

**IBM Language Environment for MVS & VM
Fortran Run-Time Migration Guide
Release 5**

Document Number SC26-8499-00

Note!

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

First Edition (December 1995)

This edition applies to Version 1 Release 5 of Language Environment for MVS & VM, Program Number 5688-198, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department J58
P. O. Box 49023
San Jose, CA, 95161-9023
United States of America

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1995. All rights reserved.**

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

Contents

| | |
|---|---------|
| Notices | vi |
| Programming Interface Information | vi |
| Trademarks | vi |
| About This Book | vii |
| Using Your Documentation | vii |
| How to Read the Syntax Diagrams | viii |

Part 1. Planning the Migration 1

| | |
|--|-------|
| Chapter 1. Introducing Fortran with Language Environment | 2 |
| What Language Environment Is and How It Supports Fortran | 2 |
| Fortran Object Module Compatibility | 3 |
| Obstacles to Migrating Applications Containing Fortran | 3 |
| Fortran Load Module Compatibility | 4 |
| Chapter 2. Identifying the Incompatibilities | 5 |
| List of Incompatibilities | 5 |
| VS FORTRAN Facilities without Language Environment Counterparts | 6 |
| Chapter 3. Limited Use of Language Environment Facilities | 9 |
| Fortran Cannot Directly Call Language Environment Callable Services | 9 |
| Fortran Routines Cannot Be Included in Reentrant Load Modules | 9 |
| Fortran Routines Restricted to Initial POSIX Thread | 9 |
| Preinitialization Services Cannot Refer to Fortran Routines | 9 |

Part 2. Link-Editing and Running the Application 11

| | |
|--|--------|
| Chapter 4. Specifying the Language Environment Libraries | 12 |
| Libraries Used to Link-Edit and Run Your Application | 12 |
| Cataloged Procedures | 13 |
| Specifying Run-Time Libraries under TSO | 15 |
| Chapter 5. Removing VS FORTRAN Library Routines | 17 |
| Chapter 6. Declaring the Presence of Fortran Routines | 18 |
| Link-Editing Fortran Routines That Don't Call Fortran Library Routines | 18 |
| Dynamically Loading the First Fortran Routine in Your Application | 19 |
| Chapter 7. Resolving Conflicting Library Routine References | 21 |
| Step 1: Identifying the Conflicting References | 22 |
| Step 2: Recompiling Programs to Eliminate Conflicting References | 24 |
| Step 3: Automatically Resolving the Conflicting References | 25 |
| Step 4: Manually Resolving the Conflicting References | 30 |
| Chapter 8. Migrating VS FORTRAN Run-Time Options | 36 |
| Coding the Option String | 36 |
| Comparing the Individual Run-Time Options | 37 |
| Providing Default Run-Time Options for Your Application | 40 |

| | |
|--|----|
| Chapter 9. Interpreting Return Codes and Completion Codes | 41 |
| Specifying How Unhandled Conditions Should Be Reported | 41 |
| Return Codes | 42 |
| Completion (Abend) Codes | 44 |

Part 3. Changing Source Programs 47

| | |
|---|----|
| Chapter 10. Handling Run-Time Errors | 48 |
| Overview of the Language Environment Condition Handling Model | 48 |
| Overview of the VS FORTRAN Extended Error Handling Facility | 57 |
| Fortran-Specific Services for Error Handling | 59 |
| Handling Run-Time Errors from Your Fortran Routines | 63 |
| Regaining Control for Conditions Not Handled by a Subprogram | 73 |
| Chapter 11. Making Other Source Program Changes | 76 |
| Texts of Run-Time Error Messages | 76 |
| Values Returned through the IOSTAT Specifier | 77 |
| Permissible Input/Output to the Error Message Unit | 77 |
| Data Set Attributes for the Message File | 79 |
| Fix-Up for Misaligned Vector Instruction Operands | 80 |
| Fixed-Point Overflow | 80 |
| DVCHK and OVERFL Subroutines | 81 |
| Assembler Language Routines That Find Program Arguments | 82 |
| Run-Time Initialization from Assembler Language | 82 |

Part 4. Appendixes 87

| | |
|--|-----|
| Appendix A. Fortran Callable Services and Functions | 88 |
| AFHCEE—Invoke a Callable Service Passing the Feedback Code | 89 |
| AFHCEEN—Invoke a Callable Service Omitting the Feedback Code | 90 |
| QDFETCH Callable Service—Retrieve a Qualifying Datum of Any Type | 91 |
| QDLEN Function—Determine the Length of a Qualifying Datum | 92 |
| QDLOC Function—Obtain the Address of a Qualifying Datum | 92 |
| QDSTORE Callable Service—Update a Qualifying Datum | 93 |
| QDTYPE Function—Determine the Data Type of a Qualifying Datum | 94 |
| QDxxxx Functions—Retrieve a Qualifying Datum of a Specific Type | 95 |
| Appendix B. Qualifying Data for Language Environment Conditions | 97 |
| q_data Structure for Abends | 97 |
| q_data Structure for Arithmetic Program Interruptions | 98 |
| q_data Structure for Square-Root Exception | 101 |
| q_data Structure for Math and Bit-Manipulation Conditions | 102 |
| Format of q_data Descriptors | 105 |
| Appendix C. Message Number Mappings | 107 |
| Language Environment Conditions for VS FORTRAN Message Numbers | 107 |
| VS FORTRAN Message Numbers for Language Environment Conditions | 110 |
| Appendix D. VS FORTRAN Error Handling Behavior | 113 |
| Bibliography | 128 |
| Language Products Publications | 128 |

| | |
|--|------------|
| Related Publications | 129 |
| Softcopy Publications | 129 |
| Language Environment Glossary | 130 |
| Index | 139 |

Notices

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

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department J01, 555 Bailey Avenue, San Jose, CA 95161-9023. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming Interface Information

This book is intended to help with application programming. This book documents General-Use Programming Interface and Associated Guidance Information provided by Language Environment for MVS & VM.

General-Use programming interfaces allow you to write programs that obtain the services of Language Environment for MVS & VM.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

| | |
|-----------|----------------------|
| C/370 | IMS/ESA |
| CICS | Language Environment |
| CICS/ESA | MVS/ESA |
| DB2 | OpenEdition |
| DFSMS/MVS | VM/ESA |
| IBM | |

About This Book

IBM Language Environment for MVS & VM (Language Environment) provides common services and language-specific routines in a single run-time environment for C, C++, COBOL, Fortran, PL/I, and assembler applications. It offers consistent and predictable results for language applications, independent of the language they are written in.

This book describes what you must do to link-edit and run applications with Language Environment on MVS when these applications contain Fortran routines compiled by one of these Fortran compilers:

- FORTRAN IV G1
- FORTRAN IV H Extended
- VS FORTRAN Version 1
- VS FORTRAN Version 2

As you read this book, you will learn:

- Which Fortran load modules can run with Language Environment without link-editing them with Language Environment
- Which VS FORTRAN facilities aren't available in Language Environment
- Which Fortran object modules can be link-edited and run with Language Environment
- What source code changes are needed to run with Language Environment
- Where to find more detailed information

Before you read this book, you should be aware of the VS FORTRAN Version 1 or Version 2 facilities that are used by your applications. You should also have some understanding of the features of Language Environment and of the basics of link-editing and running applications.

Using Your Documentation

The publications provided with Language Environment are designed to help you:

- Manage the run-time environment for applications generated with a Language Environment-conforming compiler.
- Write applications that use the Language Environment callable services.
- Develop interlanguage communication applications.
- Plan for, install, customize, and maintain Language Environment.
- Debug problems in applications that run with Language Environment.
- Migrate your high-level language applications to Language Environment.

Language programming information is provided in the supported high-level language programming manuals, which provide language definition, library function syntax and semantics, and programming guidance information.

Each publication helps you perform a different task, some of which are listed in Table 1 on page viii. All books are available in both hardcopy and softcopy. For a complete list of publications that you may need, see “Bibliography” on page 128.

Table 1. How to Use Language Environment for MVS & VM Publications

| To ... | Use ... |
|--|---|
| Evaluate Language Environment | <i>Specification Sheet</i> <i>Concepts Guide</i> |
| Plan for Language Environment | <i>Concepts Guide</i> <i>Installation and Customization on MVS</i> <i>Run-Time Migration Guide</i> |
| Plan for installation, install, customize, and maintain Language Environment on MVS/ESA | <i>Installation and Customization on MVS</i> |
| Understand the Language Environment program models and concepts | <i>Concepts Guide</i> <i>Programming Guide</i> |
| Find syntax for Language Environment run-time options and callable services | <i>Programming Reference</i> |
| Develop applications that run with Language Environment | <i>Programming Guide</i> <i>Fortran Run-Time Migration Guide</i> and your language programming guide |
| Debug applications that run with Language Environment, get details on run-time messages, diagnose problems with Language Environment | <i>Debugging Guide and Run-Time Messages</i> |
| Develop interlanguage communication (ILC) applications | <i>Writing Interlanguage Communication Applications</i> , and your language programming guide |
| Understand warranty information | <i>Licensed Program Specifications</i> |
| Migrate applications to Language Environment | <i>Run-Time Migration Guide</i> <i>Fortran Run-Time Migration Guide</i> and other language migration guides |

How to Read the Syntax Diagrams

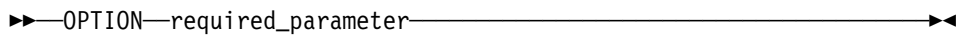
The following rules apply to the notation used in the syntax diagrams contained in this book:

- Read the syntax diagrams from left to right, top to bottom following the path of the line.
- Each syntax diagram begins with a double arrowhead (▶▶).
- An arrow (→) at the end of a line indicates that the option, service, or macro syntax continues on the next line. A continuation line begins with an arrow (▶).

- IBM-supplied default keywords appear **above** the main path or options path (see the sample on page x). In the parameter list, IBM-supplied default choices are underlined.
- Keywords appear in nonitalic capital letters and should be entered exactly as shown. However, some keywords may be abbreviated by truncation from the right as long as the result is unambiguous. In this case, the unambiguous truncation is shown in capital letters in the keyword, for example:

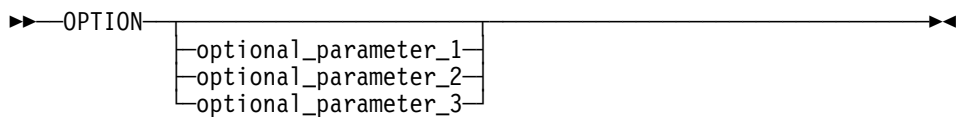
ANyheap

- Words in lowercase letters represent user-defined parameters or suboptions.
- Enter parentheses, arithmetic symbols, colons, semicolons, commas, and greater-than signs where shown.
- Required parameters appear on the same horizontal line (the main path) as the option, service, or macro:

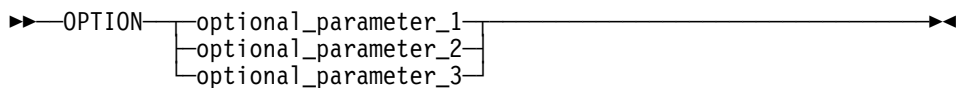


- If you can choose from two or more parameters, the choices are stacked one above the other.

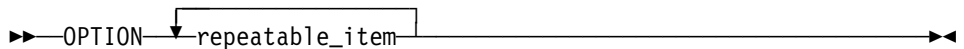
If choosing one of the items is optional, the entire stack appears below the main line.



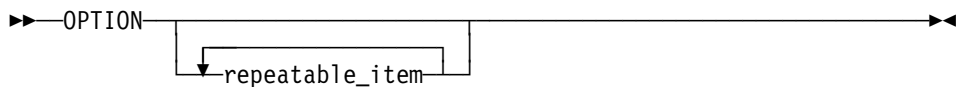
If you *must* choose one of the items, one item of the stack appears on the main path:



- An arrow returning to the left above a line indicates that an item can be repeated:



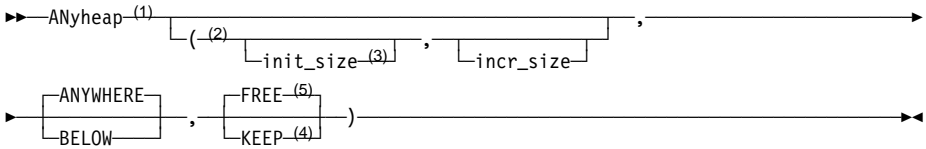
OR



- A comma or semicolon included in the repeat symbol indicates a separator that you must include between repeated parameters. These separators must be coded where shown.
- When entering commands, parameters and keywords must be separated by at least one blank if there is no intervening punctuation.
- A double arrow (→▶) at the end of a line indicates the end of the syntax diagram.

The following example demonstrates how to read the syntax notation. Numbers in the example correspond to explanations supplied below the example.

Format



The diagram illustrates the syntax for heap allocation. It shows the keyword `ANYheap` followed by an opening parenthesis. Inside the parentheses, there is an optional parameter `init_size` and an optional keyword `incr_size`, separated by a comma. After the closing parenthesis, there is another comma followed by two optional keywords: `ANYWHERE` and `KEEP`, which are separated by a vertical bar. The entire syntax is enclosed in a large right-pointing arrow.

Notes:

- 1 Keyword with minimum unambiguous truncation shown in capital letters
- 2 Opening parenthesis (must be specified if any parameters are specified)
- 3 Optional parameter
- 4 Optional keyword
- 5 Optional keyword (IBM-supplied default)

Part 1. Planning the Migration

| | |
|---|---|
| Chapter 1. Introducing Fortran with Language Environment | 2 |
| What Language Environment Is and How It Supports Fortran | 2 |
| Fortran Object Module Compatibility | 3 |
| Obstacles to Migrating Applications Containing Fortran | 3 |
| Fortran Load Module Compatibility | 4 |
| | |
| Chapter 2. Identifying the Incompatibilities | 5 |
| List of Incompatibilities | 5 |
| VS FORTRAN Facilities without Language Environment Counterparts | 6 |
| Parallel Programs | 6 |
| Extended Common Blocks (EC Compile-Time Option) | 6 |
| Alternative Mathematical Routines | 7 |
| Restricted Source Materials Tape | 7 |
| Internal Run-Time Library Interfaces | 7 |
| Self-Contained Load Modules (Link Mode) | 7 |
| Static Debug Packets | 8 |
| | |
| Chapter 3. Limited Use of Language Environment Facilities | 9 |
| Fortran Cannot Directly Call Language Environment Callable Services | 9 |
| Fortran Routines Cannot Be Included in Reentrant Load Modules | 9 |
| Fortran Routines Restricted to Initial POSIX Thread | 9 |
| Preinitialization Services Cannot Refer to Fortran Routines | 9 |

Chapter 1. Introducing Fortran with Language Environment

This chapter introduces you to Language Environment for MVS & VM and provides an overview of how applications containing Fortran routines can run with Language Environment. You will learn about the following:

- What Language Environment is and how it supports Fortran
- Fortran object module compatibility
- Obstacles to migrating applications containing Fortran
- Fortran load module compatibility

What Language Environment Is and How It Supports Fortran

Language Environment is a single run-time environment for applications written in C, C++, COBOL, Fortran, and PL/I. In addition to support for many existing applications, Language Environment provides common condition handling and improved interlanguage communication (ILC). Application development is simplified by the use of common conventions, common run-time facilities, and a set of callable services that can be used from various languages.

Prior to Language Environment, a high-level language product, such as VS COBOL II or VS FORTRAN Version 2, included a compiler, a run-time library, and, in some cases, a debugger. Language Environment is a single product that provides the functions of the run-time libraries for all five supported high-level languages.

C, C++, COBOL, and PL/I all offer new compiler products that don't include run-time library components, but depend instead on Language Environment for that support. For example, the new COBOL compiler product is COBOL for MVS & VM, which produces object modules that use Language Environment internal interface conventions. Therefore, you must link-edit and run these new object modules with Language Environment rather than with the run-time libraries from any of the predecessor COBOL products. With some restrictions, you can link-edit and run the code produced by certain earlier COBOL compilers.

In the case of Fortran, there is no new compiler that produces code specifically for Language Environment. However, on MVS but not on VM, you can use Language Environment to link-edit and run much of the code that various Fortran compilers, such as VS FORTRAN Version 2, produce. You can also construct applications in which Fortran routines call or are called by routines written in C, C++, COBOL, or PL/I. The considerations for these interlanguage applications are discussed in *Language Environment for MVS & VM Writing Interlanguage Communication Applications*.

To link-edit and run Fortran routines with Language Environment, you might need to change your source programs, JCL statements, or TSO commands. In some cases, these changes could be significant. This book helps you understand these changes. For a list of what might affect the migration of your applications that contain Fortran routines, see Chapter 2, "Identifying the Incompatibilities" on page 5.

Fortran Object Module Compatibility

Subject to the migration considerations discussed in this book, you can use Language Environment to link-edit and run the code produced by any of the following Fortran compilers:

- FORTRAN IV G1
- FORTRAN IV H Extended
- VS FORTRAN Version 1 ¹
- VS FORTRAN Version 2

When you link-edit your application with Language Environment, the resulting load module must not contain any run-time library routines from the VS FORTRAN Version 1 or Version 2 products. If your input to the linkage editor includes a VS FORTRAN executable load module, use Language Environment's set of linkage editor REPLACE statements to remove the VS FORTRAN library routines. For more information, see Chapter 5, "Removing VS FORTRAN Library Routines" on page 17.

Obstacles to Migrating Applications Containing Fortran

A number of VS FORTRAN Version 2 facilities are not available when you link-edit and run Fortran routines with Language Environment. For some of your applications, this might make the migration difficult or even impossible. Following are the major VS FORTRAN Version 2 facilities that aren't available with Language Environment:

- Parallel programs
- Extended common blocks (EC compile-time option)
- Extended error handling facility subroutines
- Automatic error fix-up actions
- Self-contained load modules (link mode)
- Interactive Debug
- Support for Fortran on VM

For a detailed list of the VS FORTRAN Version 2 facilities that either aren't available or require you to make some changes, see Chapter 2, "Identifying the Incompatibilities" on page 5.

Because the code produced by the Fortran compilers doesn't conform to the Language Environment interface conventions, you can't use certain Language Environment facilities, primarily the callable services, directly from a Fortran routine. For further information, see Chapter 3, "Limited Use of Language Environment Facilities" on page 9.

¹ There's one exception to the object module compatibility for VS FORTRAN Version 1: An object module cannot be link-edited with Language Environment if *both* of the following conditions are true:

- The program was compiled with VS FORTRAN Version 1 prior to Release 3
- The program either passes character arguments to a subprogram or is a subprogram that receives character arguments.

If you have such an object module, recompile it with VS FORTRAN Version 2.

Fortran Load Module Compatibility

On MVS (but not on VM), can use the Language Environment product as the run-time library for load modules that were link-edited with either the VS FORTRAN Version 1 or Version 2 library and that were link-edited so that library routines are loaded at run time. The existing load modules that you can run in this way are those that were link-edited in one of these ways:

- With the VS FORTRAN Version 1 Release 2, 3, or 3.1 library to use the MVS reentrant I/O library (sometimes called the IFYVRENT facility)
- With the VS FORTRAN Version 1 Release 4 or Release 4.1 library to run in load mode
- With the VS FORTRAN Version 2 library to run in load mode

To run your VS FORTRAN load module, the only change you need to make is to change your STEPLIB DD statement to refer to the Language Environment load library, CEE.V1R5M0.SCEERUN. For the applications that you choose to run in this way, none of the migration considerations in this book apply because the applications will run exactly as they did with the VS FORTRAN Version 2 Release 6 library. You can even run applications using Fortran facilities, like parallel programs, that aren't available to applications that are link-edited with Language Environment. However, once you link-edit one of your VS FORTRAN load modules with Language Environment, your application is subject to all of the considerations described in this book, such as the unavailability of certain VS FORTRAN Version 2 facilities, like parallel programs.

If you have an application that dynamically loads other parts of the application at run time, you must not link-edit any of the load modules with Language Environment unless you link-edit all of them with Language Environment.

Load Modules with Languages Other Than Fortran: If your VS FORTRAN load module contains one or more routines written in C, COBOL, or PL/I, in most cases you must link-edit the load module with Language Environment if you want to run it with Language Environment. The requirement to link-edit applies to load modules link-edited to run either in load mode or in link mode.

Load Modules That Run in Link Mode: Load modules link-edited with VS FORTRAN Version 1 or Version 2 to run in link mode (as opposed to load mode) do not require a library to be available at run time. These load modules are not affected by the presence of Language Environment.

Chapter 2. Identifying the Incompatibilities

This chapter lists the incompatibilities that you might encounter in migrating to Language Environment, and it explains how to deal with some of the VS FORTRAN facilities for which there are no Language Environment counterparts.

List of Incompatibilities

Table 2 shows various VS FORTRAN facilities that, if used in your application, could affect your migration to Language Environment. The “Status” column indicates that the facility either is different or is not available with Language Environment. On the referenced page, you can find information on how to address the incompatibility.

Table 2. Summary of Fortran Incompatibilities

| Facility | Status | Page |
|---|---------------|-------------|
| Extended error handling facility subroutines | Not available | 48 |
| Automatic error fix-up actions | Not available | 48 |
| Parallel programs | Not available | 6 |
| Extended common blocks (EC compile-time option) | Not available | 6 |
| Alternative mathematical routines | Not available | 7 |
| Values returned through the IOSTAT specifier | Changed | 77 |
| Permissible input/output to the error message unit | Changed | 77 |
| Data set attributes for the message file | Changed | 79 |
| Fixed-point overflow | Changed | 80 |
| Restricted source materials tape | Not available | 7 |
| Internal run-time library interfaces | Changed | 7 |
| Run-time options specified at program invocation | Changed | 36 |
| Link-editing an application containing Fortran routines | Changed | 11 |
| Self-contained load modules (link mode) | Not available | 7 |
| Return codes and completion (abend) codes | Changed | 41 |
| Loading modules through the ddname FORTLIB | Not available | 15 |
| Run-time initialization from assembler language | Changed | 82 |
| Assembler language routines obtaining program arguments | Changed | 82 |
| Texts of run-time error messages | Changed | 76 |
| Fix-up for misaligned vector instruction operands | Not available | 80 |
| DVCHK and OVERFL subroutines | Not available | 81 |
| Static debug packets | Not available | 8 |

VS FORTRAN Facilities without Language Environment Counterparts

The following sections discuss VS FORTRAN facilities that are not available and have no counterparts in Language Environment. In some cases, the only solutions are either:

- Continue running the application with VS FORTRAN Version 2.
- On MVS, without link-editing your application with Language Environment, change your STEPLIB DD statement to refer to the Language Environment load library, CEE.V1R5M0.SCEERUN. In this case, you cannot use any of the Language Environment facilities, such as the callable services or the improved communication among routines written in different languages.

Parallel Programs

You cannot link-edit and run parallel programs with Language Environment.

A *parallel program* is one that was compiled with the VS FORTRAN Version 2 Release 5 or 6 compiler and that either:

- Contains parallel language constructs
- Invokes any of the parallel callable services (PEORIG, PEPOST, PEWAIT, PETERM, PLCOND, PLFREE, PLLock, PLORIG, or PLTERM)
- Was compiled with the PARALLEL compile-time option

If you want to continue to run the program as a parallel program, link-edit the application with VS FORTRAN Version 2 Release 6, and run it either with the VS FORTRAN Version 2 Release 6 library or with the MVS version of Language Environment. However, you cannot use any of the Language Environment facilities in this case.

Programs using the Fortran multitasking facility (MTF) are not considered parallel programs. If your program uses MTF, you can link-edit and run it with Language Environment.

Extended Common Blocks (EC Compile-Time Option)

With Language Environment, you cannot link-edit and run programs that use extended common blocks. An *extended common block* is a common block whose name is specified in the EC compile-time option and that is created in a data space.

If your extended common blocks are small enough to fit in the primary address space, then recompile the programs that refer to them and provide the names of the common blocks in the DC compile-time option. If possible, reduce the size of your programs or the size of other data to make more space available in the primary address space.

If you can't restructure your programs and their data to fit in the primary address space, link-edit the application with VS FORTRAN Version 2 Release 6, and run it either with the VS FORTRAN Version 2 Release 6 library or with the MVS version of Language Environment. However, you cannot use any of the Language Environment facilities in this case.

Alternative Mathematical Routines

The VS FORTRAN Version 1 and VS FORTRAN Version 2 products each include a set of alternative mathematical routines that you can use instead of the standard set:

- With VS FORTRAN Version 1, the alternative set includes several routines that provide improved performance and accuracy.
- With VS FORTRAN Version 2, the alternative set includes several routines whose results are compatible with those of the standard set of routines in VS FORTRAN Version 1.

Neither of the alternative sets is available with Language Environment.

When no errors are detected, the mathematical routines used when you link-edit your Fortran routines with Language Environment provide exactly the same results as the standard set of routines in VS FORTRAN Version 2.

Restricted Source Materials Tape

There is no Language Environment counterpart to the VS FORTRAN restricted source materials tape, which contains the assembler language source code for many of the VS FORTRAN library routines. The source code isn't available for any of the components of Language Environment: the common component, the Fortran run-time library, or any of the other languages' run-time libraries.

If you've used the restricted source materials to modify the VS FORTRAN product, determine whether the general-use programming interfaces in Language Environment provide the functions you need. If they don't, contact your IBM representative for assistance.

Internal Run-Time Library Interfaces

Your applications probably won't run if they depend on any internal VS FORTRAN product interfaces that aren't documented in either *VS FORTRAN Version 2 Language and Library Reference* or *VS FORTRAN Version 2 Programming Guide for CMS and MVS*. If you've used any undocumented interfaces, restructure your applications to use only the general-use programming interfaces in Language Environment. In many cases, the Language Environment callable services should satisfy your needs. If they don't, contact your IBM representative for assistance.

Self-Contained Load Modules (Link Mode)

With VS FORTRAN, you can link-edit your program to operate in link mode, which produces a load module that contains all the required VS FORTRAN library routines; such a load module can run on a system that doesn't have the VS FORTRAN library installed. Language Environment has no link mode equivalent; the required run-time library routines are always loaded at run time. Therefore, you must have Language Environment installed on the system on which you run your application.

Static Debug Packets

The static debug facility of VS FORTRAN is not available with Language Environment. (Static debug includes the AT, DEBUG, DISPLAY, END DEBUG, and TRACE statements. The DEBUG and END DEBUG statements surround one or more debug packets, each of which begins with an AT statement.)

With Language Environment, you can link-edit and run object modules whose source code contains a debug packet. However, the debug packet, including all the statements in it, is ignored.

Chapter 3. Limited Use of Language Environment Facilities

This chapter lists some of the Language Environment facilities that aren't available to Fortran routines, and it describes some techniques that let you take advantage of much of what Language Environment offers.

Fortran Cannot Directly Call Language Environment Callable Services

A Fortran routine cannot directly call any of the Language Environment callable services, all of which have names beginning with CEE. The reason for this restriction is that Language Environment has extended the standard linkage convention for subroutine calls; currently, no Fortran compiler produces code that conforms to these new conventions.

The two Fortran-specific callable services AFHCEE and AFHCEEN permit Fortran routines to call most of the Language Environment callable services described in *Language Environment for MVS & VM Programming Reference*. For some examples of the use of the AFHCEE and AFHCEEN callable services, see "Fortran Services for Calling Language Environment Callable Services" on page 63. These and other Fortran-specific callable services are described in detail in Appendix A, "Fortran Callable Services and Functions" on page 88.

Fortran Routines Cannot Be Included in Reentrant Load Modules

Do not mix non-Fortran reentrant routines with Fortran routines in the same load module if you want the resulting load module to be reentrant. This restriction applies even if you compile the Fortran routine with the RENT compile-time option.

Including any Fortran routine in a load module makes the load module nonreentrant, and a nonreentrant load module must not be placed in a read-only area, such as a link pack area.

If you have a reentrant routine written in a language other than Fortran and you want to include it in a reentrant load module, dynamically load any Fortran routines that it calls. Use the other language's dynamic loading facility, such as the C `fetch()`, the COBOL dynamic call, or the PL/I `fetch` facility.

Fortran Routines Restricted to Initial POSIX Thread

Fortran routines can communicate with OpenEdition–conforming C routines, but the Fortran routines are restricted to the initial thread, and the main routine must be written in C.

Preinitialization Services Cannot Refer to Fortran Routines

The Language Environment preinitialization services let you use an assembler language routine to initialize the run-time environment once, perform multiple executions of routines within the environment, and explicitly terminate the environment. The routines that are invoked directly by these services cannot be Fortran routines. To circumvent this restriction, structure the application as shown in Figure 1 on page 10.

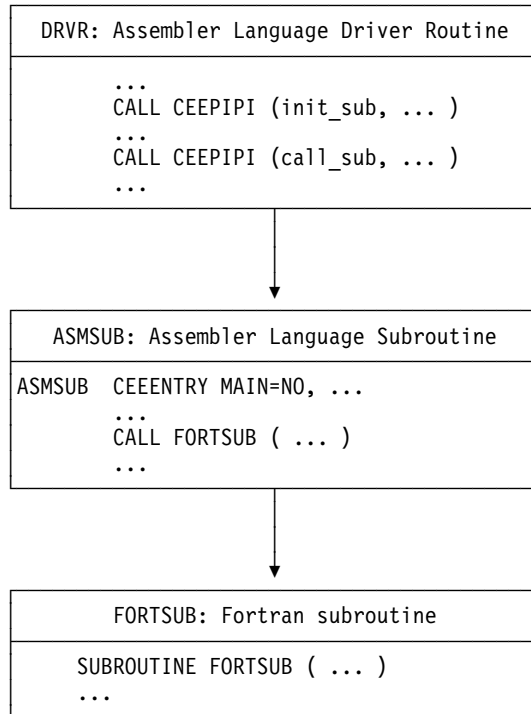


Figure 1. Assembler Language Routine to Invoke a Fortran Subroutine

Notes on the Example:

1. The first call to CEEPIPI from the routine DRVR initializes the run-time environment, allowing subroutines that use or depend on Language Environment services to be called.
2. The routine DRVR is not considered to operate as part of the run-time environment because it is neither a main routine nor is it called (directly or indirectly) by a main routine. Therefore, it can't call any Language Environment services other than the preinitialization services.
3. The second call to CEEPIPI invokes the assembler language subroutine ASMSUB. Because the *init_sub* function is specified in the first call to CEEPIPI, only subroutines (rather than main routines) can be called.
4. Because the routine called by CEEPIPI must conform to the Language Environment linkage conventions, the assembler language subroutine ASMSUB uses the CEEENTRY macro to establish these conventions. It then calls the Fortran subroutine FORTSUB.
5. After control is returned to it, the driver program DRVR can call ASMSUB again, or it can call other subroutines. While the subroutines are executing, they operate within the run-time environment established by the first call to CEEPIPI.

The preinitialization services provide several variations on the preceding scenario, such as allowing a main routine to be called. For further information on the preinitialization services and on the CEEENTRY macro, see *Language Environment for MVS & VM Programming Guide*.

Part 2. Link-Editing and Running the Application

Because the Fortran run-time library is part of Language Environment, there are numerous changes in the way you compile, link-edit, and run your Fortran programs. The following chapters explain the changes you'll have to make as you migrate to Language Environment. Most of the comparisons are between VS FORTRAN Version 2 Release 6 and Language Environment, although migration from VS FORTRAN Version 1 is similar.

| | |
|--|----|
| Chapter 4. Specifying the Language Environment Libraries | 12 |
| Libraries Used to Link-Edit and Run Your Application | 12 |
| Cataloged Procedures | 13 |
| Resolving Library Name Conflicts When Link-Editing Fortran Routines | 14 |
| Creating Cataloged Procedures for Fortran Compilations | 14 |
| Specifying Run-Time Libraries under TSO | 15 |
| | |
| Chapter 5. Removing VS FORTRAN Library Routines | 17 |
| | |
| Chapter 6. Declaring the Presence of Fortran Routines | 18 |
| Link-Editing Fortran Routines That Don't Call Fortran Library Routines | 18 |
| Dynamically Loading the First Fortran Routine in Your Application | 19 |
| | |
| Chapter 7. Resolving Conflicting Library Routine References | 21 |
| Step 1: Identifying the Conflicting References | 22 |
| Examining Your Programs to Find Conflicting References | 22 |
| Step 2: Recompiling Programs to Eliminate Conflicting References | 24 |
| Step 3: Automatically Resolving the Conflicting References | 25 |
| Fortran Library Routines but No C/C++ Library Routines | 26 |
| DFSMS/MVS and the Interface Validation Exit | 27 |
| Step 4: Manually Resolving the Conflicting References | 30 |
| Using the Conflicting References Removal Tool | 30 |
| Including Fortran Library Routines To Resolve Conflicting References | 34 |
| | |
| Chapter 8. Migrating VS FORTRAN Run-Time Options | 36 |
| Coding the Option String | 36 |
| Where to Code the Option String | 36 |
| Format of Option String | 37 |
| Comparing the Individual Run-Time Options | 37 |
| Providing Default Run-Time Options for Your Application | 40 |
| | |
| Chapter 9. Interpreting Return Codes and Completion Codes | 41 |
| Specifying How Unhandled Conditions Should Be Reported | 41 |
| Return Codes | 42 |
| Detecting the Return Code from the Completed Application | 42 |
| Specifying the Return Code in Your Fortran Routine | 43 |
| Interpreting Return Codes for Unhandled Conditions | 44 |
| Completion (Abend) Codes | 44 |

Chapter 4. Specifying the Language Environment Libraries

The Fortran run-time library, the run-time libraries of the other supported languages, and the common component of Language Environment are installed together in a single set of libraries. This chapter discusses the following topics to help you understand how to use these libraries to link-edit and run your application:

- Libraries used to link-edit and run your application
- Cataloged procedures
- Specifying load libraries under TSO

Libraries Used to Link-Edit and Run Your Application

Use the following libraries to link-edit and run your application with Language Environment. The names shown are the names supplied by IBM, but check with those who installed Language Environment at your site to see whether these names have been changed.

CEE.V1R5M0.SCEELKED

Contains the run-time library routines that the linkage editor includes in your load module along with your own routines. These library routines are sometimes called *resident run-time library routines* or simply *resident routines*.

CEE.V1R5M0.SCEELKED contains the resident routines for all of the high-level languages installed at your site and for the common library component of Language Environment.

When you link-edit your application, specify the data set CEE.V1R5M0.SCEELKED in the DD statement or ALLOCATE command with the ddname SYSLIB. This data set is similar to but is not an exact replacement for the VS FORTRAN Version 2 data set SYS1.VSF2FORT.

Important: Before you use CEE.V1R5M0.SCEELKED to link-edit an application containing Fortran routines, be sure you understand how to deal with the conflicting names that exist in the C/C++ and Fortran run-time libraries. For further information, see Chapter 7, “Resolving Conflicting Library Routine References” on page 21.

CEE.V1R5M0.SAFHFORT

Contains certain Fortran-specific run-time library routines that the linkage editor includes in your load module along with your own routines. It has routines, such as SQRT and EXIT, for which there are C-specific run-time library routines of the same name. You must use this data set in certain cases to correctly resolve potential name conflicts, as discussed in Chapter 7, “Resolving Conflicting Library Routine References” on page 21.

In some cases you must concatenate CEE.V1R5M0.SAFHFORT ahead of CEE.V1R5M0.SCEELKED in the SYSLIB input to the linkage editor to include the Fortran library routines rather than the corresponding C library routines in CEE.V1R5M0.SCEELKED.

The data set CEE.V1R5M0.SAFHFORT has no VS FORTRAN counterpart. However, for a Fortran-only application, the following concatenation of data sets replaces the VS FORTRAN Version 2 data set SYS1.VSF2FORT:

```
//SYSLIB DD DSN=CEE.V1R5M0.SAFHFORT,DISP=SHR
// DD DSN=CEE.V1R5M0.SCEELKED,DISP=SHR
```

CEE.V1R5M0.SCEERUN

Contains the library routines that are loaded at run-time. These routines are sometimes called *transient run-time library routines* or simply *transient routines*.

CEE.V1R5M0.SCEERUN contains the transient routines for all of the high-level languages installed at your site and for the common library component of Language Environment.

