple FORTRAN program with a single program named MAIN, the VS FORTRAN compiler would generate the following:

- A reentrant CSECT with the name @MAIN
- A nonreentrant CSECT with the name MAIN

All of this output is in the same text file and, when loaded, all address constants will be properly resolved and the program will run as it has been coded. If, however, the text file is passed to the separation tool, the following is output:

- A file containing the nonreentrant CSECT, MAIN
- A file containing two CSECTs: IFYZRENT and the reentrant CSECT, @MAIN

There are two ways of invoking the separation tool (IFYVSFST). The first is without any module name parameter, and the second is with a module name parameter.

With the module name parameter, RENTPART, the following is output:

- A file containing the nonreentrant CSECT, MAIN, but with an extra record with the module name (RENTPART) in it
- A file containing the reentrant CSECTs, IFYZRENT and @MAIN, and a record with "NAME RENTPART(R)" on it

Without the module name parameter, the following is output:

- A file containing the nonreentrant CSECT, MAIN
- A file containing the reentrant CSECTs, IFYZRENT AND @MAIN, and a record with "NAME @MAIN(R)" on it

There are several ways to load the modules; in each case, several things must be done to ensure access to the modules.

1. The first possibility is as follows:

    a. A module name was specified.

    b. The reentrant output file has a filename the same as the module name and a filetype of TEXT.

    c. The appropriate GLOBAL TXTLIB and GLOBAL LOADLIB commands are issued.

    d. The nonreentrant text output went into a file with a filetype of TEXT.

    e. You issue:

        1) LOAD nrname (NOAUTO START

           or

        2) LOAD nrname (NOAUTO CLEAR
           GENMOD nrname
           nrname

        where *nrname* is the nonreentrant filename.

    f. When execution begins, the file with the reentrant CSECTs will be loaded by the VS FORTRAN library and the program will execute.

    g. If you want to keep a record of where the reentrant portion is located, keep a copy of the LOAD MAP created by the LOAD command.

2. The second possibility is as follows:

    a. A module name was specified when invoking the separation tool.

    b. The reentrant output file has a filename the same as the module name and a filetype of TEXT.

    c. The reentrant output file is link-edited with the LKED command.

    d. The appropriate GLOBAL TXTLIB and GLOBAL LOADLIB commands are issued with the LOADLIB list containing the name of the new file.

    e. The nonreentrant text output went into a file with a filetype of TEXT.

    f. You issue:

        1) LOAD nrname (NOAUTO START

           or

        2) LOAD nrname (NOAUTO CLEAR
           GENMOD nrname
           nrname

        where *nrname* is the nonreentrant filename.

g. When execution begins, reentrant module(s) will be loaded from the LOADLIB by the VS FORTRAN library and the program will execute.

h. If you want to keep a record of where the reentrant portion is located, keep a copy of the LOAD MAP created by the LOAD command.

3. The third possibility is as follows:

   a. No module name was specified.

   b. The reentrant output file has a filename the same as the module name and a filetype of TEXT.

   c. The reentrant output file is link-edited with the LKED command.

   d. The appropriate GLOBAL TXTLIB and GLOBAL LOADLIB commands are issued with the LOADLIB list containing the name of the new file.

   e. The nonreentrant text output went into a file with a filetype of TEXT.

   f. You issue:

      1) LOAD nrname (NOAUTO START

         or

      2) LOAD nrname (NOAUTO CLEAR
         GENMOD nrname
         nrname

      where *nrname* is the nonreentrant filename.

   g. When execution begins, reentrant module(s) will be loaded from the LOADLIB by the VS FORTRAN library and the program will execute.

   h. If you want to keep a record of where the reentrant portion is located, keep a copy of the LOAD MAP created by the LOAD command.

In the scenario example of one MAIN CSECT without subprograms, the second and third possibilities are practically the same. The only difference is that in the second, a module name was specified when RENTPART was invoked; in the third, because no module name was specified, the separation tool uses the name generated by the compiler.

If the input text file contained many routines with reentrant portions, the difference between the second and third possibilities becomes noticeable. In the second possibility, one large load module with the locator CSECT (IFYZRENT) and all of the reentrant CSECTs would be created. In the third possibility, a large number of load modules, each with its own locator CSECT and its own name, would be created. In either case, the VS FORTRAN library handles the conditions properly—with a trade-off of disk space versus main storage space.

In the module name case, one large locator CSECT along with all of the reentrant CSECTs would be created. In the no module name case, many small modules, each with its own locator CSECT, would be created. At first, the difference in file size would be small, but a number of CSECTs the size of the locator CSECTs could accumulate and cause disk storage size problems.

If the reentrant portion is to be entered into a shared segment, you must decide the best size of the segment and choose whether to provide one name for a large segment or a lot of names for a large number of small segments.

There is a limit of 255 ESDIDs per loaded module. If you choose the module name possibility, you must have 254 or fewer reentrant CSECTs in the text file. If the text file is to be used to generate a LOADLIB member, there is no 255 ESDID count limit. Note that this 255 ESDID limit would affect a user who intends to put the module into a shared segment (DCSS). The only solution would be to compile fewer than 255 routines with reentrant parts with the VS FORTRAN compiler; or to compile as usual, but partition the compiler output text file into two or more parts.

CMS EXEC Files to Execute the Separation Tool

Two sample EXECs are shown, in Figure 40 on page 240 and Figure 41 on page 243. They are:

FOVSRCS which compiles a FORTRAN program with the RENT compiler option and then separates the reentrant and nonreentrant CSECTs.

FOVSRSEP which separates the reentrant and nonreentrant CSECTs from a TEXT file that was created previously from a compilation with the RENT compiler option.

You can execute the FOVSRCS EXEC for compilation of a reentrant program, followed by separation, as follows:

```
FOVSRCS ftnname renttext (options ...
```

where:

ftnname is the filename of the FORTRAN source program. This file must have a filetype of FORTRAN.

renttext is the filename of the TEXT file into which the reentrant object code is to be placed. This name must be different from *ftnname* above.

options may be any VS FORTRAN compiler options. RENT need not be given because the EXEC already provides it. In addition, either of the following may be coded to specify the structure of the reentrant output file. This controls the result when the file is subsequently used as input to the LKED or to the TXTLIB ADD command.

ONEMEMB	Only one LOADLIB or TXTLIB member (of name *renttext*) is desired. This is the default.
MULTMEMB	A separate LOADLIB or TXTLIB member is desired for each reentrant CSECT (that is, for each main program or subprogram that is compiled).

```
&CONTROL OFF NOMSG
&GOTO -START
*
*  COMPILE A REENTRANT FORTRAN PROGRAM AND SEPARATE THE REENTRANT
*   AND NON-REENTRANT CSECTs
*
-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
      non-reentrant and its reentrant parts.
Execute it as follows:

&END
&TYPE &EXEC ftnname renttext (options, ...
&BEGTYPE

 ftnname   is the file name of the FORTRAN source program.  This file
           must have a file type of FORTRAN.

 renttext  is the file name of the TEXT file into which the reentrant
           object code is to be placed.  This name must be different
           from "ftnname", above.

 options   may be any VS FORTRAN compiler options.  RENT need not be
           given.  In addition, either of the following may be coded
           to specify the structure of the reentrant output file.
           This controls the result when that file is subsequently
           used as input to the LKED or to the TXTLIB ADD command.

            ONEMEMB    Only one LOADLIB or TXTLIB member (of name
                       "renttext") is desired.  This is the default.

            MULTMEMB   A separate LOADLIB or TXTLIB member is desired
                       for each reentrant CSECT (that is, for each
                       main program or subprogram that is compiled).
&END
&EXIT  &RCVALUE
-START
```

Figure 40 (Part 1 of 2). FOVSRCS, Compile Reentrant Program and Separate CSECTs

```
*
*  CHECK FOR VALID OR OMITTED PARAMETERS
*
&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 ONEMEMB OR MULTMEMB
*
&P  =  4
-NEXTOPT
&IF  &P   GT  &INDEX     &GOTO -ENDOPT
&IF  &&P  EQ  ONEMEMB    &GOTO -SETPARM
&IF  &&P  NE  MULTMEMB   &GOTO -INCRPAR
-SETPARM
&SEPPARM =  &&P
&&P  =
-INCRPAR
&P  =  &P + 1
&GOTO -NEXTOPT
-ENDOPT
*
*  ASSURE THAT THE INPUT SOURCE FILE NAME DIFFERS FROM REENTRANT NAME
*
&IF  &1  NE  &2  &GOTO -INOUT
&BEGTYPE
The input FORTRAN file name must differ from the output reentrant
TEXT file name.
&END
&EXIT 12
-INOUT
*
*  COMPILE THE PROGRAM
*
&F  =  FORTVS
&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 REENTRANT AND NON-REENTRANT CSECTS
*
EXEC  FOVSRSEP  &1 &2 (&SEPPARM
&EXIT &RETCODE
*
*  INCORRECT INPUT PARAMETERS
*
-BADINP
&BEGTYPE
Invalid input parameter format.

&END
&RCVALUE  =  8
&GOTO -RULES
```

Figure 40 (Part 2 of 2). FOVSRCS, Compile Reentrant Program and Separate CSECTs

You execute the FOVSRSEP EXEC for separation using an existing TEXT file as input as follows:

```
FOVSRSEP inptext renttest (option
```

inptext	is the filename of the input TEXT file which is to be separated. This file is updated to contain only the nonreentrant CSECTs.
renttext	is the filename of the TEXT file into which the reentrant object code is to be placed. This name must be different from *inptext* above.
option	may be either of the following to specify the structure of the reentrant output file. This controls the result when that file is subsequently used as input to the LKED or to the TXTLIB ADD command.

ONEMEMB	Only one LOADLIB or TXTLIB member (of name *renttext*) is desired. This is the default.
MULTMEMB	A separate LOADLIB or TXTLIB member is desired for each reentrant CSECT that is found in the input TEXT file.

```
&CONTROL OFF NOMSG
&GOTO -START
*
*  SEPARATE THE REENTRANT AND NON-REENTRANT CSECTs IN A TEXT FILE
*     CREATED BY A COMPILATION DONE WITH THE "RENT" OPTION
*
-RULES
&BEGTYPE
This EXEC separates an input TEXT file into two parts:  one with the
non-reentrant CSECTs and the other with the reentrant CSECTs.
Execute it as follows:

&END
&TYPE  &EXEC inptext renttext (option
&BEGTYPE

 inptext   is the file name of the input TEXT file which is to be
           separated.  This file is updated to contain only the
           non-reentrant CSECTs.

 renttext  is the file name of the TEXT file into which the reentrant
           object code is to be placed.  This name must be different
           from "inptext", above.

 option    may be either of the following to specify the structure
           of the reentrant output file.  This controls the result
           when that file is subsequently used as input to the LKED
           or to the TXTLIB ADD command.

            ONEMEMB     Only one LOADLIB or TXTLIB member (of name
                        "renttext") is desired.  This is the default.

            MULTMEMB    A separate LOADLIB or TXTLIB member is desired
                        for each reentrant CSECT that is found in the
                        input TEXT file.
&END
&EXIT  &RCVALUE
-START
```

Figure 41 (Part 1 of 4). FOVSRSEP, Separate Reentrant and Nonreentrant CSECTs

```
*
*  CHECK FOR VALID OR OMITTED PARAMETERS
*
&SEPPARM  =  ONEMEMB
&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 ONEMEMB OR MULTMEMB
*
&P  =  4
-NEXTOPT
&IF  &P   GT  &INDEX     &GOTO -ENDOPT
&IF  &&P  EQ  ONEMEMB    &GOTO -SETPARM
&IF  &&P  EQ  MULTMEMB   &GOTO -INCRPAR
&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
*
*  ASSURE THAT THE INPUT TEXT FILE NAME DIFFERS FROM REENTRANT NAME
*
&IF  &1  NE  &2  &GOTO -INOUT
&BEGTYPE
The input TEXT file name must differ from the output (reentrant) name.
&END
&EXIT 12
-INOUT
```

Figure 41 (Part 2 of 4). FOVSRSEP, Separate Reentrant and Nonreentrant CSECTs

```
*
*  DETERMINE WHETHER TO SUPPLY LOAD MODULE NAME
*
&IF  &SEPPARM  NE  ONEMEMB    &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 NRENTEXT A
FILEDEF  SYSUT2    DISK  &2 TEXT     A
FILEDEF  SYSUT3    DISK  &1 TEMP     A
*
*  EXECUTE THE SEPARATION TOOL PROGRAM
*
LOAD  IFYVSFST IFYVSFIO (CLEAR
START * &LMNAME
*
*  ASSURE THAT SEPARATION WAS SUCCESSFUL
*
&IF  &RETCODE  EQ  0   &GOTO -SEPOK
&BEGTYPE
Errors occurred during execution of the separation tool.
&END
&RCVALUE  =  &RETCODE
ERASE  &1 NRENTEXT 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 TEXT     A
RENAME &1 NRENTEXT A   &1 TEXT A
```

Figure 41 (Part 3 of 4). FOVSRSEP, Separate Reentrant and Nonreentrant CSECTs

```
*
*  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 41 (Part 4 of 4). FOVSRSEP, Separate Reentrant and Nonreentrant CSECTs

Selection of 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 executed. Execution-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 library routines included within your executable program, you are operating in **link mode**. If, on the other hand, you choose to have the library routines loaded during execution of your program, you are operating in **load mode**. You make the choice of link mode or load mode by making the appropriate combination of libraries available when you create your executable program from your TEXT files.

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 execute the program using the START command. No permanent copy of the executable program is made. Execution 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 which is stored as a nonrelocatable (MODULE) file on a CMS disk. You may execute the program later by issuing a CMS

command with the same name as the MODULE file. Execution can be in link mode or load mode.

3. By using the LKED command to create—that is, to link-edit—an executable program which is stored as a load module in a member of a CMS LOADLIB. You may execute the program later by using the OSRUN command. Execution can only be done in load mode.

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:

- VLNKMLIB, a text library (TXTLIB), which contains library modules used for creating a program that is to operate in link mode.
- VFORTLIB, a text library (TXTLIB), which contains library modules used for creating a program that is to operate in link mode, and for creating a program that is to operate in load mode.
- VFLODLIB, a CMS LOADLIB, which contains the library modules that may be loaded into virtual storage during execution of your program.

Your system programmer must tell you which CMS minidisk contains these libraries so that you may gain access to this minidisk. In addition, your system programmer may have given these libraries names that are 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 make the appropriate VS FORTRAN Library text libraries, as well as your own text libraries, available by means of a GLOBAL command. If you want your program to execute in link mode, use the following:

```
GLOBAL TXTLIB VLNKMLIB VFORTLIB CMSLIB userlib ...
```

If you want your program to execute in load mode, use the following command:

```
GLOBAL TXTLIB VFORTLIB 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 which 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 executing the temporary copy of your executable program, you must issue the following GLOBAL command if execution is to be in load mode:

```
GLOBAL LOADLIB VFLODLIB
```

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

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

```
START *
```

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, however, you must make the appropriate VS FORTRAN Library text libraries, as well as your own text libraries, available by means of a GLOBAL command. If you want your program to execute in link mode, use the following command:

```
GLOBAL TXTLIB VLNKMLIB VFORTLIB userlib ...
```

If you want your program to execute in load mode, use the following command:

```
GLOBAL TXTLIB VFORTLIB 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 which 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 executed at any time.

You may be required to issue one or more GLOBAL commands prior to executing 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 which does not have these instructions:

```
GLOBAL TXTLIB CMSLIB
```

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

```
GLOBAL LOADLIB VFLODLIB
```

To execute 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. Such a program can execute only in load mode. Prior to issuing the LKED command, you must issue the following FILEDEF command:

```
FILEDEF SYSLIB DISK VFORTLIB TXTLIB fm
```

where *fm* is the filemode of the CMS disk that contains the library VFORTLIB. Then 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 whose object code is 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
```

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 into 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 you can execute the program that was created with the LKED command, you must issue the following GLOBAL command:

```
GLOBAL LOADLIB VFLODLIB 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 execute your program:

```
OSRUN membname
```

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

Specifying Execution-Time Options under CMS

You can specify an execution-time option (XUFLOW, NOXUFLOW, DEBUG, or NODEBUG), as follows:

- When executing your program using a START command:

  ```
  START * option
  ```

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

```
modname option
```

where modname is the name of your VS FORTRAN program, and option is XUFLOW, NOFLOW, DEBUG, or NODEBUG.

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

  ```
  OSRUN membname PARM=option
  ```

For more information, see "Using the Execution-Time Options" on page 200.

Execution-Time Files

When you execute your program using the techniques described above, you may need many different files, as outlined in the following sections.

The following describes how files are defined through the use of the CMS file definition command, and how they are identified to the system through the use of file identifiers and VS FORTRAN input/output statements.

Predefined Files

All execution-time files that you want to use must be defined to the operating system. Three files are predefined:

Sequential

- Terminal input
- Terminal output
- Punched card output

The three predefined files are provided for you by the VS FORTRAN initialization routine, which also supplies the system definition command for these files. These files are recognized by CMS when you refer to them in VS FORTRAN input or output statements with the VS FORTRAN data set reference number that has been assigned to them. Since they are predefined, you do not have to supply a system definition command for them, unless you want to change their definition or create new files to replace them (see Figure 42).

Data Set Reference Number	Input and Output Statements	Identifies	Required Record Format and Maximum Length
5	READ (5,b)list READ (5,*)list	Terminal input	Fixed-length, unblocked (F), 80 characters long.
6	WRITE (6,b)list WRITE (6,*)list	Terminal output	Fixed-length, unblocked (FA), 133 characters long.
7	WRITE (7,b)list	Punched card output	Fixed-length, unblocked (F), 80 characters long.

Figure 42. Predefined Files

Notes to Figure 42:

1. The numbers shown in this table under Data Set Reference Number are default numbers, but can be different for your installation.

2. The forms of the READ or WRITE statements that have * for the unit number refer to these predefined files—but bear in mind that the defaults may have been changed for your installation.

User-Defined Files

User-defined files containing data that you will want your program to process may already exist in your system. Conversely, you may want your program to create a file to hold data that was generated during its execution.

All such files must be defined with a system definition command; since they are not predefined, they cannot be identified by CMS and associated with your program. System definition is used in conjunction with the data set reference numbers in your VS FORTRAN input and output statements and the identifier of the file that you want to use or create.

You can, in addition, define the following files to be used in place of the system's predefined files or change them to suit your own needs:

Sequential

- Terminal input
- Terminal output
- Punched card output

Whether a file is sequential or direct access will, to a great extent, determine how a record is defined, the way it is identified, and how it is referred to in a VS FORTRAN program.

Regardless of the type of file you are using, there are several general guidelines that must be followed in defining and using files.

- Define each file used in your program to the system (either through system-supplied definition or one that you supply).
- Do not use the same file on more than one type of device in the same program.
- You may refer to the same file from more than one program using sequential and direct access if you refer to it with different unit numbers and with separate data definition statements.

Specifying a File Identifier

Each file that you use is referred to by a file identifier in the following format:

```
filename filetype [filemode]
```

where filename is the name that identifies the file to the system; filetype defines the kind of file this is.

The filetypes you'll most often use are:

DATA	for your data files
FORTRAN	for all your FORTRAN source programs

The filemode is optional and can be any valid CMS filemode. You need to specify the filemode when the same filename (filetype) is located on more than one disk and the first access file is not the desired file.

Using the FILEDEF (File Definition) Command

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.

The FILEDEF command can be shortened to FI.

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

```
FILE FTxxF001 A1
```

where xx is the unit number to which you are putting out the data.

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

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 reference numbers in the source program.

- For the xx field, see Figure 54 on page 286.
- 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 parameter 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 parameter.

For sequential disk files defined with a record format other than undefined or fixed unblocked, the file mode number must be specified as 4; for example, A4.

The options are any FILEDEF options valid for disk files. In particular, the maximum LRECL and BLKSIZE that can be specified is 32760.

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
```

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

- For the xx field, see Figure 54 on page 286.
- 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).

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]
```

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

For the xx field, see Figure 54 on page 286.

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]
```

For Card Punch Files:

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

For Printer Files:

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

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

For the xx field, see Figure 54 on page 286.

The options are any FILEDEF options valid for the type of unit record file you're processing.

Execution-Time Output

The output that execution of your program gives you depends upon whether or not there are errors in your program.

Execution without Error

If your program executes without error, and gives the results you expect, your task of program development is completed.

Execution with Errors

When your program has errors in it, your execution-time output may be incorrect, or nonexistent.

You may or may not get error messages as well. Any VS FORTRAN execution-time error messages you get come from the VS FORTRAN Library.

If you get output from the program itself, it may be exactly what you expected, or (if there are logic errors in the program) it may be output you didn't expect at all. When this happens, you must proceed to the next step in program development, described in Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185.

Chapter 12. Using VS FORTRAN under MVS

You can compile your programs under MVS and run them under VM or VSE; or you can compile and execute them under MVS. For special MVS/XA considerations, see "MVS/XA Considerations" on page 311.

Executing Your Program with Job Control Statements or Cataloged Procedures

The simplest way to execute your program is to use one of the cataloged procedures described in "Using and Modifying Cataloged Procedures" on page 264.

However, the cataloged procedures may not give you the programming flexibility you need for your more complex data processing jobs, and you may need to specify your own job control statements, or write your own cataloged procedures.

Job Processing

Three basic steps are taken to process a FORTRAN program:

1. Compiling
2. Linkage editing
3. Load module execution (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 that is executed in the go step. If the loader is used in place of the linkage editor, the last two steps (link-edit and load module execution) are combined into one step.

Each step is called a *job step*—the execution of one program. Each job step may be executed 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 execution, or of many steps, such as compiler execution followed by linkage editor execution and load module execution.

The programmer defines the requirements of each job to the operating system through *job control statements*.

Job control statements provide a communication link between the FORTRAN programmer and the operating system. The FORTRAN programmer uses these statements to 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:

JOB
EXEC
DD
PROC
Comment
Delimiter (End-of-Data)
Null (End-of-Job)

For a complete description of the JOB, EXEC, DD, and PROC job control statements, see one of the following system publications:

OS/VS2 MVS JCL
MVS/Extended Architecture JCL

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 the appropriate JCL publication.

The parameters let you request the following:

- Accounting information for this job
- Your name
- The type of system messages to be written
- Conditions for terminating job execution
- Assignment of input and output classes
- Job priority
- Main storage requirements
- Time limit for the job

Assigning Default Values—PROC Statement

You use the PROC job control statement to assign default values to symbolic parameters.

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

The name identifies a cataloged procedure; name is optional. 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.

Modifying PROC Statements

You can modify a PROC statement parameter by specifying a change in the EXEC statement that calls the procedure. When you change a PROC statement parameter, you're assigning an overriding value to a symbolic parameter; when the cataloged procedure is executed, this value is transferred to the appropriate parameter in the EXEC or DD statement.

For example, to change the region size of the compiler to 200K bytes and the card punch output in the load module from output class B to output class C, you can use the following statement:

```
//  EXEC  FORTVCLG,FVREGN=200K,
//        GOF7DD='SYSOUT=C'
```

Note that you don't code the ampersand preceding a symbolic parameter, and that you use apostrophes to enclose a value containing a special character, as in SYSOUT=C.

Before you execute the call, the appropriate statements in FORTVCLG appear as follows:

```
//FORT   EXEC   PGM=&FVPGM,REGION=&FVREGN,...
                 .
                 .
                 .
//FT07F001  DD   &GOF7DD
```

When the cataloged procedure is called, the statements appear as though they were coded:

```
//FORT   EXEC   PGM=FORTVS,REGION=200K,...
                 .
                 .
                 .
//FT07F001  DD   SYSOUT=C
```

Note that a symbolic parameter not changed (PGM=&FVPGM) retains its default value.

Another method you can use to change a parameter value is to assign the new value directly to the parameter itself, not the symbolic parameter associated with it. For example, you can change the region size by specifying REGION=200K in place of FVREGN=200K. (In this example, you'd be changing the region size for all job steps; to change the region size for the compile job step only, you must code the appropriate job step name, FORT, in the parameter; for example, REGION.FORT=200K.)

Requesting Execution—EXEC Statement

You use the EXEC job control statement to request that compilation of a program or procedure is to begin.

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

The stepname identifies this job step.

The procname is the name of a cataloged procedure you want executed.

To request a FORTRAN compilation, you specify one of the procedures listed in table under "Using and Modifying Cataloged Procedures" on page 264.

The PARM parameter lets you specify any compiler options that differ from the defaults.

(See "Using the Compiler Options" on page 157 and "Link-Editing Your Program" on page 278.)

The other parameters let you request other information:

- A job step name (when it's necessary for a later job step to refer to information from this job step)
- Conditions for bypassing execution of this job step
- Accounting information for this job step
- Time limit for this step
- Main storage requirements

Modifying EXEC Statements

To modify EXEC statement parameters, you must change the EXEC statement that calls the procedure.

The following rules apply to EXEC statement modifications:

- Parameters are overridden in their entirety. If you want to retain some options while changing others, you must specify those options to be retained. (However, if you don't override them, default options remain in effect.)
- To specify parameters for individual job steps, use the form:

  keyword.stepname=value

where:

keyword
: indicates the parameter name

stepname
: indicates the procedure stepname, for example,

```
REGION.FORT=value
```

Parameters not specifying stepname are assumed to apply to all steps in the procedure; for example, REGION=value applies to the entire cataloged procedure.

- To make changes to more than one step, you must specify all changes for an earlier step before those for later steps.
- You can combine changes to symbolic parameters and EXEC statement parameters on the same card.

You're allowed to make the following modifications:

1. **Override existing parameters**: For example, to modify the LKED step by raising the condition code from 4 to 8, use the statement:

   ```
   //SOMENAME EXEC FORTVCLG,
   //              COND.LKED=(8,LT)
   ```

2. **Add new parameters**: For example, to modify FORT by specifying the TIME parameter, use the statement:

   ```
   //ANYNAME  EXEC  FORTVCLG,TIME.FORT=5
   ```

3. **Change more than one parameter**: For example, to modify FORT by changing the region to 200K bytes and the PARM option NOLIST to LIST, use the statement:

   ```
   //SOME EXEC FORTVCLG,
   //     FVREGN=200K,
   //     FVPOLST=LIST
   ```

4. **Change more than one step**: For example, to modify FORT by specifying TIME and to modify LKED by raising the condition code from 4 to 8, use the statement:

   ```
   //ANY  EXEC  FORTVCLG,TIME.FORT=5,
   //           COND.LKED=(8,LT)
   ```

   Note that you can add a parameter while revising an existing one.

5. **Combine changes to symbolic parameters and EXEC statement parameters**: For example, to modify the symbolic parameter FVREGN, and to add the TIME parameter to the FORT EXEC statement, use the statement:

   ```
   //ANY  EXEC  FORTVCLG,FVREGN=200K,
   //           TIME.FORT=5
   ```

6. **Change execution-time options**: For example, to modify GO by specifying the PARM parameter, use the statement:

   ```
   //ANY EXEC FORTVCLG,GO.PARM='NOXUFLOW'
   ```

Defining Files—DD Statement

To define files you may need, you specify the DD statement:

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

The ddname identifies the data sets defined by this DD statement to the compiler, linkage editor, loader, or to your program. The ddnames you can use for VS FORTRAN are shown in Figure 50 on page 274, Figure 51 on page 275, and Figure 52 on page 282.

The procstep identifies the procedure step.

The data-set-name is the qualified name you've given the data set that contains your data files; for example, the name of the library containing the files you use in your INCLUDE statements.

The other parameters let you request additional information:

- The location of this data set in the system configuration
- The status of this data set at the beginning and end of the job step
- Label information for this data set's volume
- Optimization of input/output channel usage
- Device type
- Space allocation (for data sets on direct access devices)
- Characteristics of the data set records

Job control statements are described under "Job Processing" on page 257.

Modifying DD Statements

You modify the DD statement by submitting new DD statements after the EXEC statement that calls the procedure. As with modifications to EXEC statements, you can override or add parameters to DD statements in one or many steps. In addition, you can add entirely new DD statements to any step (whenever you supply a SYSIN DD statement, you're adding a new DD statement.)

The following rules apply to DD statement modifications:

- Parameters are overridden in their entirety, except for the DCB parameter, in which individual subparameters can be overridden.

- Parameters are nullified by specifying a comma after the equal sign in the parameter; for example, UNIT=,.

- Parameters are overridden when mutually exclusive parameters are specified in their place; for example, SPLIT overrides SPACE.

- DD statements must indicate the related procedure step, using the form *//procstep.ddname*; for example, //FORT.SYSIN.

- To make changes in more than one step, you must specify all changes for an earlier step before those for later steps.

- To modify more than one DD statement in a job step, you must specify the applicable DD statements in the same sequence as they appear in the cataloged procedure.

You can make the following modifications:

1. **Override existing parameters**. For example, to modify SYSLMOD so that the load module is stored in a private library rather than in the system library, you can specify the statement:

   ```
   //LKED.SYSLMOD DD DSNAME=PRIV(PROG),
   //                DISP=(MOD,PASS)
   ```

   In this example, the library PRIV is assumed to be an old library and is cataloged (that is, VOLUME and UNIT parameters need not be specified). Note that, in subsequent uses of the library you must submit a JOBLIB DD statement defining the private library, to make the library available to the system.

2. **Add new parameters**. For example, to store the load module in a new, uncataloged library, you must specify the VOLUME, UNIT, and SPACE parameters. For example:

   ```
   //LKED.SYSLMOD DD DSNAME=MYLIB(FIRST),
   //                DISP=(NEW,PASS),
   //                VOLUME=SER=11234,
   //                UNIT=SYSDA,
   //                SPACE=(TRK,(50,10,2))
   ```

3. **Add new DD statements**. For example, to add new data sets having data set reference numbers 10 and 15 for processing in the GO step, you can specify the statements:

   ```
   //GO.FT10F001  DD  DSNAME=DSET1,
   //                 DISP=(NEW,DELETE),
   //                 VOLUME=SER=T1132,
   //                 UNIT=TAPE
   //GO.FT15F001  DD  DSNAME=DSET2,
   //                 DISP=(,DELETE),
   //                 VOLUME=SER=DA45,
   //                 UNIT=3350,
   //                 SPACE=(TRK,(10,10))
   ```

   Note that you can explicitly define a data set as new (DISP parameter for FT10F001), or, alternatively, permit the system to assume a new data set by default (DISP in FT15F001).

Using and Modifying Cataloged Procedures

The table below is an index to a number of cataloged procedures that you can use to compile, link-edit or load, and execute your VS FORTRAN programs. Figure 43 on page 265 through Figure 49 on page 270 contain these procedures in *load* mode.

Action	**Procedure**	**Figure**
Compile only	FORTVC	Figure 43
Compile and link-edit	FORTVCL	Figure 44
Compile, link-edit, and execute	FORTVCLG	Figure 45
Link-edit and execute	FORTVLG	Figure 46
Execute only	FORTVG	Figure 47
Compile and load	FORTVCG	Figure 48
Load only	FORTVL	Figure 49

As the figures show, many of the JCL parameters in these procedures are coded as symbolic parameters (the parameter name is preceded by an ampersand). The IBM-supplied default value for each symbolic parameter is defined in the PROC statement that begins each procedure.

This means that you can execute the procedures without making any changes to them, or you can modify them for any particular run. (The use of symbolic parameters is explained in the job control language publications for the system you're operating under.)

If you want to change the procedures to operate in link mode, you have to concatenate SYS1.VLNKMLIB ahead of SYS1.VFORTLIB in the SYSLIB DD statement for the link-edit steps. In this case, SYS1.VFORTLIB is not needed in the GO steps.

The procedures are coded to reference the standard mathematical subroutines in SYS1.VFORTLIB, by means of the SYSLIB data set in the link step when the linkage editor is used, or in the GO step when the loader is used. If you want to use the alternative mathematical library subroutines in SYS1.VALTLIB (instead of the standard routines), concatenate SYS1.VALTLIB as SYSLIB ahead of SYS1.VFORTLIB.

Before making either of these changes, you should be aware of how your system administrator installed VS FORTRAN, as that will determine what libraries are available. The procedures may have been changed already to match the needs of your VS FORTRAN installation.

Note: Unless you increase the permissible condition code in the COND parameter of the compilation EXEC statement, severity levels higher than level 4 prevent link-edit processing.

For more information about job processing statements, see "Job Processing" on page 257.

```
//FORTVC    PROC FVPGM=FORTVS,FVREGN=1200K,FVPDECK=NODECK,
//              FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',
//              FVLNSPC='3200,(25,6)'
//*
//*             PARAMETER  DEFAULT-VALUE      USAGE
//*
//*               FVPGM    FORTVS             COMPILER NAME
//*               FVREGN   1200K              FORT-STEP REGION
//*               FVPDECK  NODECK             COMPILER DECK OPTION
//*               FVPOLST  NOLIST             COMPILER LIST OPTION
//*               FVPOPT   0                  COMPILER OPTIMIZATION
//*               FVTERM   SYSOUT=A           FORT.SYSTERM OPERAND
//*               FVLNSPC  3200,(25,6)        FORT.SYSLIN SPACE
//*
//FORT    EXEC  PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//              PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM       DD &FVTERM
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
```

Figure 43. Cataloged Procedure FORTVC—Compile Only

```
//FORTVCL  PROC FVPGM=FORTVS,FVREGN=1200K,FVPDECK=NODECK,
//         FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',
//         PGMNAME=MAIN,PGMLIB='&&GOSET',FVLNSPC='3200,(25,6)'
//*
//*            PARAMETER  DEFAULT-VALUE      USAGE
//*
//*              FVPGM    FORTVS             COMPILER NAME
//*              FVREGN   1200K              FORT-STEP REGION
//*              FVPDECK  NODECK             COMPILER DECK OPTION
//*              FVPOLST  NOLIST             COMPILER LIST OPTION
//*              FVPOPT   0                  COMPILER OPTIMIZATION
//*              FVTERM   SYSOUT=A           FORT.SYSTERM OPERAND
//*              FVLNSPC  3200,(25,6)        FORT.SYSLIN SPACE
//*              PGMLIB   &&GOSET            LKED.SYSLMOD DSNAME
//*              PGMNAME  MAIN               LKED.SYSLMOD MEMBER NAME
//*
//FORT   EXEC  PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//             PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT     DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM      DD &FVTERM
//SYSPUNCH     DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN       DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//LKED   EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD      DD DSN=&PGMLIB.(&PGMNAME),DISP=(,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//             DD DDNAME=SYSIN
```

Figure 44. Cataloged Procedure FORTVCL—Compile and Link-Edit

```
//FORTVCLG PROC FVPGM=FORTVS,FVREGN=1200K,FVPDECK=NODECK,
//              FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',GOREGN=100K,
//              FVLNSPC='3200,(25,6)',
//              GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',
//              GOF7DD='SYSOUT=B'
//*
//*             PARAMETER  DEFAULT-VALUE     USAGE
//*
//*               FVPGM    FORTVS            COMPILER NAME
//*               FVREGN   1200K             FORT-STEP REGION
//*               FVPDECK  NODECK            COMPILER DECK OPTION
//*               FVPOLST  NOLIST            COMPILER LIST OPTION
//*               FVPOPT   0                 COMPILER OPTIMIZATION
//*               FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*               FVLNSPC  3200,(25,6)       FORT.SYSLIN SPACE
//*               GOREGN   100K              GO-STEP REGION
//*               GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*               GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*               GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*
//FORT   EXEC   PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//              PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM       DD &FVTERM
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//LKED   EXEC   PGM=IEWL,REGION=200K,COND=(4,LT),
//              PARM='LET,LIST,MAP,XREF'
//SYSPRINT      DD SYSOUT=A
//SYSLIB        DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1        DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD       DD DSN=&&GOSET(MAIN),DISP=(,PASS),UNIT=SYSDA,
//              SPACE=(TRK,(10,10,1),RLSE)
//              DD DDNAME=SYSIN
//GO     EXEC   PGM=*.LKED.SYSLMOD,REGION=&GOREGN,COND=(4,LT)
//STEPLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001      DD &GOF5DD
//FT06F001      DD &GOF6DD
//FT07F001      DD &GOF7DD
```

Figure 45. Cataloged Procedure FORTVCLG—Compile, Link-Edit, and Execute

```
//FORTVLG PROC LKLNDD='DDNAME=SYSIN',GOPGM=MAIN,GOREGN=100K,
//             GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B'
//*
//*
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*              LKLNDD   DDNAME=SYSIN      LKED.SYSLIN OPERAND
//*              GOPGM    MAIN              OBJECT PROGRAM NAME
//*              GOREGN   100K              GO-STEP REGION
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*
//*
//LKED   EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD      DD DSN=&&GOSET(&GOPGM),DISP=(,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD &LKLNDD
//GO     EXEC  PGM=*.LKED.SYSLMOD,REGION=&GOREGN,
//             COND=(4,LT,LKED)
//STEPLIB      DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 46. Cataloged Procedure FORTVLG—Link-Edit and Execute

```
//FORTVG PROC  GOPGM=MAIN,GOREGN=100K,
//             GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B'
//*
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*
//*              GOPGM    MAIN              PROGRAM NAME
//*              GOREGN   100K              GO-STEP REGION
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 DD OPERAND
//*              GOF6DD   SYSOUT=A          GO.FT06F001 DD OPERAND
//*              GOF7DD   SYSOUT=B          GO.FT07F001 DD OPERAND
//*
//*
//GO     EXEC  PGM=&GOPGM,REGION=&GOREGN,COND=(4,LT)
//STEPLIB      DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 47. Cataloged Procedure FORTVG—Execute Only

```
//FORTVCG  PROC FVPGM=FORTVS,FVREGN=1200,FVPDECK=NODECK,
//             FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',
//             FVLNSPC='3200,(25,6)',
//             GOF5DD='DDNAME=SYSIN',
//             GOF6DD='SYSOUT=A',
//             GOF7DD='SYSOUT=B',GOREGN=100K
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*
//*              FVPGM    FORTVS            COMPILER NAME
//*              FVREGN   1200              FORT-STEP REGION
//*              FVPDECK  NODECK            COMPILER DECK OPTION
//*              FVPOLST  NOLIST            COMPILER LIST OPTION
//*              FVPOPT   0                 COMPILER OPTIMIZATION
//*              FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*              FVLNSPC  3200,(25,6)       FORT.SYSLIN SPACE
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*              GOREGN   100K              GO-STEP REGION
//*
//FORT   EXEC  PGM=&FVPGM,REGION=&FVREGN,COND=(4,LT),
//             PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT)'
//SYSPRINT     DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM      DD &FVTERM
//SYSPUNCH     DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN       DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//GO     EXEC  PGM=LOADER,REGION=&GOREGN,COND=(4,LT),
//             PARM='LET,NORES,EP=MAIN'
//STEPLIB      DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSLIN       DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSLOUT      DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 48. Cataloged Procedure FORTVCG—Compile and Load

```
//FORTVL   PROC GOF5DD='DDNAME=SYSIN',
//              GOF6DD='SYSOUT=A',
//              GOF7DD='SYSOUT=B',GOREGN=100K
//*
//*
//*             PARAMETER  DEFAULT-VALUE     USAGE
//*
//*               GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*               GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*               GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*               GOREGN   100K              GO-STEP REGION
//*
//*
//GO      EXEC  PGM=LOADER,REGION=&GOREGN,
//              PARM='LET,NORES,EP=MAIN'
//STEPLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSLOUT       DD SYSOUT=A
//SYSLIB        DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001      DD &GOF5DD
//FT06F001      DD &GOF6DD
//FT07F001      DD &GOF7DD
```

Figure 49. Cataloged Procedure FORTVL—Load Only

Cataloging Your Source

You can create partitioned data sets for use in your SYSLIB data set. You can then catalog your source program, and source statement sequences you'll use in FORTRAN INCLUDE statements, as members in that library.

The library in which you catalog the source programs or statement sequences is SYSLIB. Then, when you compile a program using the FORTRAN INCLUDE statement, you must specify SYSLIB in a DD statement.

Requesting Compilation

In one job step, you can request compilation for a single source program or for a series of source programs.

Compiling a Single Source Program

For compiling a single source program, the sequence of job control statements is:

```
//JOB Statement
//EXEC Statement (to execute the VS FORTRAN compiler)
//DD Statements for Compilation (as required)

  (Source program to be compiled)

/*Data Delimiter Statement (only if source program is on cards)
//End-of-Job Statement
```

Job control statements are described under "Job Processing" on page 257.

Batch Compilation of More Than One Source Program

You can compile more than one source program during the execution of one job. The sequence of job control statements you use is:

```
//JOB Statement
//EXEC Statement (to execute the VS FORTRAN compiler)
//DD Statements (as required)
@PROCESS Statement (if needed to modify compiler options)

  (First source program to be compiled)

@PROCESS Statement (if needed to modify compiler options)

  (Second source program to be compiled)

@PROCESS Statement (if needed to modify compiler options)

  (Third source program to be compiled)

/*Data Delimiter Statement (only if source program is on cards)
//End-of-Job Statement
```

These job control statements are decribed under "Job Processing" on page 257.

The @PROCESS statement is described in "Modifying Compilation Options—@PROCESS Statement" on page 164.

Requesting Compilation Only—MVS

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

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

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

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

Printing on the IBM 3800 Printing Subsystem under MVS

Additional run-time parameters are required to support the IBM 3800 Printing Subsystem under MVS. These parameters are the DCB and CHARS parameters on the GO statement.

```
//ddname DD ...
//        ,DCB=(OPTCD=J)
//        ,CHARS=(cat0[,cat1,...])
```

DCB=(OPTCD=J)
tells JES to interpret the second byte of each record as a table reference character, TRC, rather than as the first printable character. The existence of a TRC must be accounted for in the record length specified for the record. For a variable spanned record, only the first segment can contain a TRC.

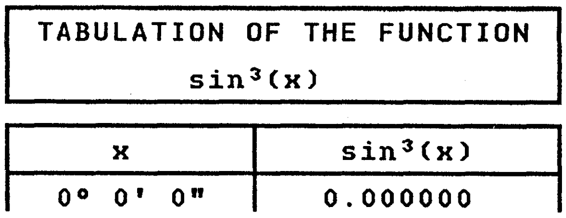
CHARS=(cat0[,cat1, ...])

identifies the character arrangement table to be used to print each line. The tables are described in *IBM 3800 Printing Subsystem Programmer's Guide.* Up to four names can be specified. A TRC is an index (0 to 3) into this list of names.

A sample FORTRAN program using the 3800 follows:

```
C   SAMPLE PROGRAM FOR THE IBM 3800 PRINTING SUBSYSTEM
C
 100  FORMAT ('12','   W66666666666666666666666666666X'
     1    / ' 2','     7                             7'
     2    / '+1','       TABULATION OF THE FUNCTION  '
     3    / ' 2','     7                             7'
     4    / ' 2','     7                             7'
     5    / '+0','               sin',A1,'(x)      '
     6    / ' 2','     Z66666666666666666666666666666Y'
     7    / ' 2','     W666666666666166666666666666666X'
     8    / ' 2','     7            7                7'
     9    / '+0','           x          sin',A1,'(x) '
     A    / ' 2','     366666666666656666666666666664')

 200  FORMAT (' 2','   7            7                7'
     1    / '+0','     ',I3,A1,I2,'''',I2,'"',5X,F9.6)

      CHARACTER*1 DEG, U3
      DATA DEG/ZA1/, U3/ZB3/

      WRITE (6,100) U3, U3

      WRITE (6,200) 0, DEG, 0, 0, 0.

      END
      .
      .
      .
//GO.FT06F001 DD DCB=(OPTCD=J),CHARS=(TN,GS10,FM10),...
```

The sample program above produces the following 3800 output:



```
TABULATION OF THE FUNCTION
         sin³(x)

      x          sin³(x)
 0° 0' 0"        0.000000
```

IBM Extension

Using the FORTRAN INCLUDE Statement

If your source program uses the INCLUDE statement, you must create a library containing the INCLUDE source code.

1. Create one or more members of a partitioned data set. This can be done in a similar fashion to that used for creating your source program.

2. Create a FORTRAN source program.

```
    .
    .
    .
INCLUDE (member1)
Z=A1 * B1
    .
    .
    .
END
```

3. Define the library to the system (in this case, MVS batch) by means of the SYSLIB ddname:

```
//FORT.SYSLIB DD DSN=user.lib.fort,DISP=SHR
```

   In this case, *member1* of *user.lib.fort* is brought into your inline source program.

Blocked INCLUDE

The included files may be fixed blocked or fixed unblocked by specifying RECFM FB or RECFM F when you create the members.

End of IBM Extension

Compilation Data Sets

For compilation under MVS, there are two required data sets and several optional ones.

Required Compiler Data Sets

You must ensure that the following ddnames are available during compilation. They may be available through the cataloged procedure you're using for compilation (check with your system administrator). If they aren't, you must specify them through a DD statement:

- SYSIN—to define the source input data set

- SYSPRINT—to define the printed output data set

Optional Compiler Data Sets

You specify the following data sets, through a DD statement, only if you're requesting specific compilation features:

- SYSLIN—if you're requesting that an object module be produced (through the OBJECT compiler option); it defines the object module data set.

- SYSPUNCH—if you're requesting an object module punched on cards (through the DECK compiler option); it defines the card image data set on which the object module is punched.

- SYSTERM—when the TERMINAL compiler option is in effect, you specify SYSTERM as the data set that will contain printed compiler output (that is, error messages and compiler statistics).

- SYSLIB—if your source program uses the INCLUDE statement, this ddname defines the library input data set.

Compiler Output

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

- The source program listing—as you entered it, but with compiler-generated internal sequence numbers prefixed at the left; the sequence numbers identify the line-numbers referred to in compiler messages.

- An object module—a translation of your program in machine code.

- Messages about the results of the compilation.

- Other listings helpful in debugging.

These listings are described in "Identifying User Errors" on page 167, and Chapter 10, "Sample Programs and Subroutines" on page 213; examples of output for each feature are also given.

If your compilation was completed without error messages, see your appropriate operating system chapter for information on link-editing.

If your compilation caused error messages to appear online, you may have to fix up your errors, as described in "Identifying User Errors" on page 167.

Compiler Data Sets

The compiler uses up to eight system data sets. Many of these data sets are defined in cataloged procedures. Figure 50 lists the function, device types, and allowable device classes for each data set.

ddname	Function	Device Types	Device Class	Defined
SYSIN	Reading input source data set	Card reader Magnetic tape Direct access	Input stream (defined as DD *, or DD DATA)	No
SYSLIB	Reading INCLUDE data sets (Required if INCLUDE is specified)	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
SYSPRINT	Writing source, object, and cross reference listings, storage maps, messages	Printer Magnetic tape Direct access	A SYSSQ SYSDA	Yes
SYSPUNCH[2]	Punching the object module deck (Required if DECK is specified)	Card punch Magnetic tape Direct access	B SYSCP SYSSQ SYSDA	Yes
SYSUDUMP or SYSABEND	Writing dump in event of abnormal termination (Required if DUMP is specified)	Printer Magnetic tape Direct access	A SYSSQ SYSDA	No
SYSTERM	Writing error message and compiler statistics (Required if TERM or TRMFLG is specified)	Printer Magnetic tape Direct access	A SYSSQ SYSDA	Yes

Figure 50. Compiler Data Sets

Notes to Figure 50:

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.

DCB Default Values

The DCB subparameters define record characteristics of a data set. Figure 51 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	-

Figure 51. Compiler Data Set DCB Default Values

Note to Figure 51 :

1 These default block size values correspond to the BLKSIZE values specified on the DD statements in the distributed cataloged procedures. The compiler defaults to:

BLKSIZE = LRECL

Execution-Time Loading of Library Modules

In Release 4.0, all library modules (other than the mathematical routines) can be either link-edited into the load module with compiler-generated code, or loaded dynamically at execution time. Execution-time loading has several advantages. It reduces auxiliary storage requirements for load modules, speeds execution in compile-link-go mode, and, in an MVS/XA environment, allows some library routines to be placed in the extended link pack area.

This new feature replaces the previous technique used to load the reentrant library. In order to maintain compatibility with the Release 2 and 3 load modules that use IFYVRENT, the Release 4.0 library includes an IFYVRENT module with all necessary maintenance functions, but no new Release 4.0 functions. After your load module contains *any* code compiled with Release 4.0 or *any* Release 4.0 library modules, then *all* library modules linked into that load module must be at the Release 4.0 level. The former IFYVRENT mechanism will then no longer be used for that load module.

Selection of Link Mode or Load Mode

During installation of the VS FORTRAN library, your system programmer may have specified the libraries needed for use 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.

Specifying Libraries in Load Mode

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

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

- To execute a program link-edited in load mode, make VFORTLIB available for the execution step by performing one of the following steps.

  1. Concatenate SYS1.VFORTLIB to SYS1.LINKLIB in the system link list so that SYS1.VFORTLIB will be searched as part of the link library without JOBLIB or STEPLIB DD statements. The reentrant composite modules IFYVRENA (MVS/XA only), IFYVRENB (MVS/XA only), and IFYVRENC (non-XA), as well as selected individual reentrant modules, may be placed in the link pack area (SYS1.LPALIB). The copy of the modules in the link pack area will be used without searching SYS1.VFORTLIB. (If maintenance affects any modules in the link pack area, the updated copies of the modules must be copied into the link pack area from SYS1.VFORTLIB.)

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

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

     or place the following STEPLIB DD statement in the JCL for the step which executes the VS FORTRAN program:

     ```
     //STEPLIB  DD   DSN=SYS1.VFORTLIB,DISP=SHR
     ```

     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* or *MVS/Extended Architecture Supervisor Services and Macro Instructions* in the discussion of program management.)

  3. If you want to use a step library or job library in addition to loading reentrant modules from the link pack area, you must do the following:

     a. After tailoring the composite modules, place the reentrant composite modules IFYVRENA (MVS/XA only), IFYVRENB (MVS/XA only), and IFYVRENC (non-XA) in the link pack area (library SYS1.LPALIB).

     b. Optionally, place any reentrant modules that are *not* in a composite module into the link pack area.

     c. Create a new library that contains all modules from SYS1.VFORTLIB *less* the modules (either composite modules or individual modules) that have been placed in the link pack area. Make this library available as a step library or as a job library for the execution of the VS FORTRAN program.

If maintenance affects any of the modules in the link pack area or your new library, then the updated modules must be copied from SYS1.VFORTLIB.

For examples of link-edit JCL, see Figure 43 on page 265 through Figure 49 on page 270.

Specifying Libraries in Link Mode

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