When you run your application, make CEE.V1R5M0.SCEERUN available in one of the following ways:

- Refer to CEE.V1R5M0.SCEERUN in a DD statement with ddname STEPLIB or JOBLIB.
- From TSO, use the MVS/TSO Dynamic STEPLIB Facility program offering (5798-DZW) to add CEE.V1R5M0.SCEERUN to your STEPLIB allocation. For an example of using this program offering, see “Specifying Run-Time Libraries under TSO” on page 15.
- Have your system programmer include CEE.V1R5M0.SCEERUN in the system link list so it is accessible without being referenced through a STEPLIB or JOBLIB allocation.

CEE.V1R5M0.SCEERUN is the Language Environment replacement for the VS FORTRAN Version 2 data set SYS1.VSF2LOAD.

Cataloged Procedures

Language Environment includes cataloged procedures for link-editing and running your programs. The C/C++, COBOL, and PL/I products also include procedures for compiling, link-editing, and running programs written in those languages. For details, see *Language Environment for MVS & VM Programming Guide*.

Table 3 summarizes the procedures that you're likely to use for an application containing Fortran routines:

Table 3. Cataloged Procedures Often Used with Fortran

| Name | Function of the Procedure |
|-------------|---|
| CEEWG | Load and run a program written in a language supported by Language Environment. |
| CEEWL | Link-edit a program written in a language supported by Language Environment. |
| CEEWLG | Link-edit and run a program written in a language supported by Language Environment. |
| AFHWL | Link-edit an application containing routines written in Fortran and possibly in languages other than C/C++. |
| AFHWLG | Link-edit and run an application containing routines written in Fortran and possibly in languages other than C/C++. |
| AFHWN | Link-edit with NCAL to facilitate changing external names in conflict between C/C++ and Fortran to names recognized by Fortran. |
| AFHWRL | Separate the nonshareable and shareable parts of a Fortran object module, then link-edit the program. |
| AFHWRLG | Separate the nonshareable and shareable parts of a Fortran object module, then link-edit and run the program. |

Resolving Library Name Conflicts When Link-Editing Fortran Routines

There are several resident run-time library routines that have the same name in both the Fortran and the C/C++ run-time libraries. In many cases, you must take special action to ensure that you link-edit the correct routine into your load module. For example, if the procedures CEEWG, CEEWL, and CEEWLG haven't been customized at your site, they might cause the C/C++ library routines to be included into your load module instead of the required Fortran routines. To avoid this problem when there are no C/C++ routines in your load module, you can sometimes use the procedures AFHWL and AFHWLG instead. For complete information on this subject, see Chapter 7, "Resolving Conflicting Library Routine References" on page 21.

Creating Cataloged Procedures for Fortran Compilations

None of the cataloged procedures that are part of Language Environment include a job step to compile a Fortran program. However, you can combine existing cataloged procedures to include the job steps you need. For example, if you need a cataloged procedure that compiles a Fortran program with the VS FORTRAN Version 2 compiler and then link-edits and runs that program with Language Environment, you could create a cataloged procedure, say VSF2ECLG, from the following:

1. Procedure step FORT from the VSF2CLG cataloged procedure (supplied as part of VS FORTRAN Version 2)
2. Procedure steps LKED and GO from the AFHWLG cataloged procedure (supplied as part of Language Environment)

In the combined procedure, make the SYSLIN DD statement in the LKED step refer to the data set (&&LOADSET in this case) referenced by the SYSLIN DD statement in the FORT step. Figure 2 on page 15 shows what that combined procedure might look like (with some of the symbolic parameters in the original VSF2CLG cataloged procedure removed to simplify the example).

In the example, the SYSLIB DD statement in the FORT step refers to the data set CEE.V1R5M0.SCEESAMP, which contains the following files that you can include in your Fortran source program:

- xxxFORCT—symbolic feedback code files. See "Symbolic Feedback Codes" on page 51.
- AFHCQDSB—type declarations for qualifying data functions. See page 61.

```

//VSF2ECLG PROC LIBPRFX='CEE.V1R5M0',
//          PGMLIB='&&GOSET',GOPGM=GO
//FORT EXEC PGM=FORTVS2,REGION=2100K,COND=(4,LT),
//          PARM='NODECK,NOLIST,OPT(0)'
//STEPLIB DD DSN=SYS1.VSF2COMP,DISP=SHR
//SYSPRINT DD SYSOUT=*,DCB=BLKSIZE=3429
//SYSTEM DD SYSOUT=*
//SYSLIB DD DSN=CEE.V1R5M0.SCEESAMP,DISP=SHR
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//          SPACE=(CYL,(1,1)),DCB=BLKSIZE=3200
//LKED EXEC PGM=HEWL,REGION=1024K
//SYSLIB DD DSNAME=&LIBPRFX..SAFHFORT,DISP=SHR
//          DD DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SCEESAMP DD DSNAME=&LIBPRFX..SCEESAMP,DISP=SHR
//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,PASS)
//          DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&PGMLIB(&GOPGM),
//          SPACE=(TRK,(10,10,1)),
//          UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED),REGION=2048K
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*

```

Figure 2. Sample Cataloged Procedure to Compile, Link-Edit, and Run a Fortran Program

Important: Before you create and use a cataloged procedure such as the one in Figure 2, be sure you understand how to deal with the conflicting names that exist in the C/C++ and Fortran run-time libraries. See Chapter 7, “Resolving Conflicting Library Routine References” on page 21.

Specifying Run-Time Libraries under TSO

Under TSO, VS FORTRAN Version 2 lets you provide an ALLOCATE statement with ddname FORTLIB to refer to the data sets that contain the dynamically loaded run-time library routines and your own dynamically loaded routines. There is no Language Environment equivalent of ddname FORTLIB. Instead, the Language Environment transient routines (in CEE.V1R5M0.SCEERUN) and your own dynamically loaded routines are loaded using the standard MVS search order:

1. Task libraries
2. STEPLIB
3. JOBLIB (if no STEPLIB)
4. Link pack area
5. System link list

There are three ways to make the required load libraries available to your programs when you run them under TSO:

- If you can update your LOGON procedure, include CEE.V1R5M0.SCEERUN and any of your own load libraries in its STEPLIB DD statement to make these libraries available during your TSO session. If you don't have the authority to update your LOGON procedure, ask your system programmer to make the change for you.

- If your site chooses to make the required load libraries available to all users, the system programmer should specify them in the system link list. Doing so makes the load libraries available without the need to refer to them in a batch job or TSO session.
- If it's not practical either to add the STEPLIB allocation to your LOGON procedure or to make the load libraries available in the system link list, use the MVS/TSO Dynamic STEPLIB Facility (5798-DZW). This program offering lets you modify your STEPLIB allocation to refer to the data sets you need to run your application.

For example, to add the data set CEE.V1R5M0.SCEERUN to your STEPLIB allocation, use the following commands, where the ddname SL is an arbitrary name:

```
ALLOCATE FILE(SL) DATASET('CEE.V1R5M0.SCEERUN') SHR  
STEPLIB SET(SL)
```

The STEPLIB allocation persists only during your current TSO session or until you change it with another STEPLIB command. For further information, see *MVS/TSO Dynamic STEPLIB Facility Program Description/Operations Manual*.

Chapter 5. Removing VS FORTRAN Library Routines

When you link-edit your application with Language Environment, your input to the linkage editor could include an executable load module that was link-edited with VS FORTRAN. For example, you would have to use such a load module if you didn't have the original object modules available and if you couldn't recompile one or more of your routines. To successfully link-edit and run such an application, you must remove the VS FORTRAN run-time library routines from the load module.

To remove library routines from a load module, use the Fortran library module replacement tool. This tool is a set of linkage editor REPLACE statements for removing Fortran library routines for these products:

- VS FORTRAN Version 1
- VS FORTRAN Version 2
- Language Environment

The REPLACE statements are in member AFHWRLK in data set CEE.V1R5M0.SCEESAMP. The following example shows how to remove the library routines from load module MYLMO and to link-edit it with Language Environment using cataloged procedure AFHWL:

```
//RELINK EXEC PROC=AFHWL,PGMLIB=USER.APPL.LOAD,GOPGM=MYLMO
//VSFLMO DD DSN=USER.VSF.LOAD,DISP=OLD
//SYSIN DD *
INCLUDE SCEESAMP(AFHWRLK)
INCLUDE VSFLMO(MYLMO)
/*
```

You don't need to include a DD statement referring to the data set CEE.V1R5M0.SCEESAMP because the AFHWL and AFHWLG cataloged procedures both contain one with ddname SCEESAMP.

The following example shows how to use the TSO LINK command to perform the same task:

```
LINK ('CEE.V1R5M0.SCEESAMP(AFHWRLK)', 'USER.VSF.LOAD(MYLMO)') +
LOAD ('USER.APPL.LOAD(MYLMO)') +
LIB ('CEE.V1R5M0.SAFHFORT', 'CEE.V1R5M0.SCEELKED') +
NOTERM
```

Important: Because certain names exist in both the C/C++ and the Fortran libraries, the preceding examples work properly only when there are no C/C++ routines in the load module. For further information, see Chapter 7, "Resolving Conflicting Library Routine References" on page 21.

Chapter 6. Declaring the Presence of Fortran Routines

When you link-edit an application that contains a Fortran routine, there are two cases in which you must include in your load module the *Fortran signature CSECT* (CEESG007), a resident library routine that Language Environment uses to recognize the presence of a Fortran routine in your application. Include the Fortran signature CSECT in your load module if either of the following is true:

- All of the Fortran routines in your load module are subprograms that don't use any Fortran library routines. For information on determining whether your subprograms use Fortran library routines and on including the Fortran signature CSECT, see “Link-Editing Fortran Routines That Don't Call Fortran Library Routines.”
- Your main program is written in a language other than Fortran, and the first Fortran routine in your application is dynamically loaded. See “Dynamically Loading the First Fortran Routine in Your Application” on page 19.

Link-Editing Fortran Routines That Don't Call Fortran Library Routines

You must include the Fortran signature CSECT in any load module for which all of the following conditions are true:

- The load module doesn't contain a Fortran main program.
- The load module contains Fortran subprograms (subroutines or functions).
- None of the Fortran subprograms calls a Fortran library routine.

If your Fortran routine does any of the following, you can assume that it calls a Fortran library routine (and therefore you don't have to include the Fortran signature CSECT):

- Refers to any mathematical or bit-manipulation intrinsic function other than simple ones (such as ABS, REAL, and MOD) for which inline code is generated
- Contains any I/O statement
- Calls any Fortran-specific callable service, such as FILEINF, AFHCEEF, or SYSRCS
- Contains a STOP or PAUSE statement
- Declares a common block that the DC compile-time option specifies as a dynamic common block

It might not always be obvious whether your Fortran subprograms use Fortran library routines; however, including the Fortran signature CSECT is permissible even when you don't have to do so.

The example in Figure 3 on page 19 shows how to compile and link-edit a COBOL main program that calls a Fortran subroutine (FSUB) that doesn't use any Fortran library routines.

```

//CMAIN EXEC IGYWCL
//SYSIN DD *
    IDENTIFICATION DIVISION.
    PROGRAM-ID. CBFC32I.
    ENVIRONMENT DIVISION.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    1 X PIC S9(9) USAGE IS BINARY.
    PROCEDURE DIVISION.
        MOVE 5 to X.
        CALL "FSUB" USING X.
        DISPLAY "UPDATED VALUE IN COBOL: ", X.
        GOBACK.
    END PROGRAM CBFC32I.
/*
//LKED.SYSIN DD *
    INCLUDE SYSLIB(CEESG007)
/*

```

Figure 3. Link-Editing a Fortran Subprogram That Calls No Library Routines

Notes on the Example:

1. The cataloged procedure IGYWCL compiles and link-edits the COBOL main program, which calls the Fortran subroutine FSUB. IGYWCL is described in *COBOL for MVS & VM Programming Guide*.
2. The INCLUDE statement directs the linkage editor to include the Fortran signature CSECT (CEESG007) in the load module. This INCLUDE statement should be used no matter which language is used to call the Fortran subroutine.

Dynamically Loading the First Fortran Routine in Your Application

If the first Fortran routine in your application is dynamically loaded, include the Fortran signature CSECT in at least one load module invoked prior to the dynamically loaded Fortran routine.

The example in Figure 4 on page 20 shows how to assemble and link-edit an assembler language routine that dynamically loads and executes a Fortran subroutine:

```

//AMAIN EXEC ASMACL
//C.SYSLIB DD
//      DD DSN=CEE.V1R5M0.MACLIB,DISP=SHR
//SYSIN  DD *
AMAIN   CEEENTRY MAIN=YES
        CEELoad NAME=FORTSUB
        LA    1,=A(ARG1,ARG2,ARG3+X'80000000')
        BALR 14,15
        CEETERM RC=0

        :
PPA     CEEPPA
        LTORG
        CEEDSA
        CEECAA
        END

/*
//L.SYSLIB DD DSN=CEE.V1R5M0.SCEELKED,DISP=SHR
//SYSIN  DD *
        INCLUDE SYSLIB(CEESG007)
/*

```

Figure 4. Link-Editing a Routine That Dynamically Loads a Fortran Subroutine

Notes on the Example:

1. The cataloged procedure ASMACL compiles and link-edits the assembler language routine. ASMACL is described in *High Level Assembler for MVS & VM & VSE Programmer's Guide, MVS & VM Edition*.
2. The CEELoad macro instruction dynamically loads the routine FORTSUB. The BALR instruction invokes the loaded routine.
3. The macros whose names begin with CEE are described in *Language Environment for MVS & VM Programming Guide*.
4. The INCLUDE statement directs the linkage editor to include the Fortran signature CSECT (CEESG007) in the load module. This INCLUDE statement should be used no matter which language is used to dynamically load the Fortran routine.

Chapter 7. Resolving Conflicting Library Routine References

The Fortran object modules you link-edit with Language Environment might contain references to some Fortran library routines with the same names as in the C/C++ library. For example, if one of your Fortran routines uses the SQRT function and was compiled with the FORTRAN IV H Extended compiler, the object module has a reference to the library routine SQRT. Although the Language Environment data set CEE.V1R5M0.SCEELKED, which is normally used to resolve such references, contains a library routine named SQRT, this library routine is a version of SQRT that can be link-edited only with C/C++ routines. Because the Fortran and the C/C++ SQRT routines are not interchangeable, you must ensure that the Fortran SQRT library routine is the one link-edited with your Fortran routines. This chapter discusses how to do this.

There are 20 library routines with conflicting names (listed in Table 4 on page 22); all of these routines in CEE.V1R5M0.SCEELKED are the C/C++ rather than the Fortran versions. Therefore, a potential problem arises whenever your application requires or might require the Fortran routines. When in doubt about the requirement for these routines, assume that they are required, and follow the process described in this chapter to resolve the conflict. (If the C/C++ component of Language Environment isn't installed at your site, you still must follow the process described here to include the Fortran routines.)

In the following discussion, the term *conflicting reference* means an external reference from an object module to one of the 20 conflicting names when the intended resolution is to a Fortran library routine rather than to a C/C++ library routine. Conflicting references aren't restricted to Fortran programs; a conflicting reference also occurs when an assembler language routine refers to one of the names and was written to use the Fortran (rather than C) library routine.

To solve the problem of conflicting references, use the following process when you link-edit applications that might require Fortran library routines:

1. Determine whether Fortran or assembler language code contains any conflicting references. See "Step 1: Identifying the Conflicting References" on page 22. If there are no conflicting references, no special actions are required.
2. If any conflicting references are present, determine whether they can be eliminated by recompiling the programs. See "Step 2: Recompiling Programs to Eliminate Conflicting References" on page 24. If recompiling the programs solves the problem, no further special actions are required.
3. If you can't recompile the programs to remove the conflicting references, determine whether the conflicting references are resolved automatically during link-editing. See "Step 3: Automatically Resolving the Conflicting References" on page 25. If the conflicting references are resolved automatically, no further special actions are required.
4. If steps 1, 2, and 3 don't provide a solution, see "Step 4: Manually Resolving the Conflicting References" on page 30.

The following sections discuss the preceding steps in detail and suggest the simplest possible solution for each of several cases that you need to identify.

Step 1: Identifying the Conflicting References

Table 4 lists the 20 Fortran library routine names that conflict with names in the C/C++ library. It also shows which products could generate each of the conflicting references.

Table 4. Existence of Fortran Conflicting References

| Library Routine Name | Product Used for Compilation | | | |
|---|------------------------------|--|---|----------------------------------|
| | Assembler (Any) | FORTRAN IV G1 FORTRAN IV H Extended | VS FORTRAN Version 1 VS FORTRAN Version 2 Rel. 1–4 | VS FORTRAN Version 2 Rel. 5 or 6 |
| ABS EXP ACOS GAMMA ASIN LOG ATAN LOG10 ATAN2 SIN COS SINH COSH SQRT ERF TAN ERFC TANH | Conflicting reference | Conflicting reference | If passed as an argument, ¹ conflicting reference Otherwise, no conflicting reference | No conflicting reference |
| CLOCK EXIT | Conflicting reference | Conflicting reference | Conflicting reference | Conflicting reference |

Note:

1. *Passed as an argument* means that one of the library routine names is provided as an actual argument in a call to a subroutine. For example, when compiled with the indicated compilers, the following code results in a conflicting reference to the SQRT library routine:

```

INTRINSIC SQRT
REAL*4 A
  ⋮
CALL SUB (A, SQRT)

```

Because passing an intrinsic function name as an argument is an infrequently used feature of the language, there are unlikely to be conflicting references in object modules produced by the VS FORTRAN Version 1 and Version 2 compilers.

If you're sure that none of your modules contain any of the 20 conflicting references, you don't have to worry about a solution to this problem and can skip the rest of this chapter. Otherwise, read the following sections.

Examining Your Programs to Find Conflicting References

If Table 4 suggests that your Fortran or assembler language routines might contain conflicting references, read this section to determine whether they really do. No matter whether your routines are available as source programs, object modules, or load modules, you can determine whether there are any conflicting references.

The analysis described in the following paragraphs might prove that you have no conflicting references. In this case, no further special actions are required.

Source Programs: If your Fortran or assembler language source programs are available, examine them for the use of the library routine names shown in Table 4.

Object Modules: If your Fortran or assembler language routines are in the form of object modules,² use an editor to examine the object modules. In the records with the characters ESD in positions 2 through 4, look for the 20 names that can be conflicting references. If you find one of these names, it is a conflicting reference unless it's the name of one of your own routines.

Load Modules: If your Fortran or assembler language routines are part of a load module and you don't have them in either source program or object module form, check the load module to determine which conflicting references, if any, are present:

- If you have the linkage editor's printed output from the creation of the load module (or if you can re-link-edit your load module to get the printed output), and if this output has a cross-reference listing, examine the references from the CSECTs for your Fortran or assembler language routines. Compare these references with the list of conflicting references in Table 4 on page 22.
- If the linkage editor output isn't available, use the AMBLIST service aid to get a listing of the references made by each CSECT in the load module. The following example shows how to use AMBLIST to get a cross-reference listing for the load module MOD1 in the data set USER.VSF.LOAD:

```
//L1      EXEC  PGM=AMBLIST
//SYSPRINT DD  SYSOUT=*
//SYSLIB  DD   DSN=USER.VSF.LOAD,DISP=SHR
//SYSIN   DD   *
          LISTLOAD OUTPUT=XREF,MEMBER=MOD1
/*
```

In the cross-reference listing produced by AMBLIST, look for the CSECTs that you know are Fortran or assembler language routines. Examine the references from these CSECTs for the use of the conflicting references shown in Table 4 on page 22.

For more information on the AMBLIST service aid, see one of the following:

- *MVS/ESA Diagnosis: Tools and Service Aids, MVS/ESA System Product: JES2 Version 4, JES3 Version 4*
- *MVS/ESA Diagnosis: Tools and Service Aids, MVS/ESA System Product: JES2 Version 5, JES3 Version 5*

No further special actions are required if your routines prove to have no conflicting references. If you do find conflicting references, then, as directed by step 2 of the process outlined on page 21, see “Step 2: Recompiling Programs to Eliminate Conflicting References” on page 24.

² The term *object module* means the output of a compiler or assembler prior to the output's being link-edited. Object modules consist of 80-character records; the first position of each record has the value X'02'.

Step 2: Recompiling Programs to Eliminate Conflicting References

Fortran Routines: If your Fortran source programs are available and they don't call CLOCK or EXIT, recompile them with the VS FORTRAN Version 2 Release 6 compiler to eliminate all of the conflicting references.

As Table 4 on page 22 shows, recompiling your Fortran programs won't eliminate conflicting references to the EXIT and CLOCK callable services, so:

- If your routines use the CLOCK callable service, then as directed by step 3 of the process outlined on page 21, skip to "Step 3: Automatically Resolving the Conflicting References" on page 25.
- If your routines use EXIT but not CLOCK, replace the use of the EXIT callable service either with a STOP statement or with a call to the SYSRCX callable service. Then recompile your routines with the VS FORTRAN Version 2 Release 6 compiler.

Assembler Language Routines: For assembler language routines with conflicting references, change the source programs as follows. Then reassemble the programs.

| Instead of this name: | Use this name: |
|----------------------------------|---------------------------|
| ABS | A#ABS |
| ACOS | A#COS |
| ASIN | A#SIN |
| ATAN | A#TAN |
| ATAN2 | A#ATAN2 |
| CLOCK | CLOCK# |
| COS | C#OS |
| COSH | C#OSH |
| ERF | E#RF |
| ERFC | E#RFC |
| EXIT | EXIT# |
| EXP | E#XP |
| GAMMA | G#AMMA |
| LOG | A#LOG |
| LOG10 | A#LOG1 |
| SIN | S#IN |
| SINH | S#INH |
| SQRT | S#QRT |
| TAN | T#AN |
| TANH | T#ANH |

The following examples show how to change two typical assembler language statements to remove conflicting references:

| Change this: | To this: |
|---------------------|---------------------|
| ABSADDR DC V(ABS) | ABSADDR DC V(A#ABS) |
| L 15,=V(COS) | L 15,=V(C#OS) |

After Recompiling or Reassembling

After changing your programs as necessary and recompiling them to remove any conflicting references, simply link-edit your programs using the method of your choice. If you're not yet familiar with link-editing with Language Environment, see Chapter 4, "Specifying the Language Environment Libraries" on page 12.

When Recompiling Does Not Provide a Solution

For routines whose conflicting references can't be removed by recompiling (such as routines for which you don't have the source code), continue with the following sections as directed by steps 3 and 4 of the process outlined on page 21.

Step 3: Automatically Resolving the Conflicting References

Even when you can't recompile your programs to eliminate the conflicting references, you'll still find that in many cases there is an easy solution. Following are summaries of two common cases; these summaries are followed by more detailed explanations.

Your load module requires Fortran library routines but no C/C++ library routines

In your link-edit step, concatenate CEE.V1R5M0.SAFHFORT ahead of CEE.V1R5M0.SCEELKED in the SYSLIB input, or use the cataloged procedures AFHWL or AFHWLG, which do the concatenation. This resolves the conflicting references to the Fortran library routines, and no further action is required. For detailed information, see "Fortran Library Routines but No C/C++ Library Routines" on page 26.

DFSMS/MVS Version 1 Release 3 is installed at your site and you don't have any conflicting references from assembler language routines

For your link-edit step, ensure that the Language Environment interface validation exit is activated in one of these places, as described in detail in "DFSMS/MVS and the Interface Validation Exit" on page 27:

- In your JCL for the link-edit step
- In your TSO LINK or LOADGO command
- In your site's cataloged procedures such as CEEWL and CEEWLG

This resolves the conflicting references to the Fortran library routines, and it resolves references from any C/C++ routines to the C/C++ library routines. No further action is required. For detailed information, see "DFSMS/MVS and the Interface Validation Exit" on page 27.

If neither case applies to your situation (for example, your load module has C/C++ routines, and it also has assembler language routines with conflicting references), you can still handle the conflicting references. In this case, as directed by step 4 of the process outlined on page 21, see "Step 4: Manually Resolving the Conflicting References" on page 30.

Fortran Library Routines but No C/C++ Library Routines

Even though Fortran or assembler language routines in your application contain conflicting references, you can include the Fortran library routines into your load module as long as there are no references to any C/C++ library routines.³ Depending on whether you're using one of the Language Environment cataloged procedures, there are two different ways to do this:

Without using Language Environment cataloged procedures

When you link-edit your load module, concatenate the data set CEE.V1R5M0.SAFHFORT ahead of CEE.V1R5M0.SCEELKED in the SYSLIB input. This causes the linkage editor to include the Fortran library routines from CEE.V1R5M0.SAFHFORT rather than the C/C++ library routines from CEE.V1R5M0.SCEELKED. Here is how you would do this using DD statements in your JCL:

```
//SYSLIB DD DSN=CEE.V1R5M0.SAFHFORT,DISP=SHR
//      DD DSN=CEE.V1R5M0.SCEELKED,DISP=SHR
```

Here is how you would do this using the LIB parameter in a TSO LINK or LOADGO command:

```
LIB('CEE.V1R5M0.SAFHFORT','CEE.V1R5M0.SCEELKED')
```

Using Language Environment cataloged procedures

Language Environment provides two cataloged procedures, AFHWL and AFHWLG, that include the concatenation of CEE.V1R5M0.SAFHFORT in the SYSLIB input to the linkage editor. Use these procedures instead of CEEWL and CEEWLG to include Fortran library routines rather than the C/C++ library routines.

The following example illustrates:

- Using the cataloged procedure AFHWL to link-edit a load module that contains Fortran but no C/C++ routines
- Including an existing load module, MYLMOD, as input to the linkage editor
- Removing the VS FORTRAN library routines by including AFHWRLK during the link-edit⁴

```
//RELINK EXEC PROC=AFHWL,PGMLIB=USER.APPL.LOAD,GOPGM=MYLMOD
//VSFLOAD DD DSN=USER.VSF.LOAD,DISP=OLD
//SYSIN DD *
INCLUDE SCEESAMP(AFHWRLK)
INCLUDE VSFLOAD(MYLMOD)
/*
```

³ If you have any C/C++ routines in your load module, assume that there are references to C/C++ library routines. Another example of references to C/C++ library routines is an assembler language routine that was written to use these routines.

⁴ As discussed in Chapter 5, "Removing VS FORTRAN Library Routines" on page 17, if your input to the linkage editor includes a load module containing VS FORTRAN library routines, you must remove these library routines by including AFHWRLK during the link-edit process.

The following example shows how to use the TSO LINK command to perform the same task:

```
LINK ('CEE.V1R5M0.SCEESAMP(AFWRLK)', 'USER.VSF.LOAD(MYLMOD)') +  
LOAD ('USER.APPL.LOAD(MYLMOD)') +  
LIB ('CEE.V1R5M0.SAFHFORT', 'CEE.V1R5M0.SCEELKED') +  
NOTERM
```

If you have C/C++ routines in your load module, read the following section for an alternative method for correctly resolving conflicting references.

DFSMS/MVS and the Interface Validation Exit

When both of the following conditions are satisfied, you can use the Language Environment interface validation exit to automatically resolve conflicting references:

- DFSMS/MVS Version 1 Release 3 is installed at your site.
- Your load module doesn't contain any assembler language routines with conflicting references.

If either condition isn't satisfied, then as directed by step 4 of the process outlined on page 21, see "Step 4: Manually Resolving the Conflicting References" on page 30.

Function of the Interface Validation Exit

The interface validation exit is a tool that, when used with the binder⁵ in DFSMS/MVS Version 1 Release 3, automatically resolves the conflicting references in your Fortran routines. It does this during the link-edit process by:

1. Recognizing object modules produced by one of the Fortran compilers whose object modules can be run with Language Environment
2. In these Fortran object modules, identifying any conflicting references not already resolved to one of your own routines
3. Directing the binder to resolve each identified reference to an equivalent Fortran-specific library routine with a different name
4. Allowing the corresponding library routine references in C/C++ routines (and from any non-Fortran routines) to be resolved to the C/C++ library routines in CEE.V1R5M0.SCEELKED

Activating and Using the Interface Validation Exit

If DFSMS/MVS Version 1 Release 3 is installed at your site, those who install Language Environment are directed to update the cataloged procedures such as CEEWL and CEEWLG to make certain the interface validation exit is available to everyone who uses the procedures. (Tailoring the cataloged procedures is discussed in *Language Environment for MVS & VM Installation and Customization on MVS*.) Once you ensure that the cataloged procedures have been updated, you can use them without taking any special action to activate the interface validation exit.

⁵ *Binder* is the term used in DFSMS/MVS for the replacement for the linkage editor. It has many features beyond those in the linkage editor (such as the ability to invoke an interface validation exit), although for most of the discussion in this book, the linkage editor and the binder are equivalent.

If the JCL or cataloged procedures used to link-edit your applications haven't been updated to activate the interface validation exit, do both of the following to activate it:

- Add the EXITS(INTFVAL(CEEPINTV)) binder option as follows to direct the binder to invoke the exit:
 - In your JCL for the link-edit step, include the option in the PARM parameter as follows:

```
PARM='...EXITS(INTFVAL(CEEPINTV))...'
```
 - For the TSO LINK or LOADGO command, include the option among any other options that you specify.

- Include the data set CEE.V1R5M0.SCEELKED in a STEPLIB allocation as follows to make the interface validation exit available to the binder:

- In your JCL for the link-edit step, include the following DD statement:

```
//STEPLIB DD DSN=CEE.V1R5M0.SCEELKED,DISP=SHR
```

- Under TSO, use the MVS/TSO Dynamic STEPLIB Facility (5798-DZW) program offering as follows:

```
ALLOCATE FILE(SL) DATASET('CEE.V1R5M0.SCEELKED') SHR  
STEPLIB SET(SL)
```

For further information on the Dynamic STEPLIB Facility, see *MVS/TSO Dynamic STEPLIB Facility Program Description/Operations Manual*.

Don't add CEE.V1R5M0.SCEELKED to your LOGON procedure, and don't ask your system programmer to add it to the system link list in order to avoid using the Dynamic STEPLIB Facility. Making CEE.V1R5M0.SCEELKED available outside the link-edit process could cause unpredictable results because there are names in CEE.V1R5M0.SCEELKED that conflict with the names of certain system components.

The following example illustrates:

- Using the cataloged procedure CEEWL (which is assumed to have been updated to activate the interface validation exit) to link-edit a load module that contains a Fortran subroutine (possibly with conflicting references), a C main routine, but no assembler language routines with conflicting references
- Including an existing load module, MYLMOD, as input to the linkage editor
- Removing the VS FORTRAN library routines by including AFHWRLK during the link-edit⁶
- Replacing certain C library routines during the link-edit

⁶ As discussed in Chapter 5, "Removing VS FORTRAN Library Routines" on page 17, if your input to the linkage editor includes a load module containing VS FORTRAN library routines, you must remove these library routines by including AFHWRLK during the link-edit process.

```

//RELINK EXEC PROC=CEEWL,PGMLIB=USER.APPL.LOAD,GOPGM=MYLMO
//SCEESAMP DD DSN=CEE.V1R5M0.SCEESAMP,DISP=SHR
//VSFLOAD DD DSN=USER.VSF.LOAD,DISP=OLD
//SYSIN DD *
INCLUDE SYSLIB(EDCSTART)
INCLUDE SYSLIB(CEER00TB)
INCLUDE SYSLIB(@@FTOC)
INCLUDE SYSLIB(@@CTOF)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE VSFLOAD(MYLMO)
ENTRY CEESTART
/*

```

The first four INCLUDE statements are needed only when link-editing an existing load module that contains C routines. For further information, see *C/C++ for MVS/ESA Compiler and Run-Time Migration Guide*.

If your main routine is written in Fortran, then specify its name rather than CEESTART in the ENTRY statement.

Conflicting References in Assembler Language Routines

If you have assembler language routines with conflicting references that are intended to be resolved to Fortran library routines, you cannot use the interface validation exit to correctly resolve these references. (This is because the exit cannot determine whether the assembler language routine intends to call the Fortran library routines or the C/C++ library routines.) In this case, as directed by step 4 of the process outlined on page 21, see “Step 4: Manually Resolving the Conflicting References” on page 30.

Step 4: Manually Resolving the Conflicting References

This section describes what you must do to resolve conflicting references in your Fortran or assembler language routines if you can't link-edit your application using the approach described in the preceding section. Depending on the form in which your routines are available as input to the linkage editor, you'll have to take different actions, which are summarized here. These summaries are followed by more detailed explanations.

Modules without any references to C library routines

If your Fortran or assembler language routines don't refer to any C/C++ library routines and if you have your routines available in either of the following forms:

- Individual Fortran or assembler language object modules, or
- Load modules that, in addition to your Fortran or assembler language routines, don't contain any routines with intended references to C/C++ library routines,⁷

use the conflicting references removal tool provided by Language Environment to remove the conflicting references. The tool changes the conflicting references to names that are not in conflict with C/C++ library routine names. Once the conflicting references have been removed in this way, no further special action is required in the link-edit process. For further information, see "Using the Conflicting References Removal Tool."

Load modules that also contain C library references

If your Fortran and assembler language routines with the conflicting references are available only as part of a load module containing intended references to C library routines, determine which Fortran library routines are needed and include them by name when you link-edit the load module. This doesn't remove the conflicting references, but it provides a way to link-edit your load module in spite of them. For information on how to do this, see "Including Fortran Library Routines To Resolve Conflicting References" on page 34.

Using the Conflicting References Removal Tool

The *conflicting references removal tool*, member AFHWNCH in data set CEE.V1R5M0.SCEESAMP, is a set of linkage editor CHANGE statements used during a link-edit step to change conflicting references to their corresponding unambiguous Fortran names. Once you have used this tool to process a module containing conflicting references, the conflicting references are gone, and no further special actions are required to link-edit the module.

You can use the conflicting references removal tool only if your linkage editor input does not contain any intended references to C/C++ library routines. This linkage editor input can be in one of these forms:

- Individual Fortran or assembler language object modules
- Load modules⁷

If your linkage editor input does contain intended references to C/C++ library routines, bypass this section, and resolve the conflicting references using the

⁷ If you have any C/C++ routines in your load module, assume that there are references to C/C++ library routines. In this case, you can't use the conflicting references removal tool.

technique described in “Including Fortran Library Routines To Resolve Conflicting References” on page 34.

The examples in the following sections show two different ways to use the conflicting references removal tool:

- Removing the conflicting references from your routines and saving the resulting module in a library in a form that's intended to be link-edited into other load modules later. Use this approach if you plan to use your routines in several different load modules or if your routines are stable but you plan to link-edit the application frequently.
- In a single step, removing the conflicting references from your routines and creating an executable load module that contains C routines. Use this approach if you don't plan to link-edit the routines with the conflicting references into more than one load module.

Saving the Load Module for Use in Other Load Modules

The following example shows how to use the conflicting references removal tool to remove any conflicting references from a Fortran or assembler language routine (SUB1) and to save the resulting load module as a member of a library so that the routine can subsequently be incorporated into various applications.

```
//CHGNAM EXEC PROC=AFHWN,PGMLIB=USER.LE.LOAD,GOPGM=SUB1
//VSFLOAD DD DSNAME=USER.VSF.LOAD,DISP=SHR
//SYSIN DD *
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB1)
/*
```

Notes on the Example:

1. The linkage editor INCLUDE statements include the following:

AFHWRLK

The Fortran library module removal tool, needed only if the input module (SUB1 in this case) is a load module that was link-edited with VS FORTRAN and that contains Fortran library routines. For further information, see Chapter 5, “Removing VS FORTRAN Library Routines” on page 17.

AFHWNCH

The conflicting references removal tool.

SUB1

The module with the conflicting references. This is the load module from which library routines and conflicting references are removed.

2. The linkage editor stores the resulting load module in member SUB1 in data set USER.LE.LOAD.
3. The use of the AFHWN cataloged procedure, which specifies the NCAL option, causes the linkage editor's automatic library call to be suppressed. Therefore, required library routines aren't included in the resulting load module (SUB1), and you cannot invoke SUB1 until you link-edit it into a fully executable load module.

Removing Conflicting References from More Than One Routine: If you need to remove the conflicting references from more than one routine, say SUB1, SUB2, and SUB3, you can do this in one job step as follows:

```
//CHGNAM EXEC PROC=AFHWN,PGMLIB=USER.LE.LOAD,GOPGM=SUB1
//VSFLOAD DD DSN=USER.VSF.LOAD,DISP=SHR
//SYSIN DD *
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB1)
NAME SUB1(R)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB2)
NAME SUB2(R)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB3)
NAME SUB3(R)
/*
```

You must include AFWRLK (if needed) and AFWNCH before each of your input modules whose conflicting references are to be removed. Also provide a linkage editor NAME statement for each new member of the data set USER.LE.LOAD.

The following TSO LINK commands give the same result:

```
LINK ('CEE.V1R5M0.SCEESAMP(AFWRLK)', +
      'CEE.V1R5M0.SCEESAMP(AFWNCH)', +
      'USER.VSF.LOAD(SUB1)') +
LOAD ('USER.LE.LOAD(SUB1)') +
NOTERM LET NCAL
LINK ('CEE.V1R5M0.SCEESAMP(AFWRLK)', +
      'CEE.V1R5M0.SCEESAMP(AFWNCH)', +
      'USER.VSF.LOAD(SUB2)') +
LOAD ('USER.LE.LOAD(SUB2)') +
NOTERM LET NCAL
LINK ('CEE.V1R5M0.SCEESAMP(AFWRLK)', +
      'CEE.V1R5M0.SCEESAMP(AFWNCH)', +
      'USER.VSF.LOAD(SUB3)') +
LOAD ('USER.LE.LOAD(SUB3)') +
NOTERM LET NCAL
```

Using Modules from Which Conflicting References Have Been Removed:

Once you have removed the conflicting references from your routines, you can link-edit them into your applications without taking any further special actions. In the following example, assume that you have a C main routine whose object module is in member CMAIN in data set USER.APPL.OBJ and that CMAIN calls the Fortran or assembler routines SUB1, SUB2, and SUB3, which the preceding example put in USER.LE.LOAD:

```
//LINKAPP EXEC PROC=CEEWL,PGMLIB=USER.APPL.LOAD,GOPGM=CMAIN
//SYSLIB DD
// DD DSN=USER.LE.LOAD,DISP=SHR
//APPLOBJ DD DSN=USER.APPL.OBJ,DISP=SHR
//SYSIN DD *
INCLUDE APPLOBJ(CMAIN)
/*
```

The use of the CEEWL cataloged procedure to link-edit the C application causes automatic library call processing; therefore, the linkage editor includes the required library routines from CEE.V1R5M0.SCEELKED and the routines SUB1, SUB2, and

SUB3 from USER.LE.LOAD. The resulting load module is given the member name CMAIN in data set USER.APPL.LOAD.

The following example link-edits CMAIN under TSO:

```
LINK ('USER.APPL.OBJ(CMAIN)') +
LIB ('CEE.V1R5M0.SCEELKED', 'USER.LE.LOAD') +
LOAD ('USER.APPL.LOAD(CMAIN)') +
NOTERM
```

This completes the resolution of the conflicting references using the conflicting references removal tool. The following section describes another way to use the tool.

Creating an Executable Load Module

This section shows how to use the conflicting references removal tool to remove the conflicting references and to create an executable load module, all in one step. The technique shown here is simpler than the technique described in the preceding section because there's no need for an intermediate library. However, if you want to use the same Fortran or assembler language routines in other load modules, you have to remove the conflicting references each time you link-edit them into another load module.

The following example creates an executable load module from the same routines (CMAIN, SUB1, SUB2, and SUB3) as the previous example; however, this example bypasses putting SUB1, SUB2, and SUB3 in USER.LE.LOAD:

```
//CHGLINK EXEC PROC=CEEWL,PGMLIB=USER.APPL.LOAD,GOPGM=CMAIN
//SCEESAMP DD DSN=CEE.V1R5M0.SCEESAMP,DISP=SHR
//APPLOBJ DD DSN=USER.APPL.OBJ,DISP=SHR
//VSFLOAD DD DSNAME=USER.VSF.LOAD,DISP=SHR
//SYSIN DD *
INCLUDE APPLOBJ(CMAIN)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB1)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB2)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE SCEESAMP(AFWNCH)
INCLUDE VSFLOAD(SUB3)
/*
```

Notes on the Example:

1. The first INCLUDE statement includes the C main routine, CMAIN.
2. For each of the three Fortran or assembler language routines with conflicting references, there are linkage editor INCLUDE statements to:
 - a. Remove VS FORTRAN library routines using AFWRLK
 - b. Remove the conflicting references using AFWNCH
 - c. Include the Fortran or assembler language routine (SUB1, SUB2, or SUB3) with the conflicting references

The following TSO LINK command gives the same result:

```
LINK ('USER.APPL.OBJ(CMAIN)' +
      'CEE.V1R5M0.SCEESAMP(AFWRLK)', +
      'CEE.V1R5M0.SCEESAMP(AFWNCH)', +
      'USER.VSF.LOAD(SUB1)', +
      'CEE.V1R5M0.SCEESAMP(AFWRLK)', +
      'CEE.V1R5M0.SCEESAMP(AFWNCH)', +
      'USER.VSF.LOAD(SUB2)', +
      'CEE.V1R5M0.SCEESAMP(AFWRLK)', +
      'CEE.V1R5M0.SCEESAMP(AFWNCH)', +
      'USER.VSF.LOAD(SUB3)') +
LIB ('CEE.V1R5M0.SCEELKED') +
LOAD ('USER.APPL.LOAD(CMAIN)') +
NOTERM
```

This completes the resolution of the conflicting references using the conflicting references removal tool.

Including Fortran Library Routines To Resolve Conflicting References

This section shows how to deal with the conflicting references when both of the following conditions are satisfied:

- Your input to the linkage editor is a load module containing both of the following:
 - One or more C routines
 - Fortran or assembler language routines with conflicting references
- You don't have the Fortran or assembler language routines available as source modules or as individual object modules.⁸

Follow these steps to resolve the conflicting references to the proper Fortran library routines:

1. Identify exactly which of the 20 names in Table 4 on page 22 are referenced by the Fortran or assembler language routines. To find this information, look at cross-reference information produced when the load module was link-edited. (If you don't have the output from the linkage editor, use the AMBLIST service aid as shown in "Load Modules" on page 23.)
2. From the data set CEE.V1R5M0.SAFHFORT, specifically include into your load module each library routine identified in step 1. This resolves the conflicting references to the correct Fortran library routines, but the conflicting references remain in the resulting load module.

The following example shows how to link-edit a load module MOD1 from the data set USER.VSF.LOAD. Assume that the main routine is written in C and that you've found three conflicting references: SIN, LOG, and CLOCK.

⁸ If you have the Fortran or assembler language source programs and can recompile them, following step 2 of the process outlined on page 21 leads you to an easier technique for resolving conflicting references than is described in this section. See "Step 2: Recompiling Programs to Eliminate Conflicting References" on page 24 instead.

```

//FORTC EXEC PROC=CEEWL,PGMLIB=USER.APPL.LOAD,GONAME=MOD1
//SCEESAMP DD DSNAME=CEE.V1R5M0.SCEESAMP,DISP=SHR
//SAHFHFORT DD DSNAME=CEE.V1R5M0.SAHFHFORT,DISP=SHR
//VSFLOAD DD DSNAME=USER.VSF.LOAD,DISP=SHR
//SYSIN DD *
INCLUDE SAHFHFORT(SIN)
INCLUDE SAHFHFORT(LOG)
INCLUDE SAHFHFORT(CLOCK)
INCLUDE SYSLIB(EDCSTART)
INCLUDE SYSLIB(CEER00TB)
INCLUDE SYSLIB(@@FTOC)
INCLUDE SYSLIB(@@CTOF)
INCLUDE SCEESAMP(AFWRLK)
INCLUDE VSFLOAD(MOD1)
ENTRY CEESTART
NAME MOD1(R)
/*

```

Notes on the Example:

1. The linkage editor INCLUDE statements include the following:

SIN, LOG, CLOCK

The Fortran library versions of the routines for which there are conflicting references. These included routines replace VS FORTRAN routines with the same names.

EDCSTART, CEER00TB, @@FTOC, @@CTOF

C library routines that must replace C library routines with the same names in order to link-edit the application with Language Environment. These library routines are included from data set CEE.V1R5M0.SCEELKED because the SYSLIB DD statement in the CEEWL cataloged procedure refers to it. For further information on link-editing existing load modules that contain C routines, see *C/C++ for MVS/ESA Compiler and Run-Time Migration Guide*.

AFHWRLK

The Fortran library module removal tool, needed because the input load module (MOD1) contains Fortran library routines.

MOD1

The load module with the conflicting references.

2. All required library routines not specifically included with INCLUDE statements are included from CEE.V1R5M0.SCEELKED through the automatic library call process.
3. The references to SIN, LOG, and CLOCK are not removed; they are just resolved to the proper Fortran library routines.

Chapter 8. Migrating VS FORTRAN Run-Time Options

Run-time options are parameters that control certain run-time behavior of Language Environment and of your routines. Many of the run-time options apply to all languages in the application. An example is XUFLOW, which controls whether an exponent-underflow exception should be allowed to occur. Others apply only to routines written in a certain language, such as Fortran. An example is PRTUNIT, which indicates the unit number of the print unit, that is, the unit to which output from a Fortran PRINT statement is directed.

With Language Environment you can use most VS FORTRAN Version 2 run-time options but there are the following differences, which this chapter discusses:

- The format in which you code the string of run-time options when you invoke your application is slightly different.
- A few of the run-time options from VS FORTRAN Version 2 have no meaning with Language Environment, and others have replacement options that provide similar functions.
- The process that you use to create a set of default run-time options that you can link-edit into your load module is different.

Coding the Option String

When you invoke your application, you can provide a string that consists of both run-time options and program arguments. (The *program arguments* are the data that can be retrieved in your program as a string of characters using either the Fortran-specific ARGSTR callable service or the CEE3PRM callable service.)

Where to Code the Option String

If you want to supply either run-time options or program arguments when you invoke your application, code the option string, *opt_str*, in one of the following ways, depending on how you invoke the application:

- As the value of the PARM parameter on the EXEC statement in your JCL:

```
//step EXEC PGM=name,PARM='opt_str'
```
- As the parameter string on the TSO CALL command:

```
CALL dsname(name) 'opt_str'
```
- As the parameter string on the TSO LOADGO command:

```
LOADGO name 'opt_str'
```
- In a standard parameter list passed from an assembler language routine to a main program when these two routines are link-edited together:

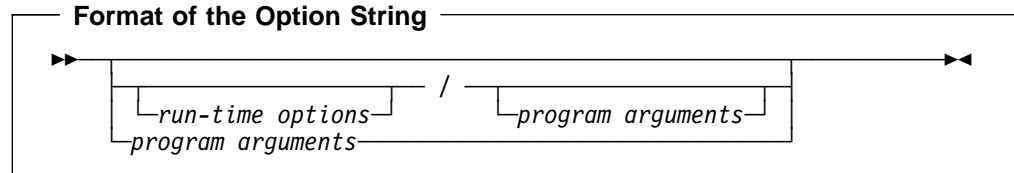
```
LA 1,PARMLIST
L 15,=V(assm_name)
BALR 14,15

PARMLIST DC A(PARMDATA+X'80000000')
PARMDATA DC Y(L'OPTIONS)
OPTIONS DC C'opt_str'
```

For a Fortran main program, *assm_name* is the name in the PROGRAM statement. (If there was no PROGRAM statement in the Fortran main program, *assm_name* is MAIN# for programs compiled with the VS FORTRAN Version 2 Release 5 or Release 6 compiler; *assm_name* is MAIN for programs compiled with previous Fortran compilers.)

Format of Option String

The option string has the following format, which is slightly different than its counterpart in VS FORTRAN:



Here are some examples of providing either run-time options only, program arguments only, or both:

- Both run-time options and program arguments:
`N00CSTATUS,MSGFILE(FT06F001)/ANNUAL-REPORT`

- Run-time options only:
`N00CSTATUS,MSGFILE(FT06F001)/`

Note that the slash (/) isn't required following the run-time options with VS FORTRAN Version 2, but that it is required with Language Environment.

- Program arguments only:
`/ANNUAL-REPORT`

or

`ANNUAL-REPORT`

If the main program is written in COBOL, there are some differences in the format of the option string (even when Fortran-specific options are present).

Comparing the Individual Run-Time Options

Most of the run-time options that you can use in VS FORTRAN Version 2 are available with Language Environment. However, there are a few that aren't supported in Language Environment, and a few whose functions are provided by different options in Language Environment. Table 5 on page 38 lists the options from VS FORTRAN Version 2 and shows their equivalents, where applicable, in Language Environment. As the tables indicates, Language Environment automatically maps some VS FORTRAN run-time options into the equivalent Language Environment run-time options. In other cases, there is no automatic mapping, and you must specify the replacement run-time option. If you specify a VS FORTRAN run-time option that has no Language Environment equivalent and isn't automatically mapped to another option, Language Environment prints an information message at run time.

Table 5 (Page 1 of 2). Fortran and Language Environment Options

| Fortran Option | Language Environment Equivalent | Notes |
|-----------------------|---------------------------------|---|
| ABSDUMP NOABSDUMP | TERMTHDACT | TERMTHDACT(DUMP) replaces ABSDUMP to produce a Language Environment dump at termination, but there is no automatic mapping. TERMTHDACT with suboptions TRACE, QUIET, or MSG replaces NOABSDUMP to avoid getting a Language Environment dump at termination. |
| AUTOTASK NOAUTOTASK | AUTOTASK NOAUTOTASK | AUTOTASK NOAUTOTASK provides behavior compatible with VS FORTRAN Version 2. |
| CNVIOERR NOCNVIOERR | Not applicable | There is no Language Environment equivalent for CNVIOERR NOCNVIOERR. Fortran semantics are as though CNVIOERR were in effect. |
| DEBUG NODEBUG | Not applicable | There is no debugger support for Fortran. |
| DEBUNIT | Not applicable | There is no Language Environment equivalent for DEBUNIT. |
| ECPACK NOECPACK | Not applicable | There is no Language Environment equivalent for ECPACK NOECPACK. You cannot run programs with Language Environment that use access registers or that were compiled with the EC or EMODE compiler options. |
| ERRUNIT | ERRUNIT | ERRUNIT provides behavior compatible with VS FORTRAN Version 2. |
| FAIL | ABTERMENC | ABTERMENC replaces FAIL, but there is no automatic mapping. ABTERMENC controls whether a condition of severity 2 or greater is terminated with a return code or an abend. ABTERMENC(RETCODE) is similar to FAIL(RC), and ABTERMENC(ABEND) is similar to FAIL(ABEND). |
| FILEHIST NOFILEHIST | FILEHIST NOFILEHIST | FILEHIST NOFILEHIST provides behavior compatible with VS FORTRAN Version 2. |
| INQPCOPN NOINQPCOPN | INQPCOPN NOINQPCOPN | INQPCOPN NOINQPCOPN provides behavior compatible with VS FORTRAN Version 2. |
| IOINIT NOIOINIT | Not applicable | There is no Language Environment equivalent for IOINIT NOIOINIT. The message file is opened either when the first record is written to it or when an OPEN statement refers to error message unit. If no allocation for the ddname has been made for the message file, it is dynamically allocated to the terminal (under TSO) or to SYSOUT=* (under MVS batch). |
| OCSTATUS NOOCSTATUS | OCSTATUS NOOCSTATUS | OCSTATUS NOOCSTATUS provides behavior compatible with VS FORTRAN Version 2. |
| PARALLEL NOPARALLEL | Not applicable | There is no Language Environment equivalent for PARALLEL NOPARALLEL. Parallel programs cannot be run with Language Environment. |
| PRTUNIT | PRTUNIT | PRTUNIT provides behavior compatible with VS FORTRAN Version 2. |

Table 5 (Page 2 of 2). Fortran and Language Environment Options

| Fortran Option | Language Environment Equivalent | Notes |
|---------------------------------|-------------------------------------|---|
| PTRACE NOPTRACE | Not applicable | There is no Language Environment equivalent for PTRACE NOPTRACE. Parallel programs cannot be run with Language Environment. |
| PUNUNIT | PUNUNIT | PUNUNIT provides behavior compatible with VS FORTRAN Version 2. |
| RDRUNIT | RDRUNIT | RDRUNIT provides behavior compatible with VS FORTRAN Version 2. |
| RECPAD NORECPAD RECPAD(VAR) | RECPAD(OFF NONE VAR ALL ON) | NORECPAD automatically maps to RECPAD(OFF). RECPAD(VAR) provides behavior compatible with VS FORTRAN Version 2. RECPAD must be changed to RECPAD(ON). |
| SPIE NOSPIE, STAE NOSTAE | TRAP(ON OFF) | If either SPIE or STAE is specified in input, TRAP is set to TRAP(ON). If both NOSPIE and NOSTAE are specified, TRAP is set to TRAP(OFF). TRAP(ON) must be in effect for many applications to run successfully. |
| XUFLOW NOXUFLOW | XUFLOW(ON AUTO) XUFLOW(OFF) | There is no automatic mapping of XUFLOW to the Language Environment XUFLOW. NOXUFLOW maps to the Language Environment XUFLOW(OFF), which provides compatible behavior. |

Providing Default Run-Time Options for Your Application

With VS FORTRAN Version 2, you can create a set of run-time options that you can use in one or more applications without specifying the options when you invoke the application. You do this by assembling a VSF2PARAM macro instruction with the options you want, which produces an object module with the name AFBVLPRM.⁹ You then include this object module into the load module that contains your main program.

You can't use the VS FORTRAN Version 2 macro or an existing copy of AFBVLPRM with Language Environment, but there is a comparable way of providing the run-time options. Follow these steps:

1. Code a CEEXOPT macro instruction with the run-time options that you want to use.
2. Assemble the file containing the CEEXOPT macro instruction to produce a CEEUOPT object module.
3. Link-edit the CEEUOPT object module into the load module that contains your main program.

The specified options take effect when you invoke the application. You can use the same copy of the CEEUOPT object module for different applications, or you can create copies with different sets of run-time options for different applications. For further information, see the chapter on using run-time options in *Language Environment for MVS & VM Programming Guide*.

⁹ Those who install VS FORTRAN Version 2 can assemble a VSF2PARAM macro instruction to provide run-time options that apply to your whole site. (In this case, the assembly creates the module AFBVGPRM.) With Language Environment, there is a different process for providing site-wide run-time options, as described in *Language Environment for MVS & VM Installation and Customization on MVS*.

Chapter 9. Interpreting Return Codes and Completion Codes

When your application completes, either successfully or unsuccessfully, it returns an indication of how the application terminated to the invoker of the application. An application that terminates normally¹⁰ provides a return code. One that terminates abnormally provides a system or user completion code.

This chapter provides an overview of how return codes and completion codes are set and shows how certain Language Environment return codes and completion codes differ from the VS FORTRAN ones.

Specifying How Unhandled Conditions Should Be Reported

With VS FORTRAN, the FAIL run-time option lets you control whether certain kinds of failures result in normal termination with a return code of 16 or an abnormal termination (abend) with a user completion code of 240. The default is for abnormal terminations to be translated into a user completion code 240 and for all other failures to cause normal termination with return code 16.

Language Environment has the ABTERMENC run-time option, which has a similar function. This option controls what happens when termination occurs because of an unhandled condition with a severity of 2 or greater:

ABTERMENC(ABEND)

Causes abnormal termination to occur. The completion code depends on the type of the original failure, such as program interruption, software-detected error, or abend.

ABTERMENC(RETCODE)

Causes normal termination with a nonzero return code. For the value of the return code, see “Interpreting Return Codes for Unhandled Conditions” on page 44.

As supplied by IBM, the default value of the ABTERMENC run-time option is RETCODE. Therefore, with Language Environment, an unhandled condition of severity 2 or greater causes a normal termination with a nonzero return code; with VS FORTRAN, the same error might cause an abnormal termination. For example, with Language Environment, a program interruption due to a protection exception causes normal termination with a return code of 3000. With VS FORTRAN, the same error causes an abnormal termination with user completion code 240. If you have JCL or CLISTS that interpret return codes or abends, then revise your code to detect failures as defined by the ABTERMENC run-time option.

Language Environment lets you write an assembler user exit called CEEBXITA, which can override the actions of the ABTERMENC run-time option. For further information, see the section on the termination behavior for unhandled conditions in *Language Environment for MVS & VM Programming Guide*.

¹⁰ To say that an application terminated normally doesn't necessarily mean that it completed successfully from your point of view. *Normal termination* is used in the MVS sense to mean that no abend (or *abnormal termination*) occurred.

Return Codes

An application terminates normally for one of these reasons:

- Some language construct, such as a STOP statement, requests termination and possibly provides a return code to be used.
- The ABTERMENC(RETCODE) run-time option is in effect and there is an unhandled condition of severity 2 or greater.

Every application that terminates normally provides a return code that indicates the success or failure of the application. By convention, a return code of 0 usually indicates successful completion, but when your program provides the return code, the values can have whatever meaning you choose.

This section discusses the following topics:

- Detecting the return code from the completed application
- Specifying the return code in your Fortran routine
- Interpreting return codes for unhandled conditions

Detecting the Return Code from the Completed Application

Your application's return code is presented to you in different ways, depending on how you invoked your application. Here are some common ways:

- When you use the EXEC statement in your JCL to specify the name of the load module containing your main program, the return code is *nnnn* in the following message in the job log:

```
COND CODE nnnn
```

You can test the return code by using the COND parameter on the EXEC statement for subsequent job steps.

- When you use a CALL command in a TSO CLIST to specify the name of the load module containing your main program, the return code is available to your CLIST in the variable &LASTCC.
- When you invoke your main program using the REXX language, the return code is available to your REXX exec in the REXX special variable RC.
- When you call your main program directly from an assembler language program, the return code is provided to the calling program in general register 15.

The following sections explain what causes different return code values to be set.

Specifying the Return Code in Your Fortran Routine

With Language Environment, the following Fortran statements terminate your application with the return code shown. These return codes are the same as in VS FORTRAN.

Table 6. Return Codes from Fortran Statements That Terminate an Application

| Fortran Statement | Return Code |
|--|---|
| STOP | 0 |
| END in a main program | 0 |
| STOP <i>n</i> | <i>n</i> |
| STOP 'message' (message is a string of up to 72 characters.) | 0 |
| CALL SYSRCX (<i>n</i>) | <i>n</i> if <i>n</i> is between 0 and 4095, inclusive. Otherwise, undefined. |
| CALL EXIT or CALL SYSRCX | The most recent user return code that was set either by a call to SYSRCS or CEE3SRC or by some other language's service |
| CALL DUMP or CALL CDUMP | 0 |

At some point while your application is running, you can set a value that might be used later as the return code when the application finally terminates. The value that you set in this way is called the *user return code* to contrast it with the final return code when the application terminates. The statements in the following table set a user return code without terminating immediately:

Table 7. Fortran Statements That Set a User Return Code

| Statement | User Return Code |
|---|---|
| CALL SYSRCS (<i>n</i>) or CALL CEE3SRC (<i>n</i> , <i>fc</i>) | <i>n</i> if <i>n</i> is between 0 and 4095, inclusive. Otherwise, undefined. |

You can't call the CEE3SRC callable service directly from a Fortran routine, but you can call it using the AFHCEEFF or AFHCEEN callable service, or you can call it from an assembler language routine.

Other languages also provide services for setting the user return code. An example is the PLIRETC built-in subroutine in PL/I. Regardless of the language that was used to set the user return code, the most recently set value is used when the application terminates.

Interpreting Return Codes for Unhandled Conditions

Whenever the ABTERMENC(RETCODE) run-time option is in effect and the application terminates due to an unhandled condition of severity 2 or greater, a normal termination occurs, and the return code is set according to the following formula:

$$1000 \times \textit{severity} + \textit{user_return_code}$$

severity

The severity (2, 3, or 4) of the unhandled condition. The value $1000 \times \textit{severity}$ is called the *return code modifier*.

user_return_code

The user return code that was set with the most recent call either to the SYSRCS or CEE3SRC callable service or to another language's service, such as PL/I's PLIRETC built-in subroutine.

For example, the following program sets the user return code to 201 and is then terminated due to the unhandled condition FOR1916, the severity 3 condition that results from the excessively large unit number in the OPEN statement:

```
CALL SYSRCS (201)
OPEN (10000)
END
```

The return code for the application is 3201. With VS FORTRAN, this same program terminates normally with return code 16 if the number of allowed occurrences of the error is 1.

Completion (Abend) Codes

An application abnormally terminates (abends) for one of these reasons:

- One of your routines requests the abnormal termination and provides the user completion code, such as with a call to the SYSABN callable service.
- The ABTERMENC(ABEND) run-time option is in effect and there is an unhandled condition of severity 2 or greater.
- The CEEBXITA assembler user exit, which is described in *Language Environment for MVS & VM Programming Guide*, requests that an unhandled condition of severity 2 or greater cause an abnormal termination rather than a normal termination with a return code.
- Language Environment detects some unusual situation for which continued execution is impossible.

An application that terminates abnormally provides a completion code, often called an abend code, that indicates the cause of the failure. Many completion codes are accompanied by a reason code that provides more detailed information on the error. There are two types of completion codes: system completion codes and user completion codes. The system completion codes are produced by operating system components and are printed as three hexadecimal digits. For the meaning of the system completion codes and reason codes, see one of the following:

- *MVS/ESA System Codes, MVS/ESA System Product: JES2 Version 4, JES3 Version 4*

- *MVS/ESA System Codes, MVS/ESA System Product: JES2 Version 5, JES3 Version 5*

User completion codes are dependent on the application that detected the error and are usually printed as four decimal digits. Language Environment produces user completion codes above 4000 to report problems for which continued processing is impossible. For the meaning of these codes, see the chapter on Language Environment abend codes in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*. For user completion codes produced by your application, see whatever information describes the code. Be aware that your application could produce a user completion code that overlaps with Language Environment's 40nn codes; if it does so, there could be some confusion about the meaning of the code.

Part 3. Changing Source Programs

Some VS FORTRAN facilities aren't available in Language Environment, and their absence will affect the migration of your Fortran routines that depend on them. The most significant of these features is the extended error handling facility, which provides corrective actions and other services for Fortran applications. The following chapters discuss how to change your Fortran source programs to use Language Environment features when you want behavior similar to that of VS FORTRAN. Chapter 10 discusses how to deal with run-time errors using the condition handling services that are part of Language Environment, and Chapter 11 discusses several other VS FORTRAN facilities and how to use Language Environment services to get similar results.

| | |
|---|----|
| Chapter 10. Handling Run-Time Errors | 48 |
| Overview of the Language Environment Condition Handling Model | 48 |
| Conditions and the Enablement Step | 48 |
| Stack Frames, Cursors, and the Condition Step | 49 |
| The Condition Token | 50 |
| Interaction with a User-Written Condition Handler | 52 |
| Unhandled Conditions | 56 |
| Overview of the VS FORTRAN Extended Error Handling Facility | 57 |
| Error Option Table | 58 |
| Extended Error Handling Facility Subroutines | 59 |
| Automatic Error Correction | 59 |
| Fortran-Specific Services for Error Handling | 59 |
| Callable Services to Retrieve and Update Qualifying Data | 60 |
| Fortran Functions to Retrieve Qualifying Data | 61 |
| Fortran Functions That Provide Information about Qualifying Data | 62 |
| Fortran Services for Calling Language Environment Callable Services | 63 |
| Handling Run-Time Errors from Your Fortran Routines | 63 |
| Sample Condition Handler for a Square-Root Exception | 64 |
| Sample Condition Handlers for a Fixed-Point Overflow Exception | 66 |
| Sample Condition Handler for an Exponent-Overflow Exception | 68 |
| Sample Condition Handlers for an Invalid Character in a Numeric Field | 70 |
| Regaining Control for Conditions Not Handled by a Subprogram | 73 |
| | |
| Chapter 11. Making Other Source Program Changes | 76 |
| Texts of Run-Time Error Messages | 76 |
| Values Returned through the IOSTAT Specifier | 77 |
| Permissible Input/Output to the Error Message Unit | 77 |
| Data Set Attributes for the Message File | 79 |
| Fix-Up for Misaligned Vector Instruction Operands | 80 |
| Fixed-Point Overflow | 80 |
| DVCHK and OVERFL Subroutines | 81 |
| Assembler Language Routines That Find Program Arguments | 82 |
| Run-Time Initialization from Assembler Language | 82 |
| Assembler Language Routine Acting as a Main Program | 83 |
| Assembler Language Routine for an Active Environment | 84 |
| Assembler Language Routine Operating outside the Environment | 85 |
| COBOL Routine Invoking an Assembler Language Routine for Initialization | 86 |

Chapter 10. Handling Run-Time Errors

You can use the Language Environment condition handling model to replace much of the error handling that is part of VS FORTRAN. If your Fortran programs depend on the extended error handling facility of VS FORTRAN, either for fix-up actions or for continued execution when an error occurs, change these programs to use the services that are part of the Language Environment condition handling model. This chapter discusses the following topics:

- Overview of the Language Environment condition handling model
- Overview of the VS FORTRAN extended error handling facility
- Fortran-specific services for error handling
- Handling run-time errors from your Fortran routines
- Regaining control for conditions not handled by a subprogram

Overview of the Language Environment Condition Handling Model

This section introduces you the Language Environment condition handling model. It discusses the terminology involved, the operation of the Language Environment condition manager, and the interaction with your routines. Emphasis is placed on the interaction with Fortran routines.

There is detailed information on condition handling in *Language Environment for MVS & VM Programming Guide*.

Conditions and the Enablement Step

A *condition* is some exceptional situation detected by the one of the following and processed through the Language Environment condition handling model:

Hardware

These are the program interruptions that are detected by the central processing unit. Examples are the exponent-overflow and addressing exceptions.

Operating system

These are software errors that are reported as abends. An example is the failure of an OPEN macro instruction which could fail with a system completion (abend) code of 813-04 to indicate that a requested data set isn't on the tape volume.

Other software

These are errors detected either by Language Environment (including the language-specific libraries, such as the Fortran library) or by your own routines.

Not all exceptional situations necessarily become conditions.

There are several participants in the Language Environment condition handling model. The first is the *Language Environment condition manager* (or just *condition manager*), which controls the actions defined by the model. Other participants include language-specific condition handlers and user-written condition handlers.

A condition is *signaled* when it is brought to the attention of the condition manager through the CEESGL callable service. The condition manager's first action is to determine whether the condition should continue to be processed as a condition.

This is called the *enablement step*. A participant at this step is the *language-specific enablement routine*, a language-specific library routine that is invoked immediately after the condition is signaled. A specific enablement routine, such as the one for Fortran, is invoked only for a condition resulting from the execution of one of your routines written in that specific language. A condition is said to be *enabled* if the language-specific enablement routine indicates that the condition should be processed further as a condition. The decision whether to enable a specific condition is based on the semantics of the language involved.

If an enablement routine does not to enable a condition, then no further condition handling occurs, and execution continues at the next instruction or as defined by the language semantics.

The Fortran-specific enablement routine enables all conditions that are presented to it except for cases in which the language standard requires that control be returned to the program. An example of a condition that is not enabled is a system completion (abend) code 813-04 that occurs during the execution of a Fortran OPEN statement with an ERR specifier. This condition is not enabled, and control passes to the label indicated by the ERR specifier.

Stack Frames, Cursors, and the Condition Step

The Language Environment condition handling model views the invoked routines of an application as residing in a last-in-first-out (LIFO) stack. Two pointers, or *cursors*, point at various levels of the stack. Each entry in the stack, called a *stack frame*, represents a routine (either a user or library routine) in the call chain. The most recent stack frame represents the most recently invoked routine. The condition manager processes the stack frames of the stack in LIFO order. The language-specific enablement routine involved is the one associated with the most recent stack frame.

The *handle cursor* points to the stack frame whose condition handling actions are to be performed. The *resume cursor* points to the stack frame and the instruction where execution will resume if a condition handler decides that it is appropriate to resume execution. Language Environment provides the following callable services to let a program move the resume cursor:

| Name | Purpose |
|-------------|---|
| CEEMRCE | Moves the resume cursor to the point following a call to the CEE3SRP callable service. (A routine must use CEE3SRP to establish the resume point before the condition occurs.) You can't call CEE3SRP from a Fortran routine. |
| CEEMRCR | Moves the resume cursor to the return point in an earlier stack frame. There's a Fortran example of the use of the CEEMRCR callable service in "Regaining Control for Conditions Not Handled by a Subprogram" on page 73. |

For further details on these callable services, see *Language Environment for MVS & VM Programming Reference*.

During the condition step, the condition handling model defines for each stack frame a logical queue of condition handlers which might in turn get a chance to handle a condition. The sequence of condition handlers in the queue for each stack frame is as follows:

1. Registered user-written condition handlers, if any, in LIFO order
2. The language-specific condition handler

A *user-written condition handler* is a routine that you can write to deal with a condition presented to it. You register your condition handler on behalf of a given stack frame using the CEEHDLR callable service, and you unregister it using the CEEHDLU callable service. It is automatically unregistered when the routine that called CEEHDLR returns to its caller. You can register more than one user-written condition handler for a given stack frame.

The Condition Token

When a condition is signaled, it is represented by a condition token. A *condition token* is an instance of a 12-byte datum which represents a single occurrence of some condition. (The detailed format of a condition token can be found in the chapter on using condition tokens in *Language Environment for MVS & VM Programming Guide*.) The condition token contains identifying information, such as the following, that identifies a particular condition:

- The *facility ID*, a three-character identifier of a product or component
- A unique message number that identifies a condition and its corresponding message

Another component of the condition token is the condition's severity, which is a value from 0 through 4. The severities have the following meanings:

| Severity | Meaning |
|----------|----------------|
| 0 | Information |
| 1 | Warning |
| 2 | Error |
| 3 | Severe error |
| 4 | Critical error |

Severity 4 usually refers to situations where continued execution of the application is not likely to be successful, such as when some internal control block has been destroyed. The other severities usually refer to impact on the particular service being performed, not necessarily to the impact on the overall application. For example, a mathematical routine might signal a severity 2 condition, but this could be a catastrophic error from the point of view of your application if your application can't continue without the computed value. On the other hand, the library routine that processes your OPEN statement might signal a severity 3 condition while opening your file, but your program could ignore this error if it's necessary to do other processing even without the file.

The condition token also refers to the following *instance-specific information* about the condition:

- The variable data to be substituted in a message
- The qualifying data associated with the condition. The *qualifying data* are unique pieces of information about a particular occurrence of a condition. One example of a qualifying datum is the input argument to the service that

detected the condition. For example, if -4.0 is passed as an argument to a real square root routine, one qualifying datum associated with the condition token for this error has the value -4.0 .

A user-written condition handler uses qualifying data to determine what corrective actions to take and to provide the data needed for the corrective actions.

Condition Tokens as Feedback Codes

The condition token is also used in another context, which is outside the condition manager's control. Most of the Language Environment callable services have an optional argument, called a *feedback code*, which is a condition token that indicates the success or failure of the execution of the service. By convention, if the feedback code is present in the call to a callable service and if some error is detected by the service, the error is reported through the feedback code when the service returns to its caller. In this case, the calling routine must interpret the feedback code (a condition token) and take whatever action it chooses. However, if the feedback code is omitted from the call and if some error is detected, the service does not return to the caller; instead, it signals the condition that represents the error.

Symbolic Feedback Codes

For any condition, the first eight bytes of the 12-byte condition token are always the same and can be used to uniquely identify the condition. The last four bytes have information that is unique to a particular occurrence (or instance) of the condition. To assist you in writing programs that must identify specific conditions or that create condition tokens, each condition is given a name, called the *symbolic feedback code*. The names CEE0CF and FOR1003 are two examples. The value of the symbolic feedback code is eight bytes long and is in the first eight bytes of the condition token.

The symbolic feedback codes are shown along with the messages in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*. For example, CEE0CF is shown as the symbolic feedback code associated with the message CEE399W, and FOR1003 is shown as the symbolic feedback code associated with message FOR1003S. For every Fortran-specific condition, the symbolic feedback code consists of the characters FOR followed by the four-digit message number.