```
//SYSLIB   DD  DSN=SYS1.VLNKMLIB,DISP=SHR
//         DD  DSN=SYS1.VFORTLIB,DISP=SHR
```

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

Link-Editing Your Program

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

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

The following sections also show how to catalog your object module or load module, and how to use the linkage editor or loader.

Note: For TSO considerations on loading and executing your program, see Chapter 13, "Using VS FORTRAN under TSO" on page 317.

Automatic Cross-System Support

In VS FORTRAN, you can compile your source program under any supported operating system. You can then link-edit the resulting object module under the same system, or under any other supported system.

For example, you could request compilation under VM and then link-edit the resulting object module for execution under VSE.

You don't have to request anything special during compilation to do this; VS FORTRAN uses the execution-time library for all system interfaces, so the operating system under which you link-edit determines the system under which you execute.

Linkage Editor Input

Your input to the linkage editor can be the object module in machine-language format (which you request through the OBJECT compiler option), or as a machine-language input data set (which you request through the DECK compiler option).

You request the DECK option when you want to catalog the object module and save it for future link-edit runs.

You request the OBJECT option when you want to combine the link-edit task with the compilation task. You can then catalog and/or execute the load module produced.

The data set is a copy of the object module, in card image format, which consists of dictionaries, text, and an end-of-module indicator. (See Appendix B, "Object Module Records" on page 403, for additional detail.)

You can create partitioned data sets for use in your SYSLIB data set. You can then catalog your object module as a member in that library.

The library in which you catalog your object module is SYSLIB. Then, when you link-edit and execute, you must specify SYSLIB in a DD statement.

Cataloging Your Object Module

You request an object module data set by specifying the DECK or OBJECT compiler option.

You can use the object data set as input to the linkage editor or loader in a later job step, or you can catalog it for later reference.

The data set is a copy of the object module, in card image format, which consists of dictionaries, text, and an end-of-module indicator. (See "Object Module as Link-Edit Data Set" on page 183 for additional details.)

After you've created the object module data set, you can catalog it in a system or private library for future reference.

You can create partitioned data sets for use in your SYSLIB data set. You can then catalog your object module as a member in that library. The library in which you catalog your object module is SYSLIB. Then, when you link-edit and execute, you must specify SYSLIB in a DD statement.

For TSO considerations, see Chapter 13, "Using VS FORTRAN under TSO" on page 317.

Cataloging Your Load Module

You can catalog the data set containing the load module by defining it in a SYSLMOD DD statement during link-edit processing.

Executing 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 executing immediately.

VS FORTRAN supplies you with cataloged procedures that let you link-edit or load your programs easily. See "Using and Modifying Cataloged Procedures" on page 264 for details.

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 load module execution, even when abnormal conditions have been detected that could cause execution to fail.

NCAL—specifies that the linkage editor is not to attempt to resolve external references.

If your program attempts to call external routines, you'll get an abnormal termination.

LIST—specifies that the linkage editor control statements are to be listed in the SYSPRINT data set.

OVLY—specifies that the load module is to be in overlay format. That is, that segments of the program will share the same storage at different times during processing. (For more details, see Chapter 6, "Subprograms and Shared Data" on page 113.)

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.VFORTLIB or VALTLIB 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 and the compiler statistics listing. This data set can be on a direct access, magnetic tape, or printer device.

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.

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
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 52 lists the function, device types, and allowable device classes for each linkage editor data set.

ddname	Function	Device Types	Device Class	Defined.
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.FORTLIB)	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
User-defined	Additional libraries and object modules	Direct access Magnetic tape	SYSDA SYSSQ	No

Figure 52. Linkage Editor Data Sets

Note to Figure 52:

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

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 execution. The loader combines your object module with other modules into one load module, and then places the load module into main storage and executes it.

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

Loader Options

When you execute the loader, you can specify the following options through the PARM parameter of the EXEC statement:

MAP | NOMAP—specifies whether a map of the load module is to be produced on SYSPRINT, giving the length and location of the main program and all subprograms.

LET | NOLET—specifies whether the linkage editor is to allow load module execution, even when abnormal conditions that could cause execution to fail have been detected.

CALL | NCAL—specifies whether or not the loader is to attempt to resolve external references. If you specify NCAL and your program attempts to call external routines, you'll get an abnormal termination.

SIZE—lets you specify the amount of storage to be allocated for loader processing.

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

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

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

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

Loader Data Sets

The loader normally 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) and the SYSPRINT data set (used for printed output) are available.

In addition, depending on what you want the loader to do for you, you can, optionally, specify the data sets in Figure 53 on page 285. This figure lists the function, device types, and allowable device classes for each data set.

ddname	Function	Device Types	Device Class	Defined
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.FORTLIB)	Direct access	SYSDA	Yes
SYSLOUT	Writing listings	Printer Magnetic tape Direct access	A SYSSQ	Yes
SYSPRINT	Writing messages	Printer Magnetic tape Direct access	A SYSSQ	No
SYSIN	Input data to load module function	Card reader Magnetic tape Direct access	Input stream (defined as DD *) SYSSQ SYSDA	No
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 53. Loader Data Sets

Notes to Figure 53:

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

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 execution job step executes a load module. The load module may be passed directly from a preceding link-edit job step, or it may be called from a library of programs, or it may form part of the loader job step.

The load module execution job step may use many data sets. Figure 54 on page 286 lists the function, device types, and usage for each data set.

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

Figure 54. Load Module Execution Data Sets

User-Defined Data Sets

You must define the SYSIN DD statements to complete the description of the input data set begun by the DD statement FT05F001. If no input data set is to be submitted, omit the SYSIN DD statement, and the operating system will treat the FT05F001 DD statement as though DD DUMMY had been specified.

You must also ensure that the JCL for the load module execution step includes a DD statement for error code diagnostic output. This DD statement must be present when the data set is opened for each execution of a FORTRAN load module. Usually, the FT06F001 DD statement defines this data set; however, this assignment can be modified when VS FORTRAN is installed. Check with your system administrator.

DCB Default Values

Figure 55 lists the DCB default values for load module execution **sequential** 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 55. Load Module Execution Sequential Data Set DCB Default Values

Notes to Figure 55:

1 For records not under FORMAT control, the default is VS.

2 For records not under FORMAT control, the default is 4 less than shown.

Figure 56 lists the DCB default values for load module execution *direct access* data sets.

ddname	RECFM	LRECL or BLKSIZE	BUFNO
FT05Fyyy	F	The value specified as the maximum size	2
FT06Fyyy	F	of a record in the OPEN statement.	2
FT07Fyyy	F		2
all others	F		2

Figure 56. Load Module Execution Direct Access Data Set DCB Default Values

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

Using Load Module Data Sets

If you're using cataloged procedures, or if you're using the device assignments as shipped by IBM, you must use the DD names shown in Figure 54 on page 286.

Making the VS FORTRAN Library Available at Execution

The load module requires the VS FORTRAN library during execution in MVS under the following circumstances:

1. Specify the following in job control language if you're operating in load mode:

   ```
   //STEPLIB DD DSN=SYS1.VFORTLIB,DISP=SHR
   ```

2. If you have a load module created from a version of the VS FORTRAN library prior to Release 4.0 that used the reentrant I/O library facility, specify routines in SYS1.VFORTLIB rather than in SYS1.VRENTLIB as in past releases.

Using Cataloged Load Modules

You can execute cataloged load modules using either a STEPLIB DD or a JOBLIB DD statement.

Using JOBLIB DD: If you specify a JOBLIB DD statement for the load module, the JOBLIB library is available through all job steps of the job.

To ensure that the library remains available, you must specify the JOBLIB DD statement immediately after the JOB statement.

Using STEPLIB DD: If you specify a STEPLIB DD statement for the load module, the STEPLIB library is available for only this one step of the job.

You can place the STEPLIB DD statement anywhere among the DD statements for this job step.

Specifying Execution-Time Options

To specify an execution-time option (XUFLOW, NOXUFLOW, DEBUG, or NODEBUG), use the following method:

```
//ANY EXEC PGM=MAIN,PARM='NOXUFLOW'
```

For more information, see "Using the Execution-Time Options" on page 200.

Executing the Load Module

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

VS FORTRAN supplies you with cataloged procedures that let you compile, link-edit or load, and/or execute easily. See "Using and Modifying Cataloged Procedures" on page 264 for details.

Execute Only

The job control statements you use are:

```
//JOB Statement
//EXEC Statement         (load module)
//DD Statements          (as required for execution)

  (Input data to be processed)

/*End-of-Data Statement (if input data is on cards)
//End-of-Job Statement
```

Link-Edit and Execute

The job control statements you use are:

```
//JOB Statement
//EXEC Statement        (linkage editor)
//DD Statements         (as required for linkage editing)

  (Link-edit is performed)

//EXEC Statement        (load module)
//DD Statements         (as required for execution)

  (Input data to be processed)

/*Statement             (if input data is on cards)
//End-of-Job Statement
```

Compile, Link-Edit, and Execute

The job control statements you use are:

```
//JOB Statement
//EXEC Statement        (VS FORTRAN Compiler)
//DD Statements         (as required for compilation)

  (Source program to be compiled)

/*End-of-Data Statement (if source program is on cards)
//EXEC Statement        (linkage editor)
//DD Statements         (as required for link-editing)

  (Link-edit is performed)

//EXEC Statement        (load module)
//DD Statements         (as required for load module execution)

  (Input data to be processed)

/*End-of-Data Statement (if input data is on cards)
//End-of-Job Statement
```

Load Module Execution-Time Output

The output that execution of your load module gives you depends upon whether or not there are errors in your program.

Execution without Errors

If your program executes without any errors and gives the results you expect, your task of program development is completed.

Execution with Errors

When your program has errors in it, your execution-time output may be incorrect, or nonexistent.

You may or may not get error messages as well. Any VS FORTRAN execution-time error messages you get come from the VS FORTRAN Library.

If you get output from the program itself, it may be exactly what you expected, or (if there are logic errors in the program) it may be output you didn't expect at all. When this happens, you must proceed to the next step in program development, described in "Identifying User Errors" on page 167.

Requesting an Abnormal Termination Dump

Program interrupts causing abnormal termination produce a dump, called an indicative dump, which displays the completion code and the contents of registers and system control fields.

To display the contents of main storage as well, you must request an abnormal termination (ABEND) dump by including a SYSUDUMP DD statement in the appropriate job step. The following example shows how the statement may be specified for IBM-supplied cataloged procedures:

```
//GO.SYSUDUMP  DD  SYSOUT=A
```

Information on interpreting dumps is found in the appropriate debugging guide, as listed under "Preface" on page iii.

Using the VS FORTRAN Separation Tool

If you have compiled your program with the RENT compiler option, and wish to run it in a reentrant fashion, the separation tool is located in SYS1.VFORTLIB. The separation tool consists of two modules, IFYVSFST and IFYVSFIO.

For more general information about the separation tool, see "VS FORTRAN Separation Tool (for Both VM and MVS)" on page 189.

To invoke the separation tool in batch mode, the following Job Control Language (JCL) is needed.

```
//SEPARATE JOB (1,1),'MYJOB'
//SEP       EXEC PGM=IFYVSFST
//STEPLIB   DD   DSN=SYS1.VFORTLIB,DISP=SHR
//SYSPRINT DD    SYSOUT=A
//SYSUT1    DD   DSN=&NRENT,DISP=(NEW,PASS),UNIT=SYSDA,
//               SPACE=(3200,(25,1)),DCB=BLKSIZE=3200
//SYSUT2    DD   DSN=&RENT,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=userid.pgmname.OBJ,DISP=SHR
```

This JCL only separates the input file identified by ddname SYSIN, and puts the results into files &RENT and &NRENT. To add these object files to a load library, the following step must be run to link-edit the reentrant portion of the program. To retain a record of the location of the reentrant modules, keep the SYSPRINT output from this step.

```
//LKEDR     EXEC  PGM=IEWL,PARM='XREF,LET,LIST,RENT,REUS'
//SYSPRINT DD     SYSOUT=A
//SYSUT1    DD    UNIT=SYSDA,SPACE=(TRK,(10,1))
//SYSLIN    DD    DSN=&RENT,DISP=(OLD,PASS)
//SYSLIB    DD    DSN=SYS1.VFORTLIB,DISP=SHR
//SYSLMOD   DD    DSN=&GOFILE,DISP=(NEW,PASS),UNIT=SYSDA,
//                SPACE=(TRK,(10,10,5))
```

The following JCL describes the procedure to link-edit the nonreentrant portion.

```
//LKEDN     EXEC  PGM=IEWL,PARM='XREF,LET,LIST'
//SYSPRINT DD     SYSOUT=A
//SYSUT1    DD    UNIT=SYSDA,SPACE=(TRK,(10,1))
//SYSLIN    DD    DSN=&NRENT,DISP=(OLD,PASS)
//SYSLIB    DD    DSN=SYS1.VFORTLIB,DISP=SHR
//SYSLMOD   DD    DSN=&GOFILE(MAIN),DISP=(MOD,PASS)
```

To execute your program, use the following JCL:

```
//GO        EXEC  PGM=MAIN
//STEPLIB   DD    DSN=&GOFILE,DISP=(OLD,PASS)
//          DD    DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001 DD     DDNAME=SYSIN
//FT06F001 DD     DDNAME=SYSPRINT
```

You must supply GO.SYSIN and GO.SYSPRINT DD statements for the conditions of your system and your program. Note that this is not the only way to process your compiler text file. Further information can be found in the section below.

Advanced Topics

This section tells more about the separation tool, its implementation, and its usage, and gives some information about using the reentrant portion of the compiler output text file.

The separation tool is composed of two independent but related CSECTs. Both IFYVSFST and IFYVSFIO are reentrant and thus may be installed in an MVS link pack area (LPA) or in an MVS/XA LPA (IFYVSFIO) and in the MVS/XA extended LPA (IFYVSFST). The separation tool is invoked by calling IFYVSFST.

The calling program may supply up to two parameters. The parameters are in the standard MVS format of a pointer pointing to a halfword length, followed by the string of data. The first parameter is the module name to associate with the reentrant CSECTs in the input text file; this may be from 1 to 8 characters in length. All characters after the first 8 characters are ignored. The second parameter is the list of alternate ddnames for the program in the standard format.

Examples are as follows:

```
Register 1 points to
  Parameter 1 pointer, which points to
          length/data string
             0008RENTPART
  Parameter 2 pointer, which points to
          length/data string
             0050ddname list   (hex 0050 is decimal 80)
```

The high-order bit of the last parameter in the list must be on. If it isn't and you do not have a ddname list, the separation tool will fail.

Register 1 should be zero if no parameters are specified.

The order of the ddnames for substitution is as follows:

Displacement	**Original ddname**
0	Not used
8	Not used
16	Not used
24	Not used
32	SYSIN
40	SYSPRINT
48	Not used
56	SYSUT1
64	SYSUT2
72	SYSUT3

Positions for the "not used" entries must be replaced by 8 bytes of binary zeros. If you do not wish to change ddname SYSIN, for example, just put binary zeros in the 8 bytes that the replacement would occupy. Replacement ddnames must be 1 to 8 characters in length, left-justified in the field, and padded on the right with blanks. The maximum list length is 80 characters (decimal) for all the entries. If you want to change fewer entries, you may enter a shorter list with a smaller length and fewer entries. For example, if you wish only to change the SYSIN and SYSPRINT ddnames, a length of 48 characters would be the minimum number needed. You may always pass a length of 80 characters, but the unchanged ddnames must then be zeros.

Messages issued by the separation tool are documented in the *VS FORTRAN Language and Library Reference*.

A Simple Scenario That Might Occur

The tool expects the text file to contain entries in a certain order: the reentrant CSECT followed by the corresponding nonreentrant CSECT. If this does not occur because of a routine's not being compiled for reentrancy, the tool will accommodate the difference.

A simple scenario that might occur is: Given a simple FORTRAN program with a single program named MAIN, the VS FORTRAN compiler would generate the following:

- A reentrant CSECT with the name @MAIN
- A nonreentrant CSECT with the name MAIN

All this output is in the same text file and, when loaded, all address constants will be properly resolved and the program will run as it has been coded. If, however, the text file is passed to the separation tool, the following is output:

- A file containing the nonreentrant CSECT, MAIN
- A file containing two CSECTs: IFYZRENT and the reentrant CSECT, @MAIN

There are two ways of invoking the separation tool (IFYVSFST). The first is without any module name parameter; the second is with a module name parameter.

With the module name parameter, RENTPART, for example, the following is output:

- A file containing the nonreentrant CSECT, MAIN, but with an extra record with the module name (RENTPART) in it
- A file containing the reentrant CSECTs, IFYZRENT and @MAIN, and a record with "NAME RENTPART(R)" on it

Without the module name parameter, the following is output:

- A file containing the nonreentrant CSECT, MAIN
- A file containing the reentrant CSECTs, IFYZRENT AND @MAIN, and a record with "NAME @MAIN(R)" on it

Both text files must be passed separately to the linkage editor with the correct parameters for the linkage editor. After the files are link-edited successfully, the modules can be passed to the operating system for invocation and running. The GO step must point to the correct libraries to ensure the modules are found.

The table below provides an index to a number of cataloged procedures that you can use to compile, separate, link-edit, and execute your VS FORTRAN programs. Figure 57 on page 294 through Figure 59 on page 298 contain these procedures in *load* mode.

Action	**Procedure**	**Figure**
Compile, separate, and link	VFTNRCL	Figure 57
Compile, separate, link, and go	VFTNRCLG	Figure 58
Separate, link, and go	VFTNRLG	Figure 59

```
//VFTNRCL  PROC FVPGM=FORTVS,FVREGN=1200K,FVPDECK=NODECK,
//         FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',FVNAME='RENTPART',
//         FVLNSPC='3200,(25,6)',FVRENT='RENT',GONAME='MAIN'
//*
//*
//*********************************************************************
//*  PROC DESCRIPTION: FORTRAN COMPILE, SEPARATE AND LINK             *
//*********************************************************************
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*
//*              FVPGM    FORTVS            COMPILER NAME
//*              FVREGN   1200K             FORT-STEP REGION
//*              FVPDECK  NODECK            COMPILER DECK OPTION
//*              FVPOLST  NOLIST            COMPILER LIST OPTION
//*              FVPOPT   0                 COMPILER OPTIMIZATION
//*              FVTERM   SYSOUT=A          FORT.SYSTERM OPERAND
//*              FVLNSPC  3200,(26,6)       SPACE ALLOCATION
//*              FVNAME   RENTPART          REENTRANT MODULE NAME
//*              FVRENT   RENT              RENT COMPILER OPTION
//*              GONAME   MAIN              EXECUTION MODULE NAME
//*
//*            COMPILATION STEP
//*
//FORT   EXEC  PGM=&FVPGM,REGION=&FVREGN,
//             PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT),&FVRENT'
//STEPLIB      DD DSN=SYS1.FORTVS,DISP=SHR
//SYSPRINT     DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM      DD &FVTERM
//SYSPUNCH     DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN       DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
```

Figure 57 (Part 1 of 2). JCL Procedure to Compile, Separate, and Link

```
//*
//*            SEPARATION STEP
//*
//SEPARATE     EXEC PGM=IFYVSFST,PARM='&FVNAME',COND=(4,LT)
//STEPLIB      DD  DSN=SYS1.VFORTLIB,DISP=SHR
//SYSPRINT     DD  SYSOUT=A
//SYSIN        DD DSN=&&LOADSET,DISP=(OLD,PASS)
//SYSUT1       DD DSN=&NRENT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//SYSUT2       DD DSN=&RENT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//SYSUT3       DD DSN=&TEMP,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//*
//*            NON-REENTRANT PART LINKEDIT
//*
//LKEDN  EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD      DD DSN=&&GOSET(&GONAME),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD DSN=&NRENT,DISP=(OLD,DELETE)
//             DD DDNAME=SYSIN
//*
//*            REENTRANT PART LINKEDIT
//*
//LKEDR  EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF,RENT,NCAL'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD      DD DSN=&&GOSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD DSN=&RENT,DISP=(OLD,DELETE)
//             DD DDNAME=SYSIN
```

Figure 57 (Part 2 of 2). JCL Procedure to Compile, Separate, and Link

```
//VFTNRCLG PROC FVPGM=FORTVS,FVREGN=1400K,FVPDECK=NODECK,
//              FVPOLST=NOLIST,FVPOPT=0,FVTERM='SYSOUT=A',GOREGN=500K,
//              FVLNSPC='3200,(25,6)',FVRENT=RENT,GONAME=MAIN
//              GOF5DD='DDNAME=SYSIN',GOF6DD='SYSOUT=A',
//              GOF7DD='SYSOUT=B',FVNAME='RENTPART'
//*
//*********************************************************************
//*  PROC DESCRIPTION: REENTRANT COMPILATION,SEPARATE,LKED AND GO     *
//*********************************************************************
//*
//*             PARAMETER  DEFAULT-VALUE      USAGE
//*
//*               FVPGM    FORTVS             COMPILER NAME
//*               FVREGN   880K               FORT-STEP REGION
//*               FVPDECK  NODECK             COMPILER DECK OPTION
//*               FVPOLST  NOLIST             COMPILER LIST OPTION
//*               FVPOPT   0                  COMPILER OPTIMIZATION
//*               FVTERM   SYSOUT=A           FORT.SYSTERM OPERAND
//*               FVLNSPC  3200,(25,6)        FORT.SYSLIN SPACE
//*               FVNAME   RENTPART           REENTRANT MODULE NAME
//*               FVRENT   RENT               REENTRANT OPTION
//*               GOREGN   100K               GO-STEP REGION
//*               GOF5DD   DDNAME=SYSIN       GO.FT05F001 OPERAND
//*               GOF6DD   SYSOUT=A           GO.FT06F001 OPERAND
//*               GOF7DD   SYSOUT=B           GO.FT07F001 OPERAND
//*               GONAME   MAIN               EXECUTION TIME NAME
//*
//*             COMPILATION STEP
//*
//FORT    EXEC  PGM=&FVPGM,REGION=&FVREGN,
//              PARM='&FVPDECK,&FVPOLST,OPT(&FVPOPT),&FVRENT'
//STEPLIB       DD DSN=SYS1.FORTVS,DISP=SHR
//SYSPRINT      DD SYSOUT=A,DCB=BLKSIZE=3429
//SYSTERM       DD &FVTERM
//SYSPUNCH      DD SYSOUT=B,DCB=BLKSIZE=3440
//SYSLIN        DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//              SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
```

Figure 58 (Part 1 of 2). JCL Procedure to Compile, Separate, Link-Edit, and Go