Language Environment includes several symbolic feedback code files, each of which contains a single language's data declarations of a set of symbolic feedback codes. These files are in the data set CEE.V1R5M0.SCEESAMP. For example, there is a file with Fortran language declarations of the Fortran conditions, another with the Fortran language declarations of the conditions for the common component of Language Environment, and so on. The names of these files are based on:

- Which component's symbolic feedback codes are declared in the file, and
- The language in which the declarations are coded

The names of the symbolic feedback code files have the following format:

`xxxyyyCT`

`xxx` The facility ID of the component whose symbolic feedback codes are declared in the file. `xxx` can be one of the following:

| <u>xxx</u> | <u>Component</u> |
|------------|--|
| CEE | Common component of Language Environment |
| EDC | C and C++ |
| FOR | Fortran |
| IBM | PL/I |
| IGZ | COBOL |

`yyy` The facility ID of the language in which the declarations are coded. `yyy` can be one of the following:

| <u>yyy</u> | <u>Language</u> |
|------------|-----------------|
| BAL | Assembler |
| EDC | C and C++ |
| FOR | Fortran |
| IBM | PL/I |
| IGZ | COBOL |

The two symbolic feedback code files that you will use most often in your Fortran programs are FORFORCT and CEEFORCT; these are the Fortran declarations for the Fortran-specific conditions and for the common component conditions, respectively. To make these files available to the Fortran compiler, provide the following INCLUDE lines among the data declarations in your Fortran source program:

```
INCLUDE (FORFORCT)
INCLUDE (CEEFORCT)
```

and provide the following DD statement in your JCL for the compilation step:

```
//SYSLIB DD DSN=CEE.V1R5M0.SCEESAMP,DISP=SHR
```

Interaction with a User-Written Condition Handler

A user-written condition handler receives control from the condition manager as a subroutine with four arguments, the first of which is the condition token that, along with the qualifying data, represents the condition being processed. Based on this information, the condition handler can decide how to deal with the condition. The following sections discuss the different actions that a condition handler can request the condition manager to take and the detailed interface between a user-written condition handler and the condition manager.

Actions That a User-Written Condition Handler Can Request

Through the third argument passed to it, the condition handler requests one of the following actions that it wants the condition manager to take to continue processing the condition:

resume

The condition handler considers the condition to have been handled, and program execution should continue at the instruction to which the resume

cursor points. If the resume cursor has not been moved by a condition handler using the CEEMRCE or CEEMRCR callable service, the resume cursor points to the next instruction following the call to CEESGL.

The condition handler can move the resume cursor for any condition. For some conditions, this is the only way to request the *resume* action. Other conditions do allow the *resume* action when the resume cursor hasn't been moved, but there usually isn't any corrective action other than ignoring the failing statement or service and continuing execution. For any of the Fortran conditions, you can determine whether *resume* is allowed by referring to the permissible resume actions listed with the message in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.

percolate

The condition handler declines to handle the condition, and the next condition handler defined by condition handling model should be given a chance to handle it.

The *percolate* action has two variations. The normal case is to invoke the next condition handler, if any, on the stack frame to which the handle cursor points. The alternative is to bypass any remaining condition handlers on that stack frame, to move the handle cursor to the stack frame just before the one to which it currently points, and to present the condition to first condition handler, if any, for the stack frame to which the handle cursor was moved.

Important: Always request the *percolate* action for any condition that your condition handler either does not understand or chooses not to process.

promote

The condition handler declines to handle the current condition, but it requests that this condition be converted to a different condition, which it specifies through one of the arguments.

The *promote* action has three variations. After the condition has been converted, the normal case is to invoke the next condition handler, if any, on the stack frame to which the handle cursor points. One alternative is to bypass any remaining condition handlers on that stack frame, to move the handle cursor to the stack frame just before the one to which it currently points, and to present the condition to first condition handler, if any, for the stack frame to which the handle cursor was moved. Another alternative is to keep the handle cursor at the same stack frame but to invoke the first condition handler on that same stack frame. In this case, the current condition handler could be invoked again for the new condition.

fix-up and resume

To handle the condition, the condition handler requests that one of several possible corrective actions should be taken.

In the fourth argument passed to it, the condition handler provides a condition token that corresponds to the specific corrective action that it wants the condition manager to take. For the Fortran conditions, there are three possible *fix-up and resume* actions, not all of which apply to any one condition:

resume with new input value

The failing service should be retried with a new input value, which the condition handler provides through a qualifying datum.

The condition token for requesting this action has the symbolic feedback code CEE0CE; its message number is 398.

resume with new output value

Execution should continue but with a specified result instead of what the failing instruction or service would have provided. The condition handler provides this result through a qualifying datum.

The condition token for requesting this action has the symbolic feedback code CEE0CF; its message number is 399.

resume with Fortran-specific correction

Execution should continue but with a Fortran-specific action that is unique to a particular condition. This action applies to a few Fortran conditions, such as FOR1002, that are signaled during the processing of an I/O statement.

The condition token for requesting this action has the symbolic feedback code FOR0070; its message number is 70.

Not all conditions allow any or all of the *fix-up and resume* actions. Before you write a condition handler that requests one of the *fix-up and resume* actions, ensure that the condition being processed allows that particular *fix-up and resume* action. When a *fix-up and resume* action is allowed for a condition, your condition handler must provide the qualifying data needed for the corrective action; in most cases, this is the new input value or the new output value. Depending on the condition, look for the information on how to do this in one of these places:

- For the Fortran conditions, refer to the permissible resume actions listed with the messages in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.
- For mathematical routine errors and arithmetic program interruptions, such as the exponent-overflow exception, see Appendix B, “Qualifying Data for Language Environment Conditions” on page 97.

The *fix-up and resume* action is allowed only if the resume cursor still points to the location where the error was originally detected.

User-Written Condition Handler Interface

This section discusses the details of the interface between the Language Environment condition manager and a user-written condition handler. Before you read this section, be sure that you understand the actions that a condition handler can request of the condition manager.

Here is the format of the Fortran SUBROUTINE statement for a condition handler and the Fortran declarations of the dummy arguments:

Fortran Declarations for a User-Written Condition Handler

```
SUBROUTINE name ( curr_ctok, token, result_code, new_ctok )  
CHARACTER*12 curr_ctok  
INTEGER*4 token  
INTEGER*4 result_code  
CHARACTER*12 new_ctok
```


name

The name of the condition handler. This is the name specified either as the first argument in the call to the CEEHDLR callable service or as the value of the USRHDLR run-time option.

curr_ctok (input)

A 12-byte condition token that represents the condition being processed by the condition manager and for which the condition handler is entered. This condition token has associated with it all of the qualifying data that applies to the condition.

token (input)

A 4-byte integer whose value is the token that was included as the second argument in the call to the CEEHDLR callable service that registered this condition handler. Using *token*, you can communicate data between the condition handler and the routine that registered the condition handler.

result_code (output)

A 4-byte integer that the condition handler sets to indicate how it wants the condition manager to continue its processing of the condition. Table 8 shows the permissible values for *result_code*. For details on the associated actions, see “Actions That a User-Written Condition Handler Can Request” on page 52.

Table 8. Values of *result_code* Set by a User-Written Condition Handler

| <i>result_code</i> | Response | Action |
|--------------------|--------------------------|---|
| 10 | <i>resume</i> | Resume execution at the resume cursor. |
| 20 | <i>percolate</i> | Percolate the condition to the next condition handler for the same stack frame as the current condition handler. |
| 21 | <i>percolate</i> | Move the handle cursor to the stack frame just before the one to which it currently points. Then percolate the condition to first condition handler, if any, for the stack frame to which the handle cursor was moved. |
| 30 | <i>promote</i> | Promote the condition to the one indicated by the <i>new_ctok</i> argument. Then provide this new condition to the next condition handler for the same stack frame as the current condition handler. |
| 31 | <i>promote</i> | Promote the condition to the one indicated by the <i>new_ctok</i> argument. Then move the handle cursor to the stack frame just before the one to which it currently points, and present this new condition to first condition handler, if any, for the stack frame to which the handle cursor was moved. |
| 32 | <i>promote</i> | Promote the condition to the one indicated by the <i>new_ctok</i> argument. Then provide this new condition to the first condition handler for the same stack frame as the current condition handler. |
| 60 | <i>fix-up and resume</i> | Provide the fix-up actions specified by the <i>new_ctok</i> argument and by the applicable qualifying data. Then resume execution at the resume cursor, which must not have been moved from its original point. |

If the condition handler doesn't set *result_code*, a value of 20 is assumed.

new_ctok (output)

A 12-byte condition code that the condition handler sets as follows based on the action it requests through the *result_code* argument:

- For *promote*, the condition handler sets *new_ctok* to the condition token for the condition to which the current condition should be changed.
- For *fix-up and resume*, the condition handler sets *new_ctok* to the condition token that indicates the type of fix-up that should be done. Table 9 shows the symbolic feedback codes that represent the permissible condition tokens for *new_ctok*.

Table 9. Actions for fix-up and resume

| Symbolic Feedback Code | Action |
|-------------------------------|--|
| CEE0CE | <i>resume with new input value</i> |
| CEE0CF | <i>resume with new output value</i> |
| FOR0070 | <i>resume with Fortran-specific correction</i> |

For details about the meaning of the three *fix-up and resume* actions, see the discussion of these actions on page 53.

For information on how to use symbolic feedback codes to set *new_ctok*, see “Symbolic Feedback Codes” on page 51.

The *new_ctok* argument isn't used for the *resume* and *percolate* actions.

Unhandled Conditions

During the condition step, the condition manager processes the stack frames in LIFO order, that is, from the most recent to the earliest in the call chain. In each stack frame, user condition handlers, if any, are entered in LIFO order prior to a language-specific condition handler.

If a condition isn't handled by any condition handler registered for any stack frame in the call chain, there is still one more user-written condition handler that can get control. If you've specified the name of a condition handler in the USRHDLR run-time option, then this condition handler gains control after the condition manager processes all of the stack frames. This condition handler can handle the condition if it chooses. If it doesn't, there's another language-specific handler that enforces default actions for certain languages; for conditions that arise from Fortran routines, no action is taken at this point.

If the condition still hasn't been handled, the condition is said to be *unhandled*. For the conditions signaled by Language Environment (as opposed to signaled by your own code), the default action taken for the unhandled condition depends on the severity of the condition as shown in Table 10 on page 57:

Table 10. Condition Manager Actions for an Unhandled Condition

| Severity | Action Taken by the Condition Manager |
|----------|--|
| 0 | Resumes execution without printing the message. |
| 1 | If the condition occurred in a stack frame associated with a COBOL routine, prints the message and resumes execution. If the condition occurred in a stack frame associated with a non-COBOL routine, resumes execution without printing the message. |
| 2, 3, 4 | Takes the following steps to terminate the application: <ol style="list-style-type: none">1. Promotes the unhandled condition to the T_I_U (CEE066, Termination Imminent due to Unhandled Condition) condition.2. Processes each stack frame in LIFO order beginning with the one where the condition occurred, giving control to the condition handlers (including the condition handler, if any, specified with the USRHDLR run-time option). During this step, any condition handler can abandon the termination by moving the resume cursor and requesting the <i>resume</i> action.3. Prints the message for the original condition unless the TERMTHDACT(QUIET) run-time option is in effect.4. Terminates the application in one of these ways:<ul style="list-style-type: none">• If the ABTERMENC(RETCODE) run-time option, terminates normally with a return code based on the user return code and the severity of the original condition.• If the ABTERMENC(ABEND) run-time option, terminates abnormally (abends) with one of several possible completion (abend) codes. <p>For more information on the return codes and the completion codes based on the ABTERMENC run-time option, see Chapter 9, "Interpreting Return Codes and Completion Codes" on page 41.</p> |

Overview of the VS FORTRAN Extended Error Handling Facility

Language Environment does not have an equivalent of the VS FORTRAN extended error handling facility, which includes the following:

- Automatic corrective actions that are taken when an error occurs. There are fix-up actions to correct errors in the arguments for the mathematical routines and to provide result values for certain program interruptions, such as the exponent-underflow and exponent-overflow exceptions.
- The ERRSET, ERRSTR, ERRSAV, and ERRMON subroutines.¹¹ These subroutines are used to control error handling characteristics, such as the number of times certain messages should be printed and the number of times certain errors should be allowed to occur before the application is terminated.
- The error option table, which contains the information that controls the error handling characteristics.

With Language Environment, unless your application takes specific action to deal with errors, it is terminated after the first occurrence of an error. Having your

¹¹ However, the ERRTRA subroutine, which produces a traceback of the active called routines, is available in Language Environment.

application terminated in this way is often exactly the way you want your application to behave. If so, then the only changes that you'll have to make will be to remove any references to the extended error handling facility subroutines (ERRSET, and so on) from your source programs.

As you saw earlier in this chapter, the Language Environment condition handling model defines a set of services for error handling. You can create a user-written condition handler that provides some or all of the functions of the VS FORTRAN extended error handling facility. This condition handler can perform fix-up actions that are similar to those in VS FORTRAN, and it can control when termination occurs.

If you're not already familiar with the VS FORTRAN extended error handling facility but must convert existing Fortran applications that use it, read the following sections to learn more about how the error option table, the subroutines, and the automatic error correction interact with each other. For a list of the VS FORTRAN error numbers and their associated corrective actions, see Appendix D, "VS FORTRAN Error Handling Behavior" on page 113.

Error Option Table

The extended error handling facility maintains a table called the *error option table* with an entry for each VS FORTRAN error number. Each entry contains the following information to control what happens when the error occurs:

- The number of occurrences of the error to allow before terminating the application
- The number of times the error message is to be printed before suppressing further printing of it
- A count of the number of times this error has occurred
- Whether the I/O buffer is to be printed after the message is printed
- Whether the traceback is to be printed after the message
- The name of the user error exit routine, if any, which is to be invoked after the message is printed

The information in the error option table controls the handling of errors throughout the entire application (unless the information is changed using the extended error handling facility subroutines), an approach to error handling sometimes called the *global error table* model. This differs from the Language Environment condition handling model where the actions of a condition handler do not apply to stack frames that are earlier in the call chain than the stack frame that registered the condition handler.

Those who install or customize VS FORTRAN at your site can customize the values in the error option table to apply selected error handling characteristics to all Fortran applications.

Extended Error Handling Facility Subroutines

The extended error handling facility includes the following five subroutines to manipulate the error option table and to control the handling of errors:

| Name | Purpose |
|-------------|--|
| ERRSAV | Copies an entry from the error option table into an 8-byte storage area accessible to the Fortran programmer. |
| ERRSTR | Stores an entry into the error option table. |
| ERRSET | Modifies certain information in the error option table for a specific error or for a range of errors. |
| ERRMON | Prints text provided by the user for a given error number and processes the error using the information in the corresponding error option table entry. This subroutine also returns an indication of whether the standard corrective action or a user corrective action should be taken. |
| ERRTRA | Prints a traceback of the called routines. ERRTRA is available in Language Environment. |

A call to the ERRSET subroutine can register a user error exit routine to be given control when a specific error occurs. For each different error, a unique set of arguments is passed to the user error exit routine. The exit routine can modify these arguments to request that some error corrective action, either a default action or a user-specified action, be taken.

Automatic Error Correction

Another feature of the VS FORTRAN extended error handling facility is the automatic correction of many errors. As an example, consider the case of the real-valued square root function. If the argument has a negative value, no mathematically correct square root can be computed. However, after printing an error message, the VS FORTRAN library automatically returns the square root of the absolute value of the argument. Unless overridden (either by customizing the error option table default values for your site or by calling the ERRSTR or ERRSET subroutine), this action occurs as many as nine times; then on the tenth occurrence of the error, the application is terminated.

For the details of the corrective actions that the extended error handling facility provides for each error, see Appendix D, “VS FORTRAN Error Handling Behavior” on page 113.

Fortran-Specific Services for Error Handling

Because Language Environment doesn't include the VS FORTRAN extended error handling facility, you must use Language Environment condition handling services to provide the error handling required by your Fortran routines. Although Fortran routines cannot call any of the Language Environment callable services directly, there are some Fortran-specific callable services that Fortran routines can use for calling the Language Environment callable services. There are also some services for retrieving and updating qualifying data. The following sections show you how to use these Fortran-specific services. For the complete descriptions of these services, see Appendix A, “Fortran Callable Services and Functions” on page 88.

Callable Services to Retrieve and Update Qualifying Data

From a user-written condition handler you'll usually have to use the qualifying data for the condition in order to provide the fix-up actions that your application needs. You are likely to use qualifying data in one or more of these ways:

- To determine which mathematical function or I/O statement was in use when the error was detected
- To examine the input data that caused the error to be detected
- To provide the condition manager a new argument value when you request the *resume with new input value* action
- To provide the condition manager a result value when you request the *resume with new output value* action

Language Environment provides the following callable services for retrieving and updating a single qualifying datum:

| Name | Purpose |
|-------------|---|
| QDFETCH | Returns the value of a specific qualifying datum to your program. |
| QDSTORE | Sets a specific qualifying datum to a value that your program supplies. |

For both services, the arguments are:

1. The condition token with which the qualifying datum is associated
2. A specification of which qualifying datum (first, second, and so on) you want to retrieve or update
3. The qualifying datum (the QDFETCH output value; the QDSTORE input value)

The QDFETCH and QDSTORE callable services are intended to be called by Fortran routines, but you can call them from assembler language routines if you follow the Fortran conventions for argument lists with character arguments. See "Passing Character Arguments Using the Standard Linkage Convention" in *VS FORTRAN Version 2 Programming Guide for CMS and MVS*.

The following example shows how to use the QDFETCH callable service.

```
CHARACTER*12 COND_INP
INTEGER*4    CHAR_POSITION
:
CALL QDFETCH ( COND_INP, 7, CHAR_POSITION )
```

Following are the arguments in the call to the QDFETCH callable service:

| | |
|---------------|---|
| COND_INP | The condition token, which, for this example, is assumed to be for condition FOR1003 (the error in which an integer input field in a READ statement contains a character other than an integer or a blank). |
| 7 | The specification that the seventh qualifying datum for condition FOR1003 is to be retrieved. |
| CHAR_POSITION | The variable in which the callable service returns the value of the seventh qualifying datum, a 4-byte integer containing the relative character position (first, second, and so on) of the invalid character within the whole input field. |

When you use the QDFETCH and QDSTORE callable services, be sure that the type and length of the third argument matches what is defined for the actual qualifying datum. Depending on the particular condition, determine the type, length, and content of the qualifying data as follows:

- For the Fortran conditions, see the Fortran messages in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.
- For abends, mathematical routine errors, and arithmetic program interruptions, such as the exponent-overflow exception, see Appendix B, "Qualifying Data for Language Environment Conditions" on page 97.

Fortran Functions to Retrieve Qualifying Data

There are a number of Fortran functions for retrieving a single qualifying datum. These provide exactly the same information as QDFETCH; however, because they're functions, you'll find them easier to use in many cases. These functions let you retrieve qualifying data of the indicated types:

| Name | Return Value Type |
|---------|-------------------|
| QDCH1 | CHARACTER*1 |
| QDCH6 | CHARACTER*6 |
| QDCH8 | CHARACTER*8 |
| QDCH31 | CHARACTER*31 |
| QDCH62 | CHARACTER*62 |
| QDCH255 | CHARACTER*255 |
| QDCX8 | COMPLEX*8 |
| QDCX16 | COMPLEX*16 |
| QDCX32 | COMPLEX*32 |
| QDINT1 | INTEGER*1 |
| QDINT2 | INTEGER*2 |
| QDINT4 | INTEGER*4 |
| QDINT8 | INTEGER*8 |
| QDR4 | REAL*4 |
| QDR8 | REAL*8 |
| QDR16 | REAL*16 |
| QDUS1 | UNSIGNED*1 |

The input arguments for these functions are:

1. The condition token with which the qualifying datum is associated
2. A specification of which qualifying datum (first, second, and so on) you want to retrieve

Because these functions are not intrinsic functions, you must declare their data types in your Fortran program by using member AFHCQDSB of data set CEE.V1R5M0.SCEESAMP. To do this, provide the following INCLUDE line among the data declarations in your Fortran program:

```
INCLUDE (AFHCQDSB)
```

and provide the following DD statement in your JCL for the compilation step:

```
//SYSLIB DD DSN=CEE.V1R5M0.SCEESAMP,DISP=SHR
```

The following example shows how to use the QDINT4 function:

```
CHARACTER*12 COND_INP
CHARACTER*1  INPUT_CHAR(255)
INCLUDE (AFHCQDSB)
:
PRINT *, 'The invalid character is ', INPUT_CHAR(QDINT4(COND_INP, 7))
```

For the Fortran condition FOR1003 (nonnumeric character in an integer input field), the QDINT4 function returns the character position of the invalid character within the input field.

Using the QDINT4 function can be more convenient than using the QDFETCH callable service because you don't need an intermediate variable to hold the value of the seventh qualifying datum.

When you use one of these functions to retrieve a qualifying datum, be sure that the function you use corresponds to the data type and length of the actual qualifying datum. Because the preceding example retrieved a 4-byte qualifying datum, it used the QDINT4 function.

Fortran Functions That Provide Information about Qualifying Data

The following Fortran functions return information about a single qualifying datum:

| Name | Information Returned |
|--------|---|
| QDLOC | Address, a 4-byte pointer |
| QDTYPE | Data type, a 4-byte integer as defined on page 94 |
| QDLEN | Length, a 4-byte integer |

Use the QDTYPE and QDLEN functions for a qualifying datum that can have different data types or lengths, depending on the arguments to the service that detected the error. You can use these functions only for a qualifying datum that has an associated q_data descriptor, which indicates the qualifying datum's type and length.

The input arguments for these functions are:

1. The condition token with which the qualifying datum is associated
2. A specification of which qualifying datum's (first, second, and so on) information you want returned

The QDLOC, QDTYPE, and QDLEN functions are not intrinsic functions. Therefore, use the AFHCQDSB file described on page 61 to provide data type declarations for the functions.

The following example shows how to use the QDLOC and QDLEN functions for the Fortran condition FOR1003 (nonnumeric character in an integer input field):

```
CHARACTER*12 COND_INP
POINTER*4    (INPUT_FIELD_LOC, INPUT_FIELD)
CHARACTER*255 INPUT_FIELD
INCLUDE (AFHCQDSB)
:
INPUT_FIELD_LOC = QDLOC(COND_INP, 6)
PRINT *, 'The invalid field is ', INPUT_FIELD(1:QDLEN(COND_INP, 6))
```


Fortran Services for Calling Language Environment Callable Services

There are two Fortran-specific callable services that Fortran routines can use to call most of the Language Environment callable services (in particular, the condition handling services) described in *Language Environment for MVS & VM Programming Reference*.

| Name | Purpose |
|---------|--|
| AFHCEEN | Passes control to any Language Environment callable service if you omit the optional feedback code when it is the last argument. |
| AFHCEEF | Passes control to any Language Environment callable service if you provide all of the arguments, including the optional feedback code. |

Both of these callable services have the following arguments:

1. The name of the Language Environment callable service that you want to call
2. The arguments required by the Language Environment callable service

Following is an example of using AFHCEEN to call CEERAN0, which generates a random number:

```
INTEGER*4 SEED / 0 /
REAL*8 RANDOM_NUM
EXTERNAL CEERAN0
:
CALL AFHCEEN (CEERAN0, SEED, RANDOM_NO)
```

The variables SEED and RANDOM_NUM are the CEERAN0 callable service's first two arguments. The last of the CEERAN0 callable service's three arguments, the feedback code, is omitted, as required for a call to AFHCEEN.

If you want to pass a feedback code to CEERAN0, and thus gain control if an error is detected, use the AFHCEEF callable service, and provide the feedback code as a character variable of length 12 as shown in this example:

```
INTEGER*4 SEED / 0 /
REAL*8 RANDOM_NO
CHARACTER*12 FC
EXTERNAL CEERAN0
:
CALL AFHCEEF (CEERAN0, SEED, RANDOM_NO, FC)
```

Handling Run-Time Errors from Your Fortran Routines

This section shows how to use Language Environment's condition handling to provide error handling similar to what you have in VS FORTRAN. There are several examples of user-written condition handlers that provide fix-up actions using the concepts and services discussed earlier in this chapter. The examples cover the following errors:

- Square-root exception
- Fixed-point overflow exception
- Exponent-overflow exception
- Invalid character in a numeric input field

Several of the examples use Fortran language extensions added to VS FORTRAN Version 2 in Release 6. These extensions include pointer variables, the LOC intrinsic function, and the use of hexadecimal constants in assignment statements.

Sample Condition Handler for a Square-Root Exception

Figure 5 shows a condition handler that handles the condition that is signaled when a negative argument is passed to the Fortran SQRT function. For this condition, the condition handler provides a new argument value that is the absolute value of the negative argument value, and it requests the *resume with new input value* action. This fix-up action is the same as what VS FORTRAN provides, but you could provide whatever fix-up values you choose. This example has a main program that registers the condition handler, invokes the SQRT function with an argument value of -4.0 , prints the final result after the fix-up, and finally prints the message about the error.

This example of the square-root exception is typical of how you would write condition handlers for errors detected by most of the mathematical routines.

For information on the qualifying data that are used in this example, see “q_data Structure for Math and Bit-Manipulation Conditions” on page 102.

```
PROGRAM SQRTXMP
REAL*4    ARG    / -4.0E0 /
REAL*4    RESULT
CHARACTER*12 ERROR_CTOK
EXTERNAL  SQRTHDL
EXTERNAL  CEEHDLR
EXTERNAL  CEEMSG
CALL AFHCEEN(CEEHDLR, SQRTHDL, LOC(ERROR_CTOK))
RESULT = SQRT(ARG)
PRINT *, 'The result of SQRT(', ARG, ') is', RESULT
CALL AFHCEEN(CEEMSG, ERROR_CTOK, 2)
END

SUBROUTINE SQRTHDL(HDLR_CTOK, COMM_TOK, RESPONSE, FIXUP_CTOK)
INCLUDE (CEEFORCT)
INCLUDE (AFHCQDSB)
CHARACTER*12 HDLR_CTOK
POINTER*4    (COMM_TOK, MAIN_CTOK)
INTEGER*4    RESPONSE
CHARACTER*12 FIXUP_CTOK
CHARACTER*12 MAIN_CTOK
IF (HDLR_CTOK(1:8) .EQ. CEE1UQ
1 .AND. QDCH8(HDLR_CTOK,2) .EQ. 'SQRT') THEN
    IF (QDLEN(HDLR_CTOK,6) .EQ. 4 ) THEN
        CALL QDSTORE(HDLR_CTOK, 6, ABS(QDR4(HDLR_CTOK,6)) )
    ELSE IF (QDLEN(HDLR_CTOK,6) .EQ. 8 ) THEN
        CALL QDSTORE(HDLR_CTOK, 6, DABS(QDR8(HDLR_CTOK,6)) )
    ELSE
        CALL QDSTORE(HDLR_CTOK, 6, QABS(QDR16(HDLR_CTOK,6)) )
    FIXUP_CTOK(1:8) = CEE0CE
    FIXUP_CTOK(9:12) = Z'00000000'
    RESPONSE = 60
ENDIF
MAIN_CTOK = HDLR_CTOK
RETURN
END
```

Figure 5. Condition Handler Giving New Argument Value for Square-Root Exception

Notes on the Example:

1. In the main program, the first call to the AFHCEEN callable invokes the CEEHDLR callable service to register the condition handler with the name SQRTHDL. The second and third arguments for AFHCEEN represent the first two arguments for CEEHDLR. The third argument for CEEHDLR is the feedback code, which is omitted in this case. (If you want to supply the feedback code, then use AFHCEEF instead of AFHCEEN.) Because the feedback code is omitted, a condition is signaled if the CEEHDLR callable service fails.
2. The three EXTERNAL statements are required because the names of the three subroutines are supplied as arguments to AFHCEEN.
3. When the main program invokes the SQRT function with the argument of -4.0 , condition CEE1UQ is signaled, and the condition manager gives control to the condition handler SQRTHDL.
4. In the condition handler SQRTHDL, the INCLUDE line includes the symbolic feedback code file CEEFORCT. As described in “Symbolic Feedback Codes” on page 51, this file contains Fortran language declarations for the symbolic feedback codes for the condition tokens created by the common component of Language Environment. These symbolic feedback codes have names beginning with CEE.
5. The first IF statement in the condition handler determines whether the condition is the square-root exception that the condition handler is prepared to handle. The square-root exception is identified by condition CEE1UQ and by a value of SQRT in the second qualifying datum. For any other condition, the condition handler returns without requesting any action; this is the same as requesting *percolate*.
6. Even though the main program uses only a REAL*4 argument for SQRT, the condition handler handles real variables of length 4, 8, or 16. The QDLEN function returns the length of the argument so that the condition handler can set a new argument value of the proper length.
7. The second argument for QDLEN refers to the qualifying datum, the sixth in this case, whose length is needed; it does not refer directly to the corresponding q_data descriptor, which is the fifth of the qualifying data.
8. The condition handler uses the QDSTORE callable service to update the sixth qualifying datum with the absolute value of the negative argument.
9. The condition handler sets the third dummy argument, RESPONSE, to 60 to request the *fix-up and resume* action, and it sets the fourth dummy argument, FIXUP_CTOK, to condition CEE0CE to request the *resume with new input value* action.
10. Before returning to the condition manager, the condition handler provides the main program a copy of the condition token for the condition being processed. It uses the second dummy argument, COMM_TOK, which is the value passed as the second argument in the call to CEEHDLR. This value is a pointer to the main program's variable ERROR_CTOK. Using a pointer in this way is a convention for communicating between this main program and this condition handler. When you write a condition handler, you can use the value that's passed as second argument to CEEHDLR in any way you choose.
11. To provide the *resume with new input value* action, the condition manager invokes the SQRT function again with the new argument value. This time, no

error is detected, and control returns to the main program where the SQRT function was invoked.

12. Through the final call to AFHCEEN, the main program invokes the CEEMSG callable service to print the message associated with the condition that was just handled. If the program hadn't printed the message itself, then no message would have been printed because Language Environment doesn't print the message for a handled condition.

Sample Condition Handlers for a Fixed-Point Overflow Exception

Requesting *resume* for Fixed-Point Overflow Exception

Figure 6 shows a condition handler that simply requests the *resume* action for a fixed-point overflow condition. The results are equivalent to the behavior of VS FORTRAN, where the program mask is set to disable the program interruption that occurs due to fixed-point overflow. Because this condition handler takes no action for the fixed-point overflow other than to resume execution, it doesn't use any of the qualifying data associated with the condition.

```
PROGRAM FOVXMP1
  INTEGER*4  INT_VAR    / 0 /
  INTEGER*4  BIG        / 2000000000 /
  EXTERNAL  CEEHDLR, FOVHDL1,
  CALL AFHCEEN(CEEHDLR, FOVHDL1, 0)
  INT_VAR = BIG + BIG
  PRINT *, 'SUM IS', INT_VAR
  END

SUBROUTINE FOVHDL1(HDLR_CTOK, DUMMY1, RESPONSE, DUMMY2)
  INCLUDE (CEEFORCT)
  CHARACTER*12 HDLR_CTOK
  INTEGER*4    DUMMY1
  INTEGER*4    RESPONSE
  CHARACTER*12 DUMMY2
  IF (HDLR_CTOK(1:8) .EQ. CEE348) THEN
    RESPONSE = 10
  ENDIF
  END
```

Figure 6. Condition Handler Requesting *resume* for Fixed-Point Overflow Exception

Notes on the Example:

1. The addition of the large numbers in FOVXMP1 causes a fixed-point overflow exception. Condition CEE348 is signaled, and the condition manager gives control to the condition handler FOVHDL1.
2. The IF statement in the condition handler determines whether the condition is the fixed-point overflow exception that the condition handler is prepared to handle. The exponent-overflow exception is identified by condition CEE348. For any other condition, the condition handler returns without requesting any action; this is the same as requesting *percolate*.
3. The condition handler sets the third dummy argument, RESPONSE, to 10 to request the *resume* action.
4. To provide the *resume* action, the condition manager restores the registers to the values they had at the time of the interruption and returns to the next sequential instruction after the failing instruction. In the event that the failing

machine instruction was a vector instruction, the condition manager resumes the execution of that instruction.

5. The result of the addition in the main program is a negative number with a large magnitude.

Providing Information about Fixed-Point Overflow Exceptions

In Figure 7, the condition handler for the fixed-point overflow condition requests the *resume* action, and it sets increments or decrements one of the main program's variables to indicate whether the fixed-point overflow is in a positive or negative direction. After the computation, the main program determines whether the overflows in the positive and negative directions cancel each other so that the final result of the computation is correct.

For information on the qualifying data that are used in this example, see “q_data Structure for Arithmetic Program Interruptions” on page 98.

```
PROGRAM FOVXMP2
INTEGER*4  DIRECTION
INTEGER*4  INT_VAR      / 0 /
INTEGER*4  POS          / 2000000000 /
INTEGER*4  NEG          / -2000000000 /
EXTERNAL  FOVHDL2
EXTERNAL  CEEHDLR
CALL AFHCEEN(CEEHDLR, FOVHDL2, LOC(DIRECTION))
DIRECTION = 0
INT_VAR = POS + POS + POS + NEG + NEG + NEG
IF (DIRECTION .EQ. 0) THEN
  PRINT *, 'OVERFLOWS BALANCE. THE RESULT IS', INT_VAR
ELSE IF (DIRECTION .GT. 0) THEN
  PRINT *, 'THE RESULT IS', INT_VAR, '+', DIRECTION, '*2**32'
ELSE
  PRINT *, 'THE RESULT IS', INT_VAR, '-', -DIRECTION, '*2**32'
ENDIF
END

SUBROUTINE FOVHDL2(HDLR_CTOK, PASSED_DIR_PTR, RESPONSE, DUMMY)
INCLUDE (CEEFORCT)
INCLUDE (AFHCQDSB)
CHARACTER*12 HDLR_CTOK
POINTER*4    (PASSED_DIR_PTR, PASSED_DIRECTION)
INTEGER*4    RESPONSE
CHARACTER*12 DUMMY
POINTER*4    (INST_RESULT_PTR, INST_RESULT)
INTEGER*4    INST_RESULT
INTEGER*4    PASSED_DIRECTION
IF (HDLR_CTOK(1:8) .EQ. CEE348) THEN
  INST_RESULT_PTR = QDLOC(HDLR_CTOK, 3)
  IF (INST_RESULT .LT. 0) THEN
    PASSED_DIRECTION = PASSED_DIRECTION + 1
  ELSE
    PASSED_DIRECTION = PASSED_DIRECTION - 1
  ENDIF
  RESPONSE = 10
ENDIF
END
```

Figure 7. Condition Handler Providing Information on Fixed-Point Overflow Exceptions

Notes on the Example:

1. The additions of the large numbers in FOVXMP2 cause fixed-point overflow exceptions. Condition CEE348 is signaled, and the condition manager gives control to the condition handler FOVHDL2.
2. The outermost IF statement in the condition handler determines whether the condition is the fixed-point overflow exception that the condition handler is prepared to handle. The exponent-overflow exception is identified by condition CEE348. For any other condition, the condition handler returns without requesting any action; this is the same as requesting *percolate*.
3. The condition handler uses the QDLOC function to get the address of the data that the failing machine instruction left in the general register. Based on whether this data is either positive or negative, the condition handler either decrements or increments one of the main program's variables to indicate how many times the result is in error by a value of 2^{32} . It does this by using the second dummy argument, PASSED_DIR_PTR, which is the value given as the second argument in the call to CEEHDLR. In this example, this value is a pointer to the main program's variable DIRECTION.
4. The condition handler sets the third dummy argument, RESPONSE, to 10 to request the *resume* action.
5. To provide the *resume* action, the condition manager restores the registers to the values they had at the time of the interruption and returns to the next sequential instruction after the failing instruction. In the event that the failing machine instruction was a vector instruction, the condition manager resumes the execution of that instruction.

Sample Condition Handler for an Exponent-Overflow Exception

Figure 8 on page 69 shows a condition handler that handles the exponent-overflow exception. The condition handler requests the same fix-up action that VS FORTRAN provides, that is, it sets the result of the failing floating-point instruction to the value with the largest magnitude. Depending on the sign that the correct mathematical result would have had, this result value is either positive or negative.

For information on the qualifying data that are used in this example, see “q_data Structure for Arithmetic Program Interruptions” on page 98.

```

PROGRAM XOVMXMP
INTEGER*4    OVERFLOWS / 0 /
REAL*4      BIG_4     / 1E50 /, RESULT_4
REAL*8      BIG_8     / 1D50 /, RESULT_8
REAL*16     BIG_16    / 1Q50 /, RESULT_16
EXTERNAL    CEEHDLR, XOVMFHD
CALL AFHCEEN(CEEHDLR, XOVMFHD, LOC(OVERFLOWS))
RESULT_4 = BIG_4 * BIG_4
RESULT_8 = -BIG_8 * BIG_8
RESULT_16 = BIG_16 * BIG_16
PRINT *, OVERFLOWS, ' exponent-overflow exceptions.'
PRINT *, 'REAL*4 result is ', RESULT_4
PRINT *, 'REAL*8 result is ', RESULT_8
PRINT *, 'REAL*16 result is ', RESULT_16
END

SUBROUTINE XOVMFHD(HDLR_CTOK, COUNTER_PTR, RESPONSE, FIXUP_CTOK)
INCLUDE (CEEFORCT)
INCLUDE (AFHCQDSB)
CHARACTER*12 HDLR_CTOK
POINTER*4    (COUNTER_PTR, PASSED_COUNTER)
INTEGER*4    RESPONSE
CHARACTER*12 FIXUP_CTOK
INTEGER*4    PASSED_COUNTER
INTEGER*4    RESULT_LENGTH
REAL*4       MAX_POS_4 /Z7FFFFFFFF/
REAL*4       MAX_NEG_4 /ZFFFFFFFF/
REAL*8       MAX_POS_8 /Z7FFFFFFFFFFFFFFFF/
REAL*8       MAX_NEG_8 /ZFFFFFFFFFFFFFFFF/
REAL*16      MAX_POS_16 /Z7FFFFFFFFFFFFFFFF71FFFFFFFFFFFFFFFF/
REAL*16      MAX_NEG_16 /ZFFFFFFFFFFFFFFFFF1FFFFFFFFFFFFFFFF/
EXTERNAL    CEEMSG
IF (HDLR_CTOK(1:8) .EQ. CEE34C) THEN
    RESULT_LENGTH = QDLEN(HDLR_CTOK,5)
    IF (RESULT_LENGTH .EQ. 4) THEN
        IF (QDR4(HDLR_CTOK,5) .GT. 0E0) THEN
            CALL QDSTORE(HDLR_CTOK, 5, MAX_POS_4)
        ELSE
            CALL QDSTORE(HDLR_CTOK, 5, MAX_NEG_4)
        ENDIF
    ELSE IF (RESULT_LENGTH .EQ. 8) THEN
        IF (QDR8(HDLR_CTOK,5) .GT. 0D0) THEN
            CALL QDSTORE(HDLR_CTOK, 5, MAX_POS_8)
        ELSE
            CALL QDSTORE(HDLR_CTOK, 5, MAX_NEG_8)
        ENDIF
    ELSE
        IF (QDR16(HDLR_CTOK,5) .GT. 0Q0) THEN
            CALL QDSTORE(HDLR_CTOK, 5, MAX_POS_16)
        ELSE
            CALL QDSTORE(HDLR_CTOK, 5, MAX_NEG_16)
        ENDIF
    ENDIF
    FIXUP_CTOK(1:8) = CEE0CF
    FIXUP_CTOK(9:12) = Z'00000000'
    RESPONSE = 60
    PASSED_COUNTER = PASSED_COUNTER + 1
    CALL AFHCEEN(CEEMSG, HDLR_CTOK, 2)
ENDIF
END

```

Figure 8. Condition Handler Giving New Result Value for Exponent-Overflow Exception

Notes on the Example:

1. Each of the three multiplications in X0VFXMP causes an exponent-overflow exception. Condition CEE34C is signaled, and the condition manager gives control to the condition handler X0VFHDL.
2. The first IF statement in the condition handler determines whether the condition is the exponent-overflow exception that the condition handler is prepared to handle. The exponent-overflow exception is identified by condition CEE34C. For any other condition, the condition handler returns without requesting any action; this is the same as requesting *percolate*.
3. The condition handler sets the third dummy argument, RESPONSE, to 60 to request the *fix-up and resume* action, and it sets the fourth dummy argument, FIXUP_CTOK, to condition CEE0CF to request the *resume with new output value* action.
4. The condition handler increments the main program's counter that indicates the number of occurrences of condition CEE34C. It does this by using the second dummy argument, COUNTER_PTR, which is the value given as the second argument in the call to CEEHDLR. In this example, this value is a pointer to the main program's variable OVERFLOWS.
5. Before returning to the condition manager, the condition handler calls AFHCEEN to invoke the CEEMSG callable service to print the message associated with the condition.
6. To provide the *resume with new output value* action, the condition manager updates the floating-point register used in the failing machine instruction with the value that the condition handler stored in the fifth qualifying datum. Then the condition manager passes control to the next sequential machine instruction. In the event that the failing machine instruction was a vector instruction, the condition manager resumes the execution of that instruction.

Sample Condition Handlers for an Invalid Character in a Numeric Field

The section has two examples showing condition handlers that handle the condition that's signaled for a READ statement when the input field corresponding to a numeric edit descriptor contains an invalid character.

Replacing a Nonnumeric Input Character with Zero

Figure 9 on page 71 shows a condition handler whose fix-up action is to replace the invalid numeric character in the input field with a 0 and then to repeat the data conversion with the field's updated value. This is VS FORTRAN's standard corrective action for this error.

For information on the qualifying data that are used in these examples, see condition FOR1003 in the section with the Fortran run-time messages in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.

```

PROGRAM ICHXMP1
INTEGER*2      INT_ITEM
CHARACTER*20   INPUT_FILE / ' 12X4 ' /
EXTERNAL      ICHHDL1
EXTERNAL      CEEHDLR
CALL AFHCEEN(CEEHDLR, ICHHDL1, 0)
READ (INPUT_FILE, *) INT_ITEM
PRINT *, 'INT_ITEM has the value', INT_ITEM
END

SUBROUTINE ICHHDL1(HDLR_CTOK, DUMMY, RESPONSE, FIXUP_CTOK)
INCLUDE (FORFORCT)
INCLUDE (CEEFORCT)
INCLUDE (AFHCQDSB)
CHARACTER*12   HDLR_CTOK
INTEGER*4      DUMMY
INTEGER*4      RESPONSE
CHARACTER*12   FIXUP_CTOK
POINTER*4      (INPUT_FIELD_PTR, INPUT_FIELD)
CHARACTER*1    INPUT_FIELD(255)
IF (HDLR_CTOK(1:8) .EQ. FOR1003) THEN
    INPUT_FIELD_PTR = QDLOC(HDLR_CTOK, 6)
    INPUT_FIELD(QDINT4(HDLR_CTOK, 7)) = '0'
    FIXUP_CTOK(1:8) = CEE0CE
    FIXUP_CTOK(9:12) = Z'00000000'
    RESPONSE = 60
ENDIF
END

```

Figure 9. Condition Handler Replacing a Nonnumeric Input Character with Zero

Notes on the Example:

1. After registering the condition handler ICHHDL1, the main program executes a list-directed READ statement referring to an internal file. This internal file, INPUT_FILE, contains a single field with the value 12X4. The READ statement tries to interpret this value as an integer, but condition FOR1003 is signaled because of the nonnumeric character in the value.
2. The IF statement in the condition handler determines whether the condition is the invalid-character condition, FOR1003, that the condition handler is prepared to handle. For any other condition, the condition handler returns without requesting any action; this is the same as requesting *percolate*.
3. Using the QDLOC function, the condition handler obtains the address of the sixth qualifying datum, which contains the whole input field, and stores this address in the pointer INPUT_FIELD_PTR so that the variable INPUT_FIELD contains the input data.
4. The seventh qualifying datum is the subscript value that refers to the character in error within the input field. The condition handler gets this value with the QDINT4 function and replaces the invalid character with a 0.
5. The condition handler sets the third dummy argument, RESPONSE, to 60 to request the *fix-up and resume* action, and it sets the fourth dummy argument, FIXUP_CTOK, to condition CEE0CE to request the *resume with new input value* action.
6. To provide the *resume with new input value* action, the condition manager retries the data conversion with the updated input field, and execution of the READ statement continues.

Giving New Result Value for Nonnumeric Input Character

In Figure 10 the fix-up action for an invalid numeric character is to provide a value of 1 as the result of the data conversion.

```
PROGRAM ICHXMP2
COMMON      / ICHCOM2 / INPUT_LENGTH, ERROR_INDEX, INPUT_FIELD
INTEGER*2   INPUT_LENGTH
INTEGER*2   ERROR_INDEX
CHARACTER*255 INPUT_FIELD
INTEGER*2   INT_ITEM
CHARACTER*20 INPUT_FILE / ' 12X4 ' /
EXTERNAL    CEEHDLR, ICHHDL2
CALL AFHCEEN(CEEHDLR, ICHHDL2, 0)
ERROR_INDEX = 0
READ (INPUT_FILE, *) INT_ITEM
PRINT *, 'INT_ITEM has the value', INT_ITEM
IF (ERROR_INDEX .NE. 0) THEN
    PRINT *, 'Invalid character at position', ERROR_INDEX
    PRINT *, 'Input field: "', INPUT_FIELD(1:INPUT_LENGTH), '"'
ENDIF
END

SUBROUTINE ICHHDL2(HDLR_CTOK, DUMMY, RESPONSE, FIXUP_CTOK)
INCLUDE (FORFORCT)
INCLUDE (CEEFORCT)
INCLUDE (AFHCQDSB)
COMMON      / ICHCOM2 / INPUT_LENGTH, ERROR_INDEX, INPUT_FIELD
INTEGER*2   INPUT_LENGTH
INTEGER*2   ERROR_INDEX
CHARACTER*255 INPUT_FIELD
CHARACTER*12 HDLR_CTOK
INTEGER*4   DUMMY
INTEGER*4   RESPONSE
CHARACTER*12 FIXUP_CTOK
INTEGER*4   RESULT_LENGTH
INTEGER*1   INT1_1   / 1 /
INTEGER*2   INT2_1   / 1 /
INTEGER*4   INT4_1   / 1 /
INTEGER*8   INT8_1   / 1 /
IF (HDLR_CTOK(1:8) .EQ. FOR1003 .AND.
1 QDTYPE(HDLR_CTOK,11) .EQ. QDTYPE_INTEGER) THEN
    RESULT_LENGTH = QDLEN(HDLR_CTOK,11)
    IF (RESULT_LENGTH .EQ. 1) THEN
        CALL QDSTORE(HDLR_CTOK, 11, INT1_1)
    ELSE IF (RESULT_LENGTH .EQ. 2) THEN
        CALL QDSTORE(HDLR_CTOK, 11, INT2_1)
    ELSE IF (RESULT_LENGTH .EQ. 4) THEN
        CALL QDSTORE(HDLR_CTOK, 11, INT4_1)
    ELSE
        CALL QDSTORE(HDLR_CTOK, 11, INT8_1)
    ENDIF
    FIXUP_CTOK(1:8) = CEE0CF
    FIXUP_CTOK(9:12) = Z'00000000'
    RESPONSE = 60
    INPUT_LENGTH = QDLEN(HDLR_CTOK,6)
    CALL QDFETCH(HDLR_CTOK, 6, INPUT_FIELD(1:INPUT_LENGTH))
    ERROR_INDEX = QDINT4(HDLR_CTOK, 7)
ENDIF
END
```

Figure 10. Condition Handler Giving New Result Value for Nonnumeric Input Character

Notes on the Example:

1. The main program and the condition handler ICHHDL2 share the common block ICHCOM2 to communicate information about the input field that was in error. The condition handler provides the field's length and contents as well as the subscript that refers to the invalid character.
2. As in the earlier example, the condition handler determines whether the condition is FOR1003. For simplicity, this example handles input items of integer type only. It determines whether the condition applies to an integer item by using the QDTYPE function to obtain the data type of the eleventh qualifying datum, the result field. If this result field is not of integer type, then instead of dealing with the condition, the condition handler lets the condition percolate.
3. The file AFHCQDSB contains declarations for named constants, such as QDTYPE_INTEGER, for the values returned by the QDTYPE function. For a list of these named constants, see the description of the QDTYPE function in Appendix A on page 94.
4. After determining the length of the integer result field with the QDLEN function, the condition handler uses the QDSTORE callable service to set an integer value 1 of the appropriate length into the eleventh qualifying datum, which is the result of the data conversion.
5. The condition handler sets the third dummy argument, RESPONSE, to 60 to request the *fix-up and resume* action, and it sets the fourth dummy argument, FIXUP_CTOK, to condition CEE0CF to request the *resume with new output value* action.
6. Before returning to the condition manager, the condition handler updates the fields in the common block ICHCOM2 with information about the input field that was in error. This information includes the length of the input field, the actual contents of the input field, and the subscript value of the character in error within the input field.
7. To provide the *resume with new output value* action, the condition manager provides the value in the eleventh qualifying datum as the result of the data conversion. This result becomes the value of the input item INT_ITEM in the main program.
8. There's an implicit assumption that ICHXMP2 is compiled with optimization level 0 or 1, that is, with the OPT(0) or OPT(1) compile-time option. If ICHXMP2 were compiled with a higher optimization level, the compiler would assume that ERROR_INDEX doesn't change between the assignment statement (where it's set to 0) and the IF statement (where it's tested); therefore, the compiler wouldn't generate any code to test the value of ERROR_INDEX. The problem is that the compiler isn't aware of the hidden transfer of control to the condition handler ICHHDL2, which sets ERROR_INDEX to a different value. At optimization level 0 or 1, the compiler doesn't do the analysis that causes this problem.

Regaining Control for Conditions Not Handled by a Subprogram

You can structure your application with a control program that continues running in spite of conditions that aren't handled by routines it calls. The example in Figure 11 on page 74 shows how to do this with a user-written condition handler that uses the CEEMRCR callable service. (CEEMRCR moves the resume cursor relative to the position of the handle cursor so that control is passed to the point at

which one routine calls another. CEEMRCR is described in *Language Environment for MVS & VM Programming Reference*.) In this example, CNTLPGM is the control program to which the control should return even if conditions signaled in routines it calls aren't handled.

```

PROGRAM CNTLPGM
INCLUDE (CEEFORCT)
EXTERNAL    PROCRTN
REAL*4      INARG    / -4.0 /
REAL*4      OUTARG
CHARACTER*12 PROCRTN_FC
EXTERNAL    CEEMSG
CALL ROUTER (PROCRTN, INARG, OUTARG, PROCRTN_FC)
IF (PROCRTN_FC(1:8) .NE. CEE000) THEN
    PRINT *, 'PROCRTN FAILED. MESSAGE FOLLOWS.'
    CALL AFHCEEN (CEEMSG, PROCRTN_FC, 2)
ENDIF
END

SUBROUTINE ROUTER (ROUTINE, ARG1, ARG2, ROUTINE_FC)
INCLUDE (CEEFORCT)
EXTERNAL    ROUTINE
REAL*4      ARG1, ARG1
CHARACTER*12 ROUTINE_FC
EXTERNAL    GETFC
EXTERNAL    CEEHDLR
CALL AFHCEEN(CEEHDLR, GETFC, LOC(ROUTINE_FC))
ROUTINE_FC(1:8) = CEE000
ROUTINE_FC(9:12) = Z'00000000'
CALL ROUTINE (ARG1, ARG2)
END

SUBROUTINE GETFC(HDLR_CTOK, PASS_FC_PTR, RESPONSE, FIXUP_CTOK)
CHARACTER*12 HDLR_CTOK
POINTER*4    (PASS_FC_PTR, PASSED_FC)
CHARACTER*12 PASSED_FC
INTEGER*4    RESPONSE
CHARACTER*12 FIXUP_CTOK
EXTERNAL    CEEMRCR
PASSED_FC = HDLR_CTOK
CALL AFHCEEN (CEEMRCR, 1)
RESPONSE = 10
END

SUBROUTINE PROCRTN (INPUT, OUTPUT)
REAL*4      INPUT, OUTPUT
OUTPUT = SQRT(INPUT)
END

```

Figure 11. Using CEEMRCR to Return to a Common Point for Conditions in a Subroutine

Notes on the Example:

1. The control program CNTLPGM calls ROUTER, passing to it the following:
 - a. The name of the subroutine to be called (PROCRTN in this case)
 - b. The actual arguments to be passed to PROCRTN
 - c. The feedback code in which ROUTER returns the feedback code that results from calling PROCRTN

2. Through the Fortran-specific callable service AFHCEEN (discussed in “Fortran Services for Calling Language Environment Callable Services” on page 63), the routine ROUTER uses the CEEHDLR callable service to register the user-written condition handler GETFC. Using CEEHDLR's second argument (AFHCEEN's third argument), ROUTER passes to GETFC the address of the feedback code argument provided by CNTLPGM.
3. After setting the feedback code argument to CEE000, ROUTER calls the subroutine that CNTLPGM specified should be called (PROCRTN).
4. If the subroutine PROCRTN experiences a condition that isn't handled, the condition is percolated to the condition handler GETFC, which returns the condition through the feedback code argument provided by CNTLPGM. In this example, PROCRTN calls the SQRT function with the negative argument that CNTLPGM provided; because PROCRTN hasn't registered a condition handler, the condition for the square-root exception is presented to GETFC.
5. Before returning to the condition manager, GETFC uses the CEEMRCR callable service to move the resume cursor to the point in CNTLPGM immediately following the call to ROUTER.
6. Because GETFC requests the *resume* action (by setting RESPONSE to 10), the condition manager passes control to the statement where the resume cursor points—just beyond the call to ROUTER.
7. After it regains control, the control program CNTLPGM checks the feedback code to determine whether the subprogram PROCRTN completed successfully.

In the example, ROUTER receives and passes along exactly two arguments for the routine requested by CNTLPGM. However, ROUTER could be generalized to receive and pass along an arbitrary number of arguments.

Chapter 11. Making Other Source Program Changes

Chapter 10 discussed how to use the Language Environment condition handling model to replace the VS FORTRAN extended error handling facility. Besides error handling, there are a few other differences between VS FORTRAN and Language Environment; some of these might require changes in your source programs. This chapter discusses the following topics:

- Texts of run-time error messages
- Values returned through the IOSTAT specifier
- Permissible input/output to the error message unit
- Data set attributes for the message file
- Fix-up for misaligned vector instruction operands
- Fixed-point overflow
- DVCHK and OVERFL subroutines
- Run-time initialization from assembler language
- Assembler language routines that find program arguments

Texts of Run-Time Error Messages

With Language Environment, the Fortran run-time error messages differ from the corresponding VS FORTRAN messages in several ways:

- The message number prefixes and the message numbers are different. For example, Language Environment produces the message FOR1002E instead of the AFB212I that VS FORTRAN Version 2 produces.
- The message texts more accurately describe the error that was detected.
- The message texts have more information describing a given occurrence of the error. For example, the message texts for most I/O conditions include additional information, such as file names, unit numbers, and record lengths, that isn't in the VS FORTRAN message texts.

As an example, consider the following VS FORTRAN Version 2 message that's printed when the record read by a formatted READ statement doesn't contain enough data for all the input items:

```
AFB212I  VCOMH : FORMATTED I/O, END OF RECORD, FILE MYINPUT
```

This is the equivalent Language Environment message:

```
FOR1002E  The READ statement for unit 10, which was connected to MYINPUT,
          failed. An input item required data from beyond the end of the
          data that was available in a record. The length of the available
          data was 56.
```

If any of your programs read and interpret files containing error messages, change these programs either to interpret the new messages or to obtain the required information using Language Environment callable services.

Values Returned through the IOSTAT Specifier

All of the integer values provided when an I/O statement contains the IOSTAT specifier are different from their VS FORTRAN values. The values provided by Language Environment are the new message numbers that are shown for the Fortran run-time messages in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*. For example, the following Fortran statements cause condition FOR1002 to be signaled if the READ statement encounters a record that's too short:

```
INTEGER*4  ERROR_NO
REAL*4    A
:
OPEN (10, FILE='MYINPUT')
READ (10, '(56X, F8.2)', IOSTAT=ERROR_NO) A
:
```

VS FORTRAN sets the variable ERROR_NO to 212 while Language Environment sets it to 1002.

If you have any programs that depend on the error numbers returned through the IOSTAT specifier, change them to refer to the Language Environment numbers. For the mapping of the VS FORTRAN Version 2 message numbers to the Language Environment conditions, see Appendix C on page 107. This appendix also shows the mapping of the Language Environment conditions to the VS FORTRAN Version 2 message numbers on page 110.

As shown in Appendix C, there isn't a one-to-one mapping between the Language Environment and the VS FORTRAN Version 2 error numbers. For example, the VS FORTRAN Version 2 error AFB212 is detected for a short record with either a READ or a WRITE statement. With Language Environment, a short record detected during execution of a READ statement causes condition FOR1002 to be signaled while a short record during execution of a WRITE statement causes condition FOR1001 to be signaled.

Permissible Input/Output to the Error Message Unit

With Language Environment, error messages and other diagnostic information are written to the Language Environment message file along with certain printed output from programs written in other high-level languages. In a Fortran routine, all output directed to the Fortran error message unit is written to the message file. Fortran also has another unit, the print unit, which is used for output from certain Fortran output statements. The unit numbers of the error message unit and the print unit are specified by the ERRUNIT and PRTUNIT run-time options. Depending on whether the Fortran print unit is the same unit as the error message unit or a different unit, printed output from your Fortran program either can be interspersed with other message file output or can be written to a separate file. Table 11 on page 78 summarizes the destination of various forms of printed output. In the table, *print file* means the file associated with the print unit when the error message unit and the print units are different units.

Table 11. Destination of Printed Output from Fortran Routines

| Fortran Statement or Callable Service | Print Unit Same as the Error Message Unit | Destination of Output |
|---|---|-----------------------|
| PRINT statement | Yes | Message file |
| | No | Print file |
| WRITE ([UNIT=]*, [FMT=] <i>fmt</i> , ...) where <i>fmt</i> is any format identifier. | Yes | Message file |
| | No | Print file |
| WRITE ([UNIT=] <i>emu</i> , [FMT=] <i>fmt</i> , ...) where <i>emu</i> has same value as the error message unit number, and <i>fmt</i> is any format identifier. | Either | Message file |
| CDUMP / CPDUMP callable service DUMP / PDUMP callable service SDUMP callable service | Either | Message file |
| Error messages from any component of Language Environment | Either | Message file |

The message file can be used only for output, and the only Fortran statements that can refer to the error message unit are those listed in the preceding table plus the INQUIRE, OPEN, and CLOSE statements. This differs from VS FORTRAN where the error message unit can be used for both input and output.¹²

You must make changes if your programs use the following statements to refer to the error message unit:

- READ
- BACKSPACE
- REWIND
- ENDFILE
- CLOSE with a STATUS specifier value of DELETE

If you want to use the preceding I/O statements for the file that contains the printed output produced by WRITE and PRINT statements in your Fortran routines, make the following changes:

- Provide different values for the ERRUNIT and PRTUNIT run-time options so that the error message unit and the print unit are different units.
- On the I/O statements that refer to the error message unit, provide a unit number that is the same as what you've used as the value of the PRTUNIT run-time option.

The suggested changes separate the message file output from the print file output and allow you to manipulate the print file output. The results won't be identical to those of VS FORTRAN because you still can't use statements other than those listed in Table 11 for the error message unit.

¹² There are additional Fortran statements, such as an unformatted WRITE statement or a direct access WRITE statement, that can't refer to the error message unit with either VS FORTRAN or Language Environment.

Data Set Attributes for the Message File

As shown in Table 11 on page 78, certain printed output from your Fortran routines is written to the Language Environment message file. This file is shared with routines written in other languages and is equivalent to the file that is connected to the error message unit in VS FORTRAN. However, with Language Environment, as shipped by IBM, the data set attributes shown in Table 12 have values that differ from those with VS FORTRAN:

Table 12. IBM-Supplied Attributes for Message File

| Data Set Attribute | DCB Parameter | VS FORTRAN Value | Language Environment Value |
|--------------------|---------------|------------------|----------------------------|
| Record format | RECFM | UA | FBA |
| Record length | LRECL | 133 | 121 |
| ddname | — | FTnnF001 | SYSOUT |

If your Fortran routine writes to the message file, such as with a PRINT statement, and if your output record is longer than what a message file record holds, the condition FOR1001 is signaled, and your application is terminated if the condition isn't handled. To avoid this problem, increase the message file's record length in one of the following ways:

- In the DD statement or ALLOCATE command for the message file, provide an LRECL parameter that's large enough to hold your records.
- As the third subparameter of the MSGFILE run-time option, provide a larger value for the record length. For example, the following run-time option specifies values for the record format and the record length that are the same as in VS FORTRAN:

```
MSGFILE(SYSOUT,UA,133)
```
- Prepare a set of default run-time options that you link-edit with your application, and include a MSGFILE run-option such as the preceding one. For further information, see "Providing Default Run-Time Options for Your Application" on page 40 and the chapter on run-time options in *Language Environment for MVS & VM Programming Guide*.
- Ensure that Language Environment is installed at your site with a MSGFILE run-time option default value such as the preceding one. If you do this, the default values will apply to all applications that run with Language Environment. For information on setting default run-time options for your site, see *Language Environment for MVS & VM Installation and Customization on MVS*.

If you want to make the ddname of the message file the same as it was in VS FORTRAN, provide the ddname as the first suboption of the MSGFILE run-time option as follows:

```
MSGFILE(FTnnF001)
```

where *nn* is the unit number of the error message unit. The value *nn* is the value given for the ERRUNIT run-time option:

```
ERRUNIT(nn)
```

Fix-Up for Misaligned Vector Instruction Operands

Unlike VS FORTRAN Version 2, there is no fix-up action in Language Environment to provide error recovery when a vector instruction operand isn't aligned on the proper boundary in storage. For example, if an array with REAL*8 elements is used in vectorized code, the array must be aligned so its elements are on doubleword storage boundaries. If the elements aren't properly aligned and if a vector instruction refers to the incorrectly aligned data, a specification exception is detected, and a program interruption occurs.

Usually data is incorrectly aligned because of its declaration either in a common block or with an EQUIVALENCE statement. In the following example, the array R8 isn't aligned on doubleword storage boundaries because of its position within the COM1 common block. When the array is passed to the DOADD subroutine, which in this example is assumed to be compiled with the VECTOR compile-time option, a specification exception occurs.

```
COMMON / COM1 / I4, R8
INTEGER*4 I4
REAL*8 R8(10000)
:
CALL DOADD (R8)
:
END
SUBROUTINE DOADD (X8)
REAL*8 X8(10000)
DO I = 1, 10000
  X8(I) = X8(I) + 1.0D0
ENDDO
END
```

To correct this problem, add padding in the common block as shown to align the array R8 on doubleword storage boundaries:

```
COMMON / COM1 / I4, DUMMY4, R8
INTEGER*4 I4
INTEGER*4 DUMMY4
REAL*8 R8(10000)
:
CALL DOADD (R8)
:
END
SUBROUTINE DOADD (X8)
REAL*8 X8(10000)
DO I = 1, 10000
  X8(I) = X8(I) + 1.0D0
ENDDO
END
```

If you change the common block in this way, recompile all programs that use it.

Fixed-Point Overflow

When there is a Fortran routine in an application, the program mask bit for fixed-point overflow is on, which causes a program interruption when fixed-point overflow occurs. This is different from VS FORTRAN, where the mask bit is off, and no program interruption occurs.

If the logic of your Fortran routines depends on continued execution when fixed-point overflow occurs, there are two solutions. The first is to disable the program interruption due to fixed-point overflow by changing the program mask. Use the CEE3SPM callable service as follows:

```

CHARACTER*80  DISABLE_FIXED_PT_OVERFLOW / 'NOF' /
INTEGER*4    SET_PGM_MASK
PARAMETER    (SET_PGM_MASK = 1)
EXTERNAL     CEE3SPM
:
:
CALL AFHCEEN (CEE3SPM, SET_PGM_MASK, DISABLE_FIXED_PT_OVERFLOW)

```

If you disable the interruption, routines written in languages other than Fortran might not operate correctly because some languages depend on the interruption to provide certain language semantics. If this is a problem in your application, use the push and pop features of the CEE3SPM callable service to disable the interruption for limited portions of your Fortran routines. Do this as follows:

```

CHARACTER*80  DISABLE_FIXED_PT_OVERFLOW / 'NOF' /
CHARACTER*80  BLANK_80 / ' ' /
INTEGER*4    PUSH_SET_PGM_MASK
PARAMETER    (PUSH_SET_PGM_MASK = 5)
INTEGER*4    POP_PGM_MASK
PARAMETER    (POP_PGM_MASK = 4)
EXTERNAL     CEE3SPM
:
:
CALL AFHCEEN (CEE3SPM, PUSH_SET_PGM_MASK,
1           DISABLE_FIXED_PT_OVERFLOW)
:
:
CALL AFHCEEN (CEE3SPM, POP_PGM_MASK, BLANK_80)

```

Between the two calls to CEE3SPM, no program interruption occurs due to fixed-point overflow. The second call to CEE3SPM restores the program mask to the setting it had before the first call.

Another way to handle fixed-point overflow is with a user-written condition handler that resumes execution after the condition is signaled for the fixed-point overflow exception. For a simple example providing the same results as VS FORTRAN, see Figure 6 on page 66. However, processing the program interruption in this way can degrade run-time performance.

DVCHK and OVERFL Subroutines

Language Environment does not provide the DVCHK and OVERFL subroutines, which in VS FORTRAN report the occurrence of the following program interruptions:

- Fixed-point divide exception
- Floating-point divide exception
- Fixed-point overflow exception
- Exponent-overflow exception

The condition handling facilities in Language Environment include services for writing condition handlers to detect and report various conditions, such as the divide and overflow exceptions. To learn how to provide the error handling that's similar to what is available with VS FORTRAN's DVCHK and OVERFL subroutines, read Chapter 10, "Handling Run-Time Errors" on page 48. There are several examples of user-written condition handlers beginning on page 63.

Assembler Language Routines That Find Program Arguments

An assembler language routine that searches backward through the save area chain to find the option string¹³ and returns all or part of the string to the caller will probably not work with Language Environment. This is because Language Environment inserts an additional save area in the save area chain just after the operating system's save area and just before the main program's save area. This additional save area interferes with the logic of an assembler language routine that extracts the general register 1 value from a save area that's a fixed number of save areas prior to the assembler language routine's save area. There are several ways to correct this problem:

- Change your Fortran routine to call the ARGSTR callable service which is described in *VS FORTRAN Version 2 Language and Library Reference*, to get the program arguments as shown in the following example:

```
CHARACTER*40  PROG_ARGS
INTEGER*4     ARGSTR_RC
:
CALL ARGSTR (PROG_ARGS, ARGSTR_RC)
:
```

If the PARM parameter in your EXEC statement is coded as follows:

```
PARM='NOOCSTATUS,MSGFILE(FT06F001)/ANNUAL_REPORT'
```

the variable PROG_ARGS is assigned the value ANNUAL_REPORT.

- Change the assembler language routine to call the ARGSTR callable service. Be sure to follow the Fortran conventions for argument lists with character arguments. These conventions are described in the section “Passing Character Arguments Using the Standard Linkage Convention” in *VS FORTRAN Version 2 Programming Guide for CMS and MVS*.
- Change the assembler language routine to bypass the additional save area in the save area chain.

Run-Time Initialization from Assembler Language

If your VS FORTRAN application has an assembler language routine rather than a Fortran main program to initialize the Fortran run-time environment, you might have to change the application to run it with Language Environment. Such an assembler language routine calls the VFEIN# callable service (or the equivalent VSCOM# or IBCOM# callable service) for initialization. Depending on the relationship of the assembler language routine to the rest of the application, there are different changes that you must make.

The following sections discuss different scenarios involving initialization using VFEIN#. In the figures, the term *operating system* means the invoker of the high-level language application. The invoker is usually MVS, but it can be any routine that isn't running under the control of Language Environment.

¹³ The *option string* is the information given in the PARM parameter of the EXEC statement in your JCL or as the parameter string on the TSO CALL or LOADGO command. For further information on the format of the option string with Language Environment, see “Coding the Option String” on page 36.

Assembler Language Routine Acting as a Main Program

Figure 12 shows an assembler language routine acting as a Fortran main program:

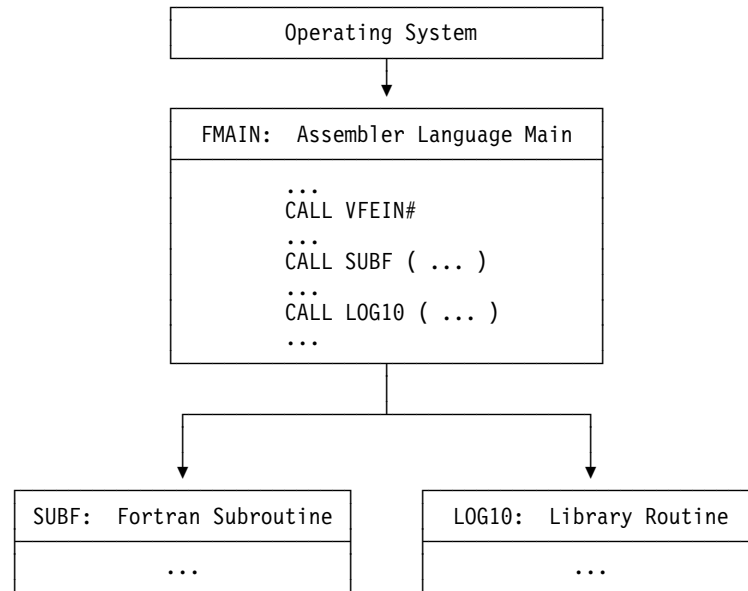


Figure 12. Assembler Language Routine Acting as a Main Program

The assembler language routine FMAIN calls the VFEIN# callable service for initialization of the Fortran run-time environment. This allows either Fortran subroutines or Fortran run-time library routines, such as SUBF or LOG10 in this example, to be called. In this case, VS FORTRAN treats FMAIN as a replacement for a Fortran main program. When the application terminates (because of a STOP statement, for example), control returns to the caller of FMAIN.

For an application structured like the one shown in Figure 12, no change is required in the assembler language routine to run the application with Language Environment. When FMAIN is called by the operating system or by some other program that's not running with Language Environment, the VFEIN# callable service initializes the environment as it does with VS FORTRAN. When the application terminates, control returns to the caller of FMAIN.

If you're writing a new assembler language main program, use the CEEENTRY macro to generate a main program prolog that conforms to the Language Environment's standard linkage conventions. In this case, Language Environment automatically initializes the run-time environment, including the Fortran-specific portion, and you don't have to use the VFEIN# callable service. For more information on the CEEENTRY and related macros, see *Language Environment for MVS & VM Programming Guide*. When you link-edit your assembler language routine, might have to include the Fortran signature CSECT as described in Chapter 6, "Declaring the Presence of Fortran Routines" on page 18.

Assembler Language Routine for an Active Environment

When the assembler language routine that calls VFEIN# isn't called directly by the operating system but rather by a routine written in a high-level language such as COBOL, the Language Environment behavior differs from the VS FORTRAN behavior. Consider the structure in Figure 13.