```
//*
//*            SEPARATION STEP
//*
//SEPARATE     EXEC PGM=IFYVSFST,PARM='&FVNAME',COND=(4,LT)
//STEPLIB      DD  DSN=SYS1.VFORTLIB,DISP=SHR
//SYSPRINT     DD  SYSOUT=A
//SYSIN        DD DSN=&&LOADSET,DISP=(OLD,PASS)
//SYSUT1       DD DSN=&NRENT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//SYSUT2       DD DSN=&RENT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//SYSUT3       DD DSN=&TEMP,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//*
//*            NON-REENTRANT PART LINKEDIT
//*
//LKEDN  EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD      DD DSN=&&GOSET(&GONAME),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD DSN=&NRENT,DISP=(OLD,DELETE)
//             DD DDNAME=SYSIN
//*
//*            REENTRANT PART LINKEDIT
//*
//LKEDR  EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF,RENT,NCAL'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSLMOD      DD DSN=&&GOSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD DSN=&RENT,DISP=(OLD,DELETE)
//             DD DDNAME=SYSIN
//*
//*            EXECUTION STEP
//*
//GO     EXEC  PGM=&GONAME,REGION=&GOREGN,COND=(4,LT)
//STEPLIB      DD DSN=&&GOSET,DISP=(OLD,PASS)
//             DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001     DD &GOF5DD
//FT06F001     DD &GOF6DD
//FT07F001     DD &GOF7DD
```

Figure 58 (Part 2 of 2). JCL Procedure to Compile, Separate, Link-Edit, and Go

```
//VFTNRLG PROC LKLNDD='DDNAME=SYSIN',GOPGM=MAIN,GOREGN=500K,
//             GOF5DD='DDNAME=SYSIN',FVLNSPC='3200,(25,6)',
//             GOF6DD='SYSOUT=A',FVNAME='RENTPART',
//             GOF7DD='SYSOUT=B'
//*
//**********************************************************************
//*  PROC DESCRIPTION: SEPARATE, LINK AND GO                           *
//**********************************************************************
//*
//*
//*            PARAMETER  DEFAULT-VALUE     USAGE
//*              LKLNDD   DDNAME=SYSIN      LKED.SYSLIN OPERAND
//*              GOPGM    MAIN              OBJECT PROGRAM NAME
//*              GOREGN   100K              GO-STEP REGION
//*              GOF5DD   DDNAME=SYSIN      GO.FT05F001 OPERAND
//*              GOF6DD   SYSOUT=A          GO.FT06F001 OPERAND
//*              GOF7DD   SYSOUT=B          GO.FT07F001 OPERAND
//*              FVNAME   RENTPART          REENTRANT MODULE NAME
//*              FVLNSPC  3200,(25,6)       SPACE ALLOCATION
//*
//*            SEPARATION STEP
//*
//SEPARATE     EXEC PGM=IFYVSFST,PARM='&FVNAME'
//STEPLIB      DD  DSN=SYS1.VFORTLIB,DISP=SHR
//SYSPRINT     DD  SYSOUT=A
//*SYSIN       DD DSN=&INPUT,DISP=SHR
//SYSUT1       DD DSN=&NRENT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//SYSUT2       DD DSN=&RENT,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//SYSUT3       DD DSN=&TEMP,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(&FVLNSPC),DCB=BLKSIZE=3200
//*
//*            NON-REENTRANT PART LINKEDIT
//*
//LKEDN  EXEC  PGM=IEWL,REGION=200K,COND=(4,LT),
//             PARM='LET,LIST,MAP,XREF'
//SYSPRINT     DD SYSOUT=A
//SYSLIB       DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1       DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD      DD DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN       DD DSN=&NRENT,DISP=(OLD,PASS)
```

Figure 59 (Part 1 of 2). JCL Procedure to Separate, Link-Edit, and Go

```
//*
//*           REENTRANT PART LINKEDIT
//*
//LKEDR  EXEC PGM=IEWL,REGION=200K,COND=(4,LT),
//            PARM='LET,LIST,MAP,XREF,RENT,NCAL'
//SYSPRINT    DD SYSOUT=A
//SYSLIB      DD DSN=SYS1.VFORTLIB,DISP=SHR
//SYSUT1      DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD     DD DSN=&&GOSET,DISP=(MOD,PASS),UNIT=SYSDA,
//            SPACE=(TRK,(10,10,1),RLSE)
//SYSLIN      DD DSN=&RENT,DISP=(OLD,PASS)
//*
//*     EXECUTION STEP
//*
//GO     EXEC PGM=&GOPGM,REGION=&GOREGN,
//            COND=(4,LT)
//STEPLIB     DD DSN=&&GOSET,DISP=(OLD,PASS)
//            DD DSN=SYS1.VFORTLIB,DISP=SHR
//FT05F001    DD &GOF5DD
//FT06F001    DD &GOF6DD
//FT07F001    DD &GOF7DD
```

Figure 59 (Part 2 of 2). JCL Procedure to Separate, Link-Edit, and Go

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 executing 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, using I/O statements for formatted sequential access files. The applicable statements are: READ, WRITE, CLOSE, and REWIND. Using I/O statements other than these may produce invalid results.

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 execution.

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 number 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 above.

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" on page 299. 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 299, 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 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

Note: An OPEN statement—**not** recommended for PDS I/O—results in:

- INOUT, if the opened data set exists

- OUTIN, if it does not

Using Asynchronous Input/Output

Asynchronous input/output statements let you transfer unformatted data quickly between external sequential files and arrays in your FORTRAN program, and, while the data transfer is taking place, continue other processing within your FORTRAN program.

Because the processing overlaps, you must have a method to ensure that your program doesn't make references to the data until the data transfer is complete.

The asynchronous input/output statements have special features to achieve this:

- The WAIT statement—to ensure that data transmission is complete before your program begins processing the data
- A unique identifier to identify a particular READ, WRITE, or WAIT statement—and to connect it with other related asynchronous statements

When you code asynchronous input/output statements, you must have the VS FORTRAN library available at execution time. To make this library available (and you are using load mode), use the following job control statement:

```
//STEPLIB DD DSN=SYS1.VFORTLIB,DISP=SHR
```

If you are using link mode, the STEPLIB statement is unnecessary.

Using the Asynchronous WRITE Statement

To create an asynchronous input/output file, you use a special form of the WRITE statement:

To Transfer an Entire Array:

```
WRITE (10,ID=6) ARAY1
```

To Transfer Part of an Array:

```
WRITE (10,ID=6) ARAY1(2,2)...ARAY1(5,6)

WRITE (10,ID=6) ARAY1(2,2)...
```

where, in this example:

10	is the unit number for the asynchronous file.
ID=6	is a unique identifier for this WRITE statement, used in the WAIT statement.

ARAY1	is an array whose contents are to be transferred.
	In the first WRITE statement, the contents of the entire array are transferred.
	In the second WRITE statement, the contents of ARAY1(2,2) through ARAY1(5,6) are transferred.
	In the third WRITE statement, the contents of ARAY1(2,2) through the end of ARAY1 are transferred.

Using the Asynchronous READ Statement

To retrieve an asynchronous input/output file, you use a special form of the READ statement:

To Transfer an Entire Array:

```
   READ (10,ID=6) ARAY1
```

To Transfer Part of an Array:

```
   READ (10,ID=6) ARAY1(2,2)...ARAY1(5,6)
   READ (10,ID=6) ARAY1(2,2)...
```

where, in this example:

10	is the unit number for the asynchronous file.
ID=6	is a unique identifier for this READ statement, used in the WAIT statement.
ARAY1	is an array whose contents are to be transferred.
	In the first READ statement, the contents of the entire array are transferred.
	In the second READ statement, the contents of ARAY1(2,2) through ARAY1(5,6) are transferred.
	In the third READ statement, the contents of ARAY1(2,2) through the end of ARAY1 are transferred.

Using the Asynchronous WAIT Statement

After you've executed an asynchronous WRITE or READ statement, you must ensure that the I/O operation is complete before you make any further program references to the array being processed. The WAIT statement tells the program to suspend operations until the data transfer is complete; that is, it synchronizes the WRITE or READ statement with the rest of the program.

For example, you can use the following WAIT statement with the previously described READ or WRITE statements:

```
      WAIT (10,ID=6) ARAY1
```

or

```
      WAIT (10,ID=6) ARAY1(2,2)...ARAY1(5,6)
```

where, in this example:

10	is the unit number for the asynchronous file.
ID=6	is a unique identifier for this WAIT statement; it ties this WAIT statement to the READ or WRITE statement with the same identifier (the operation the program is waiting for).
ARAY1	is an array. In the first WAIT statement, the data transfer for the entire array is being synchronized. In the second WAIT statement, the data transfer for array elements ARAY1(2,2) through ARAY1(5,6) is being synchronized.

End of IBM Extension

Sequential Files—System Considerations

Each sequential file you use must be defined to the system through job control statements. To define each file to the system, you specify a DD statement; see "Defining Files—DD Statement" on page 262 for details. The data sets you can specify and the ddnames you can use for them are shown in Figure 54 on page 286.

Direct Files—System Considerations

You must define each direct file to the system through job control statements. For information about job control statements, see "Job Processing" on page 257.

For TSO considerations, see Chapter 13, "Using VS FORTRAN under TSO" on page 317.

Before you can write records into the file, it must be initialized with empty records. The first extent of a data set will be formatted. The data set must be specified as DISP=NEW in the job control information provided by the system, and the FORTRAN OPEN statement must have STATUS='NEW' or STATUS='UNKNOWN'.

To define each file to the system, you specify a DD statement; in the DCB parameter, for a direct file, you must specify:

RECFM=F	which specifies a fixed record size.
BLKSIZE=rl	where *rl* is the record length. For example, if the record length is 80, you must specify BLKSIZE=80.

The OPEN statement provides the default block size for the file, unless you override it through the BLKSIZE parameter.

For other DD statement options you can specify, see "Defining Files—DD Statement" on page 262.

The data sets you can specify and the ddnames you can use for them are shown in Figure 54 on page 286.

Input/Output—System Considerations

For every file your program uses, you may need labels. System considerations are given in the following sections for tape labels and for direct access labels.

Tape Label Considerations

You specify magnetic tape labels through the LABEL parameter of the DD statement; through this parameter, you can specify the position of the file on the tape, the type of label, if the data set is password protected, and the type of file processing allowed.

For more information about job control statements, see "Job Processing" on page 257.

For additional detail on magnetic tape label processing, see *OS/VS Tape Labels.*

Direct Access Label Considerations

You specify direct access labels through the LABEL parameter of the DD statement; through this parameter, you can specify the position of the file on the volume, the type of label, if the data set is password protected, and the type of file processing allowed.

For additional details on direct access label processing, see the appropriate *Data Management Services Guide.*

Defining FORTRAN Records—System Considerations

Your FORTRAN programs must define the characteristics of the data records it will process: their formats, their record length, their blocking, and the type of device upon which they reside.

Record Formats

Under VS FORTRAN, you can specify the format of the data records as:

Fixed-Length Records
All the records in the file are the same size and each is wholly contained within one block. Blocks can contain more than one record, and there is usually a fixed number of records in each block. 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 record-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 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 record-descriptor word (RDW) 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 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|RDW|  (26 bytes)    |RDW|   (26 bytes)   |(36 unused bytes)|

<-------------------------block 2-------------------------------->
|   |<-----record 3----->|                                       |
|BDW|RDW|  (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 as formatted or unformatted, that is, they may or may not be defined in a FORMAT statement. List-directed I/O statements are considered formatted.

Formatted Records: You can specify formatted records as fixed (blocked or unblocked) length, variable (blocked or unblocked) length, 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.

Defining Records

You define data record characteristics through the DCB parameter of the DD statement.

Through the DCB parameter, you can specify:

- Record format—fixed length, variable length, or undefined
- Record length—either the exact length (fixed or undefined), or the length of the longest record (variable)
- Blocking information—such as the block size
- Buffer information—the number of buffers to be assigned
- Whether the data set is encoded in the EBCDIC or the ISCII/ASCII character set
- Special information for tape files
- Special information for direct access files
- Information to be used from another data set

For information about the DCB parameter, see "Job Processing" on page 257.

Cataloging and Overlaying Programs—System Considerations

In order to use the subprograms you write, you must catalog them in a library—so they're available to calling programs. For information on how to do this, see "Cataloging Your Object Module" on page 279.

Overlaying Programs in Storage

When you use the overlay features of the linkage editor, you can reduce the main storage requirements of your program by breaking the program up into two or more segments that don't need to be in main storage at the same time. These segments can then be assigned the same storage addresses and can be loaded at different times during execution of the program.

You must specify linkage editor control statements to indicate the relationship of segments within the overlay structure.

Keep in mind that, although overlays reduce storage, they also can drastically increase program execution time. In other words, you probably shouldn't use overlays unless they're absolutely necessary. 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 execution time when a subprogram not already in main storage is referred to. The reference to the subprogram may be either a FUNCTION name or a CALL statement to a SUBROUTINE subprogram name. When the subprogram reference is found, the overlay segment containing the required subprogram is loaded—as well as any segments in its path not currently in main storage.

When a segment is loaded, it overlays any segment in storage with the same relative origin. It also overlays any segments that are lower (farther from the root segment) in the path of the overlaid segment.

Whenever a segment is loaded it contains a fresh copy of the program units that it comprises; any data values that may have been established or altered during previous processing are returned to their initial values each time the segment is loaded.

For this reason, you should place subprograms whose data values must be retained for longer than a single load phase into the root segment.

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

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

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

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

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

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

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

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

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

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

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

Invoking the VS FORTRAN Compiler

VS FORTRAN can be invoked through the use of the CALL, ATTACH, or LINK MVS macro instruction. These instructions are used as part of an assembler language program that can then be assembled, link-edited, and executed by MVS or TSO.

The program must supply to the FORTRAN compiler:

- The information usually specified in the PARM parameter of the EXEC statement.

- The ddnames of the data sets 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	FORTVS, (optionaddr[,ddnameaddr]), VL

compiler-name
specifies the program name of the compiler to be invoked. FORTVS is specified for VS FORTRAN.

optionaddr
specifies the address of a variable-length list containing information usually specified in the PARM parameter of the EXEC statement.

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 the address of a variable-length list containing alternate names of the 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
2	00000000
3	00000000
4	00000000
5	SYSIN
6	SYSPRINT
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   15,12,12(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=FORTVS,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
```

MVS/XA Considerations

Every program that executes 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 execute 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 Release 2.0 and later can execute 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 Releases 1.0 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 subprograms for execution 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 Release 2.0 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 312.

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 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 Release 2.0 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 will reside below the 16-megabyte line, and will be invoked in 24-bit addressing mode.

- All VS FORTRAN main programs compiled with VS FORTRAN Release 2.0 and later have the AMODE set to ANY. The linkage editor will set the AMODE of the load module to ANY by default.

  The load module may be entered with 31-bit or 24-bit addressing mode, depending on the release of MVS/XA being used and the parameters used in TSO. The addressing mode upon entry is set to 31-bit if MVS/SP Version 2, Release 1.1, is used to load the program, and set to 24-bit if MVS/SP Version 2, Release 1.0, is used. If a FORTRAN program is called from a TSO session, the addressing mode is set to 24-bit, but if a FORTRAN program compiled with the TEST option is called from a TSO session, the addressing mode is set to 31-bit.

  If the main routine is entered in 31-bit addressing mode and calls a VS FORTRAN subroutine compiled by VS FORTRAN prior to Release 2.0, or a FORTRAN subroutine compiled by any other FORTRAN compiler or an Assembler routine with 24-bit addressing dependencies, the program may abnormally terminate during execution. The default AMODE attribute must be overriden in the link-edit step to set AMODE=24.

- The RMODE of the load module is based upon whether the load module is created for execution in link mode or load mode. For complete details concerning link mode and load mode of the VS FORTRAN Library, see "Execution-Time Loading of Library Modules" on page 276.

  A program that is link-edited to operate in link mode will always be given an RMODE of 24. Overriding this value to ANY is not permissible because there are some library routines 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 library routines that are loaded during execution 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 Library modules, the execution-time library always switches to 31-bit addressing mode while in the library routines, and to the addressing mode of the caller of the library routine 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-megabtye line according to its RMODE. For more information about AMODE and RMODE, see *MVS/Extended Architecture Supervisor Services and Macro Instructions.*

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)'
```

  See *MVS/Extended Architecture Linkage Editor and Loader* for additional detail.

- The linkage editor MODE control statement

MODE	AMODE(xxx),RMODE(yyy)

  See *MVS/Extended Architecture Linkage Editor and Loader* for additional detail.

- 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-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 Library at execution time. For details concerning dynamic common areas, see "Using Blank and Named Common (Static and Dynamic)" on page 133.

In order to use the extra storage available with MVS/XA, the load module must execute in 31-bit addressing mode. In particular, the module cannot contain subroutines compiled under FORTRAN G1, Hx or prior to VS FORTRAN Release 2.0. The storage for dynamic common areas will be obtained above the 16-megabyte line only when the program is executing in 31-bit addressing mode (regardless of the residence mode) and storage is available; storage will be obtained below the 16-megabyte line when the program is executing 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 execution 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 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 Library execution-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.

- During installation, specify ARCH=XA in the VSFORTL macro.

- Unless you specifically force an AMODE value of 24, do not mix object modules compiled with VS FORTRAN Release 2.0 and later with

  - Object modules compiled with compilers prior to VS FORTRAN Release 2.0

  - Assembler code with 24-bit dependencies

Chapter 13. Using VS FORTRAN under TSO

You can use the facilities of TSO, taking advantage of quick terminal turnaround time, to develop VS FORTRAN programs. You can compile your programs under TSO and link-edit them to run under MVS or any other supported system, or you can compile, link-edit, and execute them under TSO.

When your VS FORTRAN source programs are compiled under TSO, the compiler creates files that contain its listing and the executable code the program produces. Some of the programs, during execution, may process or create files containing data.

This section describes files in a FORTRAN context. For additional information on TSO, see:

OS/VS2 MVS TSO Command Language Reference
OS/VS2 MVS TSO Terminal User's Guide

Using the TSO Commands

The TSO commands help you create and edit your source programs, link-edit your object modules, and execute your load modules. The TSO commands you'll use most frequently are shown in Figure 60 on page 318.

User-Defined Files

Before invoking the VS FORTRAN compiler, a VS FORTRAN source program must be available in a TSO file. The source program is usually created using the TSO EDIT command, and may be written in either fixed or free format. The files may be created at the terminal just prior to compiling them, or they may be old files that need recompilation. In either case, all VS FORTRAN source files must have a TSO identifier and file characteristics that conform to VS FORTRAN compiler requirements.

User-defined files containing data that you will want your program to process, may already exist in your system. Conversely, you may want your program to create a file to hold data that was generated during its execution. You must define all such files with an ALLOCATE command; because they are not predefined, they cannot be identified by TSO and associated with your program. ALLOCATE is used in conjunction with the data set reference numbers in your FORTRAN input and output statements and the identifier of the file that you want to use or create.

TSO Command	Usage
ALLOCATE	Allocates data sets needed for compilation, link-editing, or execution.
ATTRIB	Builds a list of data sets (DCB parameters) for dynamic allocation.
CALL	Invokes compiler, linkage-editor, or load module for execution.
DELETE	Deletes one or more data set entries or one or more members of a partitioned data set.
EDIT	Puts you in EDIT mode to create and edit source program and data files, and lets you use EDIT subcommands.
FREE	Frees files allocated for a job (same as UNALLOCATED).
HELP	Provides information about commands other than EDIT subcommands.
LINK	Converts one or more object modules into a load module.
LOADGO	Loads one or more object modules into storage and executes them.
STATUS	Checks execution status of a submitted batch job.
SUBMIT	Submits a JCL file to MVS to run as a batch (background) job (requires SUBMIT logon capabilities).
TEST	Tests an object program for proper execution and locates programming errors.

Figure 60. TSO Commands Often Used with VS FORTRAN

Whether a file is sequential, direct access, or random access will, to a great extent, determine how a record is defined, the way it is identified, and how it is referred to in a VS FORTRAN program.

Regardless of the type of file you are using, there are several guidelines that must be followed in defining and using files.

- Define each file used in your program to the system (either through system-supplied definition or one that you supply).

- Do not use the same definition for more than one file.
- You may refer to the same file from more than one program through different devices and access methods, if you change the file and its definition appropriately before using it.

File Identification—TSO ALLOCATE Command

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

For the Source Program as Compiler Input:

```
ALLOCATE DATASET(myprog.fort) FILE(SYSIN) OLD
```

This ALLOCATE command tells TSO that the file named *myprog.fort* is an existing file (OLD), available on the SYSIN data set.

If you are using the INCLUDE statement, you need to allocate that data set as SYSLIB:

```
ALLOCATE FILE(SYSLIB) DATASET('user.lib.fort') SHR
```

For information on how to create a library containing the INCLUDE source code, see "Using the FORTRAN INCLUDE Statement" on page 336.

For Compiler Output Listings:

```
ALLOCATE DATASET(myprog.list) FILE(SYSPRINT) NEW   -
                 BLOCK(120)     SPACE(60,10)
```

This ALLOCATE command tells TSO that the file named *myprog.list* is a new file (NEW), to be produced on the SYSPRINT data set. The line length is 120 characters; the primary space allocation is 60 lines.

To print the listing, use the PRINT command.

For an Object Deck:

```
ALLOCATE DATASET(myprog.obj) FILE(SYSPUNCH) NEW   -
                 BLOCK(80) SPACE (120,20)
```

This ALLOCATE command tells TSO that the file named *myprog.obj* is a new file (NEW), to be produced on the SYSPUNCH data set. The record length is 80 characters.

For the Object Module:

```
ALLOCATE DATASET(myprog.obj) FILE(SYSLIN)   -
                       NEW BLOCK(80) SPACE(100,10)
```

This ALLOCATE command tells TSO that the file named *myprog.obj* is a new file (NEW), to be produced on the SYSLIN data set. The record size (and block size) must be 80 characters. The space you can specify as any size you need.

For Terminal Input/Output:

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

This ALLOCATE command tells TSO that the file identified by the asterisk (*) is available on the SYSTERM data set. You can then use the terminal to receive error message output. (The listing output is on the SYSPRINT data set.)

For terminal files, a null entry in response to a prompt is taken to be an end-of-file. Another ALLOCATE command or an explicit OPEN is required to continue processing.

For Program Data Sets:

```
ALLOCATE DATASET(identifier.name.data) FILE(FTxxFyyy)
```

This ALLOCATE command tells TSO that the file identified by the qualified name is available on the FTxxFyyy data set. Valid values for xx and yyy are documented in Figure 54 on page 286.

Before you can load a direct data set, you must preformat it. "Direct Files—System Considerations" on page 303 tells how to do this.

Creating Your Source Program—TSO EDIT Command

To create a source program data set, you use the EDIT command. Use the EDIT command whenever you want to create a new data set and also whenever you want to edit an existing one.

(You can also use the Interactive System Productivity Facility (ISPF) editor to create source program data sets. You can use SPF to allocate sequential data sets or partitioned data sets, although this isn't necessary, because the ALLOCATE and ATTRIB commands are also available.)

The EDIT subcommands (such as COPY, INPUT, INSERT, etc.) help you enter and edit the lines of source code.

To create a source program file, you can specify the qualifier of your source program file as *fort*. Alternatively, you can specify the qualifier as *data*.

For example, to create a source program data set named *myprog*, you specify:

```
EDIT myprog.fort

  or

EDIT myprog.data
```

This creates an empty data set for you, with the name *myprog*, and the descriptive qualifier *fort* or *data*. (If *myprog.fort* already exists, the EDIT command retrieves it for you and makes it available for editing.)

You can now enter your source program into the data set, line by line, according to the rules for fixed or free form source programs.

Fixed format FORTRAN files contain 80-character records; you use the first 72 characters for FORTRAN statements and continuation lines.

Compiling Your Program—TSO ALLOCATE and CALL Commands

To compile your program, use the ALLOCATE and CALL commands. For other possibilities, see "Invoking the VS FORTRAN Compiler" on page 309.

Allocating Compilation Data Sets—TSO ALLOCATE Command

First, you allocate the data sets you'll need for compilation as shown in Figure 61 and discussed under "File Identification—TSO ALLOCATE Command" on page 319.