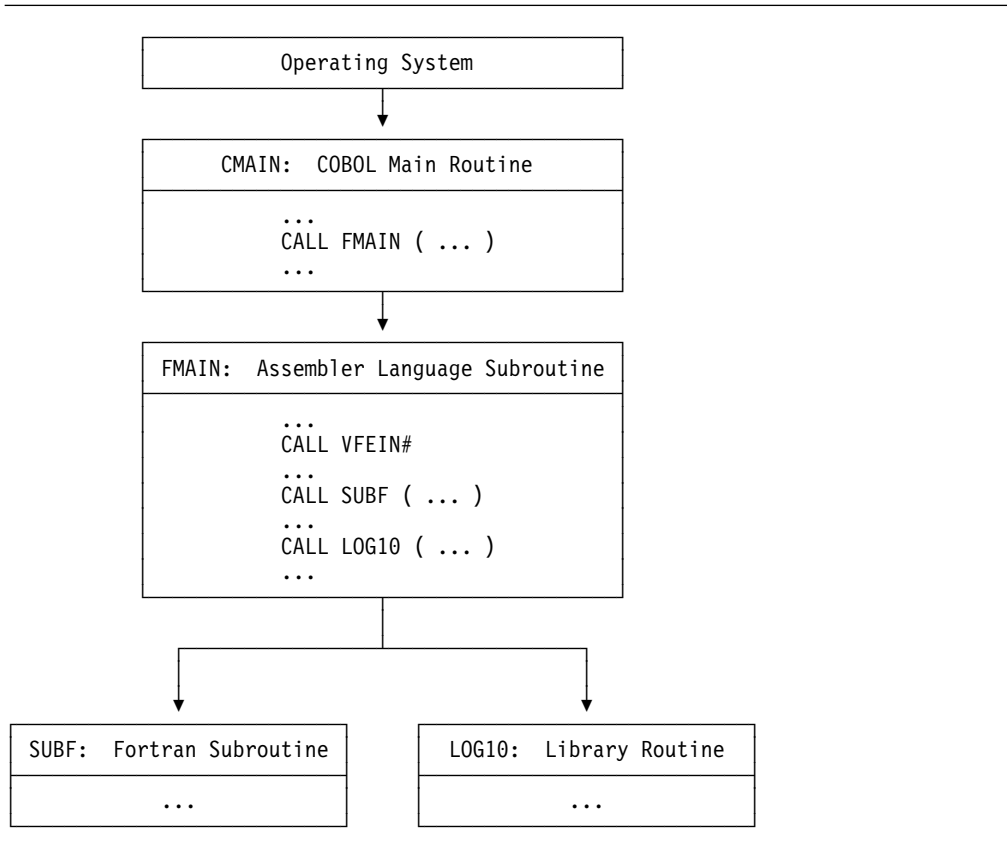


Figure 13. Assembler Language Routine Called from a Main Program

Because Language Environment provides a single run-time environment for all languages, there's only one main program in the application, and the COBOL routine CMAIN is treated as the main program. The assembler language routine FMAIN is a subroutine even though it calls VFEIN#. The effect of having CMAIN rather than FMAIN as the main routine is that when the application terminates, control returns to the caller of the COBOL main routine CMAIN. This is true even when a Fortran STOP statement is executed.

(With VS FORTRAN, the assembler language routine FMAIN is considered to be the main program from the Fortran point of view. If the application terminates within the Fortran portion, control returns to the caller of FMAIN, that is, to CMAIN.)

The call to VFEIN# in Figure 13 isn't needed because Language Environment initializes the entire run-time environment, including the Fortran-specific portion, when the main program CMAIN is entered. However, the call to VFEIN# is permitted and is ignored.

Assembler Language Routine Operating outside the Environment

Figure 14 shows a structure in which the assembler language routine that calls VFEIN# doesn't work properly as the main routine in some cases.

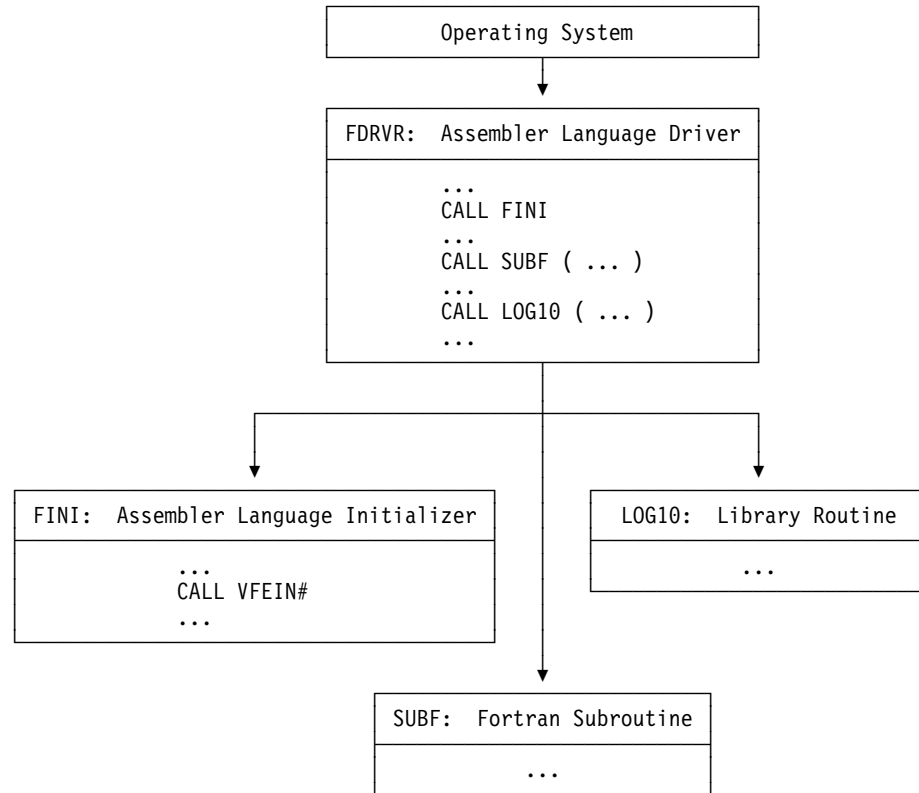


Figure 14. Assembler Language Routine Operating Outside the Environment

The assembler language driver routine FDRVR calls the assembler language routine FINI, which calls the VFEIN# callable service to initialize the Fortran run-time environment. FINI returns to FDRVR, which calls Fortran subroutines or Fortran run-time library routines, such as SUBF or LOG10. With VS FORTRAN, when the Fortran routines return directly to their callers and don't terminate the application, this approach works successfully. However, it doesn't work correctly if the application terminates from one of the Fortran routines; in this case, control returns to FDRVR, sometimes at an unexpected place.

With Language Environment, an application structured in this way does not work because FINI is treated as the main program. When FINI returns to its caller (FDRVR), the application terminates. When FDRVR calls the Fortran subroutine SUBF or the Fortran library routine LOG10, the results are unpredictable because the application has already terminated.

To enable this application to run with Language Environment, move the call to the VFEIN# callable service to FDRVR. Then the structure of the application is the same as that in Figure 12 on page 83, and FDRVR becomes the main routine.

If it's not practical to change this application in this way, consider using the Language Environment preinitialization callable services to construct an application in which calls are made from a routine, such as FDRVR, outside the environment to a

routine, such as SUBF, that's part of the environment. For information on the preinitialization callable services, see *Language Environment for MVS & VM Programming Guide*. For some limitations on the use of these services in calling Fortran routines, see "Preinitialization Services Cannot Refer to Fortran Routines" on page 9.

COBOL Routine Invoking an Assembler Language Routine for Initialization

If the assembler language driver (FDRVR) in Figure 14 on page 85 is replaced by a COBOL routine, as in Figure 15, the behavior is different.

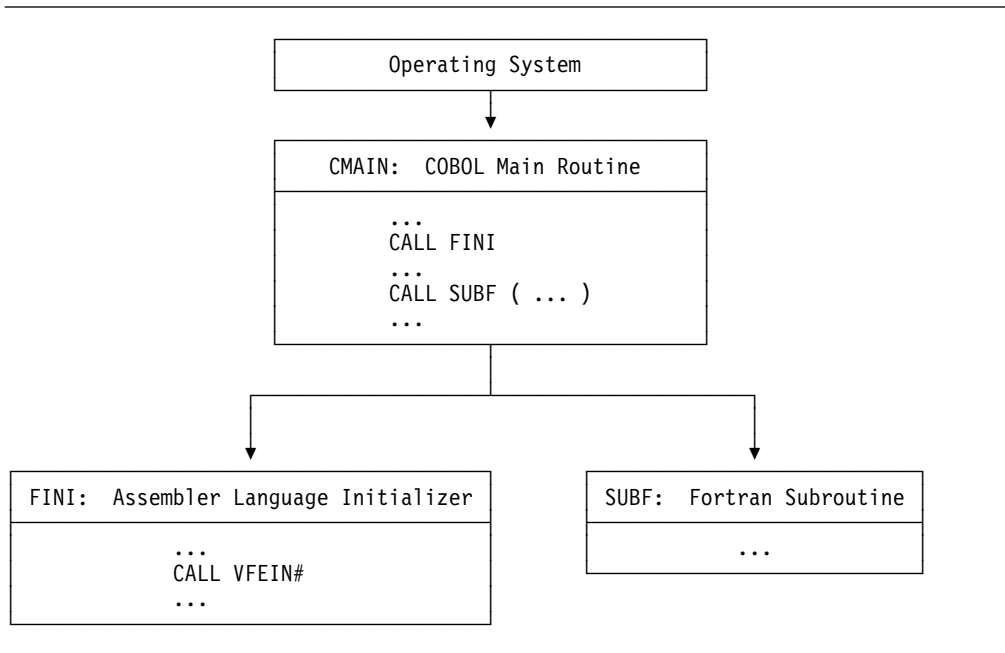


Figure 15. COBOL Routine Invoking an Assembler Language Routine for Initialization

The COBOL routine CMAIN calls the assembler language routine FINI, which calls the VFEIN# callable service to initialize the Fortran run-time environment. FINI returns to CMAIN, which calls Fortran subroutines such as SUBF. With VS FORTRAN, this is one method of incorporating Fortran routines into a COBOL application. When the Fortran routines return directly to their callers and don't terminate the application, this approach works successfully. However, it doesn't work correctly if the application terminates from one of the Fortran routines; in this case, control returns to CMAIN, sometimes at an unexpected place.

With Language Environment, the COBOL routine CMAIN is the main routine, and the initialization of the run-time environment, including the Fortran portion, occurs when CMAIN is entered. You can remove the call to VFEIN# because calling VFEIN# has no effect when the environment is already established.

When the application terminates, control returns to the caller of the COBOL main routine CMAIN. This is true even when a Fortran STOP statement is executed.

Part 4. Appendixes

| | |
|--|-----|
| Appendix A. Fortran Callable Services and Functions | 88 |
| AFHCEE—Invoke a Callable Service Passing the Feedback Code | 89 |
| AFHCEEN—Invoke a Callable Service Omitting the Feedback Code | 90 |
| QDFETCH Callable Service—Retrieve a Qualifying Datum of Any Type | 91 |
| QDLEN Function—Determine the Length of a Qualifying Datum | 92 |
| QDLOC Function—Obtain the Address of a Qualifying Datum | 92 |
| QDSTORE Callable Service—Update a Qualifying Datum | 93 |
| QDTYPE Function—Determine the Data Type of a Qualifying Datum | 94 |
| QDxxxx Functions—Retrieve a Qualifying Datum of a Specific Type | 95 |
| | |
| Appendix B. Qualifying Data for Language Environment Conditions | 97 |
| q_data Structure for Abends | 97 |
| q_data Structure for Arithmetic Program Interruptions | 98 |
| q_data Structure for Square-Root Exception | 101 |
| q_data Structure for Math and Bit-Manipulation Conditions | 102 |
| Format of q_data Descriptors | 105 |
| | |
| Appendix C. Message Number Mappings | 107 |
| Language Environment Conditions for VS FORTRAN Message Numbers | 107 |
| VS FORTRAN Message Numbers for Language Environment Conditions | 110 |
| | |
| Appendix D. VS FORTRAN Error Handling Behavior | 113 |

Appendix A. Fortran Callable Services and Functions

This appendix describes a set of Fortran-specific callable services and functions that give your Fortran routines access to most of the Language Environment features, such as its callable services.

You can use these callable services to call the Language Environment callable services from your Fortran routines:

| <u>Name</u> | <u>How Invoked</u> | <u>Purpose</u> |
|--------------------|---------------------------|--|
| AFHCEEF | CALL statement | Invoke a callable service passing the feedback code |
| AFHCEEN | CALL statement | Invoke a callable service omitting the feedback code |

You can use these callable services and functions to manipulate or get information about the qualifying data associated with a condition token:

| <u>Name</u> | <u>How Invoked</u> | <u>Purpose</u> |
|--------------------|---------------------------|---|
| QDFETCH | CALL statement | Retrieve a qualifying datum of any data type |
| QDLLEN | Function reference | Determine the length of a qualifying datum |
| QDLOC | Function reference | Obtain the address of a qualifying datum |
| QDSTORE | CALL statement | Update a qualifying datum |
| QDTYPE | Function reference | Determine the data type of a qualifying datum |

You can use these functions to retrieve the qualifying data associated with a condition token:

| <u>Name</u> | <u>How Invoked</u> | <u>Purpose</u> |
|--------------------|---------------------------|---|
| QDCH1 | Function reference | Retrieve a CHARACTER*1 qualifying datum |
| QDCH6 | Function reference | Retrieve a CHARACTER*6 qualifying datum |
| QDCH8 | Function reference | Retrieve a CHARACTER*8 qualifying datum |
| QDCH31 | Function reference | Retrieve a CHARACTER*31 qualifying datum |
| QDCH62 | Function reference | Retrieve a CHARACTER*62 qualifying datum |
| QDCH255 | Function reference | Retrieve a CHARACTER*255 qualifying datum |
| QDCX8 | Function reference | Retrieve a COMPLEX*8 qualifying datum |
| QDCX16 | Function reference | Retrieve a COMPLEX*16 qualifying datum |
| QDCX32 | Function reference | Retrieve a COMPLEX*32 qualifying datum |
| QDINT1 | Function reference | Retrieve a INTEGER*1 qualifying datum |
| QDINT2 | Function reference | Retrieve a INTEGER*2 qualifying datum |
| QDINT4 | Function reference | Retrieve a INTEGER*4 qualifying datum |
| QDINT8 | Function reference | Retrieve a INTEGER*8 qualifying datum |
| QDR4 | Function reference | Retrieve a REAL*4 qualifying datum |
| QDR8 | Function reference | Retrieve a REAL*8 qualifying datum |
| QDR16 | Function reference | Retrieve a REAL*16 qualifying datum |
| QDUS1 | Function reference | Retrieve a UNSIGNED*1 qualifying datum |

The preceding set of functions for retrieving qualifying data is described in “QDxxxxx Functions—Retrieve a Qualifying Datum of a Specific Type” on page 95.

AFHCEE—Invoke a Callable Service Passing the Feedback Code

AFHCEE calls a specified Language Environment callable service from a Fortran routine with the optional feedback code as the last argument.

Syntax

```
CALL AFHCEE(—service—, —arg—, —fc—)
```

service (input)

The name of the Language Environment callable service to be called. Declare the name *service* with an EXTERNAL statement in the Fortran routine as follows:

```
EXTERNAL service
```

arg (input or output as used by *service*)

One of the arguments for the *service* callable service. The number and type of these arguments are based on the requirements of *service* as described in *Language Environment for MVS & VM Programming Reference*.

fc (output)

A character variable or array element of length 12 that becomes defined with the feedback code from the *service* callable service. The feedback codes are listed in the description of the *service* callable service in *Language Environment for MVS & VM Programming Reference*.

Usage Notes

- The AFHCEE callable service establishes the Language Environment linkage conventions required by the Language Environment callable services and passes the arguments along to the specified callable service.
- To use the AFHCEE callable service from your Fortran routine, compile the routine with the VS FORTRAN Version 1 or VS FORTRAN Version 2 compiler using the LANGLVL(77) compile-time option.
- Do not use the AFHCEE callable service to CEE3SRP callable service.
- Do not use the AFHCEE callable service to invoke the Language Environment callable services for the mathematical routines because the mathematical routine names exceed Fortran's seven-character limit. (You shouldn't have to use the callable services for the mathematical routines because there are Fortran intrinsic functions for almost all of these routines.)
- Because a feedback code is passed to the *service* callable service, an error detected by the service is indicated by returning to the caller and providing a feedback code of other than CEE000.

AFHCEEN—Invoke a Callable Service Omitting the Feedback Code

AFHCEEN calls a specified Language Environment callable service from a Fortran routine without providing the optional feedback code as the last argument.

Syntax

```
CALL AFHCEEN ( service , arg )
```

service (input)

The name of the Language Environment callable service to be called. Declare the name *service* with an EXTERNAL statement in the Fortran routine as follows:

```
EXTERNAL service
```

arg (input or output as used by *service*)

is one of the arguments for the *service* callable service. The number and type of these arguments are based on the requirements of *service* as described in *Language Environment for MVS & VM Programming Reference*.

Usage Notes

- Do not provide the final feedback code argument for *service* in the call to AFHCEEN.
- To use the AFHCEEN callable service from your Fortran routine, compile the routine with the VS FORTRAN Version 1 or VS FORTRAN Version 2 compiler using the LANGLVL(77) compile-time option.
- Do not use the AFHCEEN callable service to CEE3SRP callable service.
- Do not use the AFHCEEN callable service to invoke the Language Environment callable services for the mathematical routines because none of the mathematical routines have the feedback code as the last argument, and because the mathematical routine names exceed Fortran's seven-character limit. (You shouldn't have to use the callable services for the mathematical routines because there are Fortran intrinsic functions for almost all of these routines.)
- The AFHCEEN callable service establishes the Language Environment linkage conventions required by the Language Environment callable services, rebuilds the argument list in a form that indicates that the final feedback code argument is omitted, and passes the arguments along to the specified callable service.
- Because the feedback code is omitted in the call to the *service* callable service, an error detected by the service is indicated by signaling a condition that reflects the error rather than by returning to the caller.

QDFETCH Callable Service—Retrieve a Qualifying Datum of Any Type

QDFETCH returns one of the qualifying data associated with a condition token.

Syntax

```
▶▶ CALL QDFETCH (—cond_rep—, —index—, —receiver—) ▶▶
```

cond_rep (input)

A character expression of length 12 whose value is the condition token with which the requested qualifying datum is associated.

index (input)

An integer variable or an integer array element of length 4. Its value is the ordinal number of the requested qualifying datum within the qualifying data associated with condition *cond_rep*. For example, if you want to retrieve the fifth qualifying datum associated with the condition *cond_rep*, provide a value of 5 for *index*.

receiver (output)

A variable, array element, or character substring which becomes defined with the value of the requested qualifying datum. The data type and length of *receiver* must be the same as the data type and length of qualifying datum number *index* defined for the condition *cond_rep*.

Usage Notes

- You can use the QDFETCH callable service only for conditions for which qualifying data are available.
- The value of *index* must be a positive number that doesn't exceed the number of elements of qualifying data associated with the condition *cond_rep*.
- The data type and length of any qualifying datum is a convention between the routine that signals a condition and the user of that qualifying datum. The QDFETCH callable service does not ensure that the type and length of *receiver* match the type and length of the qualifying datum for the condition *cond_rep*. If there is a mismatch, the results are unpredictable.
- The QDFETCH callable service is intended to be called by Fortran routines. However, you can call QDFETCH from assembler language routines if you follow the Fortran conventions for argument lists with character arguments. These conventions are described in the section “Passing Character Arguments Using the Standard Linkage Convention” in *VS FORTRAN Version 2 Programming Guide for CMS and MVS*.

QDLEN Function—Determine the Length of a Qualifying Datum

QDLEN returns the length of a specified qualifying datum associated with a condition token. The qualifying datum must have an associated `q_data` descriptor.

Syntax

```
►►...—QDLEN—(—cond_rep—,—index—)—...—◄◄
```

cond_rep (input)

A character expression of length 12 whose value is the condition token with which the qualifying datum is associated.

index (input)

An integer variable or an integer array element of length 4. Its value is the ordinal number of the qualifying datum within the qualifying data associated with condition `cond_rep`. For example, if you want to obtain the length of the fifth qualifying datum associated with the condition `cond_rep`, provide a value of 5 for `index`.

Usage Notes

- Provide the following INCLUDE line among the data declarations in your Fortran source program so the compiler will understand that the QDLEN function returns a value of integer type.

```
INCLUDE (AFHCQDSB)
```
- The QDLEN function returns the length of the qualifying datum as an integer of length 4.
- You can use the QDLEN function only for conditions for which qualifying data are available.
- The value of `index` must be a positive number that doesn't exceed the number of elements of qualifying data associated with the condition `cond_rep`.
- `index` must refer to a qualifying datum that has a `q_data` descriptor associated with it. Most qualifying data that don't have `q_data` descriptors are of fixed lengths; therefore, you don't have to determine their lengths at run time.

QDLOC Function—Obtain the Address of a Qualifying Datum

QDLOC returns the address of a specified qualifying datum associated with a condition token.

Syntax

```
►►ptr = —QDLOC—(—cond_rep—,—index—)—...—◄◄
```

ptr (output)

A pointer variable of length 4 that becomes defined with the address of the specified qualifying datum.

cond_rep (input)

A character expression of length 12 whose value is the condition token with which the qualifying datum is associated.

index (input)

An integer variable or an integer array element of length 4. Its value is the ordinal number of the qualifying datum within the qualifying data associated with condition *cond_rep*. For example, if you want to obtain the address of the fifth qualifying datum associated with the condition *cond_rep*, provide a value of 5 for *index*.

Usage Notes

- Provide the following INCLUDE line among the data declarations in your Fortran source program so the compiler will understand that the QDLOC function returns a value of integer type.

```
INCLUDE (AFHCQDSB)
```

- The QDLOC function returns the address of the qualifying datum as an integer of length 4. Use an assignment statement to assign the value returned by the QDLOC function to a pointer variable of length 4.
- You can use the QDLOC function only for conditions for which qualifying data are available.
- The value of *index* must be a positive number that doesn't exceed the number of elements of qualifying data associated with the condition *cond_rep*.

QDSTORE Callable Service—Update a Qualifying Datum

QDSTORE updates one of the qualifying data associated with a condition token using a value that you provide.

Syntax

```
CALL QDSTORE(—cond_rep—,—index—,—source—)
```

cond_rep (input)

A character expression of length 12 whose value is the condition token with which the qualifying datum is associated.

index (input)

An integer variable or an integer array element of length 4. Its value is the ordinal number of the target qualifying datum within the qualifying data associated with condition *cond_rep*. For example, if you want to update the fifth qualifying datum associated with the condition *cond_rep*, provide a value of 5 for *index*.

source (input)

A variable, array element, or character substring whose value replaces the existing value of the indicated qualifying datum. The data type and length of *source* must be the same as the data type and length of qualifying datum number *index* defined for the condition *cond_rep*.

Usage Notes

- You can use the QDSTORE callable service only for conditions for which qualifying data are available.
- The value of *index* must be a positive number that doesn't exceed the number of elements of qualifying data associated with the condition *cond_rep*.
- The data type and length of any qualifying datum is a convention between the routine that signals a condition and the user of that qualifying datum. The QDSTORE callable service does not ensure that the type and length of *source* match the type and length of the qualifying datum for the condition *cond_rep*. If there is a mismatch, the results are unpredictable.
- Do not use the QDSTORE callable service to update a qualifying datum unless that qualifying datum is specifically intended to be used as output.
- The QDSTORE callable service is intended to be called by Fortran routines. However, you can call QDSTORE from assembler language routines if you follow the Fortran conventions for argument lists with character arguments. These conventions are described in the section "Passing Character Arguments Using the Standard Linkage Convention" in *VS FORTRAN Version 2 Programming Guide for CMS and MVS*.

QDTYPE Function—Determine the Data Type of a Qualifying Datum

QDTYPE returns an integer value that represents the data type of a specified qualifying datum associated with a condition token. The qualifying datum must have an associated *q_data* descriptor.

Syntax

```
►►...—QDTYPE—(—cond_rep—,—index—)—...◄◄
```

cond_rep (input)

A character expression of length 12 whose value is the condition token with which the qualifying datum is associated.

index (input)

An integer variable or an integer array element of length 4. Its value is the ordinal number of the qualifying datum within the qualifying data associated with condition *cond_rep*. For example, if you want to obtain the data type of the fifth qualifying datum associated with the condition *cond_rep*, provide a value of 5 for *index*.

Usage Notes

- Provide the following INCLUDE line among the data declarations in your Fortran source program so the compiler will understand that the QDTYPE function returns a value of integer type.

```
INCLUDE (AFHCQDSB)
```
- The QDTYPE function returns the data type of the qualifying datum as an integer of length 4. The file AFHCQDSB contains declarations for the following named constants that represent the data types shown:

| <u>Named Constant</u> | <u>Data Type</u> |
|-----------------------|------------------|
| QDTYPE_CHAR | character |
| QDTYPE_CHARACTER | character |
| QDTYPE_COMPLEX | complex |
| QDTYPE_INT | integer |
| QDTYPE_INTEGER | integer |
| QDTYPE_REAL | real |
| QDTYPE_UNSIGNED | unsigned |
| QDTYPE_FAILURE | unknown |

- You can use the QDTYPE function only for conditions for which qualifying data are available.
- The value of *index* must be a positive number that doesn't exceed the number of elements of qualifying data associated with the condition *cond_rep*.
- *index* must refer to a qualifying datum that has a q_data descriptor associated with it. Most qualifying data that don't have q_data descriptors are of fixed data types; therefore, you don't have to determine their data types at run time.

QDXXXX Functions—Retrieve a Qualifying Datum of a Specific Type

Each of these retrieval functions returns a qualifying datum of a certain type, as indicated by the name of the function. The following functions are available in this set:

| <u>Name</u> | <u>Return Value Type</u> |
|-------------|--------------------------|
| QDCH1 | CHARACTER*1 |
| QDCH6 | CHARACTER*6 |
| QDCH8 | CHARACTER*8 |
| QDCH31 | CHARACTER*31 |
| QDCH62 | CHARACTER*62 |
| QDCH255 | CHARACTER*255 |
| QDCX8 | COMPLEX*8 |
| QDCX16 | COMPLEX*16 |
| QDCX32 | COMPLEX*32 |
| QDINT1 | INTEGER*1 |
| QDINT2 | INTEGER*2 |
| QDINT4 | INTEGER*4 |
| QDINT8 | INTEGER*8 |
| QDR4 | REAL*4 |
| QDR8 | REAL*8 |
| QDR16 | REAL*16 |
| QDUS1 | UNSIGNED*1 |

Syntax

►►...—QDxxxxx—(—*cond_rep*—,—*index*—)—...—————►►

cond_rep (input)

A character expression of length 12 whose value is the condition token with which the qualifying datum is associated.

index (input)

An integer variable or an integer array element of length 4. Its value is the ordinal number of the qualifying datum within the qualifying data associated with condition *cond_rep*. For example, if you want to retrieve the fifth qualifying datum associated with the condition *cond_rep*, provide a value of 5 for *index*.

Usage Notes

- Provide the following INCLUDE line among the data declarations in your Fortran source program so the compiler will understand the type of the data the retrieval functions return.

```
INCLUDE (AFHCQDSB)
```

- You can use these retrieval functions only for conditions for which qualifying data are available.
- The value of *index* must be a positive number that doesn't exceed the number of elements of qualifying data associated with the condition *cond_rep*.
- The data type and length of any qualifying datum is a convention between the routine that signals a condition and the user of that qualifying datum. These retrieval functions do not ensure that the type and length of the qualifying data they return match the type and length of the qualifying datum for the condition *cond_rep*. If there is a mismatch, the results are unpredictable.

Appendix B. Qualifying Data for Language Environment Conditions

This appendix shows the qualifying data associated with the conditions that are signaled for these situations:

- Abends
- Arithmetic program interruptions
- Square-root exception
- Math and bit manipulation conditions

For the qualifying data associated with the Fortran-specific condition, see the Fortran messages in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.

q_data Structure for Abends

If an abend occurs, Language Environment signals condition CEE35I (corresponding to message number 3250) and builds the q_data structure shown in Figure 16.

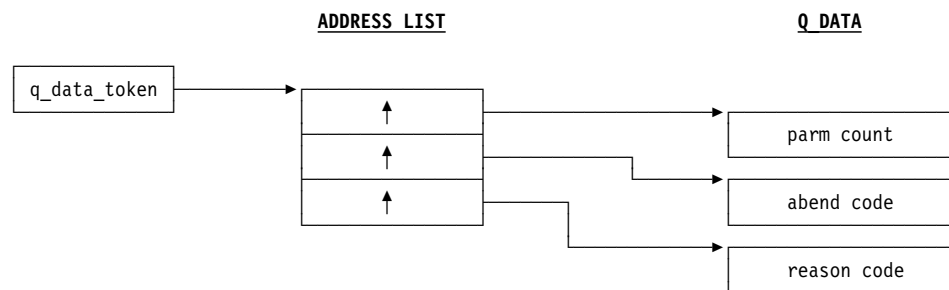


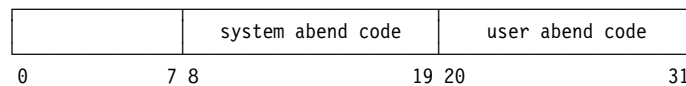
Figure 16. Structure of Abend Qualifying Data

parm count (input)

A fullword field containing the total number of parameters in the q_data structure, including *parm count*. In this case, the value of *parm count* is a fullword containing the integer 3.

abend code (input)

A 4-byte field containing the abend code in the following format:



system abend code

The 12-bit system completion (abend) code. If these bits are all zero, then the abend is a user abend.

user abend code

The 12-bit user completion (abend) code. The abend is a user abend when bits 8 through 19 are all zero.

reason code (input)

A 4-byte field containing the reason code accompanying the abend code. If a reason code is not available (as occurs, for example, in a CICS abend), *reason code* has the value zero.

Usage Notes

- You can use the CEEGQDT callable service to retrieve the q_data_token; see *Language Environment for MVS & VM Programming Reference* for further information.
- From a Fortran routine, you can retrieve the qualifying data using Fortran-specific callable services and functions, which are described in Appendix A on page 88.

q_data Structure for Arithmetic Program Interruptions

If one of the arithmetic program interruptions shown in Table 13 occurs, and the corresponding condition is signaled, Language Environment builds the q_data structure shown in Figure 17 on page 99.

Table 13. Arithmetic Program Interruptions and Corresponding Conditions

| Program Interruption ^{1, 2} | Program Interruption Code | Condition | Message Number |
|--------------------------------------|---------------------------|-----------|----------------|
| Fixed-point overflow exception | 08 | CEE348 | 3208 |
| Fixed-point divide exception | 09 | CEE349 | 3209 |
| Exponent-overflow exception | 0C | CEE34C | 3212 |
| Exponent-underflow exception | 0D | CEE34D | 3213 |
| Floating-point divide exception | 0F | CEE34F | 3215 |
| Unnormalized-operand exception | 1E | CEE34U | 3230 |

Notes:

1. The square root exception is also an arithmetic program interruption, but is treated like the condition from the square root mathematical routine.
2. An arithmetic program interruption that occurs on a vector instruction is presented to a user-written condition handler in the same form as though it had occurred on a scalar instruction. A single vector instruction could cause multiple, possibly different, program interruptions to occur, but each interruption is presented individually.

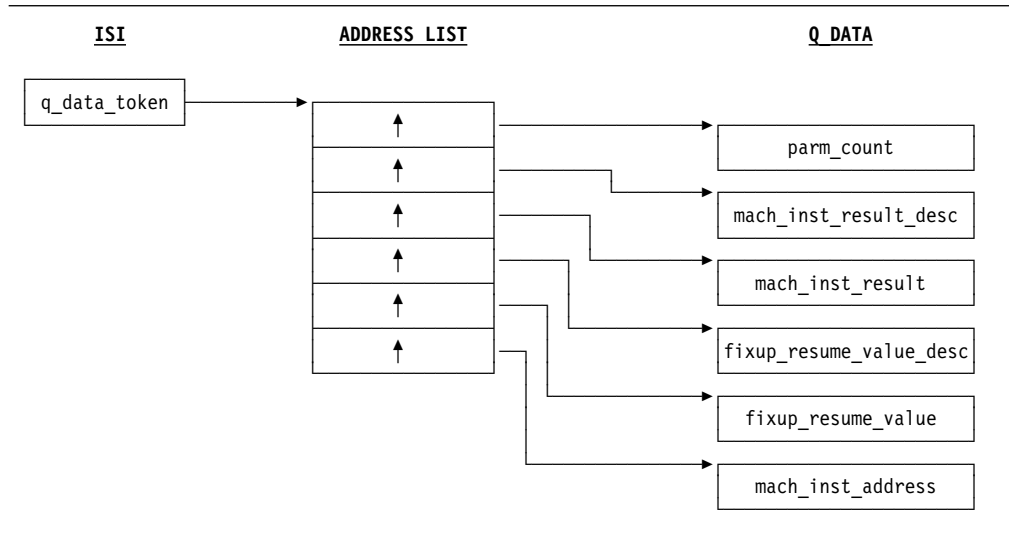


Figure 17. *q_data* Structure for Arithmetic Program Interruption Conditions. Language Environment builds this data structure for the conditions of exponent overflow, exponent underflow, floating-point divide, fixed-point overflow, fixed-point divide, and unnormalized-operand exceptions.

The following information is provided by the *q_data* structure shown in Figure 17:

q_data_token (input)

The 4-byte address of the address list. This value is returned by the CEEGQDT callable service.

parm_count (input)

A 4-byte binary integer containing the value 6, which is the total number of *q_data* fields in the *q_data* structure, including *parm_count*.

mach_inst_result_desc (input)

The *q_data* descriptor for *mach_inst_result*. (See “Format of *q_data* Descriptors” on page 105 for more information on *q_data* descriptors.)

mach_inst_result (input)

The value left in the machine register (general register, floating-point register, or element of a vector register) by the failing machine instruction. Based on the program interruption, *mach_inst_result* has one of the following lengths and types (as reflected in the *q_data* descriptor field *mach_inst_result_desc*):

Program Interruption

Length and Type

| | |
|--|--|
| Fixed-point overflow exception | 4- or 8-byte binary integer |
| Fixed-point divide exception | 8-byte binary integer |
| Exponent-overflow exception | 4-, 8-, or 16-byte floating-point number |
| Exponent-underflow exception | 4-, 8-, or 16-byte floating-point number |
| Floating-point divide exception | 4-, 8-, or 16-byte floating-point number |
| Unnormalized-operand exception ¹⁴ | 4- or 8-byte floating-point number |

This is also the result value with which execution is resumed when the user condition handler requests the resume action (result code 10).

¹⁴ The unnormalized-operand exception occurs only on vector instructions.

fixup_resume_value_desc (input)

The q_data descriptor for *fixup_resume_value*.

fixup_resume_value (input/output)

The fix-up value which, for the exceptions other than the unnormalized-operand exception, is the result value with which execution is resumed when the user condition handler requests the fix-up and resume action (result code 60 with a condition token of CEE0CF). *fixup_resume_value* initially has one of the following values:

- For an exponent-underflow exception, the value 0
- For an unnormalized-operand exception, the value 0
- For one of the other program interruptions, the same value as in *mach_inst_result*

Based on the program interruption, *fixup_resume_value* has the following lengths and types (as reflected in the q_data descriptor field *fixup_resume_value_desc*):

| Program Interruption | Length and Type |
|--|---|
| Fixed-point overflow exception | 4- or 8-byte binary integer |
| Fixed-point divide exception | 8-byte binary integer or two 4-byte binary integers (remainder, quotient) |
| Exponent-overflow exception | 4-, 8-, or 16-byte floating-point number |
| Exponent-underflow exception | 4-, 8-, or 16-byte floating-point number |
| Floating-point divide exception | 4-, 8-, or 16-byte floating-point number |
| Unnormalized-operand exception ¹⁴ | 4- or 8-byte floating-point number |

mach_inst_address (input)

The address of the machine instruction causing the program interruption.

Usage Notes

- You can use the CEEGQDT callable service to retrieve the q_data_token; see *Language Environment for MVS & VM Programming Reference* for further information.
- From a Fortran routine, you can retrieve the qualifying data using Fortran-specific callable services and functions, which are described in Appendix A on page 88.
- Using the q_data structure, a user condition handler can resume either with:
 - The resume action (result code 10) using the value in *mach_inst_result*. The effect is the same as though execution had continued without any change to the register contents left by the machine instruction.
 - The fix-up and resume action (result code 60 with a condition token of CEE0CF) for exceptions other than unnormalized-operand. This allows any value to be placed in the result register that the machine instruction used.
- You can use the CEE3SPM callable service to set or reset the exponent-underflow mask bit in the program mask; the bit controls whether a program interruption occurs when exponent-underflow occurs, as follows:

- When the bit is on, the program interruption occurs and condition CEE34D is signaled.
- When the bit is off, no program interruption occurs; therefore no condition is signaled.