```
ALLOCATE DATASET(myprog.fort) FILE(SYSIN) OLD
ALLOCATE DATASET(myprog.list) FILE(SYSPRINT) NEW BLOCK(120) SPACE(60,10)
ALLOCATE DATASET(myprog.obj) FILE(SYSLIN) NEW BLOCK(80) SPACE(100,10)
ALLOCATE DATASET(*) FILE(SYSTERM)
ALLOCATE DATASET('user.lib.fort') FILE(SYSLIB) SHR
```

Figure 61. Allocating TSO Compilation Data Sets

For any compilation, you must allocate the SYSIN AND SYSPRINT data sets.

Allocate the SYSLIN data set only if you want to produce an object module (you've specified the OBJECT compiler option).

Allocate the SYSTERM data set only if you wish to receive diagnostics at the terminal (you've specified the TERMINAL option).

Allocate the SYSLIB data set only if your compilation uses the INCLUDE statement.

You can enter these ALLOCATE commands in any order. However, you must enter all of them before you invoke the FORTRAN compiler.

Requesting Compilation—CALL Command

After you've allocated the data sets you'll need, you can issue a CALL command, requesting compilation.

Note: VS FORTRAN compiler options can be specified as options of the CALL command.

You can request compilation, using the default compiler options:

```
CALL 'SYS1.FORTVS(FORTVS)'
```

or you can request one or more compiler options explicitly:

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

which tells the compiler that:

- FREE—your source program is in free form.
- TERM—diagnostic messages are to be directed to your terminal.
- SOURCE—the source program is to be printed in the output listing.
- MAP—a storage map is to be printed in the output listing.
- LIST—the object program is to be printed in the output listing.
- OBJECT—an object module is to be produced.

Specifying TSO Line Numbers When Debugging

If you want to use TSO line numbers as breakpoints interactively when debugging your compiled program, you must specify the NOSDUMP compiler option in addition to the TEST option.

If you are using VS FORTRAN Interactive Debug (5668-903), see Chapter 16, "Using VS FORTRAN Interactive Debug with VS FORTRAN" on page 377.

Compiler Output

Depending on your organization's compile-time defaults and/or the options you select in your CALL command, you may get a LIST data set and/or an OBJ data set as output, placed in your disk storage for easy reference, under the name(s) you specified in the ALLOCATE command.

LIST Data Set

The LIST data set contains the compiler output listing; see "Identifying User Errors" on page 167 for an explanation of what the compiler output listing contains and how to use it.

It has the name of your source program, and the qualifier LIST. For example, the qualified name for MYPROG is MYPROG.LIST.

OBJ Data Set

The OBJ data set contains the object code the compiler created from your source program. The OBJ data set contents are explained in "Link-Editing Your Program" on page 278.

It is placed in your storage with the name of your source program and the qualifier OBJ. For example, the qualified name for *myprog* is *myprog.obj.*

This data set remains in your disk storage until you erase it, using the DELETE command.

You can link-edit the OBJ data set under any of the systems that VS FORTRAN supports to get a load module.

Link-Editing and Executing Your Program under TSO

To link-edit and execute your program under TSO, use the LINK command to create a load module from one or more object modules (plus any needed VS FORTRAN library modules), and then use the CALL command to execute the load module.

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

```
userid.name.OBJ
```

For information on how to invoke the VS FORTRAN compiler, see Appendix B, "Object Module Records" on page 403.

Link-Editing Your Program—TSO LINK Command

You use the LINK command to create and execute a load module. Input you use consists of your object module, VS FORTRAN library routines, and any other secondary input (such as OBJ data sets of called subprograms).

For example, if you want to load and execute the OBJ data sets for *myprog* and its subprogram *subprog*, you specify:

For load mode:

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

or

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VALTLIB','SYS1.VFORTLIB')
```

For link mode:

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('SYS1.VLNKMLIB','SYS1.VFORTLIB')
```

or

```
LINK (myprog,subprog) LOAD(myprog)+
LIB('STS1.VLNKMLIB','SYS1.VALTLIB','SYS1.VFORTLIB')
```

When the commands are executed, 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.VALTLIB and SYS1.VFORTLIB.

For information about link and load mode, see "Executing Your Program" on page 324.

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.VFORTLIB') 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.VFORTLIB') LOAD(myprog) PRINT(*)
```

When you specify PRINT(*), the linkage editor listings are displayed at your terminal.

Executing Your Program

To execute your VS FORTRAN program on TSO or TSO-E, there are certain prerequisites.

- If you are using a Release 2 or Release 3 or 3.1 load module with the VS FORTRAN Release 4 library (that is, you have not link-edited your object files with the Release 4 library), then
  - If you are **not** using the IFYVRENT routines, only IFYVASUB and 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, IFYVRENT must also be in a library on the system link list.
  - If you are using the IFYVRENT routines and IFYVRENT is in the link pack area (LPA), then IFYVPOST and IFYVASUB (IFYVASUB may go into the LPA) must be in a library on the system link list.
  - All the above statements assume that you are running a program in a load module link-edited with the VS FORTRAN Release 2 or Release 3 or 3.1 library.
- If you link-edit your nonreentrant object modules with the Release 4 libraries, you must follow these rules when you link-edit your program:
  - Link mode allows you to build a complete load module with no execution-time library module dependency.

Example 1: Link mode and standard mathematical routines

```
LINK myprog LOAD(myprog(T)) PRINT(myprog) LET LIST MAP+
LIB('SYS1.VLNKMLIB','SYS1.VFORTLIB')

CALL myprog(T)
```

Example 2: Link mode and alternative mathematical routines

```
LINK myprog LOAD(myprog(A)) PRINT(myprog) LET LIST MAP+
LIB('SYS1.VALTLIB','SYS1.VLNKMLIB','SYS1.VFORTLIB')

CALL myprog(A)
```

- Load mode allows you to build smaller load modules than link mode, but at a cost of an execution-time module dependency. This means that SYS1.VFORTLIB must be added to your system link list or concatenated to STEPLIB in the logon procedure, so that required routines at execution time may be found.

  Example 1: Load mode and standard mathematical routines

  ```
  LINK myprog LOAD(myprog(T)) PRINT(myprog) LET LIST MAP+
  LIB('SYS1.VFORTLIB')

  CALL myprog(T)
  ```

  Example 2: Load mode and alternative mathematical routines

  ```
  LINK myprog LOAD(myprog(A)) PRINT(myprog) LET LIST MAP+
  LIB('SYS1.VALTLIB','SYS1.VFORTLIB')

  CALL myprog(A)
  ```

- If you want to run a program that has been compiled with the RENT compiler option, there are several things that you may do:

  - You may use your compiler object as if it were a nonreentrant object file and link, as described previously.

  - You may process your reentrant compilation object file with the separation program. Only the nonreentrant object output needs the Release 4 library, as described previously. The reentrant object output is link-edited without VS FORTRAN libraries present.

  - You may want to use the following CLIST to do the separation. This CLIST assumes you will be creating just one load module. If you want to create multiple load modules, then the invocation should not include passing the parameter list ('&RENTPART.'), but must include the allocation using &RENTPART.

```
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.VFORTLIB(IFYVSFST)' '&RENTPART.'
FREE F(SYSIN SYSPRINT SYSUT1 SYSUT2 SYSUT3)
FREE ATTR(DCBPARMS)
```

Note: This CLIST does not link either the reentrant or nonreentrant parts, and assumes the input object file does not have an OBJ2 file associated with it.

- You may put the reentrant modules into a library that is in the STEPLIB specification of your logon procedure. This same file may be shared with two or more users.

  Example:

  ```
  LINK myprog LOAD(SHRLIB) PRINT(M1) XREF LET LIST RENT
  ```

  would insert the contents of the *myprog* object file into the SHRLIB library. Note the name(s) of the module(s) actually entered into SHRLIB have been generated by the separation program on NAME control records.

- To insert the module(s) in *myprog* into the LPA requires authority to insert them into the correct library and restarting MVS to load those modules into the LPA. The procedure used varies from installation to installation, so see your system administrator for the procedure.

When the above has been done, a CALL to the user link-edited program

```
CALL mylib(myprog)
```

will result in the user's program being properly executed and, if a failure occurs, the post-abend processor will be called to identify what happened.

Specifying Execution-Time Options

To specify an execution-time option (XUFLOW, NOXUFLOW, DEBUG, or NODEBUG), use the following method:

```
CALL pgmname 'option'
```

where pgmname is the name of your VS FORTRAN program, and option is XUFLOW, NOXUFLOW, DEBUG, or NODEBUG.

For more information, see "Using the Execution-Time Options" on page 200.

Using the CALL Command—TSO Load Module Execution

For example, to execute MYPROG.LOAD, you specify the ALLOCATE commands needed to allocate the input and output data sets it uses, as well as any work data sets. Then you issue the CALL command, as follows:

```
ALLOCATE DATASET(myprog.indata)    FILE(FT05F001)   (as needed)
ALLOCATE DATASET(myprog.outdata)   FILE(FT06F001)   (as needed)
ALLOCATE DATASET(myprog.workfil)   FILE(FT10F001)   (as needed)
CALL myprog
```

and program *tempname* from file *myprog.load* is executed.

After program execution is complete, and if you no longer need them, you should issue the DELETE command to free the disk space used by the data sets you've named in the ALLOCATE commands and in the CALL command:

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

If you do need them, don't issue the DELETE command; you can then reuse the data sets as necessary.

Using the Loader—TSO LOADGO Command

Using the LOADGO command, you can invoke the loader program to link-edit and execute your program all in one step. This is efficient when you have several object modules you want combined into one load module for a quick test. When you use the LOADGO command, the load module created is automatically deleted when program execution ends. Link and load mode apply to the LOAD command.

First, you must allocate any needed data sets, as outlined under "Allocating Compilation Data Sets—TSO ALLOCATE Command" on page 321. You must then add SYS1.VFORTLIB to your system link list, or concatenate SYS1.VFORTLIB to STEPLIB in your logon procedure.

Then you issue the LOADGO command. In this example, you're linking and executing the *myprog* object module:

Load mode:

```
LOADGO (myprog) LIB('SYS1.VFORTLIB')
```

or

```
LOADGO (myprog) LIB('SYS1.VALTLIB','SYS1.VFORTLIB')
```

Link mode:

```
LOADGO (myprog) LIB('SYS1.VLNKMLIB','SYS1.VFORTLIB')
```

or

```
LOADGO (myprog) LIB('SYS1.VALTLIB','SYS1.VLNKMLIB','SYS1.VFORTLIB')
```

The LIB operand makes the appropriate data sets available to the loader program. The loader program can then resolve any external references in *myprog* and load the needed object modules.

You can also use the LOADGO command to execute a link-edited load module; for example, one named *myprog*:

```
LOADGO myprog(tempname)
```

You can also use the LOADGO command to specify loader options. In the last example, you can request a load module map and the listings to be printed, either on the system printer or at your terminal:

On the System Printer:

```
LOADGO (myprog) MAP PRINT
```

The qualified name of the data set sent to the system printer is MYPROG.LOADLIST.

At Your Terminal:

```
LOADGO (myprog) MAP PRINT(*)
```

When you specify PRINT(*), the loader listings are displayed at your terminal.

Fixing Execution Errors under TSO

When you're developing programs using TSO, you can make use of all the FORTRAN debugging aids described in Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185.

You can use the TSO TEST command, together with its associated subcommands, to debug your object program.

The easiest way to use TEST is to establish the point in your program at which an abnormal termination occurred.

For best results, you should be familiar with the assembler language and addressing conventions.

Requesting a Command Procedure under TSO

You can create a command procedure (CLIST) for a number of different processing options under TSO. This is useful, because you can preallocate all the options you need once, when you create the command procedure. Then, every time you execute the command procedure, there's no need to respecify the options.

You can create command procedures to process your jobs either in the foreground or in the background.

Command Procedures for Foreground Processing

To create a command procedure to link-edit and run a FORTRAN program in the foreground, use one of these forms of CLIST:

Example 1:

```
PROC 1 NAME
ALLOCATE FILE(FT06f001) DA(*)
LOADGO &NAME LIB('SYS1.VFORTLIB')
```

Example 2:

```
PROC 1 NAME
LINK &NAME LOAD(temp(MAIN)) LIB('SYS1.VFORTLIB') LET MAP
ALLOCATE FILE(FT06f001) DA(*)
CALL &NAME(MAIN)
DELETE temp
```

Command Procedures for Background Execution

If your source programs are small, foreground execution can be quite convenient. However, if your programs will take some time to compile and/or to execute, you may prefer to batch process them in the background. This frees your terminal for other work while the batch job is running. For information about procedures, see "Using and Modifying Cataloged Procedures" on page 264.

System Considerations under TSO

When you're developing programs using the TSO facilities, your FORTRAN programs must not require system actions for which you are not authorized (for example, protected data set access or volume mounting).

In addition, if your FORTRAN programs make use of assembler subroutines, there are a few system considerations you must take into account, when coding the assembler routines, as follows:

- Your assembler subprograms must execute as nonauthorized problem programs.
- Your assembler subprograms must use standard MVS system service interfaces.

- Your assembler subprogram must not use TSO-specific storage subpools.
- The address spaces your assembler subprograms use must not be sensitive to MVS control block structures.
- Your assembler subprogram must use only the data set characteristics available through the ALLOCATE command.

If your FORTRAN programs are compiled with the SYM compiler option, then use the TEST option when link-editing to form a load module. This enables you to reference most of the FORTRAN variables when using TSO TEST. All the other requirements still apply.

If your FORTRAN programs are compiled with the RENT compiler option, you can run in either foreground or background.

For information about overriding the AMODE or RMODE attribute in MVS/XA, see "Overriding AMODE/RMODE Attributes" on page 314.

Chapter 14. Using VS FORTRAN under VSE

You can compile your programs under VSE and then run them under VM or MVS and vice versa; or you can compile and execute them under VSE. The term VSE, used in this context, means VSE/Advanced Functions.

Executing Your Program with Job Control Statements or Cataloged Procedures

The simplest way to execute your program is to use one of the cataloged procedures described in "Writing and Modifying Cataloged Procedures" on page 332. However, the cataloged procedures may not give you the programming flexibility you need for your more complex data processing jobs, and you may need to specify your own job control statements, or write your own cataloged procedures.

Job Processing

Job control statements provide a communication link between the FORTRAN programmer and the operating system. The programmer uses these statements to define a job, a job step within a job, and data sets required by the job. The job control statements needed to compile, link, and execute a VS FORTRAN program are:

Compilation	**Linkage Editor**	**Execution**
JOB	JOB	JOB
EXEC VFORTRAN	EXEC LNKEDT	EXEC program-name,SIZE=nK
ASSGN	ACTION	
CLOSE	ENTRY	
DLBL	INCLUDE	
EXTENT	PHASE	
OPTION		
TLBL		
LIBDEF		

For a complete description of job control statements, see *VSE/Advanced Functions System Control Statements.*

Note: You specify the VS FORTRAN compiler options as options in the PARM parameter of the EXEC job control statement for VSE/Advanced Functions Release 3.0.

Writing and Modifying Cataloged Procedures

To catalog a procedure in the procedure library, you submit a CATALP statement specifying the procedure name. Rules for naming the procedures are given in *VSE/Advanced Functions System Control Statements.*

The control statements to be cataloged follow the CATALP statement; they can be job control or linkage editor control statements or both. The end of the control statements to be cataloged must be indicated by an end-of-procedure delimiter, usually a /+ delimiter.

Each control statement cataloged in the procedure library should have a unique identity. This identity is required if you want to be able to modify the job stream at execution time. Therefore, when cataloging, identify each control statement in columns 73 through 79 (blanks may be embedded).

Retrieving Cataloged Procedures

To retrieve a cataloged procedure from the procedure library, you use the PROC parameter in the EXEC job control statement, specifying the name of the cataloged procedure.

When the job control program starts reading the job control statements in the input stream on SYSRDR and finds the EXEC statement, it knows by the operand PROC that a cataloged procedure is to be inserted. It takes the name of the procedure to be used and retrieves the procedure with that name from the procedure library. At this point, SYSRDR is temporarily assigned to the procedure library. Job control reads and processes the job control statements as usual. The statement,

```
// EXEC MYPROGM
```

causes the program MYPROGM to be loaded and given control. When execution of MYPROGM is complete, the job control program reads the next statement or statements from the procedure library and then finds the end-of-procedure indicator (/+). The end-of-procedure indicator returns the SYSRDR assignment to its permanent device, where the job control program finds the /& statement and performs end-of-job processing as usual.

Temporarily Modifying Cataloged Procedures

You can request temporary modification of statements in a cataloged procedure by supplying the corresponding modifier statements in the input stream.

Normally not all statements need to be modified; therefore, you must establish an exact correspondence between the statement to be modified and the modifier statement by giving them the same symbolic name. This symbolic name may have from one to seven characters, and must be specified in columns 73 through 79 of both statements.

A single character in column 80 of the modifier statement specifies which function is to be performed.

A	Indicates that the statement is to be inserted *after* the statement in the cataloged procedure that has the same name.
B	Indicates that the statement is to be inserted *before* the statement in the cataloged procedure that has the same name.
D	Indicates that the statement in the cataloged procedure that has the same name is to be *deleted.*

Any other character or a blank in column 80 of the modifier statement indicates that the statement is to replace (override) the statement in the cataloged procedure that has the same name.

In addition to naming the statements and indicating the function to be performed, you must inform the job control program that it has to carry out a procedure modification. This is done as follows:

1. By specifying an additional parameter (OV for overriding) in the EXEC statement that calls the procedure, and
2. By using the statement // OVEND to indicate the end of the modifier statements.

Placement of the // OVEND statement is as follows:

- Directly behind the last modifier statement or,
- If the last modifier statement overwrites a // EXEC statement and is followed by data input, between the /* and /& delimiters.

Requesting Compilation

In one job you can request compilation for a single source program or for a series of source programs.

Note: When you're requesting a VS FORTRAN compilation, you should use SIZE=AUTO. This will give the most space in the partition for the dynamic storage (GETVIS) requests from the compiler. Then, if a compilation receives message number IFX0094I for insufficient storage, increase the partition size to increase the GETVIS area available for the compilation. The major determinants of the amount of dynamic storage or GETVIS required by the compiler are program size, complexity, and optimization level.

Compiling a Single Source Program

For a single source program, the sequence of job control statements you use is:

```
// JOB Statement
// OPTION Statement (as required)
// ASSGN Statements for Compilation (as required)
// DLBL Statements for Compilation (as required)
// EXTENT Statements for Compilation (as required)
// EXEC Statement (to execute the VS FORTRAN compiler)

   (Source program to be compiled)

/* Data Delimiter Statement (only if source program is on cards)
/& End-Of-Job Statement
```

Batch Compilation of More Than One Source Program

For a series of programs, the sequence of job control statements you use is:

```
// JOB Statement
// OPTION Statement (as required)
// ASSGN Statements for Compilation (as required)
// DLBL Statements for Compilation (as required)
// EXTENT Statements for Compilation (as required)
// EXEC Statement (to execute the VS FORTRAN compiler)

   (First source program to be compiled)

@PROCESS Statement (if needed to modify compiler options)

   (Second source program to be compiled)

@PROCESS Statement (if needed to modify compiler options)

   (Third source program to be compiled)

/* Data Delimiter Statement (only if source program is on cards)
/& End-Of-Job Statement
```

The @PROCESS statement is described in "Modifying Compilation Options—@PROCESS Statement" on page 164.

Requesting Compilation Only

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

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

where the source program is on SYSIPT, and jobname is the name you're giving to this compilation-only job.

Cataloging Your Source

You can catalog source programs and source statement sequences you'll use in FORTRAN INCLUDE statements as books in the source statement library, using the CATALS function.

Compiler Files

When you're compiling, most of the the system files the compiler uses—SYSIPT, SYSLST, SYSPCH, SYSLNK, and SYSLOG—are predefined and always available; therefore, you never have to specify them.

However, if your source program uses the INCLUDE statement, you must specify SYSSLB (for the system or a private source statement library) in an ASSGN statement. The file records representing the source statements to be included must be unblocked, fixed-length, 80-character records. The file to be included must be cataloged in the G sublibrary of the library defined in the LIBDEF statement.

Printing on the IBM 3800 Printing Subsystem under VSE

Additional run-time parameters are required to support the IBM 3800 Printing Subsystem under VSE. The SETPRT job control statement is used to specify the names of the character arrangement tables.

SETPRT TRC=Y, CHARS(cat0[,cat1, ...]), ...
specifies that each record contains a Table Reference Character byte, and supplies the names of the character arrangement tables to be used.

A sample FORTRAN program using the 3800 follows:

```
C    SAMPLE PROGRAM FOR THE IBM 3800 PRINTING SUBSYSTEM
C
 100  FORMAT ('12','     W666666666666666666666666666X'
    1    / ' 2','      7                           7'
    2    / '+1','        TABULATION OF THE FUNCTION  '
    3    / ' 2','      7                           7'
    4    / ' 2','      7                           7'
    5    / '+0','                sin',A1,'(x)      '
    6    / ' 2','      Z666666666666666666666666666Y'
    7    / ' 2','      W6666666666661666666666666666X'
    8    / ' 2','      7            7              7'
    9    / '+0','           x          sin',A1,'(x) '
    A    / ' 2','      3666666666666566666666666666664')

 200  FORMAT (' 2','    7            7              7'
    1    / '+0','      ',I3,A1,I2,'''',I2,'"',5X,F9.6)

      CHARACTER*1 DEG, U3
      DATA DEG/ZA1/, U3/ZB3/

      WRITE (6,100) U3, U3

      WRITE (6,200) 0, DEG, 0, 0, 0.

      END
      .
      .
      .
//SETPRT TRC=Y,CHARS=(TN,GS10,FM10), ...
```

The sample program above produces the following 3800 output:

TABULATION OF THE FUNCTION
sin^3(x)

x	sin^3(x)
0° 0' 0"	0.000000

IBM Extension

Using the FORTRAN INCLUDE Statement

When you compile a program using the FORTRAN INCLUDE statement, you must specify SYSSLB in an ASSGN statement or use a LIBDEF statement. See "Compiler Files" above for a more detailed description.

If your source program uses the INCLUDE statement, you must put the code to be included into the source library.

1. Catalog the member or members you want to INCLUDE.

```
// ASSGN SYSSLB,X'yyy'
// LIBDEF
// EXEC MAINT
   CATALS G.V00001
   BKEND
      .
      .
      .
- to be included -
      .
      .
      .
   BKEND
```

2. Compile your source program.

```
// ASSGN SYSSLB,X'yyy'
// LIBDEF
// EXEC VFORTRAN,...
         .
         .
         .
      Z = A1 * B1
      INCLUDE(V00001)
         .
         .
         .
      END
```

End of IBM Extension

Compiler Output

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

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

These listings are described in "Identifying User Errors" on page 167 and Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185; examples of output for each feature are also given there.

If your compilation was completed without error messages and you have specified // OPTION LINK, you can proceed to "Link-Editing Your Program" on page 340.

If your compilation caused error messages, you may have to correct your source, as described in Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185.

Cataloging Your Object Module

You request an object module data set by specifying the DECK or OBJECT compiler option.

The data set is a copy of the object module, in card image format, which consists of dictionaries, text, and an end-of-module indicator. (See "Object Module as Link-Edit Data Set" on page 183 for additional details.)

You can catalog your object module in the private relocatable library using the CATALR function and LIBDEF statement.

Then, when you link-edit and execute, you must specify SYSRLB in an ASSGN statement.

Execution-Time Loading of Library Modules

In Release 4.0, all library modules (other than the mathematical routines) can be either link-edited into your load module with the compiler-generated code, or loaded dynamically at execution time. Execution-time loading has several advantages. It reduces auxiliary storage requirements for load modules, and speeds execution in compile-link-go mode.

Selection of Link Mode or Load Mode

After installation of the VS FORTRAN library, your system programmer specified the libraries needed for use in load mode or link mode. A single environment may have been established for all users, or the selection of load mode or link mode left up to individual users. If you need to specify the libraries, do the following:

Specifying Libraries in Load Mode

- For operation in load mode, provide VFORTLIB but *not* VLNKMLIB to the linkage editor for its use when it includes VS FORTRAN library modules. Make only the relocatable library VFORTLIB available for the linkage editor step.

```
// DLBL VFORTLI,'A5748FO3.SYSRLB.VFORTLIB'
// EXTENT SYSmmm,volser
// ASSGN SYSmmm,cuu
// LIBDEF RL,SEARCH=(VFORTLI),TEMP
```

- To make the relocatable library available to all linkage editor steps, put the DLBL and EXTENT statements in the standard label job, make permanent assignments, and specify PERM on the LIBDEF command instead of TEMP.

To execute a program which has been link edited in load mode, make VFLODLIB available for the execution step.

1. Use the following statements in the step which executes the VS FORTRAN program:

   ```
   // DLBL VFLODLI,'A5748FO3.SYSCLB.VFLODLIB'
   // EXTENT SYSnnn,volser
   // ASSGN SYSnnn,cuu
   // LIBDEF CL,SEARCH=(VFLODLI),TEMP
   ```

2. To make VFLODLIB available to all jobs, put the DLBL and EXTENT statements in the standard label area, make the SYSnnn assignment permanent, and specify PERM on the LIBDEF command.

3. In the execution step, specify SIZE=AUTO.

Example

```
// JOB FORTRAN ***COMPILE, LINK-EDIT AND EXECUTE IN LOAD MODE****
// DLBL VFORTLI,'A5748FO3.SYSRLB.VFORTLIB',99/365
// EXTENT SYS001,DOSRES,1,1,8018,570
// ASSGN  SYS001,140
// LIBDEF RL,SEARCH=(VFORTLI),TEMP
// DLBL VFLODLI,'A5748FO3.SYSCLB.VFLODLIB',99/365
// EXTENT SYS002,DOSRES,1,1,6555,228
// ASSGN  SYS002,140
// LIBDEF CL,SEARCH=(VFLODLI),TEMP
*
// OPTION LINK,PARTDUMP
 ACTION MAP, CANCEL
// EXEC VFORTRAN,SIZE=AUTO
   (source program)
/*
// EXEC LNKEDT
// EXEC  ,SIZE=AUTO
/&
```

Specifying Libraries in Link Mode

- For operation in link mode, concatenate VLNKMLIB ahead of VFORTLIB for use by the linkage editor when it includes VS FORTRAN library modules. Make the relocatable libraries VLNKMLIB and VFORTLIB available for the linkage editor step:

  ```
  // DLBL VFLKMLI,'A5748FO3.SYSRLB.VLNKMLIB'
  // EXTENT SYSnnn,volser
  // ASSGN SYSnnn,cuu
  // DLBL VFORTLI,'A5748FO3.SYSRLB.VFORTLIB'
  // EXTENT SYSmmm,volser
  // ASSGN SYSmmm,cuu
  // LIBDEF RL,SEARCH=(VFLKMLI,VFORTLI),TEMP
  ```

- Alternatively, put the DLBL and EXTENT statements in the standard label area, make permanent assignments, and specify PERM on the LIBDEF command. This will make the relocatable libraries available to all linkage editor steps.

- A program which was link edited in link mode does not require any VS FORTRAN libraries at execution time.

- In the execution step, specify SIZE=AUTO.

Example

```
// JOB FORTRAN ***COMPILE, LINK-EDIT AND EXECUTE IN LINK MODE***
*
// DLBL VFLKMLI,'A5748FO3.SYSRLB.VFLKMLIB',99/365
// EXTENT SYS001,DOSRES,1,1,11267,19
// ASSGN  SYS001,140
// DLBL VFORTLI,'A5748FO3.SYSRLB.VFORTLIB',99/365
// EXTENT SYS002,DOSRES,1,1,8018,570
// ASSGN  SYS002,140
// LIBDEF RL,SEARCH=(VFLKMLI,VFORTLI),TEMP
*
// OPTION LINK,PARTDUMP
 ACTION MAP, CANCEL
// EXEC VFORTRAN,SIZE=AUTO
     (source program)
/*
// EXEC LNKEDT
// EXEC ,SIZE=AUTO
/&
```

Link-Editing Your Program

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

Note: FORTRAN66 object programs are link-edited in exactly the same way as FORTRAN77 object programs.

The following sections show how to catalog your object module or load module under VSE, and how to use the VSE linkage editor.

Note: Severity levels higher than level 4 prevent link-edit processing.

You specify the FORTRAN compiler options as options in the PARM parameter of the EXEC job control statement for VSE/Advanced Functions Release 3.0.

Automatic Cross-System Support

In VS FORTRAN, you can compile your source program under any supported operating system. You can then link-edit the resulting object module under the same system, or under any other supported system.

For example, you could request compilation under VM and then link-edit the resulting object module for execution under VSE.

You don't have to request anything special during compilation to do this; VS FORTRAN uses the execution-time library for all system interfaces, so the operating system under which you link-edit determines the system under which you execute.

Link-Editing for Immediate Execution

The simplest way to link-edit for immediate execution is to use a link-edit-execute cataloged procedure.

You can also execute the program immediately after link-editing by including the execution step immediately after the link-edit step.

A third alternative is to execute the program using the GO parameter of the EXEC control statement for VSE/Advanced Functions Release 3.0. See "Executing Your Program" on page 344 for details on the execution step.

Cataloging Your Load Module

The librarian is a group of programs used for maintaining VSE libraries and for providing printed and punched output from the libraries.

- The **core image library** (system and private) contains the system control components and program phases.

  To catalog a load module (phase) in this library, specify the following statement *before* your link-edit step and before the PHASE statement for the program you want to catalog:

  ```
  // OPTION CATAL
  ```

  which places the phase output into a core image library.

  To invoke the linkage editor, specify the following statements in your link-edit step:

  ```
  // OPTION CATAL
     PHASE...
  // EXEC LNKEDT
  ```

  which opens the SYSLNK file and all of the linkage-editor statements are written out to SYSLNK.

- The **relocatable library** (system and private) contains the system control components and compiler logical input/output control system (LIOCS) object modules.

  To catalog your object module to this library, using the MAINT program, specify:

  ```
  CATALR
  ```

- The **source statement library** (system and private) contains sequences of source statements, and macro definitions.

  To catalog your source programs to this library, using the MAINT program, specify:

  ```
  CATALS
  ```

- The **procedures library** (system only) contains cataloged procedures.

  To catalog a procedure to this library, using the MAINT program, specify:

  ```
  CATALP
  ```

Maintenance and service are the major functions performed for both the private and the system libraries by the librarian programs. Maintenance includes the addition, deletion, or copying of the items in the library. Service includes the translation of information in a library to printed or punched form. Information in a library directory and in header records can also be displayed.

For a more detailed description of these libraries and how to catalog in them, see *VSE/Advanced Functions System Management Guide.*

Executing a Link-Edit

Under VSE, if you specify // OPTION LINK or // OPTION CATAL *prior* to compile time, the only control statement you need to link-edit your program is the EXEC LNKEDT statement. VSE has the autolink feature which, unless you suppress it, always resolves all object module references to external-names, after all the input modules have been read from SYSLNK, SYSIPT, and/or the relocatable library. Ordinarily, you shouldn't suppress it. However, if you use service routines or extended error handling routines, you will need INCLUDE statements, as shown in Figure 63 on page 348.

In addition to the EXEC LNKEDT statement and the autolink feature, there are other linkage editor statements you can use to control linkage editor functions:

ACTION specifies linkage editor options, as follows:

CANCEL requests immediate cancellation if errors occur.

CLEAR initializes the temporary portion of the core image library to binary zeros.

ACTION CLEAR should be used only under supervision of someone who understands the possible consequences.

BG/F1-Fn specifies where program is to be executed:

BG	background execution
F1-Fn	Execution in one of the foreground partitions, F1 through Fn

MAP specifies that a storage map and messages are to be printed.

NOMAP suppresses the MAP option; messages are to be printed.

NOAUTO suppresses the autolink feature for this run.

INCLUDE	(followed by module name) specifies that a module from the relocatable library or from SYSIPT (if the compiler DECK option was specified on an INCLUDE statement before the deck) and is to be included; an INCLUDE statement with a module name will include modules from the relocatable library.
INCLUDE	(followed by blanks) specifies that a deck is following and that the modules should be put on SYSLNK for the linkage editor.
PHASE	specifies a name for the phase (load module) to be produced; the starting address and a relocation factor can also be specified.

You can also specify control statements to control overlay requirements; for details, see Chapter 6, "Subprograms and Shared Data" on page 113.

Logical Units Used for Link-Editing

The following logical units are used during link-editing:

SYSLNK	used for linkage editor input
SYSRES	used for input in form of relocatable object modules
SYSRLB	used for linkage editor input from the private relocatable library; this is the library used for FORTRAN library subroutines. If the alternative mathematical library subroutines are preferred over the comparable VS FORTRAN library routines, the standard routines should be replaced at installation time. If you want both libraries on the system, one must be placed in a separate private library at installation time, and a LIBDEF statement must be used to select subroutines desired at execution time.
SYSLST	used for system and compiler printed output
SYSCLB	used for output placed in a private core image library
SYS001	used as a linkage editor work file

Linkage Editor Output

Output from the linkage editor is in the form of load modules (or phases) in executable form. The exact form of the output depends upon the options in effect when you requested the link-edit, as described in the previous sections.

After link-editing, one or more of the following external references may remain unresolved and may be ignored: IBCOM#, IFYVASYN, IFYVCMSE, LDFIO#, and VLDIO#.

Executing Your Program

The following sections describe the logical units you may need, and outline the job control statements you must use to execute your programs.

Specifying Execution-Time Options

To specify an execution-time option (XUFLOW or NOXUFLOW), use the following method:

```
// EXEC ,SIZE=AUTO,PARM='NOXUFLOW'
```

For more information, see "Using the Execution-Time Options" on page 200.

Load Module Logical Units

In a FORTRAN source program, input and output devices are referred to by data set reference numbers. With this form of reference, an object program compiled from a VS FORTRAN source program is not dependent upon the availability of any specific device at execution time.

A programmer can temporarily assign a logical unit name to a specific input/output device using an ASSGN job control statement. Programmers need not make assignments unless their requirements differ from the standard assignments.

There is a direct correlation between the logical unit names, such as SYSIPT, SYSPCH, and SYS001, and the data set reference numbers used by a VS FORTRAN programmer. The first two columns of Figure 62 show the logical unit name and its associated data set reference number. This figure also illustrates the relationships of logical unit names and data set reference numbers that are under systems other than VSE.

The operator may assign available input/output devices. These are assigned to meet the logical unit requirements of the operating system. One device, for example, must be assigned as SYSIPT to serve as the system's main input unit.

Notes:

1. *VSE does not support extended precision.*

2. *FORTRAN object programs with asynchronous I/O cannot run on VSE.*

FORTRAN Reference Number	Logical Unit	VSE File Name	Function (Primary)	Device Type
0	SYS000	IJSYS00	Program data set	Unit record Magnetic tape Direct access
1	SYS001	IJSYS01	Program data set	Unit record Magnetic tape Direct access
2	SYS002	IJSYS02	Program data set	Unit record Magnetic tape Direct access
3	SYS003	IJSYS03	Program data set	Unit record Magnetic tape Direct access
4	SYS004	IJSYS04	Program data set	Unit record Magnetic tape Direct access
5	SYSIPT or SYSIN	IJSYSIP	Input data set to load module	Card reader Magnetic tape Direct access
6	SYSLST	IJSYSLS	Printed output data	Printer Magnetic tape Direct access
7	SYSPCH	IJSYSPC	Punched output data	Card punch Magnetic tape Direct access
8 thru 99	SYS005 thru SYS096	IJSYS05 thru IJSYS96	Program data set	Unit record Magnetic tape Direct access

Figure 62. Load Module Logical Units

Note: Units 9 through 99 may be added by reassembling the VSFORTL macro and replacing the unit assignment table module (IFYUATBL). For more information, see "Compiler and Library Defaults" in *VS FORTRAN Compiler and Library Installation and Customization.*

Executing the Load Module

How you execute the load module (or phase) depends on the kind of job you're running: execute only; link-edit and execute; or compile, link-edit, and execute.

Execute Only

The job control statements you use are:

```
JOB Statement
LIBDEF
ASSGN Statements      (as required for execution)
DLBL/TLBL Statements  (as required for execution)
EXTENT Statements     (as required for execution)
EXEC Statement        (load module (or phase))

   (Input data to be processed)

End-of-Data Statement (if input data is on cards)
End-of-Job Statement
```

Link-Edit and Execute

The job control statements you use are:

```
JOB Statement
LIBDEF
OPTION LINK           (sets link option)
    or
OPTION CATAL          (sets link option and catalogs phase)
PHASE                 (required for // OPTION CATAL)
 INCLUDE Statements   (as required for linkage editing)
ASSGN Statements      (as required for linkage editing)
DLBL/TLBL Statements  (as required for linkage editing)
EXTENT
EXEC LNKEDT           (linkage editor)

   (Linkage editor execution)

ASSGN Statements      (as required for execution)
DLBL/TLBL Statements  (as required for execution)
EXTENT Statements     (as required for execution)
EXEC Statement        (load module)

   (Input data to be processed)

/* End-of-Data Statement (if input data is on cards)
/& End-of-Job Statement
```

Note: Unless you specify OPTION CATAL, the phase is deleted from the core image library after execution is completed.

Compile, Link-Edit, and Execute

The job control statements you use are:

```
JOB Statement
LIBDEF
OPTION LINK           (sets link option)
   or
OPTION CATAL          (sets link option and catalogs phase)
PHASE                 (required with // OPTION CATAL)
EXEC VFORTRAN         (VS FORTRAN Compiler)

   (VS FORTRAN source program)

Statement             (if source program is on cards)
ASSGN Statements      (as required for linkage editing)
DLBL/TLBL Statements  (as required for linkage editing)
EXTENT                (as required for linkage editing)
EXEC LNKEDT           (linkage editor)

   (linkage editor execution)

ASSGN Statements      (as required for execution)
DLBL/TLBL Statements  (as required for execution)
EXTENT Statements     (as required for execution)
EXEC Statement        (load module)

   (input data to be processed)

End-of-Data Statement (if input data is on cards)
End-of-Job Statement
```

Note: Unless you specify OPTION CATAL, the phase is deleted from the core image library after execution is completed.

For a complete description of job control statements, see *VSE/Advanced Functions System Control Statements.*

Load Module Execution-Time Output

The output that execution of your load module gives you depends upon whether or not there are errors in your program.

Execution without Error

If your program executes without error, and gives the results you expect, your task of program development is completed.

Execution with Errors

When your program has errors in it, your execution-time output may be incorrect, or nonexistent.

You may or may not get error messages as well. Any VS FORTRAN execution-time error messages you get come from the VS FORTRAN Library.

If you get output from the program itself, it may be exactly what you expected, or (if there are logic errors in the program) it may be output you didn't expect at all.

When this happens, you must proceed to the next step in program development, described in Chapter 9, "Executing Your Program and Fixing Execution-Time Errors" on page 185.

Extended Error Handling—VSE Considerations

When you're calling any of the extended error handling routines, or any of the service routines, you must identify the routine by its library module name through a linkage editor INCLUDE statement. Names you can use are shown in Figure 63.

VS FORTRAN Source Name	VS FORTRAN Library Name[1]
CDUMP/CPDUMP	IFYVDUMP
DUMP/PDUMP	IFYVDUMP
DVCHK	IFYVDVCH
ERRMON	IFYVMOPT
ERRSAV	IFYVMOPT
ERRSET	IFYVMOPT
ERRSTR	IFYVMOPT
ERRTRA	IFYVMOPT
EXIT	IFYVEXIT
OPSYS	IFYOPSYS
OVERFL	IFYVOVER
SDUMP	IFYSDUMP
XUFLOW	IFYVXMSK

[1]These names should be used in an INCLUDE statement when linkediting.

Figure 63. Library Names for Error Handling and Service Routines

Requesting an Abnormal Termination Dump

To request a dump, you can specify:

```
// OPTION PARTDUMP
```

Whenever abnormal termination occurs, this provides a dump of the partition storage, the registers, and the areas of the supervisor control blocks that relate to this partition. Information on interpreting dumps is found in the appropriate debugging guide, as listed under "Preface" on page iii.

Sequential Files—System Considerations

Each sequential file you use must be defined to the system through job control statements. To define each file to the system, you, optionally, specify an ASSGN statement.

You must specify a DLBL statement; however, BLKSIZE is only required if block size is different from that stated in the DTF for your program. For example, if the program asks for 1000 bytes and the file requires 3000 bytes, your DLBL statement should specify:

```
BLKSIZE   3000
```

You must also specify an EXTENT statement, which defines each area the file occupies on the direct access device.

The logical units you can specify and the names you can use for them are shown in Figure 62 on page 344.

Direct Files—System Considerations

Before you can write records into the file, you must preformat it, using the CLEAR disk utility. For documentation, see *VSE/Advanced Functions System Utilities.*

You must define each direct file to the system through job control statements. For information about job control statements, see "Job Processing" on page 257.

Note: You cannot use direct files on fixed block architecture (FBA) devices.

To define each file to the system, you optionally specify an ASSGN statement. You must also specify a DLBL statement (using the DA parameter only for DAM (direct access method) files) and an EXTENT statement to define each area the file occupies on the direct access device. In DAM files, records are accessed by cylinder head and/or record.

The logical units you can specify and the names you can use for them are shown in Figure 62 on page 344.

Input/Output—System Considerations

For every file your program uses, you may need labels. The system processes the volume and standard file labels when you open and close the file. System considerations for tape labels and for direct access labels are described in the following sections.

Tape Label Considerations

You specify magnetic tape labels through the TLBL statement; through this statement, you can specify the position of the file on the tape, the generation and version number for this file, and the expiration date.

Note: If the tape has no label, TLBL need not be specified. For more information about the TLBL statement, see *VSE/Advanced Functions System Control Statements.*

For additional details on magnetic tape label processing, see *VSE/Advanced Functions Tape Labels.*

DASD Label Considerations

You specify DASD file label information through the DLBL statement; through this statement, you can specify the identification of the file on the volume, the type of data set label to be used (sequential, direct, or VSAM), and the expiration date. For more information, see "Direct Files—System Considerations" on page 349. For additional detail on DASD label processing, see *VSE/Advanced Functions DASD Labels.*

For a complete description of job control statements, see *VSE/Advanced Functions System Control Statements.*

Defining FORTRAN Records—System Considerations

Your FORTRAN programs must define the characteristics of the data records it will process: their formats, their record length, their blocking, and the type of device upon which they reside.

Record Formats

VSE keeps control information about your data files in an internal DTF table. There is a DTF table for each logical unit, including the system logical units, that your program uses—except for SYSLOG.

The system builds the DTF tables dynamically as each logical unit is opened for the first time, using information it obtains from your program. Guidance information on the DTF tables is included in *VSE System Data Management Concepts* and *VSE/Advanced Functions Macro User's Guide*; reference information on the DTF tables is included in *VSE/Advanced Functions Macro Reference.*

DOS FORTRAN produces and accepts records that have a particular format depending on logical unit class, device type, and the type of FORTRAN input/output operations applying to that record.

The default maximum length of a formatted record depends on the logical unit or device, as shown in Figure 64.

VSE Logical Unit Name	Device Permitted	Type of Operation	Default Maximum Record Length
SYSIPT SYSIN	Card reader, tape unit, or disk storage unit	Input	80 bytes
SYSPCH	Punch card device, tape unit, or disk storage unit	Output	80 bytes
SYSLST	Printer, tape, or disk storage unit	Output	133 bytes for tape, printer, and disk

Figure 64. VSE Logical Units and Devices Allowed

By device, the default maximum size for each record follows:

Device	Default Maximum Bytes
Card reader	80
Card punch	80
Printer	Number of print position, plus one byte for carriage control
Tape	260[1]
Disk (sequential access)	260[1]
Direct access	As specified in an OPEN statement

1 This default can be changed by using the OPSYS subroutine as described in *VS FORTRAN Language and Library Reference.*

For unformatted input/output, a single WRITE or READ may cause the transfer of more bytes than are contained in a single record. VS FORTRAN organizes such data into two or more records.

VS FORTRAN provides 8 bytes of control information at the beginning of each record block. This information indicates the size of the record and whether it is part of a record that is continued in one or more other blocks. There is never more than one record for each block.

When this control information is needed, it is provided and maintained by VS FORTRAN. There's no need to consider it in writing your input/output instructions.

The first 4 bytes of control information for unformatted records constitute a block descriptor word; the next 4 bytes constitute a segment-descriptor word.

The maximum record size is 32767 for EBCDIC files, and 2048 for ISCII/ASCII files.

Cataloging and Overlaying Programs—System Considerations

In order to use the subprograms you write, you must catalog them in a library—so they're available to calling programs. For information on how to do this, see "Cataloging Your Object Module" on page 338.

Overlaying Programs in Storage

When you use the overlay features of the linkage editor, you can reduce the main storage requirements of your program by breaking the program up into two or more segments that don't need to be in main storage at the same time. These segments can then be assigned the same storage addresses and can be loaded at different times during execution of the program.

You must specify linkage editor control statements to indicate the relationship of segments within the overlay structure.

Keep in mind that, although overlays reduce storage, they also can drastically increase program execution time. In other words, you probably shouldn't use overlays unless they're absolutely necessary.

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

Specifying Overlays

The linkage-editor control statements you need to create an overlay phase in VSE are:

- PHASE linkage-editor control statement—which lets you divide your program into a number of phases.
- INCLUDE linkage-editor control statement—which lets you specify that a module from the relocatable library is to be included in the present phase.

Overlay Procedure: To keep from running into difficulty when you're using multiple phases, you should develop your overlay programs using the following procedures:

1. Select a program name for the root phase of the program, the first four characters of which are unique in the core image library.
2. Determine the phase structure of the program and assign phase names, the first four characters of which are the same as the root phase name.
3. Write the root phase to serve as a monitor, loading each of the other phases as needed.
4. Determine the subroutine requirements for each phase.
5. If any modules used are not in the root phase but in the relocatable library, specify NOAUTO in the PHASE statement for each phase that refers to a

module appearing in a later phase; when the PHASE statement is executed, the relocatable library is not searched and the later phases are not loaded.

You must explicitly specify each subsequent phase in an INCLUDE control statement for the root phase.

Using the CALL OPSYS Statement: Within the FORTRAN program, when you want to have a phase loaded, issue a CALL to the library routine OPSYS. Your program must do this before you invoke the SUBROUTINE or FUNCTION that you want to execute.

For example, to load and execute subprograms PHASE4 and PHASE6, you specify:

```
CALL OPSYS('LOAD','PHASE4  ')
     .
     .
     .
CALL PHASE4(A,B)
CALL OPSYS('LOAD','PHASE6  ')
     .
     .
     .
CALL PHASE6(D)
```

These CALL OPSYS statements result in the phases named PHASE4 and PHASE6 being loaded. (The phase name in the CALL OPSYS statement is always eight alphameric characters, with the name left-adjusted within the field and padded on the right with blanks.)

The CALL PHASE4 and CALL PHASE6 statements cause the subprograms PHASE4 and PHASE6 to be executed.

For reference documentation about the CALL OPSYS statement, see *VS FORTRAN Language and Library Reference.*

Note: You must explicitly identify the OPSYS service routine by its library module name (IFYOPSYS) and supply a linkage editor INCLUDE statement using this name to get the OPSYS module included into the root phase.

Chapter 15. Using VSAM with VS FORTRAN

VS FORTRAN 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
- For VSE only, VSE/VSAM-managed sequential files (using the VSE/VSAM Space Management for SAM feature); such files can be processed only sequentially

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 will be accessed given a department number as a key. This is permitted if alternate keys are specified as allowing 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 the corresponding primary keys of all records in the associated KSDS that contain the alternate key value. See the appropriate *Access Methods Services* manual for instructions on how to define an alternate index.

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 does not support all VSAM functions.)

VSAM Terminology

VSAM terminology is different from the terminology used for MVS files, for example, as shown in Figure 65.

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 65. VSAM Terminology

Defining VSAM Files

VSAM entry-sequenced, key-sequenced, and relative-record data sets can be processed by VS FORTRAN 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), and KSDS (INDEXED)

- 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

See your *Access Method Services* manual for further information.

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.
- Use VSE/VSAM-Managed SAM files to reduce the amount of manual control needed to organize and maintain your non-VSAM sequential files under VSE only.

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 execute 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, or a VSAM-managed sequential file.