See *Language Environment for MVS & VM Programming Reference* for further information on the CEE3SPM callable service.

q_data Structure for Square-Root Exception

If a square-root exception¹⁵ occurs and the corresponding condition as shown in Table 14 is signaled, Language Environment builds the q_data structure shown in Figure 18 on page 102.

Table 14. Square-Root Exception and Corresponding Condition

| Program Interruption | Program Interruption Code | Condition | Message Number |
|-----------------------------|----------------------------------|------------------|-----------------------|
| Square-root exception | 1D | CEE1UQ | 2010 |

For a square-root exception, Language Environment signals the same condition (CEE1UQ) as it does when one of the square root routines detects a negative argument. For this exception, a user-written condition handler can request the same resume and fix-up and resume actions that it can request when the condition is signaled by one of the square root routines.

¹⁵ A *square-root exception* is the program interruption that occurs when a square root instruction is executed with a negative argument.

q_data Structure for Math and Bit-Manipulation Conditions

For conditions that occur in the mathematical or bit manipulation routines, the Language Environment condition manager creates `q_data` that user condition handlers can use to handle the condition. The `q_data` structure is shown in Figure 18, and is the same for all entry points of the mathematical and bit manipulation routines.

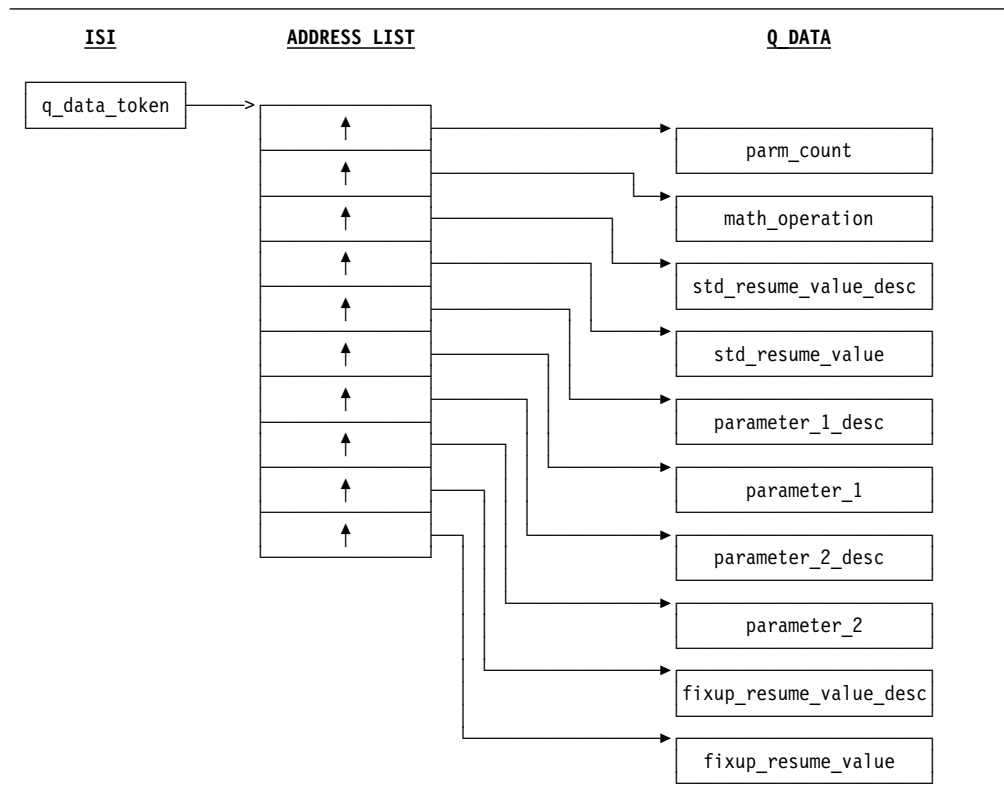


Figure 18. `q_data` Structure for Math and Bit Manipulation Routines

The following information is provided by the `q_data` structure shown in Figure 18:

q_data_token (input)

The 4-byte address of the address list. This value is returned by the CEEGQDT callable service.

parm_count (input)

A 4-byte binary integer containing the value 10, which is the total number of `q_data` fields in the `q_data` structure, including `parm_count`.

math_operation (input)

An 8-byte field containing an abbreviation for the mathematical or bit manipulation operation for which the condition occurred. The field is left-justified and padded with blanks. (See Table 15 on page 103 for a list of the abbreviations that can appear.)

std_resume_value_desc (input)

The `q_data` descriptor for `std_resume_value`.

std_resume_value (input)

A default value used as the result of the mathematical or bit manipulation function when the user condition handler requests the resume action (result

code 10). The length and type of this field are dependent on *math_operation* and are reflected in the *q_data* descriptor *std_resume_value_desc*.

parameter_1_desc (input)

The *q_data* descriptor for *parameter_1*.

parameter_1 (input/output)

The value of the first parameter provided to the mathematical or bit manipulation routine. The length and type of this field are dependent on *math_operation* and are reflected in the *q_data* descriptor *parameter_1_desc*.

This is the value of the first parameter that is used as input to the routine when the user condition handler requests a resume with new input value (result code 60 with a new condition token of CEE0CE).

parameter_2_desc (input)

The *q_data* descriptor for *parameter_2* if the mathematical or bit manipulation routine has two input parameters. (If the routine has only one parameter, the *q_data* structure has an address slot for this field, but the address is not meaningful and the field must not be referenced.)

parameter_2 (input/output)

The value of the second parameter provided to the mathematical or bit manipulation routine if the routine has two input parameters. (If the routine has only one parameter, the *q_data* structure has an address slot for this field, but the address is not meaningful and the field must not be referenced.) The length and type of the field are dependent on *math_operation* and are reflected in the *q_data* descriptor *parameter_2_desc*.

This is the value of the second parameter that is used as input to the routine when the user condition handler requests a resume with new input value (result code 60 with a new condition token of CEE0CE).

fixup_resume_value_desc (input)

The *q_data* descriptor for *fixup_resume_value*. (See “Format of *q_data* Descriptors” on page 105 for more information on *q_data* descriptors.)

fixup_resume_value (output)

The value to be used as the result of the mathematical or bit manipulation function when the user condition handler requests a resume with new output value (result code 60 with a new condition token of CEE0CF). The length and type of this field are dependent on *math_operation* and are reflected in the *q_data* descriptor *fixup_resume_value_desc*.

Table 15 (Page 1 of 2). Abbreviations of Math Operations in *q_data* Structures. Column two shows the abbreviations that can occur in field *math_operation* for the math operations shown in column one.

| Mathematical Operation | Abbreviation |
|--|--------------|
| Logarithm Base <i>e</i> | LN |
| Logarithm Base 10 | LOG |
| Logarithm Base 2 | LOG2 |
| Exponential (base <i>e</i>) | E**Y |
| Exponentiation (<i>x</i> raised to the power <i>y</i>) | X**Y |
| Arcsine | ARCSIN |

Table 15 (Page 2 of 2). Abbreviations of Math Operations in q_data Structures. Column two shows the abbreviations that can occur in field math_operation for the math operations shown in column one.

| Mathematical Operation | Abbreviation |
|---------------------------|--------------|
| Arccosine | ARCCOS |
| Arctangent | ARCTAN |
| Arctangent2 | ARCTAN2 |
| Sine | SIN |
| Cosine | COS |
| Tangent | TAN |
| Cotangent | COTAN |
| Hyperbolic Sine | SINH |
| Hyperbolic Cosine | COSH |
| Hyperbolic Tangent | TANH |
| Hyperbolic Arctangent | ARCTANH |
| Square Root | SQRT |
| Error Function | ERF |
| Error Function Complement | ERFC |
| Gamma Function | GAMMA |
| Log Gamma Function | LOGGAMMA |
| Absolute Value Function | ABS |
| Modular Arithmetic | MOD |
| Truncation | TRUNC |
| Imaginary Part of Complex | IPART |
| Conjugate of Complex | CPART |
| Nearest Whole Number | NWN |
| Nearest Integer | NINT |
| Positive Difference | POSDIFF |
| Transfer of Sign | XFERSIGN |
| Floating Complex Multiply | CPLXMULT |
| Floating Complex Divide | CPLXDIVD |
| Bit Shift | ISHFT |
| Bit Clear | IBCLR |
| Bit Set | IBSET |
| Bit Test | BTEST |

Usage Notes

- You can use the CEEGQDT callable service to retrieve the q_data_token; see *Language Environment for MVS & VM Programming Reference* for details.
- From a Fortran routine, you can retrieve the qualifying data using Fortran-specific callable services and functions, which are described in Appendix A on page 88.

- A user condition handler can request one of three different actions to continue the execution of a failing mathematical or bit manipulation routine:
 - The resume action (result code 10). The value in *std_resume_value* (either the default value provided to the user condition handler or a modified value provided by the user condition handler) becomes the final result value for the routine.
 - The resume with new input value action (result code 60 with a new condition token of CEE0CE). The values to be used as parameters for invoking the routine again are provided by the user condition handler in *parameter_1* and, if applicable, in *parameter_2*.
 - The resume with new output value action (result code 60 with a new condition token of CEE0CF). The *fixup_resume_value* value provided by the user condition handler becomes the final result value for the routine.

Format of q_data Descriptors

q_data descriptors contain additional information you need to fix up the parameter or result fields of the math q_data structures, the result field of the program interruption q_data structures, or fields for any conditions whose q_data structures contain q_data descriptors. The descriptors contain information about the length and data type of these fields. The format of the q_data descriptor is illustrated in Figure 19.

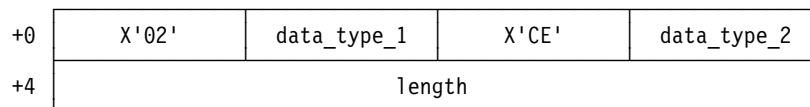


Figure 19. Format of a q_data Descriptor

The following information is provided by the q_data descriptor shown in Figure 19:

data_type_1

An integer value of length 1 that, along with *data_type_2*, indicates the data type. See Table 16 on page 106 for the values and their corresponding data types.

data_type_2

An integer value of length 1 that, along with *data_type_1*, indicates the data type. See Table 16 on page 106 for the values and their corresponding data types.

length

An integer value of length 4 that represents the length of the data.

For each type code that can occur in a q_data descriptor, Table 16 on page 106 shows the corresponding data type.

Table 16. *q_data* Descriptor Data Types

| data_type_1 Type Code | data_type_2 Type Code | Description |
|--|--|--|
| 2 | 0 | String of single-byte characters with no length prefix or ending delimiter |
| 1 | 13 | Signed binary integer whose length is 1, 2, 4, or 8 bytes |
| 1 | 14 | Floating-point number whose length is 4, 8, or 16 bytes |
| 1 | 15 | Complex number whose length is 8, 16, or 32 bytes |
| 1 | 18 | Unsigned binary integer whose length is 1 byte |

Appendix C. Message Number Mappings

The message numbers for the conditions detected during execution with Language Environment are not the same as the message numbers for the corresponding errors in VS FORTRAN Version 2. In some cases, an error represented by a single message in VS FORTRAN Version 2 is separated into several different Language Environment conditions so that the conditions more precisely describe the errors. In other cases, the errors represented by several different VS FORTRAN Version 2 messages are combined into a single Language Environment condition because they are the same condition.

The following list is in order of the VS FORTRAN Version 2 message numbers and shows the corresponding Language Environment conditions. The list starting on page 110 is in order of the Language Environment conditions and shows the corresponding VS FORTRAN Version 2 message numbers. In both lists, messages that don't have a counterpart in the other product are not listed.

Language Environment Conditions for VS FORTRAN Message Numbers

| VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition |
|--------------------------------------|---|--------------------------------------|---|--------------------------------------|---|
| AFB001I | FOR0340 | AFB105I | FOR1900 | AFB122I | FOR1921 |
| AFB002I | FOR0341 | AFB106I | FOR1340 | AFB123I | FOR1026 |
| AFB090I | FOR1276 | AFB107I | FOR1381 | AFB123I | FOR1103 |
| AFB090I | FOR1411 | AFB108I | FOR1396 | AFB124I | FOR1104 |
| AFB091I | FOR1554 | AFB108I | FOR1397 | AFB125I | FOR1106 |
| AFB091I | FOR1563 | AFB108I | FOR1398 | AFB126I | FOR1107 |
| AFB092I | FOR1250 | AFB108I | FOR1399 | AFB127I | FOR1113 |
| AFB093I | FOR1251 | AFB108I | FOR1400 | AFB128I | FOR1113 |
| AFB094I | FOR1252 | AFB108I | FOR1401 | AFB129I | FOR1112 |
| AFB095I | FOR1023 | AFB108I | FOR1402 | AFB130I | FOR1557 |
| AFB095I | FOR1024 | AFB108I | FOR1403 | AFB130I | FOR1558 |
| AFB095I | FOR1025 | AFB108I | FOR1404 | AFB131I | FOR1417 |
| AFB096I | FOR0120 | AFB108I | FOR1920 | AFB132I | FOR1418 |
| AFB096I | FOR0121 | AFB109I | FOR1341 | AFB133I | FOR1419 |
| AFB096I | FOR0122 | AFB110I | FOR1917 | AFB134I | FOR1414 |
| AFB099I | FOR2044 | AFB111I | FOR1557 | AFB135I | FOR1114 |
| AFB100I | FOR1416 | AFB111I | FOR1558 | AFB136I | FOR1382 |
| AFB101I | FOR1022 | AFB111I | FOR1559 | AFB137I | FOR1406 |
| AFB102I | FOR1395 | AFB111I | FOR1560 | AFB138I | FOR1407 |
| AFB103I | FOR1502 | AFB111I | FOR1561 | AFB139I | FOR1100 |
| AFB103I | FOR1503 | AFB111I | FOR1900 | AFB140I | FOR1102 |
| AFB103I | FOR1504 | AFB112I | CEE3206 | AFB142I | FOR0404 |
| AFB103I | FOR1505 | AFB114I | FOR1411 | AFB143I | FOR0100 |
| AFB103I | FOR1506 | AFB115I | FOR1501 | AFB143I | FOR0101 |
| AFB103I | FOR1507 | AFB116I | CEE3230 | AFB144I | FOR0102 |
| AFB103I | FOR1508 | AFB117I | CEE3230 | AFB145I | FOR0405 |
| AFB103I | FOR1509 | AFB118I | CEE2020 | AFB146I | FOR0406 |
| AFB103I | FOR1550 | AFB119I | CEE2020 | AFB147I | FOR0407 |
| AFB103I | FOR1551 | AFB120I | FOR1413 | AFB148I | FOR0414 |
| AFB104I | FOR1387 | AFB121I | FOR1405 | AFB148I | FOR0415 |

| VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition |
|--------------------------------------|---|--------------------------------------|---|--------------------------------------|---|
| AFB149I | FOR0409 | AFB192I | FOR1277 | AFB221I | FOR1225 |
| AFB152I | FOR1922 | AFB193I | FOR0500 | AFB222I | FOR1221 |
| AFB153I | FOR0402 | AFB194I | FOR1276 | AFB223I | FOR1222 |
| AFB154I | FOR0104 | AFB195I | FOR0501 | AFB224I | FOR1223 |
| AFB154I | FOR0105 | AFB196I | FOR0502 | AFB224I | FOR1224 |
| AFB154I | FOR0106 | AFB197I | FOR0503 | AFB225I | FOR1004 |
| AFB155I | FOR1408 | AFB199I | FOR0504 | AFB225I | FOR1010 |
| AFB155I | FOR1409 | AFB200I | FOR1008 | AFB225I | FOR1011 |
| AFB156I | FOR0301 | AFB200I | FOR1910 | AFB226I | FOR1005 |
| AFB157I | FOR0303 | AFB201I | FOR1001 | AFB227I | FOR1210 |
| AFB157I | FOR2040 | AFB201I | FOR1002 | AFB228I | FOR1272 |
| AFB157I | FOR2041 | AFB203I | FOR1923 | AFB230I | FOR0401 |
| AFB157I | FOR0304 | AFB204I | FOR1001 | AFB231I | FOR1020 |
| AFB158I | FOR0300 | AFB204I | FOR1002 | AFB231I | FOR1071 |
| AFB158I | FOR0302 | AFB205I | FOR1279 | AFB232I | FOR1070 |
| AFB159I | CEE2028 | AFB206I | FOR1000 | AFB233I | FOR1389 |
| AFB160I | FOR1182 | AFB207I | CEE3212 | AFB234I | FOR1925 |
| AFB161I | FOR1278 | AFB208I | CEE3213 | AFB235I | FOR1071 |
| AFB162I | FOR1331 | AFB209I | CEE3209 | AFB236I | FOR1072 |
| AFB163I | FOR1020 | AFB209I | CEE3215 | AFB238I | FOR1006 |
| AFB164I | FOR1001 | AFB210I | CEE32xx | AFB238I | FOR1007 |
| AFB164I | FOR1002 | AFB211I | FOR1181 | AFB239I | FOR1271 |
| AFB165I | FOR1411 | AFB211I | FOR1183 | AFB240I | CEE3250 |
| AFB165I | FOR1412 | AFB212I | FOR1001 | AFB241I | CEE2003 |
| AFB166I | FOR1023 | AFB212I | FOR1002 | AFB242I | CEE2004 |
| AFB167I | FOR1557 | AFB213I | FOR1001 | AFB243I | CEE2004 |
| AFB167I | FOR1558 | AFB213I | FOR1002 | AFB244I | CEE2006 |
| AFB168I | FOR1330 | AFB214I | FOR1201 | AFB245I | CEE2006 |
| AFB169I | FOR1510 | AFB214I | FOR1280 | AFB246I | CEE2008 |
| AFB169I | FOR1512 | AFB215I | FOR1003 | AFB247I | CEE2008 |
| AFB169I | FOR1555 | AFB217I | FOR1008 | AFB248I | CEE2004 |
| AFB169I | FOR1561 | AFB217I | FOR1910 | AFB249I | CEE2006 |
| AFB171I | FOR1360 | AFB218I | FOR1022 | AFB249I | CEE2020 |
| AFB172I | FOR1415 | AFB218I | FOR1027 | AFB250I | CEE2021 |
| AFB173I | FOR1200 | AFB218I | FOR1500 | AFB251I | CEE2010 |
| AFB174I | FOR1180 | AFB218I | FOR1554 | AFB252I | CEE2011 |
| AFB175I | FOR1915 | AFB218I | FOR1559 | AFB253I | CEE2012 |
| AFB180I | FOR1342 | AFB218I | FOR1560 | AFB254I | CEE2017 |
| AFB180I | FOR1383 | AFB219I | FOR1388 | AFB255I | CEE2014 |
| AFB181I | FOR1380 | AFB219I | FOR1511 | AFB256I | CEE2016 |
| AFB182I | FOR1384 | AFB219I | FOR1512 | AFB257I | CEE2016 |
| AFB183I | FOR1385 | AFB219I | FOR1554 | AFB258I | CEE2017 |
| AFB184I | FOR1386 | AFB219I | FOR1555 | AFB259I | CEE2002 |
| AFB185I | FOR1410 | AFB219I | FOR1560 | AFB260I | CEE2007 |
| AFB186I | FOR1361 | AFB219I | FOR1565 | AFB261I | CEE2010 |
| AFB187I | FOR0416 | AFB219I | FOR1570 | AFB262I | CEE2011 |
| AFB187I | FOR0417 | AFB219I | FOR1571 | AFB263I | CEE2012 |
| AFB188I | FOR0601 | AFB219I | FOR1926 | AFB264I | CEE2017 |
| AFB189I | FOR0602 | AFB220I | FOR1916 | AFB265I | CEE2014 |
| AFB191I | FOR0603 | AFB221I | FOR1220 | AFB266I | CEE2016 |

| VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition |
|--------------------------------------|---|--------------------------------------|---|
| AFB267I | CEE2016 | AFB920I | FOR2041 |
| AFB268I | CEE2017 | AFB920I | FOR2042 |
| AFB269I | CEE2002 | AFB920I | FOR2043 |
| AFB270I | CEE2008 | AFB921I | FOR2060 |
| AFB271I | CEE2009 | AFB922I | FOR2056 |
| AFB272I | CEE2015 | AFB922I | FOR2057 |
| AFB273I | CEE2018 | AFB922I | FOR2058 |
| AFB274I | CEE2019 | AFB922I | FOR2069 |
| AFB275I | CEE2013 | AFB922I | FOR2070 |
| AFB276I | CEE2009 | AFB922I | FOR2071 |
| AFB277I | CEE2013 | AFB922I | FOR2072 |
| AFB278I | CEE2018 | AFB922I | FOR2073 |
| AFB279I | CEE2019 | AFB923I | FOR1924 |
| AFB280I | CEE2013 | AFB924I | FOR2062 |
| AFB281I | CEE2009 | AFB925I | FOR2063 |
| AFB282I | CEE2015 | AFB927I | FOR2064 |
| AFB283I | CEE2018 | AFB928I | FOR2065 |
| AFB284I | CEE2019 | AFB930I | FOR2067 |
| AFB285I | CEE2013 | AFB931I | FOR2068 |
| AFB286I | FOR1273 | AFB932I | FOR0410 |
| AFB286I | FOR1274 | AFB934I | FOR2101 |
| AFB287I | FOR1270 | AFB935I | FOR2102 |
| AFB288I | FOR1275 | AFB936I | FOR1552 |
| AFB289I | CEE2010 | AFB936I | FOR1553 |
| AFB290I | CEE2005 | AFB936I | FOR1556 |
| AFB291I | CEE2005 | AFB952I | FOR1281 |
| AFB292I | CEE2011 | | |
| AFB293I | CEE2012 | | |
| AFB294I | CEE2017 | | |
| AFB295I | CEE2014 | | |
| AFB296I | CEE2016 | | |
| AFB297I | CEE2016 | | |
| AFB298I | CEE2017 | | |
| AFB299I | CEE2002 | | |
| AFB300I | CEE2005 | | |
| AFB301I | CEE2005 | | |
| AFB904I | FOR1927 | | |
| AFB905I | FOR0400 | | |
| AFB916I | FOR2122 | | |
| AFB917I | FOR2121 | | |
| AFB918I | FOR2030 | | |
| AFB918I | FOR2031 | | |
| AFB918I | FOR2032 | | |
| AFB919I | FOR2000 | | |
| AFB919I | FOR2001 | | |
| AFB919I | FOR2003 | | |
| AFB919I | FOR2004 | | |
| AFB919I | FOR2005 | | |
| AFB920I | FOR0303 | | |
| AFB920I | FOR2040 | | |

VS FORTRAN Message Numbers for Language Environment Conditions

| Language Environment Condition | VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number |
|--------------------------------|-----------------------|--------------------------------|-----------------------|--------------------------------|-----------------------|
| CEE2002 | AFB259I | CEE2017 | AFB254I | FOR0402 | AFB153I |
| CEE2002 | AFB269I | CEE2017 | AFB258I | FOR0404 | AFB142I |
| CEE2002 | AFB299I | CEE2017 | AFB264I | FOR0405 | AFB145I |
| CEE2003 | AFB241I | CEE2017 | AFB268I | FOR0406 | AFB146I |
| CEE2004 | AFB242I | CEE2017 | AFB294I | FOR0407 | AFB147I |
| CEE2004 | AFB243I | CEE2017 | AFB298I | FOR0409 | AFB149I |
| CEE2004 | AFB248I | CEE2018 | AFB273I | FOR0410 | AFB932I |
| CEE2005 | AFB290I | CEE2018 | AFB278I | FOR0414 | AFB148I |
| CEE2005 | AFB291I | CEE2018 | AFB283I | FOR0415 | AFB148I |
| CEE2005 | AFB300I | CEE2019 | AFB274I | FOR0416 | AFB187I |
| CEE2005 | AFB301I | CEE2019 | AFB279I | FOR0417 | AFB187I |
| CEE2006 | AFB244I | CEE2019 | AFB284I | FOR0500 | AFB193I |
| CEE2006 | AFB245I | CEE2020 | AFB118I | FOR0501 | AFB195I |
| CEE2006 | AFB249I | CEE2020 | AFB119I | FOR0502 | AFB196I |
| CEE2007 | AFB260I | CEE2020 | AFB249I | FOR0503 | AFB197I |
| CEE2008 | AFB246I | CEE2021 | AFB250I | FOR0504 | AFB199I |
| CEE2008 | AFB247I | CEE2028 | AFB159I | FOR0601 | AFB188I |
| CEE2008 | AFB270I | CEE32xx | AFB210I | FOR0602 | AFB189I |
| CEE2009 | AFB271I | CEE3206 | AFB112I | FOR0603 | AFB191I |
| CEE2009 | AFB276I | CEE3209 | AFB209I | FOR1000 | AFB206I |
| CEE2009 | AFB281I | CEE3212 | AFB207I | FOR1001 | AFB164I |
| CEE2010 | AFB251I | CEE3213 | AFB208I | FOR1001 | AFB201I |
| CEE2010 | AFB261I | CEE3215 | AFB209I | FOR1001 | AFB204I |
| CEE2010 | AFB289I | CEE3230 | AFB116I | FOR1001 | AFB212I |
| CEE2011 | AFB252I | CEE3230 | AFB117I | FOR1001 | AFB213I |
| CEE2011 | AFB262I | CEE3250 | AFB240I | FOR1002 | AFB164I |
| CEE2011 | AFB292I | FOR0100 | AFB143I | FOR1002 | AFB201I |
| CEE2012 | AFB253I | FOR0101 | AFB143I | FOR1002 | AFB204I |
| CEE2012 | AFB263I | FOR0102 | AFB144I | FOR1002 | AFB212I |
| CEE2012 | AFB293I | FOR0104 | AFB154I | FOR1002 | AFB213I |
| CEE2013 | AFB275I | FOR0105 | AFB154I | FOR1003 | AFB215I |
| CEE2013 | AFB277I | FOR0106 | AFB154I | FOR1004 | AFB225I |
| CEE2013 | AFB280I | FOR0120 | AFB096I | FOR1005 | AFB226I |
| CEE2013 | AFB285I | FOR0121 | AFB096I | FOR1006 | AFB238I |
| CEE2014 | AFB255I | FOR0122 | AFB096I | FOR1007 | AFB238I |
| CEE2014 | AFB265I | FOR0300 | AFB158I | FOR1008 | AFB200I |
| CEE2014 | AFB295I | FOR0301 | AFB156I | FOR1008 | AFB217I |
| CEE2015 | AFB272I | FOR0302 | AFB158I | FOR1010 | AFB225I |
| CEE2015 | AFB282I | FOR0303 | AFB920I | FOR1011 | AFB225I |
| CEE2016 | AFB256I | FOR0303 | AFB157I | FOR1020 | AFB163I |
| CEE2016 | AFB257I | FOR0304 | AFB157I | FOR1020 | AFB231I |
| CEE2016 | AFB266I | FOR0340 | AFB001I | FOR1022 | AFB101I |
| CEE2016 | AFB267I | FOR0341 | AFB002I | FOR1022 | AFB218I |
| CEE2016 | AFB296I | FOR0400 | AFB905I | FOR1023 | AFB095I |
| CEE2016 | AFB297I | FOR0401 | AFB230I | FOR1023 | AFB166I |

| Language Environment Condition | VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number | Language Environment Condition | VSF V2 Message Number |
|---------------------------------------|------------------------------|---------------------------------------|------------------------------|---------------------------------------|------------------------------|
| FOR1024 | AFB095I | FOR1341 | AFB109I | FOR1509 | AFB103I |
| FOR1025 | AFB095I | FOR1342 | AFB180I | FOR1510 | AFB169I |
| FOR1026 | AFB123I | FOR1360 | AFB171I | FOR1511 | AFB219I |
| FOR1027 | AFB218I | FOR1361 | AFB186I | FOR1512 | AFB169I |
| FOR1070 | AFB232I | FOR1380 | AFB181I | FOR1512 | AFB219I |
| FOR1071 | AFB235I | FOR1381 | AFB107I | FOR1550 | AFB103I |
| FOR1071 | AFB231I | FOR1382 | AFB136I | FOR1551 | AFB103I |
| FOR1072 | AFB236I | FOR1383 | AFB180I | FOR1552 | AFB936I |
| FOR1100 | AFB139I | FOR1384 | AFB182I | FOR1553 | AFB936I |
| FOR1102 | AFB140I | FOR1385 | AFB183I | FOR1554 | AFB091I |
| FOR1103 | AFB123I | FOR1386 | AFB184I | FOR1554 | AFB218I |
| FOR1104 | AFB124I | FOR1387 | AFB104I | FOR1554 | AFB219I |
| FOR1106 | AFB125I | FOR1388 | AFB219I | FOR1555 | AFB169I |
| FOR1107 | AFB126I | FOR1389 | AFB233I | FOR1555 | AFB219I |
| FOR1112 | AFB129I | FOR1395 | AFB102I | FOR1556 | AFB936I |
| FOR1113 | AFB127I | FOR1396 | AFB108I | FOR1557 | AFB111I |
| FOR1113 | AFB128I | FOR1397 | AFB108I | FOR1557 | AFB130I |
| FOR1114 | AFB135I | FOR1398 | AFB108I | FOR1557 | AFB167I |
| FOR1180 | AFB174I | FOR1399 | AFB108I | FOR1558 | AFB111I |
| FOR1181 | AFB211I | FOR1400 | AFB108I | FOR1558 | AFB130I |
| FOR1182 | AFB160I | FOR1401 | AFB108I | FOR1558 | AFB167I |
| FOR1183 | AFB211I | FOR1402 | AFB108I | FOR1559 | AFB111I |
| FOR1200 | AFB173I | FOR1403 | AFB108I | FOR1559 | AFB218I |
| FOR1201 | AFB214I | FOR1404 | AFB108I | FOR1560 | AFB111I |
| FOR1210 | AFB227I | FOR1405 | AFB121I | FOR1560 | AFB218I |
| FOR1220 | AFB221I | FOR1406 | AFB137I | FOR1560 | AFB219I |
| FOR1221 | AFB222I | FOR1407 | AFB138I | FOR1561 | AFB111I |
| FOR1222 | AFB223I | FOR1408 | AFB155I | FOR1561 | AFB169I |
| FOR1223 | AFB224I | FOR1409 | AFB155I | FOR1563 | AFB091I |
| FOR1224 | AFB224I | FOR1410 | AFB185I | FOR1565 | AFB219I |
| FOR1225 | AFB221I | FOR1411 | AFB090I | FOR1570 | AFB219I |
| FOR1250 | AFB092I | FOR1411 | AFB114I | FOR1571 | AFB219I |
| FOR1251 | AFB093I | FOR1411 | AFB165I | FOR1900 | AFB105I |
| FOR1252 | AFB094I | FOR1412 | AFB165I | FOR1900 | AFB111I |
| FOR1270 | AFB287I | FOR1413 | AFB120I | FOR1910 | AFB200I |
| FOR1271 | AFB239I | FOR1414 | AFB134I | FOR1910 | AFB217I |
| FOR1272 | AFB228I | FOR1415 | AFB172I | FOR1915 | AFB175I |
| FOR1273 | AFB286I | FOR1416 | AFB100I | FOR1916 | AFB220I |
| FOR1274 | AFB286I | FOR1417 | AFB131I | FOR1917 | AFB110I |
| FOR1275 | AFB288I | FOR1418 | AFB132I | FOR1920 | AFB108I |
| FOR1276 | AFB090I | FOR1419 | AFB133I | FOR1921 | AFB122I |
| FOR1276 | AFB194I | FOR1500 | AFB218I | FOR1922 | AFB152I |
| FOR1277 | AFB192I | FOR1501 | AFB115I | FOR1923 | AFB203I |
| FOR1278 | AFB161I | FOR1502 | AFB103I | FOR1924 | AFB923I |
| FOR1279 | AFB205I | FOR1503 | AFB103I | FOR1925 | AFB234I |
| FOR1280 | AFB214I | FOR1504 | AFB103I | FOR1926 | AFB219I |
| FOR1281 | AFB952I | FOR1505 | AFB103I | FOR1927 | AFB904I |
| FOR1330 | AFB168I | FOR1506 | AFB103I | FOR2000 | AFB919I |
| FOR1331 | AFB162I | FOR1507 | AFB103I | FOR2001 | AFB919I |
| FOR1340 | AFB106I | FOR1508 | AFB103I | FOR2003 | AFB919I |

| Language Environment Condition | VSF V2 Message Number |
|---------------------------------------|------------------------------|
| FOR2004 | AFB919I |
| FOR2005 | AFB919I |
| FOR2030 | AFB918I |
| FOR2031 | AFB918I |
| FOR2032 | AFB918I |
| FOR2040 | AFB920I |
| FOR2040 | AFB157I |
| FOR2041 | AFB920I |
| FOR2041 | AFB157I |
| FOR2042 | AFB920I |
| FOR2043 | AFB920I |
| FOR2044 | AFB099I |
| FOR2056 | AFB922I |
| FOR2057 | AFB922I |
| FOR2058 | AFB922I |
| FOR2060 | AFB921I |
| FOR2062 | AFB924I |
| FOR2063 | AFB925I |
| FOR2064 | AFB927I |
| FOR2065 | AFB928I |
| FOR2067 | AFB930I |
| FOR2068 | AFB931I |
| FOR2069 | AFB922I |
| FOR2070 | AFB922I |
| FOR2071 | AFB922I |
| FOR2072 | AFB922I |
| FOR2073 | AFB922I |
| FOR2101 | AFB934I |
| FOR2102 | AFB935I |
| FOR2121 | AFB917I |
| FOR2122 | AFB916I |

Appendix D. VS FORTRAN Error Handling Behavior

Table 17 on pages 114 through 127 shows how the extended error handling facility of VS FORTRAN handles the errors that have corrective actions. Use this information to determine what corrective actions (that is, fix-up actions) are needed for your Fortran routines as you migrate them to Language Environment. The table includes the following information for each error listed:

- The VS FORTRAN Version 2 error number
- The equivalent Language Environment condition
- The Fortran statement or function that detected the error
- The invalid condition stated in terms of the input arguments
- The value supplied by IBM for the number of times the error can occur before the application is terminated
- The value supplied by IBM for the number of times that the error message is to be printed
- The corrective action applied by the extended error handling facility in VS FORTRAN

In the table:

| | |
|---------------|---|
| <i>result</i> | The function value or other result value that the extended error handling facility sets for the intrinsic function, machine instruction, or data conversion for which the condition was signaled |
| <i>max</i> | The largest floating-point number that can be represented. This is approximately 7.2×10^{75} or, to be precise: Short (4-byte) format $16^{63} \times (1 - 16^{-6})$ Long (8-byte) format $16^{63} \times (1 - 16^{-14})$ Extended (16-byte) format $16^{63} \times (1 - 16^{-28})$ |
| <i>e</i> | The base of natural logarithms |
| π | The ratio of the circumference of a circle to its diameter |
| <i>i</i> | The square root of -1 |
| ALLOW | The column that shows the number of times the error is allowed to occur before the application is terminated |
| PRINT | The column that shows the number of times the error message is to be printed |
| ∞ | unlimited |
| RF | The <i>resume with Fortran-specific correction</i> action. See page 54. |

The table lists only the VS FORTRAN errors with a corrective action that manipulates the input data or the final result of a computation in some way. There are a number of errors, primarily I/O errors, for which execution continues without any corrective action. These errors are not listed.