For VSAM keyed, sequential, and direct files, the following examples assume that the data space your file is using has already been defined as VSAM space by the system administrator.

For 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 VSE/VSAM-Managed Sequential File

To define a VSE/VSAM-managed sequential file (using the VSE/VSAM Space Management for SAM Feature), you can specify:

```
DEFINE CLUSTER -
   (NAME(myfile4) -
    NONINDEXED -
    RECORDFORMAT(VB120) -
    RECORDSIZE(500) -
    RECORDS(200) -
    VOLUMES(666666))
```

which defines a sequential file named *myfile4*, suballocated in VSAM space.

RECORDFORMAT(VB120)
specifies that the file has variable blocked format with average logical records 120 bytes long.

RECORDSIZE(500)
specifies that there are four logical records in each block; 120 bytes for each logical record, plus 4 bytes for each record descriptor, plus 4 bytes for the block descriptor.

RECORDS(200)
specifies that there can be a maximum of 200 records in the space.

VOLUMES(666666)
specifies that the file is contained on volume 666666.

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 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, 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").

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 executing 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).

IBM Extension

2. By a VS FORTRAN 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 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.

End of IBM Extension

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 VSE and VM systems. The name that identifies a particular data definition statement is called a *ddname* in MVS and VM, and a *filename* in VSE.

IBM Extension

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 or VSE		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 or VSE 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.*

End of IBM Extension

Processing DEFINE Commands

After you've created your DEFINE command, you must execute 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 execute 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 executed, 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 TSO considerations, see Chapter 13, "Using VS FORTRAN under TSO" on page 317. For information about job control statements, see "Job Processing" on page 257.

For VSE: You specify the following job control statements to execute your DEFINE commands:

```
// JOB      DEFINE
// EXEC     IDCAMS,SIZE=AUTO

 (The DEFINE command as data)

/*
/&
```

To create or retrieve records in a VSAM file under VSE, you identify the file in DLBL and EXTENT statements. If *myfile1* is the name of the file defined by the DEFINE command, then this is the file ID that you specify in the DLBL control statement.

For example, to process *myfile1* in a FORTRAN program called *myprog*, you specify:

```
// JOB    VSAM1
// DLBL   filename,'myfile1',,VSAM
// EXTENT SYS015,vsamvol
// EXEC   myprog,SIZE=AUTO
//
```

When *myprog* is executed, the DLBL and EXTENT statements make *myfile1* (and the information in its catalog entry) available to the program. In the FORTRAN OPEN statement, *filename* is the name specified in the FILE parameter.

In the EXTENT statement, you need specify only the logical unit (SYS015) and the volume ID (*vsamvol*), which is the volume containing the VSAM catalog.

In the FORTRAN OPEN statement, the unit you specify must be equivalent to that specified in the EXTENT statement. Your system administrator can tell you the units valid for your organization.

For VSAM files, you must specify the SIZE parameter in the EXEC statement. Do not specify a size larger than the size of the partition the program will run in.

When a VSAM cluster is specified with the REUSE option, care must be taken when using the file for the first time. When the VSAM file is opened and the DLBL control statement describing the file has a DISP=NEW specification, the file is in Initial Load Status and can only have records written to it. To read this VSAM file, you must finish this job step and in the next job step have a DLBL control statement that has a DISP=OLD specification for the VSAM file. If you try to read the VSAM file in the **same** job step, you must have two DLBL control statements describing the same VSAM file. The WRITE DLBL control statement is specified with DISP=NEW, and the READ DLBL control statement is specified with DISP=OLD. You can read the file after all records have been written, and the file is closed and reopened with another DLBL control statement specifying DISP=OLD. The file will be empty when you CLOSE the file with the DLBL control statement specifying DISP=DELETE.

For information about job control statements, see "Job Processing" on page 331.

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 execute 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're processing VSAM files, you can use all the VS FORTRAN 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 66 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)**	**VSE/VSAM-Managed Sequential**
	Sequential Access	**Sequential Access**	**Direct Access**	**Keyed Access**	**Sequential Access**
OPEN	YES[1]	YES[1]	YES	YES	NO
WRITE	YES[6]	YES	YES[4]	YES[7]	YES
REWRITE	NO	NO	NO	YES	NO
DELETE	NO	NO	NO	YES	NO
READ	YES	NO[2] YES[3]	YES	YES	YES
BACKSPACE	YES	YES[5] YES[3]	NO	YES	YES
REWIND	YES	YES[5] YES[3]	NO	YES	YES
CLOSE	YES	YES	YES	YES	YES

Figure 66. FORTRAN Statements Valid with VSAM Files

Notes to Figure 66:

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 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 "Sequential Access I/O Statements" on page 100.

Using OPEN Statement—VSAM Sequential Files

When your program processes a VSAM sequential file, you must specify the OPEN statement. For VSAM sequential files, specify:

ACCESS='SEQUENTIAL'

Note: You cannot use an OPEN statement with a VSE/VSAM-managed sequential file. For VSE/VSAM-managed sequential files, you can use all other input/output statements valid for other non-VSAM sequential files. However, the ENDFILE statement has no meaning for a VSE/VSAM-managed sequential file and is treated as documentation. If your program contains an ENDFILE statement and processes a VSE/VSAM-managed sequential file, you will get a warning message.

Using READ Statement—VSAM Sequential Files

The READ statement for a VSAM sequential file has the same effect it has for other sequential files; records are retrieved in the order they are placed in the file. Therefore, you must use the sequential forms of the READ statement.

Using WRITE statement—VSAM Sequential Files

For VSAM sequential files, the WRITE statement places the records into the file in the order that the program writes them. If a VSAM sequential file is nonempty when your program opens it, a WRITE statement always adds a record at the end of the existing records in the file; thus you can extend the file without first reading all the existing records in the file.

After you've written a record into a VSAM sequential file, you can only retrieve it; you cannot update it. Thus, when processing a VSAM sequential file, you can't update records in place. That is, if you code the following statements:

```
READ ...
BACKSPACE ...
WRITE ...
```

the WRITE statement does *not* update the record you have just retrieved. Instead, it places the updated record at the end of the file. (If you want to update records, you should define the VSAM file as direct or keyed.

Using BACKSPACE Statement—VSAM Sequential Files

For VSAM sequential files, you can use the BACKSPACE statement to make the last record processed the current record:

- For a READ statement followed by a BACKSPACE statement, the current record is the record you've just retrieved. You can then retrieve the same record again.
- For a WRITE statement followed by a BACKSPACE statement, the current record is the record you've just written, that is, the last record in the file. You can then retrieve the record at this position.

Using REWIND Statement—VSAM Sequential Files

The REWIND statement for VSAM sequentially accessed files has the same effect it has for other sequential files: the first record in the file becomes the current record.

For VSAM sequential files, this means that you can rewind the file and then process records for retrieval only. If you attempt to update the records, you'll simply add records at the end of the file.

After a BACKSPACE or REWIND statement is executed, you cannot update the current record. If you attempt it, you'll simply add another record at the end of the file.

Processing VSAM Direct Files

VSAM direct files use VSAM relative record data sets (RRDS). You can process VSAM direct files using either direct or sequential access.

Using direct access, you supply the relative record number. 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.

Using sequential access, you access each record in turn, one after another, and you have no control over the relative record number. For this reason, if you use sequential access to load the file, there should be no gaps in the relative record number sequence.

When you're processing VSAM direct files, there are special considerations for the OPEN statement as well as for sequential, direct, and keyed access, as described in the following paragraphs. For general information, see "Direct Access I/O statements" on page 103.

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 will not be 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 can't add or update records with FORTRAN programs. (For updating and adding records, you must use direct access.)

The READ statement for a sequentially accessed VSAM direct file retrieves the records in the order they are placed in the file. The VS FORTRAN program gives you no way of determining the relative record number of any particular record you retrieve. (If you need to use the relative record number, you must use direct access.)

Except during file loading, the REWIND statement for a sequentially accessed VSAM direct file has the same effect it has for VSAM sequential files: the first record in the file becomes the current record, which is then available for retrieval. During file loading, the REWIND statement has the same effect as a CLOSE statement followed by an OPEN statement; the first record in the file is then available for retrieval.

Except during file loading, the BACKSPACE statement for a sequentially accessed VSAM file has the same effect it has for VSAM sequential files; the last record processed becomes the current record, which is then available for retrieval. During file loading, the BACKSPACE statement has the same effect as a CLOSE statement, followed by an OPEN statement, followed by file positioning to the last record written; the last record in the file is then available for retrieval.

Using Direct Access—VSAM Direct Files

You can 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 execute the BACKSPACE or REWIND statements with a directly accessed VSAM direct file; if you do, your program is terminated.

IBM Extension

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 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 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 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 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 statement usage in processing VSAM KSDS files, see "Keyed Access I/O Statements" on page 106.

End of IBM Extension

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'll get the VSAM error information for the operation attempted in the IOSTAT data item.

(If the error occurs while FORTRAN is processing it, you'll get an IOSTAT value that is the same as the VS FORTRAN 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.

IBM Extension

To inspect the codes, you can equivalence the IOSTAT variable with two integer items, each of length 2. After a VSAM input/output operation, you can then write out the two integer items, which contain the pair of VSAM codes. 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)
```

End of IBM Extension

The VSAM documentation for the system you're operating under gives the meaning of these return and reason codes. See the list of "Related Publications" at the beginning of this manual for VSAM publications titles.

Chapter 16. Using VS FORTRAN Interactive Debug with VS FORTRAN

You can use VS FORTRAN Interactive Debug (5668-903) to debug any VS FORTRAN program that executes in a CMS or TSO environment that was

- Compiled with Release 2.0 with the TEST option, or
- Compiled with Release 3.0 or 3.1 without the NOSDUMP option.

In either case, the program must be executed with the Release 3.1 VS FORTRAN library.

The operating environment may also include the Interactive System Productivity Facility (ISPF), product number 5668-960, and the Program Development Facility (PDF), product number 5665-268 for TSO and 5664-172 for CMS.

When the execution environment does include ISPF and PDF, you can do full-screen debugging, and split-screen editing or browsing of source or listing files while debugging.

The following sections describe what must be done before you can begin debugging:

- Compile the VS FORTRAN program with the appropriate options (see table below).
- Execute the program with the DEBUG option.

Complete reference information can be found in *VS FORTRAN Interactive Debug Guide and Reference*.

Compiling a VS FORTRAN Program

To use VS FORTRAN Interactive Debug with a VS FORTRAN program, you must compile the program with the SDUMP or TEST compiler option. This is the only consideration for debugging that is necessary at compile time. Any of the other VS FORTRAN compiler options may still be used. The only exception to this is: If you want to use TSO or CMS line numbers to debug your code, instead of FORTRAN internal sequence numbers (ISNs), you must specify the NOSDUMP and TEST compiler options. The following matrix illustrates the use of TEST, NOTEST, and NOSDUMP.

	SDUMP	NOSDUMP
TEST	Debug using ISNs. Allows you to run VS FORTRAN Interactive Debug.	Debug using CMS or TSO line numbers. Allows you to run VS FORTRAN Interactive Debug.
NOTEST (default)	Debug using ISNs. Allows you to run VS FORTRAN Interactive Debug.	No debugging is allowed using VS FORTRAN Interactive Debug.

Note: For more efficient VS FORTRAN execution, use the NOTEST option.

The process of compiling a VS FORTRAN program depends on the operating environment used. Examples of compilation commands for the various environments supported by VS FORTRAN Interactive Debug are shown below. In these examples, "progname" is the VS FORTRAN program name, and "option" is a VS FORTRAN compiler option.

ISPF/PDF: If the environment (either CMS or TSO) that is used includes ISPF and PDF, a panel is provided for compiling the VS FORTRAN program.

CMS: When you are executing under CMS without ISPF, the FORTVS command is used to compile a VS FORTRAN program. For example:

```
FORTVS progname (option1 option2 ...)
```

TSO: When you are executing under TSO without ISPF, the TSO CALL command is used to compile a VS FORTRAN program. For example:

```
CALL 'SYS1.FORTVS(FORTVS)' 'option1,option2,...'
```

Before using CALL, be sure to use the ALLOCATE command to define the compiler data sets; see "Requesting Compilation—CALL Command" on page 321.

Executing with the DEBUG Option

VS FORTRAN Interactive Debug is invoked by specifying DEBUG as an execution-time option when executing a VS FORTRAN program. Examples of possible execution commands are shown below. In these examples, "progname" is the program name and "testparm" may be either DEBUG or NODEBUG or it may be omitted. DEBUG invokes VS FORTRAN Interactive Debug; NODEBUG, which is the IBM default, does not.

TSO (without ISPF): When you are executing under TSO without ISPF, VS FORTRAN Interactive Debug is invoked with the CALL command:

```
CALL pgmname 'option'
```

where pgmname is the name of your VS FORTRAN program, and option is DEBUG or NODEBUG.

CMS (without ISPF): When you are executing under CMS without ISPF, VS FORTRAN Interactive Debug can be invoked in one of several ways. In any event, however, you must issue a GLOBAL TXTLIB command and a GLOBAL LOADLIB DDBALL command specifying all libraries required for normal execution and two additional libraries, CMSLIB and TSOLIB.

- Use the LOAD command to load your VS FORTRAN program.

  ```
  LOAD pgmname
  START * option
  ```

  where *pgmname* is the name of your VS FORTRAN program, and *option* is DEBUG or NODEBUG.

- Use the GENMOD command to generate a module, and then invoke the module.

  ```
  LOAD pgmname
  GENMOD pgmname
  pgmname option
  ```

- Use the LKED command to link-edit your VS FORTRAN program into a load library; issue a GLOBAL LOADLIB command, specifying the name of the load library, and then invoke the module.

  ```
  OSRUN pgmname PARM=option
  ```

ISPF/PDF: If the environment (either CMS or TSO) includes ISPF, a panel is provided for executing a VS FORTRAN program. The CMS ISPF panel is shown below.

```
    ------- FOREGROUND VS/FORTRAN INTERACTIVE DEBUG------


COMMAND ===>

ISPF LIBRARY:
  PROJECT ===>
  LIBRARY ===>
  TYPE    ===>
  MEMBER  ===>          (Blank for member selection list)

CMS FILE:
  FILE ID    ===>
  IF NOT LINKED, SPECIFY:
  OWNER'S ID ===>  DEVICE ADDR. ===>  LINK ACCESS MODE ===>

READ PASSWORD ===>

DEBUG OPTIONS:     (DEBUG or NODEBUG)
                 ===>

SYSLIB TXTLIB:  (VFORTLIB is already specified)
          ===>           ===>           ===>           ===>
```

Using the Split Screen (ISPF/PDF)

If your TSO/CMS environment with VS FORTRAN Interactive Debug includes ISPF with PDF, the screen can be split, and browsing or editing of source or listing files can be done *during* the debugging session. (The screen can be split using ISPF only, enabling you to perform some other task under ISPF; however, it is PDF that enables you to browse and edit.) The debug session may occupy one section of the screen. The other section may contain the file that you want to edit or browse.

To split the screen, type "SPLIT" on the command line. Before pressing the ENTER key, move the cursor to the position at which you want the screen to be split. The screen can also be split by moving the cursor to the position at which you want the screen to be split and pressing the PF key assigned to the split function (usually the PF2 key). After the screen is split, ISPF/PDF panels will guide you.

Chapter 17. Using VS FORTRAN under VM/PC

There are three different methods you follow to use VS FORTRAN under VM/PC. (You may want to ask your system administrator to do these tasks for you.)

1. Create the VS FORTRAN module on VM/PC. Then load the module into a nucleus extension with the NUCXLOAD command. (Creating the module is necessary only when you first access VS FORTRAN, or after a new release has been installed on the host system; however, you must issue the NUCXLOAD command after every Initial Program Load (IPL) of CMS.)

2. Copy (download) the VS FORTRAN modules onto local disk files and then invoke VS FORTRAN in local sessions. (You need to download only when you first access VS FORTRAN, or after a new release has been installed on the host system.)

3. Link to the host-system minidisk containing VS FORTRAN and then access it from the local session as a remote minidisk. Depending on your link with the system and on the system load, this often is not an efficient way to operate.

Using NUCXLOAD with VS FORTRAN

You can decrease the processing time needed to access VS FORTRAN repeatedly by installing VS FORTRAN as an extension of your nucleus.

You must first create the VS FORTRAN compiler module on your VM/PC system. After you have created the module, you can upload it into the host system; it can then be downloaded to other VM/PC stations. To create the module in your local VM/PC session, follow these steps. (You may be able to have your system administrator do this for you.)

1. Link and access the host system disk (cannot be your "A" disk) that contains the compiler text files. (You may need to read them from the installation tape and copy them onto a CMS minidisk.)

2. Issue the following commands:

```
GLOBAL TXTLIB CMSLIB
COPY LOADORD MACRO file-mode LOADORD TEXT A (REPLACE
LOAD IEAXPALL (CLEAR RLDSAVE
INCLUDE IFXOCMS IFXOCNTL LOADORD (CLEAR RESET IFXOCMS
GENMOD FORTVS (STR MAP FROM IFXOCMS RLDSAVE
RENAME LOAD MAP A FORTVS MAP A
```

These steps create a module named FORTVS that you can load into a nucleus extension of your virtual machine. For more information, see documentation on the LOAD, INCLUDE, NUCXDROP, and NUCXMAP commands in the *VM/PC User's Guide.*

After you have created the FORTVS module, you can upload it into the host system; it can then be downloaded to other VM/PC stations.

To load the FORTVS module into a nucleus extension, issue the following command:

```
NUCXLOAD FORTVS
```

You must issue this command each time you IPL CMS. (You can put the NUCXLOAD command into your PROFILE EXEC, which will issue it for you.)

Downloading VS FORTRAN into VM/PC

To use VS FORTRAN under VM/PC in local sessions, you can copy (download) certain VS FORTRAN modules into your local files. The modules you must copy are listed in Figure 67.

IFXOTRCE	TXTLIB	
FORTVS	MODULE	
VLNKMLIB	TXTLIB	
VFLODLIB	LOADLIB	
VALTLIB	TXTLIB	
VFORTLIB	TXTLIB	
IEAXPALL	TEXT (1)	extended precision module (2)
IEAXPDXR	TEXT (1)	extended precision module (2)
IEAXPSIM	TEXT (1)	extended precision module (2)

Figure 67. VS FORTRAN Modules Needed for Downloading

Notes to Figure 67:

1. See your system adminstrator; these modules should be in CMSLIB TXTLIB.

2. If you cannot find these modules, check with your system administrator.

Downloading is necessary only when you first access VS FORTRAN, or after a new release has been installed on the host system.

Note: If COPY commands fail during downloading, check with your system administrator.

See Figure 68 on page 384 for the commands you should issue. The procedure is as follows:

1. Link (if necessary) and access the local minidisk that is the target minidisk for the copy operation. If the target minidisk is your own minidisk, the link is not required.

2. Link and access the host minidisk that contains the VS FORTRAN modules and TXTLIBs.

3. Copy the VS FORTRAN modules and TXTLIBs from the host minidisk to the local minidisk. (This is known as downloading.)

4. Release the host VS FORTRAN minidisk; it is no longer required.

5. If you are compiling **and** executing, link and access the host minidisk that contains the file CMSLIB TXTLIB.

6. Copy CMSLIB TXTLIB to the local minidisk.

7. Release the host CMS minidisk; it is no longer required.

Virtual Storage Requirements: Approximately 2.0 megabytes

Minidisk Storage Requirements: Approximately 1.5 megabytes

Note: These storage requirements are for the VS FORTRAN compiler and library only; additional storage is needed for the source and/or object program files.

(1) Link and access the target minidisk.

```
CP LINK vm/pc-id ttt aaa W write-password
ACCESS aaa filemode1
```

(2) Link and access the host minidisk that contains the VS FORTRAN modules.

```
CP LINK host-id hhh bbb RR read-password REMOTE
ACCESS bbb filemode2
```

(3) Copy the files you need.

```
COPYFILE filename filetype filemode2 = = filemode1
```

(4) Release the VS FORTRAN host minidisk.

```
RELEASE filemode2 (DET
```

(5) Link and access the host minidisk that contains CMSLIB TXTLIB.

```
CP LINK host-id ccc bbb RR read-password REMOTE
ACCESS bbb filemode3
```

(6) Copy CMSLIB TXTLIB to the local VM/SP target minidisk.

```
COPYFILE CMSLIB TXTLIB filemode3 = = filemode1
```

(7) Release the CMS host minidisk.

```
RELEASE filemode3 (DET
```

Figure 68. CMS Commands to Download VS FORTRAN from a Local Session

Notes to Figure 68:

ttt	is the virtual address of the local target minidisk that will store the VS FORTRAN modules.
aaa	is an unused virtual address on the local VM/SP machine.
hhh	is the virtual address of the host minidisk that contains the VS FORTRAN modules.
bbb	is the virtual disk address you use to refer to the host disk.
ccc	is the virtual address of the host minidisk that contains CMSLIB TXTLIB.
filemode1	is the filemode of the local VM/PC machine.
filemode2	is the filemode of the host minidisk that contains the VS FORTRAN modules.
filemode3	is the filemode of the host CMS TXTLIB minidisk.

You must first make VS FORTRAN available on a minidisk you can access. For example:

```
CP LINK vm/pc-id ttt aaa RR read-password
ACCESS aaa filemode3
GLOBAL TXTLIB VFORTLIB CMSLIB
```

If VS FORTRAN is stored on your A disk, you can omit the LINK and ACCESS commands. (If you must issue these commands each time you log on to VM/PC, you can put them into your PROFILE EXEC, which issues them for you.)

Next, you can invoke VS FORTRAN through the following command:

```
FORTVS fortest
```

where *fortest* is the name of your source program (its filetype is FORTRAN).

In link mode, specify:

```
GLOBAL TXTLIB VLNKMLIB VFORTLIB CMSLIB
GLOBAL LOADLIB
```

and run your program.

You can also specify compiler options. For example:

```
FORTVS fortest (options
```

allows you to modify the default compiler options in force for your organization.

VS FORTRAN Programming Tips

You can improve compile time if you specify VS FORTRAN compiler options that do **not** request printed listings: NOSDUMP, NOLIST, NOMAP, NOSOURCE, and NOXREF.

You can also improve compile time if you specify the OPT(0) compiler option; however, this might slow down execution time.

VS FORTRAN Restrictions

Any VS FORTRAN restrictions on VM processing apply for VM/PC as well.

In addition, the following processing capabilities are not available when you are executing an object program in a local session:

- VSAM file processing is not available.
- Magnetic tape processing is not available.
- The IBM Graphical Data Display Manager (GDDM) is not available.

Appendixes

This section contains appendixes documenting the following auxiliary VS FORTRAN material:

Appendix A. Assembler Language Considerations

You can use assembler language subprograms with your FORTRAN main programs. In your FORTRAN programs, you can invoke the assembler subprogram in either of two ways: through CALL statements or through function references in arithmetic expressions.

This appendix describes the linkage conventions you must use in such assembler language subprograms to communicate with the FORTRAN program.

For documentation about assembler language programs, see "Preface" on page iii.

Subprogram References in FORTRAN

For each subprogram reference, the compiler generates:

- A contiguous argument list containing the addresses of the arguments; this makes the arguments accessible to the subprogram.

  If the calling program was compiled under Release 3.0 or later with LANGLVL(77), and if any arguments in the call to the assembler language subprogram are of character type, there will be a second argument list with pointers to the length of each character argument.

  If the length of the character arguments is not important to your assembler language program, you can write an assembler language program to reference the parameter list in the same way as for older releases of FORTRAN: G, HX, and DOS FORTRAN F, or for LANGLVL(66).

- A save area in which the subprogram can place information about the calling program.

- A calling sequence to pass control to the subprogram, using standard linkage conventions.

Argument List

The argument list contains addresses of variables, arrays, and subprogram names used as arguments. If the list contains character arguments, there is a second list that immediately follows the first containing addresses of the lengths of arguments. See the example under "Retrieving Arguments in an Assembler Program" on page 394. Each entry contains the address of an argument. The leftmost bit is set to binary 1 in the last entry.

The calling program places the address of the argument list in general register 1.

Save Area

The calling program contains a save area in which the subprogram places information: the entry point for this program, the address to which the subprogram returns, general register contents, and addresses of save areas used by programs other than this subprogram.

The calling program reserves 18 words of storage for this area.

The calling program places the address of the save area in general register 13.

Calling Sequence

The FORTRAN compiler generates a calling sequence to transfer control to the subprogram, placing the following addresses in the following registers:

- Register 13—the address of the save area.
- Register 1—the address of the argument list. (If there is no argument list, 0 (zero) is placed in general register 1.)
- Register 15—the entry address.
- Register 14—the return address.

The program then branches to the address in general register 15.

You can also use register 15 as a condition code register, and as a RETURN code register. The values you should use for these codes are:

0	when a RETURN statement is executed in the subprogram.
4*i	when a RETURN i statement is executed in the subprogram.

Linkage in Assembler Subprograms

You can use two types of assembler subprograms:

Called Subprograms—that is, assembler language subprograms that don't call another subprogram.

Called and Calling Subprograms—that is, assembler language subprograms that do call another subprogram.

The rules for coding such subprograms are somewhat different, so they are documented in separate sections following.

Called Assembler Subprograms

For assembler subprograms that don't call other subprograms, you must include the following linkage instructions:

1. An instruction naming the entry point for this subprogram.

2. Instructions to save (in the save area reserved by the calling program) any general registers this subprogram uses. (You don't need to save the contents of general registers 0 and 1.)

3. Before returning control to the calling program, instructions to restore the saved registers.

4. For a FUNCTION subprogram, instructions to return the function's value.

5. An instruction setting the first byte of the fourth word in the save area to one bits, to indicate return of control to the calling program.

6. An instruction returning control to the calling program.

In addition to these instructions, if arguments are passed, the assembler subprogram may need to transfer the arguments from the calling program and return the arguments to the calling program, using the address passed in general register 1.

Called and Calling Assembler Subprograms

An assembler language subprogram that calls another subprogram must contain the same linkage instructions as a called subprogram; it must also simulate the FORTRAN linkage conventions for calling subprograms. Therefore, it must also include:

- A save area and instructions to place entries into its save area

- A calling sequence and parameter list for the subprogram it is calling

- An instruction indicating an external reference to the subprogram it is calling

- Additional instructions in the return routine to retrieve entries from the save area

Character Argument Linkage Convention

The linkage convention for passing character arguments between subprograms is different from the linkage convention for passing arguments that are not character. FORTRAN 77 standards specify two attributes for each character argument:

The first attribute, required of all arguments, is *address*.

The second attribute, required of character arguments, is *length*.

The convention for supplying argument *address* is the same for character and noncharacter arguments. That is, in the calling subprogram, a sequence of addresses is entered in the order of the called subprogram's argument list: One word with an address for each argument in the list. A high-order bit is added to the last address to signify the end of the address list.

The convention for supplying argument *length* is to 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. A high-order bit is added to the last address to signify the end of the length list.

Note: For the case 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, a character word 'BZ00', 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 compiled object code is shown to the left, and the corresponding assembler instructions to the right.

```
LOC OBJECT CODE   ADDR2    SOURCE STATEMENT
 02   4110 C038   00038           LA     1,PL         (1ST ARG. ADDR.)
 06   58F0 C050   00050           L      15,=V(SUB)  (ADDR. OF SUB)
 0A   05EF                        BALR   14,15        (CALL SUB (A,B,C)
                                   .
                                   .
                                   .
 0C                       A       DS     CL5    (DEFINE A CHARACTER*6)
 11                       B       DS     CL11   (DEFINE B CHARACTER*11)
 1C                       C       DS     CL11   (DEFINE C CHARACTER*11)
 28                       FIVE    DS     F'5'   (DEFINE LENGTH OF A)
 2C                       ELEVEN  DS     F'11'  (DEFINE LENGTHS B,C)
 30                               DS     0F
 30  C2E90000                     DC     X'C2E90000' (BUFFWORD 'BZ')
 34  0000000C                     DC     A(PLL-PL)    (OFFSET TO PLL)
 38  0000000C             PL      DC     A(A)          (ADDR OF A)
 3C  00000011                     DC     A(B)          (ADDR OF B)
 40  8000001C                     DC     X'80',AL3(C)(ADDR OF C)
 44  00000028             PLL     DC     A(FIVE)      (ADDR OF A LEN)
 48  0000002C                     DC     A(ELEVEN)    (ADDR OF B LEN)
 4C  8000002C                     DC     X'80',AL3(ELEVEN) (A.C LEN)
                                  END
 50  00000000                            =V(SUB) (ADDRESS OF SUB)
```

Main Programs

If the main program is not a FORTRAN main program, you must establish certain FORTRAN linkages after you've established the save area and before you call the FORTRAN subroutine.

The linkages you establish cause initialization of return coding and interrupt exceptions, as well as opening of the error message data set.

If you don't do this and the FORTRAN subprogram terminates in error or with a STOP statement, any open FORTRAN data sets are not closed, and the results of the program termination are unpredictable.

An example of the "program code" might be a call to a FORTRAN subroutine SUB2:

```
FINIT    CSECT
*******************************************************************
*  THIS FIRST SECTION:
*     SAVES THE CALLER'S REGISTERS IN THEIR SAVE AREA
*     ESTABLISHES ADDRESSABILITY
*     SAVES THE ADDRESS OF PREVIOUS SAVE AREA IN OUR SAVE AREA
*     SAVES THE ADDRESS OF OUR SAVE AREA IN THE PREVIOUS SAVE AREA
*     SETS REGISTER 13 TO THE ADDRESS OF THE PREVIOUS SAVE AREA
*******************************************************************
         STM    14,12,12(13)     SAVE CALLERS REGISTERS
         BALR   12,0             ESTABLISH A BASE REG
         USING  *,12             LET ASSEMBLER KNOW ABOUT IT
         ST     13,SAVEAREA+4    STORE BACKWARD POINTER
         LR     11,13            PREVIOUS SAVE AREA TO REG 11
         LA     13,SAVEAREA      PICK UP OUR SAVE AREA
         ST     13,8(0,11)       STORE FORWARD POINTER
         LR     13,11            SAVE REG 13 ACROSS CALL TO FORTRAN
*******************************************************************
*   THIS SECOND SECTION FOLLOWS THE FIRST:
*     LOADS ROUTINE ADDRESS
*     CALLS VFEIN# TO INITIALIZE
*******************************************************************
         SR     1,1              INDICATES NO PARAMETER LIST
         L      15,=V(VFEIN#)    LOADS INITIALIZATION ROUTINE ADDR
         BALR   14,15            CALL VFEIN
*******************************************************************
*  CALL THE VS FORTRAN SUBROUTINE
*******************************************************************
         L      15,=V(SUB2)
         BALR   14,15
*******************************************************************
*  RETURN TO THE CALLER
*******************************************************************
         L      15,=V(EXIT)      CALL EXIT SERVICE RTN TO FINISH
         BALR   14,15
SAVEAREA DC     18F'0'
*******************************************************************
         END
```

Using FORTRAN Data in Assembler Subprograms

Your assembler language subprograms can use data defined in FORTRAN subprograms, data contained either in common areas or in argument lists.

Using Common Data in Assembler Subprograms

Assembler language subprograms can access data in both blank and named common areas.

Using Blank Common Data in Assembler Programs

To refer to the blank common area, the assembler language program must also define a blank common area, using the COM assembler instruction. Only one blank common area is generated; 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
name  DC V(name-of-common-area)
```

Retrieving Arguments in an Assembler Program

The argument list contains addresses of variables and arrays.

Each entry in the argument list is four bytes long and is aligned on a fullword boundary. The first bit of every entry except the last contains zero; in the first byte of the last entry, the leftmost bit is set to binary 1.

The calling program places the address of the argument list in general register 1.

The following is an example showing the form of the VS FORTRAN code:

```
ADCONS FOR PARAMETER LISTS

Reg. 1  0001B0  C2E90000          DC  AL4'C2E90000'
points  0001B4  0000001C          DC  AL4'0000001C'
here->  0001B8  00000120          DC  AL4'00000120'  R8
        0001BC  00000158          DC  AL4'00000158'  R16
        0001C0  00000128          DC  AL4'00000128'  C16
        0001C4  00000138          DC  AL4'00000138'  C32
        0001C8  000005D8          DC  AL4'000005D8'  CHAR7
        0001CC  000005DF          DC  AL4'000005DF'  CHAR17
        0001D0  800001F0          DC  AL4'800001F0'  CHAR1
        0001D4  000000E0          DC  AL4'000000E0'  8
        0001D8  000000C8          DC  AL4'000000C8'  16
        0001DC  000000C8          DC  AL4'000000C8'  16
        0001E0  000000E4          DC  AL4'000000E4'  32
        0001E4  000000D8          DC  AL4'000000D8'  7
        0001E8  000000D4          DC  AL4'000000D4'  17
        0001EC  800000DC          DC  AL4'800000DC'  1000
```

Note: Displacements to actual values are given.

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 execution 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 execution of the above instructions, 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 to be returned must be placed in a register in its internal storage representation. The register to be used is one of the following:

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 a character function, the last entry that points to the data and the last entry that points to the length in the argument list supply information needed to store the function's result.

The assembler program can return the character variable 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

To simulate the VS FORTRAN subprogram RETURN n statement, the assembler program can place the value:

```
4*n
```

in general register 15, where n is the nth asterisk argument in the dummy argument list.

For example, to return to the second statement number in the actual argument list, the assembler language program must contain:

```
LA 15,8
```

Internal Representation of VS FORTRAN Data

If you're using VS FORTRAN data in your assembler language programs, you should be aware of the formats VS FORTRAN uses within the computer.

The following examples show how VS FORTRAN data items appear in internal storage.

Character Items in Internal Storage

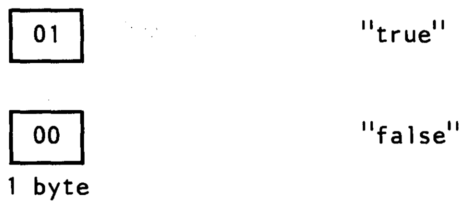
Character items are treated internally as one EBCDIC character for each character in the item.

Logical Items in Internal Storage

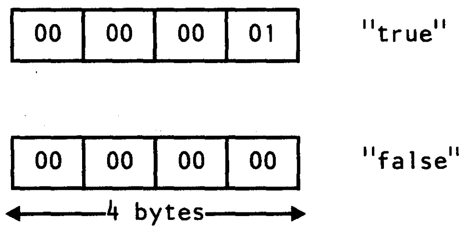
Logical items are treated internally as items either 1 byte or 4 bytes in length. Their value can be "true" or "false."

Their internal representation in hexadecimal notation is:

IBM Extension



End of IBM Extension



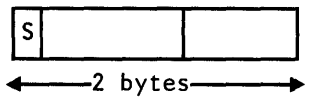
Integer Items in Internal Storage

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:

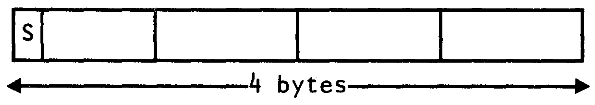
IBM Extension

INTEGER *2



End of IBM Extension

INTEGER *4



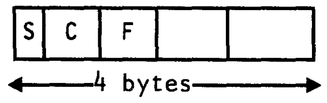
S = the sign bit

Real Items in Internal Storage

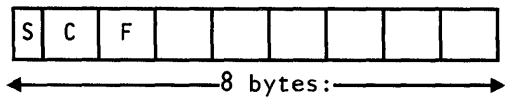
The compiler converts real items into 4-byte, 8-byte, or 16-byte floating-point numbers.

Their internal representation is:

REAL *4



DOUBLE PRECISION (REAL *8)



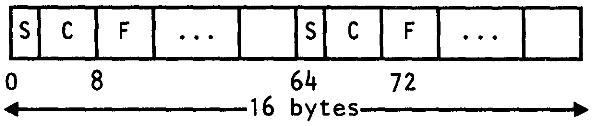
For REAL *4 and DOUBLE PRECISION items, the codes shown are:

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

IBM Extension

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

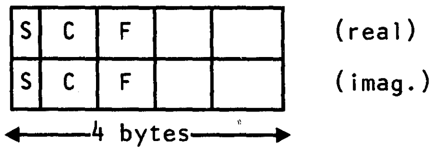
End of IBM Extension

Complex Items in Internal Storage

The compiler converts complex items into a pair of real numbers. The first number in the pair represents the real part; the second number in the pair represents the imaginary part.

The internal representations of complex numbers are:

COMPLEX *8

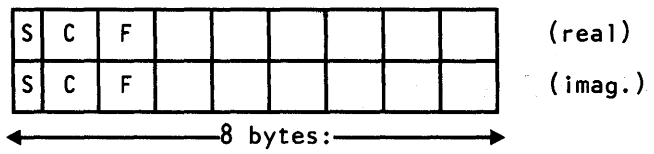


For COMPLEX *8 items, the codes shown are:

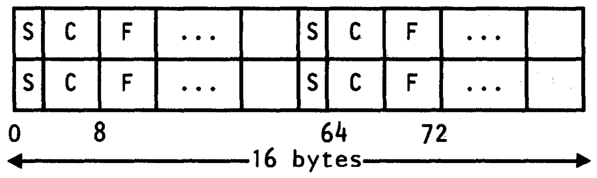
S = sign bit (bit 0)
C = characteristic, in bit positions 1 through 7
F = fraction, which occupies bit positions 8 through 31

IBM Extension

COMPLEX *16



COMPLEX *32



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

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

End of IBM Extension

Appendix B. Object Module Records

The object deck 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', which if punched on a card would produce a 12-2-9 punch. Positions 73 through 80 contain the first four characters of the program name followed by a 4-digit sequence number. The remainder of the record contains program information.

ESD Record

ESD records describe the entries of the *external symbol dictionary*, which contains one entry for each external symbol defined or referred to within a module. For example, if program MAIN calls subprogram SUBA, the symbol SUBA will appear as an entry in the external symbol dictionaries of both the program MAIN and the subprogram SUBA.

The linkage editor matches the entries in the dictionaries of other included subprograms and, when necessary, to the automatic call library.

ESD records are divided into four types, identified by the digits 0, 1, 2, or 5 in column 25 of the first entry, and column 57 of a third entry (there can be 1, 2, or 3 external-symbol entries in a record).

The contents of each type of ESD record are:

ESD Type	Contents
0	Name of the program or subprogram and indicates the beginning of the module.
1	Entry point name appearing in an ENTRY statement of a subprogram.
2	Name of a subprogram referred to by the source module through CALL statements, EXTERNAL statements, and explicit and implicit function references. (Some VS FORTRAN intrinsic functions are so complex that a function subprogram is called in place of in-line coding. Such calls are defined as *implicit function references*).
5	Information about a common block.

TXT Record

TXT records contain the constants and variables your source program uses, any constants and variables generated by the compiler, coded information for FORMAT statements, and the machine instructions generated by the compiler from the source module.

RLD Record

RLD records describe entries in the *relocation dictionary*, which contains one entry for each address that the linkage editor or loader must resolve before the module can be executed.

The relocation dictionary contains information that enables absolute storage addresses to be established when the module is loaded into main storage for execution. These addresses cannot be determined earlier, because the absolute starting address of a module cannot be known until the module is loaded.

The linkage editor or loader consolidates RLD entries in the input modules into a single relocation dictionary when it creates a load module.

RLD records contain a displacement to an area where an address is to be stored called through ESD type 2 records.

SYM Record

If you request it with the SYM compiler option, VS FORTRAN produces SYM records containing symbolic information for products like TSO TEST. SYM records are similar in form and content to those described in *Assembler H Version 2 Application Programming: Guide.*

SYM records are built for variables and arrays only. The locations of the variables or arrays are either in a LOCAL area (to the module) or in a common area. Note that if the common area is redefined from program unit to program unit, then the SYM records for the common area will 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	yyyyyy
1 byte	3 bytes	1–6 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

- yyyyyy is the name of the program (for example, MAIN, A, SUMM, FLEX, etc.).
- 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'38' indicates COMMON and no name (blank COMMON)

- yyyyyy is the name of the COMMON (for example, X, AAAA, YYYY, FFFF, etc.).

2. Data-type SYM card

Type	Displacement	Name	Variable Portion
xx	wwwwww	yyyyyy	zzzzzzzzzzzz
1 byte	3 bytes	1–6 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 name
  1 byte long
  X'81' indicates data, no multiplicity and a name
  2 bytes long
  X'82' indicates data, no multiplicity and a name
  3 bytes long
  X'83' indicates data, no multiplicity and a name
  4 bytes long
  X'84' indicates data, no multiplicity and a name
  5 bytes long
  X'85' indicates data, no multiplicity and a name
  6 bytes long

- 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

- 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, etc.).

- 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

END Record

The END record indicates:

- The end of the object module to the LOAD command
- The relative location of the main entry point
- The length (in bytes) of the object module

The format of the END record is as follows:

Columns	Contents
1	X'02'
2-4	END
5-28	Blank
29-32	Length of the CSECT

33	Character 1
34-71	5748-FO3 0400yyddd hhmmss cccccccc
73-80	Sequence number

where

yyddd	year and day of year
hhmmss	hour, minute, and second
cccccccc	8 characters reserved for a main program CSECT name such as MAIN (subprogram names are never present)

The structural order of a typical VS FORTRAN object module is shown in Figure 69 on page 410.

Record Type	Usage
ESD (Type 0)	Names object module
ESD (Type 1)	Names entry points (from ENTRY statements)
TXT	For FORMAT statements
TXT	For compiler-generated constants
ESD (Type 5)	For COMMON areas
ESD (Type 2)	For external references in CALL and EXTERNAL statements, and statements using subprograms
RLD	For external references in CALL and EXTERNAL statements, and statements using subprograms
SYM	For SYM information
TXT	For source program constants
ESD (Type 2)	For compiler-generated external references
RLD	For compiler-generated external references
TXT	For object module instructions
TXT	For the branch list
RLD	For the branch list
END	End of object module

Figure 69. Object Module Structure

Appendix C. Differences between VS FORTRAN and Other IBM FORTRANs

In VS FORTRAN, 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. See *VS FORTRAN Language and Library Reference* for a description of the bit functions.

In VS FORTRAN, the DEBUG statement and the debug packets precede the program source statements. The new END DEBUG statement delimits the debug-related source from the program source. For FORTRAN G1, the DEBUG statement and the debug packets are placed at the end of the source program.

In VS FORTRAN, evaluation of arithmetic expressions involving constants is performed at compile time (including those containing mixed-mode constants).

In VS FORTRAN, the number of arguments is checked in statement function references. The mode of arguments is checked for statement function references under LANGLVL(77) option only.

In VS FORTRAN, the form of the compiler option to name a program is NAME(nam) under LANGLVL(66).

Arguments are received only by location (or name) in LANGLVL(77). The default in LANGLVL(66) and for FORTRAN H and FORTRAN H Extended is receipt by value with the facility, to allow receipt by name by the use of slashes around the dummy argument in the SUBROUTINE, FUNCTION, or ENTRY statements.

The appearance of an intrinsic function name in a conflicting type statement has no effect in LANGLVL(77), but is considered user-supplied under LANGLVL(66) and FORTRAN H and FORTRAN H Extended.

Under VSE, direct access files must be preformatted when using LANGLVL(77). This is done by the DEFINE FILE statement under LANGLVL(66).

The extended range of a DO-loop is not part of the VS FORTRAN language. It is a valid construction under LANGLVL(66). The use of the extended range of the DO-loop under LANGLVL(77) is not diagnosed.

In VS FORTRAN, when a variable has been initialized with a DATA statement, that variable cannot appear in a subsequent explicit type statement and a severity level diagnostic is issued. FORTRAN H and FORTRAN H Extended allow typing following the data initialization. This is nonstandard usage. FORTRAN G1 issues a level 8 error diagnostic.

The record designator for direct-access I/O is required to be an integer expression for both LANGLVL(66) and LANGLVL(77). If it is not, VS FORTRAN diagnoses with a level 12 error message. FORTRAN H and FORTRAN H Extended permit this designator to be of real type. FORTRAN G1 diagnoses with a level 8 error message.

In VS FORTRAN, all calculations for arrays with adjustable dimensions are performed by a library routine called at all entry points that specify such arrays. This method was required for LANGLVL(77) because it permits redefinition of the parameters with adjustable dimensions in the subprogram but requires that the array properties do not change from those existing at the entry point.

The DOS FORTRAN IV logical unit SYSLOG is not supported by VS FORTRAN. This logical unit was used as a console log for output only. In the VSE environment, messages can now be displayed on the console by means of the PAUSE statement.

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 library follows the ANS 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 produces 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 ANS 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 produces:

```
13.70000E-06
```

The previous implementation produces:

```
13.7000E-06
```

FORTRAN G1, FORTRAN H Extended, and VS FORTRAN 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 library routine 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 library routine is used.
- VS FORTRAN generates 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.

Passing Character Arguments

In releases prior to Release 3 of VS FORTRAN for LANGLVL(77), character arguments are passed to a subprogram with both a pointer to the character string and a pointer to the length of the character string. This is required because the receiving program may have declared the dummy character arguments to have inherited length (that is, the length of the dummy argument is the length of the actual argument). The parameter list is therefore longer than for LANGLVL(66), because every character argument generates two items in the parameter list. For LANGLVL(66):

- Literal constants passed as arguments generate only one item in the parameter list.
- Hollerith constants may be passed as subroutine or function arguments.

In LANGLVL(77), a level 8 message is received if Hollerith constants are passed as arguments.

In both languages, only one item is generated in the parameter list for Hollerith arguments.

Every program that had been compiled with versions of VS FORTRAN 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 Release 3 or later.

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 prologs of FORTRAN subprograms are different in order to process these changed parameter lists. Therefore, if any FORTRAN program compiled prior to 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 Release 3 or later, then the old program must also be recompiled with Release 3 or later of VS FORTRAN.

Appendix D. Internal Limits in VS FORTRAN

Nested DO Loops

Nested DO loops and nested implied DO loops are limited to 25 levels.

Expression Evaluation

The maximum depth of the push-down stack for expression evaluation is 150. This means that, for any given expression, the maximum number of operator tokens that can be considered before any intermediate text can be put out is 150. For example, if an expression starts with 150 left parentheses before any right parentheses, this expression will exceed the push-down stack limit.

Nested Statement Function References

The total number of statement function arguments in any nested reference is limited to 50.

The total number of nested statement function references is limited to 50.

The total number of arguments in any statement function definition is limited to 20.

Nested INCLUDE Statements

The maximum number of nested INCLUDE statements is 16.

Nested Block IF Statements

Block IF statements may be nested to a depth of 25. That is, the number of IF... THEN, ELSE, and ELSEIF... THEN statements occurring before the occurrence of an ENDIF statement must be no greater than 25.

Character Constants

Character constants used in defining character symbolic names in PARAMETER statements and character constants in FORMAT statements are limited to a maximum of 255. Character constants used in PAUSE or STOP statements are limited to 72 characters. Character constants used as initialization values in DATA statements are limited to the number of characters that can be contained in 19 continuation cards.

Character constants used elsewhere in the program are limited to either the number of characters specified in the CHARLEN option (if it was used) or to the default value of 500.

Hollerith Constants

Hollerith constants are limited to a maximum of 255 in FORMAT statements.

Referenced Variables

The maximum number of referenced variables in a program unit is 2000.

Parentheses Groups

The maximum number of parentheses groups in a format is 50.

Statement Labels

Allowance has been made for up to 2000 user source labels and compiler-generated labels. However, if table overflow occurs at optimization level 2 or 3, the problem may be alleviated by removing all unreferenced user labels.

DISPLAY Statements

The maximum number of DISPLAY statements in a program unit is 100.

Repeat Count

The maximum number of times a given format code can be repeated is 255.

Index

VS FORTRAN
Programming Guide
SC26-4118-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems 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: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please 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.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ____________________

Previous TNL ____________________

Fold on two lines, tape, and mail. No postage stamp 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 directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SC26-4118-0

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 PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape **Please do not staple** Fold and tape

IBM®

VS FORTRAN
Programming Guide
SC26-4118-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems 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: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please 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.*

Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ________________

Previous TNL ________________

Fold on two lines, tape, and mail. No postage stamp 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 directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SC26-4118-0

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 PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape **Please do not staple** **Fold and tape**

IBM®

Printed in U.S.A. SC26-4118-0

VS FORTRAN
Programming Guide

File No. S370-40

SC26-4118-0

Printed in U.S.A.

SC26-4118-Ø

IBM

102721978