Table 17 (Page 1 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|----------|-------|--|--|---|
| CEE3208 | - | ∞ | 5 | Fixed-point arithmetic instruction | Fixed-point overflow exception | $result =$ the rightmost 32 bits of the mathematical result treated as a 32-bit signed integer that differs from the true result by $\pm 2^{32}$. |
| CEE2020 | 118 | 10 | 5 | $base ** exp$ where $base$ and exp are of REAL*4 type | $base < 0$ and ($0 < exp \leq 16^6 - 1$ and exp isn't a whole number) or $ exp > 16^6 - 1$) | $result = base ^{exp}$ |
| CEE2020 | 119 | 10 | 5 | $base ** exp$ where $base$ and exp are of REAL*8 type | $base < 0$ and ($0 < exp \leq 16^{14} - 1$ and exp isn't a whole number) or $ exp > 16^{14} - 1$) | $result = base ^{exp}$ |
| CEE2028 | 159 | 10 | 5 | ISHIFT(x, y) LSHIFT(x, y) RSHIFT(x, y) IBSET(x, y) IBCLR(x, y) BTEST(x, y) where x and y are of integer type | Let b be the number of bits in x For ISHIFT: $ y > b$ For LSHIFT and RSHIFT: $y > b$ or $y < 0$ For IBSET, IBCLR, and BTEST: $y > b - 1$ or $y < 0$ | For ISHIFT(x, y), $result = 0$ For LSHIFT(x, y), $result = 0$ For RSHIFT(x, y), $result = 0$ For IBSET(x, y), $result = x$ For IBCLR(x, y), $result = x$ For BTEST(x, y), $result = .FALSE.$ |
| CEE2040 | 159 | 10 | 5 | ISHFTC(x, y, z) where x, y , and z are of integer type | Let b be the number of bits in x $z > b$ or $z < 0$ | $result = 0$ |

Table 17 (Page 2 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|---------------------------------|-------|-------|---|---|---|
| CEE2041 | 159 | 10 | 5 | ISHFTC(x, y, z) where x, y, and z are of integer type | Let b be the number of bits in x $ y > z$ or $ y > b$ | $result = 0$ |
| CEE2042 | 159 | 10 | 5 | IBITS(x, y, z) where x, y, and z are of integer type | Let b be the number of bits in x $y + z > b$ | $result = 0$ |
| CEE2043 | 159 | 10 | 5 | IBITS(x, y, z) where x, y, and z are of integer type | For IBITS: $y < 0$ or $z < 0$ | $result = 0$ |
| FOR1001 | 164 201 204 212 213 | 10 | 5 | WRITE REWRITE | The length of the record to be written was greater than the maximum record length allowed for the file. | The permissible resume action RF for condition FOR1001. See <i>Language Environment for MVS & VM Debugging Guide and Run-Time Messages</i> . |
| FOR1002 | 164 201 204 212 213 | 10 | 5 | READ | The I/O list requested data from beyond the end of the record being read. | The permissible resume action RF for condition FOR1002. See <i>Language Environment for MVS & VM Debugging Guide and Run-Time Messages</i> . |

Table 17 (Page 3 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|----------|-------|---|--|---|
| FOR1000 | 206 | 10 | 5 | Integer conversion for an input list item in a formatted READ statement | The number being converted, n , had a larger magnitude than could be represented in $result$, which is of UNSIGNED*1, INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8 type. | For result of type UNSIGNED*1 If $n > 255$, $result = 255$ If $n < 0$, $result = 0$ For data type INTEGER*1 Let mi be 127 INTEGER*2 32767 INTEGER*4 2147483647 INTEGER*8 9223372036854775807 |
| CEE3212 | 207 | ∞ | 5 | Floating-point instruction | Exponent-overflow exception | If $n > mi$, $result = mi$ If $n < -(mi+1)$, $result = -(mi+1)$ If the mathematical result > 0 , $result = max$ If the mathematical result < 0 , $result = -max$ |
| CEE3213 | 208 | ∞ | 5 | Floating-point instruction | Exponent-underflow exception | $result = 0.0$ |
| CEE3209 | 209 | ∞ | 5 | Fixed-point divide instruction | Fixed-point divide exception | Let n be the dividend in $n \div 0$. $result = n$ |
| CEE3215 | 209 | ∞ | 5 | Floating-point divide instruction | Floating-point divide exception | Let n be the dividend in $n \div 0$. If $n > 0$, $result = max$ If $n = 0$, $result = 0.0$ If $n < 0$, $result = -max$ |

Table 17 (Page 4 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|----------|-------|--|---|---|
| FOR1280 | 214 | 10 | 5 | Asynchronous I/O statement | The record format specified was other than variable spanned. | The permissible resume action RF for condition FOR1280. See <i>Language Environment for MVS & VM Debugging Guide and Run-Time Messages</i> . |
| FOR1003 | 215 | ∞ | 5 | Numeric data conversion for an input list item in a formatted READ statement | In the data being converted, there was a nonnumeric character where there should have been a numeric character. | The character 0 replaces the invalid character and the conversion is done with that revised input character string. |
| FOR1004 | 225 | 10 | 5 | Hexadecimal data conversion for an input list item in a formatted READ statement | The data being converted had a character other than 0 through 9 or A through F. | The character 0 replaces the invalid character and the conversion is done with that revised input character string. |
| FOR1010 | 225 | 10 | 5 | Binary data conversion for an input list item in a formatted READ statement | The data being converted had a character other than 0 or 1. | The character 0 replaces the invalid character and the conversion is done with that revised input character string. |
| FOR1011 | 225 | 10 | 5 | Octal data conversion for an input list item in a formatted READ statement | The data being converted had a character other than 0 through 7. | The character 0 replaces the invalid character and the conversion is done with that revised input character string. |
| FOR1005 | 226 | 10 | 5 | Real or complex data conversion for an input list item in a formatted READ statement | The number being converted, n , had a magnitude that was larger or smaller than could be represented in <i>result</i> , which either is of REAL type or is the real or imaginary part of an item of COMPLEX type. | If the magnitude of n was too small, $result = 0.0$ If the magnitude of n was too large, if $n > 0$ $result = max$ if $n < 0$ $result = -max$ |
| CEE2003 | 241 | 10 | 5 | $base ** exp$ where <i>base</i> and <i>exp</i> are of integer type | $base = 0$ and $exp \leq 0$ | $result = 0$ |

Table 17 (Page 5 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|--|--|---|
| CEE2004 | 242 | 10 | 5 | <i>base ** exp</i> where <i>base</i> is of REAL*4 type and <i>exp</i> is of integer type | <i>base</i> = 0 and <i>exp</i> ≤ 0 | If <i>base</i> = 0.0 and <i>exp</i> < 0, <i>result</i> = <i>max</i> If <i>base</i> = 0.0 and <i>exp</i> = 0, <i>result</i> = 1 |
| CEE2004 | 243 | 10 | 5 | <i>base ** exp</i> where <i>base</i> is of REAL*8 type and <i>exp</i> is of integer type | <i>base</i> = 0 and <i>exp</i> ≤ 0 | If <i>base</i> = 0.0 and <i>exp</i> < 0, <i>result</i> = <i>max</i> If <i>base</i> = 0.0 and <i>exp</i> = 0, <i>result</i> = 1 |
| CEE2006 | 244 | 10 | 5 | <i>base ** exp</i> where <i>base</i> and <i>exp</i> are of REAL*4 type | <i>base</i> = 0 and <i>exp</i> ≤ 0 | If <i>base</i> = 0.0 and <i>exp</i> < 0.0, <i>result</i> = <i>max</i> If <i>base</i> = 0.0 and <i>exp</i> = 0, <i>result</i> = 1 |
| CEE2006 | 245 | 10 | 5 | <i>base ** exp</i> where <i>base</i> and <i>exp</i> are of REAL*8 type | <i>base</i> = 0 and <i>exp</i> ≤ 0 | If <i>base</i> = 0.0 and <i>exp</i> < 0.0, <i>result</i> = <i>max</i> If <i>base</i> = 0.0 and <i>exp</i> = 0, <i>result</i> = 1 |
| CEE2008 | 246 | 10 | 5 | <i>base ** exp</i> where <i>base</i> is of COMPLEX*8 type and <i>exp</i> is either of integer type or of COMPLEX*8 type | For <i>exp</i> of integer type, let <i>rexp</i> be <i>exp</i> For <i>exp</i> of complex type, let <i>rexp</i> be <i>exp</i> 's real part <i>base</i> = 0 and <i>rexp</i> ≤ 0 | If <i>base</i> = (0.0,0.0) and <i>rexp</i> < 0, <i>result</i> = <i>max</i> + 0 <i>i</i> If <i>base</i> = (0.0,0.0) and <i>rexp</i> = 0, <i>result</i> = 1 + 0 <i>i</i> |
| CEE2008 | 247 | 10 | 5 | <i>base ** exp</i> where <i>base</i> is of COMPLEX*16 type and <i>exp</i> is either of integer type or of COMPLEX*16 type | For <i>exp</i> of integer type, let <i>rexp</i> be <i>exp</i> For <i>exp</i> of complex type, let <i>rexp</i> be <i>exp</i> 's real part <i>base</i> = 0 and <i>rexp</i> ≤ 0 | If <i>base</i> = (0.0,0.0) and <i>rexp</i> < 0, <i>result</i> = <i>max</i> + 0 <i>i</i> If <i>base</i> = (0.0,0.0) and <i>rexp</i> = 0, <i>result</i> = 1 + 0 <i>i</i> |

Table 17 (Page 6 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|--|--|--|
| CEE2004 | 248 | 10 | 5 | <i>base ** exp</i> where <i>base</i> is of REAL*16 type and <i>exp</i> is of integer type | <i>base</i> = 0 and <i>exp</i> ≤ 0 | If <i>base</i> = 0.0 and <i>exp</i> < 0, <i>result</i> = <i>max</i> If <i>base</i> = 0.0 and <i>exp</i> = 0, <i>result</i> = 1 |
| CEE2006 | 249 | 10 | 5 | <i>base ** exp</i> where <i>base</i> and <i>exp</i> are of REAL*16 type | <i>base</i> = 0 and <i>exp</i> ≤ 0 | If <i>base</i> = 0.0 and <i>exp</i> < 0, <i>result</i> = <i>max</i> If <i>base</i> = 0.0 and <i>exp</i> = 0, <i>result</i> = 1 |
| CEE2020 | 249 | 10 | 5 | <i>base ** exp</i> where <i>base</i> and <i>exp</i> are of REAL*16 type | <i>base</i> < 0 and ((0 < <i>exp</i> ≤ 16 ²⁸ -1 and <i>exp</i> isn't a whole number) or <i>exp</i> > 16 ²⁸ -1) | <i>result</i> = <i>base</i> ^{<i>exp</i>} |
| CEE2021 | 250 | 10 | 5 | <i>base ** exp</i> where <i>base</i> and <i>exp</i> are of REAL*16 type | <i>exp</i> * LOG ₂ (<i>base</i>) ≥ 252 | <i>result</i> = <i>max</i> |
| CEE2010 | 251 | 10 | 5 | SQRT(<i>x</i>) where <i>x</i> is of REAL*4 type | <i>x</i> < 0 | <i>result</i> = <i>x</i> ^½ |
| CEE2011 | 252 | 10 | 5 | EXP(<i>x</i>) where <i>x</i> is of REAL*4 type | <i>x</i> > 174.673 | <i>result</i> = <i>max</i> |
| CEE2012 | 253 | 10 | 5 | LOG(<i>x</i>) ALOG(<i>x</i>) LOG10(<i>x</i>) ALOG10(<i>x</i>) where <i>x</i> is of REAL*4 type | <i>x</i> ≤ 0 | If <i>x</i> = 0, <i>result</i> = - <i>max</i> If <i>x</i> < 0, For ALOG(<i>x</i>), <i>result</i> = log <i>x</i> For ALOG10(<i>x</i>), <i>result</i> = log ₁₀ <i>x</i> |

Table 17 (Page 7 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|---|---|
| CEE2017 | 254 | 10 | 5 | SIN(x) COS(x) where x is of REAL*4 type | $ x \geq 2^{18}\pi$ | $result = 2.0^{1/2}.0$ |
| CEE2014 | 255 | 10 | 5 | ATAN2(x, y) where x and y are of REAL*4 type | $x = 0.0$ and $y = 0.0$ | $result = 0$ |
| CEE2016 | 256 | 10 | 5 | SINH(x) COSH(x) where x is of REAL*4 type | $ x > 175.366$ | For SINH(x), If $x > 0.0$, $result = max$ If $x < 0.0$, $result = -max$ for COSH(x), $result = max$ |
| CEE2016 | 257 | 10 | 5 | ASIN(x) ACOS(x) where x is of REAL*4 type | $ x > 1$ | For ACOS(x), If $x > 1.0$, $result = 0$ If $x < -1.0$, $result = \pi$ For ASIN(x), If $x > 1.0$, $result = \pi/2$ If $x < -1.0$, $result = -\pi/2.0$ |
| CEE2017 | 258 | 10 | 5 | TAN(x) COTAN(x) where x is of REAL*4 type | $ x \geq 2^{18}\pi$ | $result = 1$ |
| CEE2002 | 259 | 10 | 5 | TAN(x) COTAN(x) where x is of REAL*4 type | For TAN(x), x too close to $\pm\pi/2, \pm3\pi/2, \dots$ For COTAN(x), x too close to $0, \pm\pi, \pm2\pi, \dots$ | $result = max$ |
| CEE2007 | 260 | 10 | 5 | 2**exp where exp is of REAL*16 type | $exp \geq 2^{252}$ | $result = max$ |

Table 17 (Page 8 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|------------------------|--|
| CEE2010 | 261 | 10 | 5 | SQRT(x) DSQRT(x) where x is of REAL*8 type | x < 0 | result = x ^½ |
| CEE2011 | 262 | 10 | 5 | EXP(x) DEXP(x) where x is of REAL*8 type | x > 174.673 | result = max |
| CEE2012 | 263 | 10 | 5 | LOG(x) DLOG(x) LOG10(x) DLOG10(x) where x is of REAL*8 type | x ≤ 0 | If x = 0, result = -max If x < 0, For DLOG(x), result = log x For DLOG10(x), result = log ₁₀ x |
| CEE2017 | 264 | 10 | 5 | SIN(x) DSIN(x) COS(x) DCOS(x) where x is of REAL*8 type | x ≥ 2 ⁵⁰ π | result = 2.0 ^½ /2.0 |
| CEE2014 | 265 | 10 | 5 | ATAN2(x, y) DATAN2(x, y) where x and y are of REAL*8 type | x = 0.0 and y = 0.0 | result = 0 |
| CEE2016 | 266 | 10 | 5 | SINH(x) DSINH(x) COSH(x) DCOSH(x) where x is of REAL*8 type | x > 175.366 | For DSINH(x), If x > 0.0, result = max If x < 0.0, result = -max for DCOSH(x), result = max |

Table 17 (Page 9 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|--|---|
| CEE2016 | 267 | 10 | 5 | ASIN(x) DASIN(x) ACOS(x) DACOS(x) where x is of REAL*8 type | $ x > 1$ | For DACOS(x), If $x > 1.0$, $result = 0$ If $x < -1.0$, $result = \pi$ For DASIN(x), If $x > 1.0$, $result = \pi/2$ If $x < -1.0$, $result = -\pi/2$ |
| CEE2017 | 268 | 10 | 5 | TAN(x) DTAN(x) COTAN(x) DCOTAN(x) where x is of REAL*8 type | $ x \geq 2^{50}\pi$ | $result = 1$ |
| CEE2002 | 269 | 10 | 5 | TAN(x) DTAN(x) COTAN(x) DCOTAN(x) where x is of REAL*8 type | For DTAN(x), x too close to $\pm\pi/2$, $\pm3\pi/2$, ... For DCOTAN(x), x too close to 0, $\pm\pi$, $\pm2\pi$, ... | $result = max$ |
| CEE2008 | 270 | 10 | 5 | base ** exp where base is of COMPLEX*32 type and exp is either of integer type or of COMPLEX*32 type | For exp of integer type, let $rexp$ be exp For exp of complex type, let $rexp$ be exp's real part base = 0 and $rexp \leq 0$ | If base = (0.0,0.0) and $rexp < 0$, $result = max + 0i$ If base = (0.0,0.0) and $rexp = 0$, $result = 1 + 0i$ |
| CEE2009 | 271 | 10 | 5 | EXP(c) CEXP(c) where c is of COMPLEX*8 type | Let c be $x + y*i$ $x > 174.673$ | $result = max*(COS(x) + SIN(x)*i)$ |

Table 17 (Page 10 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|--|--|
| CEE2015 | 272 | 10 | 5 | EXP(c) CEXP(c) where c is of COMPLEX*8 type | Let c be $x + y^*i$ $ y \geq 2^{1.8}\pi$ | $result = e^x + 0i$ |
| CEE2018 | 273 | 10 | 5 | LOG(c) CLOG(c) where c is of COMPLEX*8 type | Let c be $x + y^*i$ $x = 0.0$ and $y = 0.0$ | $result = -max + 0i$ |
| CEE2019 | 274 | 10 | 5 | SIN(c) CSIN(c) COS(c) CCOS(c) where c is of COMPLEX*8 type | Let c be $x + y^*i$ $ x \geq 2^{1.8}\pi$ | For CCOS(c), $result = COSH(y) + 0i$ For CSIN(c), $result = 0 + SINH(y)^*i$ |
| CEE2013 | 275 | 10 | 5 | SIN(c) CSIN(c) COS(c) CCOS(c) where c is of COMPLEX*8 type | Let c be $x + y^*i$ $ y > 174.673$ | If $y > 0$ For CSIN(c), $result = max/2^*(SIN(x) + COS(x)^*i)$ For CCOS(c), $result = max/2^*(COS(x) - SIN(x)^*i)$ If $y < 0$ For CSIN(c), $result = max/2^*(SIN(x) - COS(x)^*i)$ For CCOS(c), $result = max/2^*(COS(x) + SIN(x)^*i)$ |
| CEE2009 | 276 | 10 | 5 | EXP(c) CQEXP(c) where c is of COMPLEX*32 type | Let c be $x + y^*i$ $x > 174.673$ | $result = max^*(COS(x) + SIN(x)^*i)$ |

Table 17 (Page 11 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|---|--|
| CEE2015 | 277 | 10 | 5 | EXP(c) CQEXP(c) where c is of COMPLEX*32 type | Let c be $x + y*i$ $ y > 2^{100}\pi$ | $result = e^x + 0i$ |
| CEE2018 | 278 | 10 | 5 | LOG(c) CQLOG(c) where c is of COMPLEX*32 type | Let c be $x + y*i$ $x = 0.0$ and $y = 0.0$ | $result = -max + 0i$ |
| CEE2019 | 279 | 10 | 5 | SIN(c) CQSIN(c) COS(c) CQCOS(c) where c is of COMPLEX*32 type | Let c be $x + y*i$ $ x \geq 2^{100}$ | For CQSIN(c), $result = 0 + DSINH(y)*i$ For CQCOS(c), $result = DCOSH(y) + 0i$ |
| CEE2013 | 280 | 10 | 5 | SIN(c) CQSIN(c) COS(c) CQCOS(c) where c is of COMPLEX*32 type | Let c be $x + y*i$ $ y > 174.673$ | If $y > 0$, For CQSIN(c), $result = max/2*(SIN(x) + COS(x)*i)$ For CQCOS(c), $result = max/2*(COS(x) - SIN(x)*i)$ If $y < 0$, For CQSIN(c), $result = max/2*(SIN(x) - COS(x)*i)$ For CQCOS(c), $result = max/2*(COS(x) + SIN(x)*i)$ |
| CEE2009 | 281 | 10 | 5 | EXP(c) CDEXP(c) where c is of COMPLEX*16 type | Let c be $x + y*i$ $x > 174.673$ | $result = max*(COS(x) + SIN(x)*i)$ |

Table 17 (Page 12 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|---|--|
| CEE2015 | 282 | 10 | 5 | EXP(c) CDEXP(c) where c is of COMPLEX*16 type | Let c be $x + y*i$ $ y \geq 2^{50}\pi$ | $result = e^x + 0i$ |
| CEE2018 | 283 | 10 | 5 | LOG(c) CDLOG(c) where c is of COMPLEX*16 type | Let c be $x + y*i$ $x = 0.0$ and $y = 0.0$ | $result = -max + 0i$ |
| CEE2019 | 284 | 10 | 5 | SIN(c) CDSIN(c) COS(c) CDCOS(c) where c is of COMPLEX*16 type | Let c be $x + y*i$ $ x \geq 2^{50}\pi$ | For CDSIN(c), $result = 0 + DSINH(y)*i$ For CDCOS(c), $result = DCOSH(y) + 0i$ |
| CEE2013 | 285 | 10 | 5 | SIN(c) CDSIN(c) COS(c) CDCOS(c) where c is of COMPLEX*16 type | Let c be $x + y*i$ $ y > 174.673$ | If $y > 0$, For CDSIN(c), $result = max/2*(SIN(x) + COS(x)*i)$ For CDCOS(c), $result = max/2*(COS(x) - SIN(x)*i)$ If $y < 0$, For CDSIN(c), $result = max/2*(SIN(x) - COS(x)*i)$ For CSCOS(c), $result = max/2*(COS(x) + SIN(x)*i)$ |
| CEE2010 | 289 | 10 | 5 | SQRT(x) QSQRT(x) where x is of REAL*16 type | $x < 0$ | $result = x ^{1/2}$ |
| CEE2005 | 290 | 10 | 5 | GAMMA(x) where x is of REAL*4 type | $(x \leq 2^{-25.2})$ or $(x > 57.5744)$ | $result = max$ |

Table 17 (Page 13 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|---|---|
| CEE2005 | 291 | 10 | 5 | ALGAMA(x) where x is of REAL*4 type | $(x \leq 0)$ or $(x > 4.2937 \times 10^{73})$ | $result = max$ |
| CEE2011 | 292 | 10 | 5 | EXP(x) QEXP(x) where x is of REAL*16 type | $x > 174.673$ | $result = max$ |
| CEE2012 | 293 | 10 | 5 | LOG(x) QLOG(x) LOG10(x) QLOG10(x) where x is of REAL*16 type | $x \leq 0$ | If $x = 0$, $result = -max$ If $x < 0$, For QLOG(x), $result = \log x $ For QLOG10(x), $result = \log_{10} x $ |
| CEE2017 | 294 | 10 | 5 | SIN(x) QSIN(x) COS(x) QCOS(x) where x is of REAL*16 type | $ x \geq 2^{100}$ | $result = 2.0^{1/2.0}$ |
| CEE2014 | 295 | 10 | 5 | ATAN2(x, y) QATAN2(x, y) where x and y are of REAL*16 type | $x = 0.0$ and $y = 0.0$ | $result = 0$ |
| CEE2016 | 296 | 10 | 5 | SINH(x) QSINH(x) COSH(x) QCOSH(x) where x is of REAL*16 type | $ x > 175.366$ | For QSINH(x), if $x > 0.0$, $result = max$ if $x < 0.0$, $result = -max$ For QCOSH(x), $result = max$ |

Table 17 (Page 14 of 14). Corrective Actions for VS FORTRAN Version 2 Error Numbers

| Lang. Env. Condition | VSF V2 Error No. | ALLOW | PRINT | Fortran Statement, Intrinsic Function, or Machine Instruction That Was Executed | Error Detected | Standard Corrective Action Provided by the VS FORTRAN Version 2 Extended Error Handling Facility |
|----------------------|------------------|-------|-------|---|---|---|
| CEE2016 | 297 | 10 | 5 | ARSIN(x) QARSIN(x) ARCOS(x) QARCOS(x) where x is of REAL*16 type | $ x > 1$ | For QARCOS(x), if $x > 1.0$, result = 0 if $x < -1.0$, result = π For QARSIN(x), if $x > 1.0$, result = $\pi/2$ if $x < -1.0$, result = $-\pi/2$ |
| CEE2017 | 298 | 10 | 5 | TAN(x) QTAN(x) COTAN(x) QCOTAN(x) where x is of REAL*16 type | $ x \geq 2^{100}$ | result = 1 |
| CEE2002 | 299 | 10 | 5 | TAN(x) QTAN(x) COTAN(x) QCOTAN(x) where x is of REAL*16 type | For QTAN(x), x too close to $\pm\pi/2$, $\pm 3\pi/2$, ... For QCOTAN(x), x too close to 0, $\pm\pi$, $\pm 2\pi$, ... | result = max |
| CEE2005 | 300 | 10 | 5 | GAMMA(x) DGAMMA(x) where x is of REAL*8 type | ($x \leq 2^{252}$) or ($x > 57.5744$) | result = max |
| CEE2005 | 301 | 10 | 5 | LGAMA(x) DLGAMA(x) where x is of REAL*8 type | ($x \leq 0$) or ($x > 4.2937*10^{73}$) | result = max |

Bibliography

Language Products Publications

Language Environment for MVS & VM

Specification Sheet, GC26-4785
Concepts Guide, GC26-4786
Licensed Program Specifications, GC26-4774
Programming Guide, SC26-4818
Programming Reference, SC26-3312
Installation and Customization on MVS, SC26-4817
Debugging Guide and Run-Time Messages, SC26-4829
Run-Time Migration Guide, SC26-8232
Writing Interlanguage Communication Applications, SC26-8351
Fortran Run-Time Migration Guide, SC26-8499
Online Product Library, SK2T-2389

C/C++ for MVS/ESA

General Information, GC09-2060
Licensed Program Specifications, GC09-2064
Library Reference, SC23-3881
Compiler and Run-Time Migration Guide, SC09-2002
Diagnosis Guide, SC09-1839

C++/MVS

Programming Guide, SC09-1994
User's Guide, SC09-1993
Language Reference, SC09-1992
Class Library User's Guide, SC09-2000
Class Library Reference, SC09-2001
Master Index, SC09-2003
Support for SOMobjects under C++/MVS, SC09-2126

C/MVS

Language Reference, SC09-2063
User's Guide, SC09-2061
Programming Guide, SC09-2062
Reference Summary, SX09-1303
Master Index, SC09-2065

AD/Cycle C/370

General Information, GC09-1358
Licensed Program Specifications, GC09-1357
Programming Guide for LE/370 V1R3 Library, SC09-1840
Programming Guide for C/370 V2R2 Library, SC09-1841
User's Guide, SC09-1763
Language Reference, SC09-1762
Library Reference, SC09-1761
Reference Summary, SX09-1247
Migration Guide, SC09-1359
Diagnosis Guide, LY09-1806
Master Index, GC09-1760
Portable Operating System Interface (POSIX) Part 1: API for C Language, IEEE Std 1003.1

COBOL for MVS & VM (Release 2)

Licensed Program Specifications, GC26-4761
Programming Guide, SC26-4767
Language Reference, SC26-4769
Compiler and Run-Time Migration Guide, GC26-4764
Installation and Customization under MVS, SC26-4766
Diagnosis Guide, SC26-3138

COBOL/370 (Release 1)

General Information, GC26-4762
Licensed Program Specifications, GC26-4761
Programming Guide, SC26-4767
Language Reference, SC26-4769
Reference Summary, SX26-3788
Compiler and Run-Time Migration Guide, GC26-4764
Planning for Installation and Customization, SC26-4766
Diagnosis Guide, LY26-9596

Debug Tool

Debug Tool User's Guide and Reference, SC09-2137

VS FORTRAN Version 2

Language and Library Reference, SC26-4221

Programming Guide for CMS and MVS, SC26-4222

PL/I for MVS & VM

Licensed Program Specifications, GC26-3116

Programming Guide, SC26-3113

Language Reference, SC26-3114

Reference Summary, SX26-3821

Compiler and Run-Time Migration Guide,
SC26-3118

Installation and Customization under MVS,
SC26-3119

Compile-Time Messages and Codes, SC26-3229

Diagnosis Guide, SC26-3149

CoOperative Development Environment/370

Specification Sheet, GC09-1861

General Information, GC09-2048

Debug Tool, SC09-1623

Using Debug Tool, SC26-4662

Debug Tool Reference, SC26-4664

Installation, SC09-1624

Licensed Program Specifications, GC09-1898

*Using CoOperative Development Environment/370
with VS COBOL II and OS PL/I*, SC09-1862

Self-Study Guide, SC09-2047

Related Publications

MVS/ESA

*System Codes, MVS/ESA System Product: JES2
Version 4, JES3 Version 4*, GC28-1664

*System Codes, MVS/ESA System Product: JES2
Version 5, JES3 Version 5*, GC28-1486

*Diagnosis: Tools and Service Aids, MVS/ESA
System Product: JES2 Version 4, JES3 Version 4*,
LY28-1813

*Diagnosis: Tools and Service Aids, MVS/ESA
System Product: JES2 Version 5, JES3 Version 5*,
LY28-1845

MVS/TSO Dynamic STEPLIB Facility

Program Description/Operations Manual,
SH21-0029

High Level Assembler for MVS & VM & VSE

Programmer's Guide, MVS & VM Edition,
SC26-4941

Softcopy Publications

*Language Environment for MVS & VM Online
Product Library*, SK2T-2389

*IBM Online Library Omnibus Edition MVS
Collection*, SK2T-0710

IBM Online Library Omnibus Edition VM Collection,
SK2T-2067

You can order these publications from Mechanicsburg
through your IBM representative.

Language Environment Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the index, to the glossary of the appropriate HLL manual, or to the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

A

abend. Abnormal end of application.

absolute value. The magnitude of a real number regardless of its algebraic sign.

active routine. The currently executing routine.

actual argument. The Fortran term for the data passed to a called routine at the point of call. See also *dummy argument*.

address space. Domain of addresses that are accessible by an application.

aggregate. A structured collection of data items that form a single data type. Contrast with *scalar*.

ALLOCATE command. In MVS, the TSO command that serves as the connection between a file's logical name (the ddname) and the file's physical name (the data srt name).

AMODE. Provided by the linkage editor, the attribute of a load module that indicates the addressing mode in which the load module should be entered.

application. A collection of one or more routines cooperating to achieve particular objectives.

application program. A collection of software components used to perform specific types of work on a computer, such as a program that does inventory control or payroll.

argument. The data passed to a called routine at the point of call or the data received by a called routine. See also *actual argument* and *dummy argument*.

array. An aggregate that consists of data objects, each of which may be uniquely referenced by subscripting.

array element. A data item in an array.

assembler. Translates symbolic assembler language into binary machine language. The High Level Assembler is an IBM licensed program.

assembler user exit. A routine to tailor the characteristics of an enclave prior to its establishment. The name of the routine is CEEBXITA.

automatic call. The process used by the linkage editor to resolve external symbols left undefined after all the primary input has been processed. See also *automatic call library*.

automatic call library. Contains load modules or object modules that are to be used as secondary input to the linkage editor to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library may be:

- Libraries containing object modules, with or without linkage editor control statements
- Libraries containing load modules
- The library containing Language Environment run-time routines (SCEELKED and SAFHFORT)

automatic library call. Automatic call. See also *automatic call library*.

B

binder. The DFSMS/MVS component that processes the output of the language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader in the MVS/ESA operating system.

byte. The basic unit of storage addressability. It has a length of 8 bits.

C

C language. A high-level language used to develop software applications in compact, efficient code that can be run on different types of computers with minimal change.

C++ language. An object-oriented high-level language that evolved from the C language. C++ exploits the benefits of object-oriented technology such as code modularity, portability, and reuse.

CAA. Common anchor area.

call chain. A trace of all active routines and subroutines that can be constructed by the user from information included in a system dump, such as the locations of save areas and the names of routines.

callable service stub. A short routine that is link-edited with an application and that is used to transfer control from the application to a callable service.

callable services. A set of services that can be invoked by a Language Environment-conforming high-level language using the conventional Language Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

called routine. A routine or program that is invoked by another.

callee. A routine or program that is invoked by another.

caller. A routine or program that invokes another routine.

calling routine. A routine or program that invokes another routine.

cataloged procedure. A set of job control language (JCL) statements placed in a library and retrievable by name.

character. (1) A letter, digit, or other symbol. (2) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes.

child enclave. The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CLIST. TSO command list.

COBOL. Common Business-Oriented Language. A high-level language, based on English, that is primarily used for business applications.

common anchor area (CAA). Dynamically acquired storage that represents a Language Environment thread. Thread-related storage /resources are anchored off of the CAA. This area acts as a central communications area for the program, holding addresses of various storage and error-handling routines, and control blocks. The CAA is anchored by an address in register 12.

common block. A storage area that may be referenced by one or more compilation units. It is

declared in a Fortran program with the COMMON statement. See also *external data*.

compilation unit. An independently compilable sequence of HLL statements. Each HLL product has different rules for what makes up a compilation unit. Synonymous with *program unit*.

compile-time options. Keywords that can be specified to control certain aspects of compilation. Compiler options can control the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, the destination of error messages, and other things.

compiler options. See *compiler-time options*.

component. (1) Software that is part of a functional unit. (2) A set of modules that performs a major function within a system.

condition. An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit or C `signal()` function call) invoked by the Language Environment *condition manager* to respond to conditions.

condition handling. In Language Environment, the diagnosis, reporting, and/or tolerating of errors that occur while a routine is running.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition step. The step of the Language Environment condition handling model that follows the enablement step. In the condition step, user-written condition handlers, C signal handlers, and PL/I ON-units are first given a chance to handle a condition. See also *enablement step* and *termination imminent step*.

condition token. In Language Environment, a data type consisting of 96 bits (12 bytes). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

conflicting name. One of 20 names that exist in both the Fortran and the C/C++ libraries. See also *conflicting reference*.

conflicting reference. An external reference from a Fortran or assembler language routine to a Fortran library routine with a name that is the same as the name of a C/C++ library routine. The reference is considered to be a conflicting reference only when the intended resolution is to the Fortran library routine rather than to the corresponding C/C++ library routine.

control block. A storage area used by a computer program to hold control information.

control section (CSECT). The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

control statement. (1) In programming languages, a statement that is used to alter the continuous sequential execution of statements; a control statement can be a conditional statement, such as IF, or an imperative statement, such as STOP. (2) In JCL, a statement in a job that is used in identifying the job or describing its requirements to the operating system.

CSECT. Control section.

cursor. One of two pointers managed by the condition manager as it processes a condition. See *handle cursor* and *resume cursor*.

D

data, qualifying. See *qualifying data*.

data set. Under MVS, a named collection of related data records that is stored and retrieved by an assigned name.

data type. The properties and internal representation that characterize data.

datum, qualifying. A single element of qualifying data associated with a condition. See *qualifying data*.

DBCS. Double-byte character set.

DD statement. In MVS, the data definition statement. A JCL control statement that serves as the connection between a file's logical name (the ddname) and the file's physical name (the data srt name).

ddname. Data definition name. The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file through a DD statement or ALLOCATE command.

default. A value that is used or an action that is taken when no alternative is specified.

descriptor, q_data. See *q_data descriptor*.

disabled/enabled. See *enabled/disabled*.

double-byte character set (DBCS). A collection of characters represented by a 2-byte code.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. See also *precision*, *single-precision*.

doubleword. A sequence of bits or characters that comprises eight bytes (two 4-byte words) and is referenced as a unit.

doubleword boundary. A storage location whose address is evenly divisible by 8.

DSA. Dynamic storage area.

dummy argument. The Fortran term for the data received by a called routine. See also *actual argument*.

dynamic call. A call that results in locating a called routine at run time, that is, by loading the routine into virtual storage. Contrast with *static call*.

dynamic loading. See *dynamic call*.

dynamic storage area (DSA). An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment-managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is known as a *stack frame*.

E

enabled/disabled. A condition is enabled when its occurrence will result in the execution of condition handlers or in the performance of a standard system action to handle the condition as defined by Language Environment.

A condition is disabled when its occurrence is ignored by the condition manager.

enablement. The determination by a language at run time that an exception should be processed as a condition. This is the capability to intercept an exception and to determine whether it should be ignored or not; unrecognized exceptions are always defined to be enabled. Normally, enablement is used to supplement the hardware for capabilities that it does not have and language enforcement of a language's

semantics. An example of supplementing the hardware is the specialized handling of exponent-overflow exceptions based on language standards.

enablement step. The first step of the Language Environment condition handling model. In the enablement step it is determined whether an exception is to be *enabled* and processed as a condition. See also *condition step* and *termination imminent step*.

enclave. In Language Environment, an independent collection of routines, one of which is designated as the main routine and is invoked first. An enclave is roughly analogous to an executable program.

entry name. In assembler language, a programmer-specified name within a control section that identifies an entry point and can be referred to by any control section. See also *entry point*.

entry point. (1) The address or label of the first instruction that is executed when a routine is entered for execution. (2) Within a load module, the location to which control is passed when the load module is invoked.

entry point name. The symbol (or name) that represents an entry point. See also *entry point*.

environment. A set of services and data available to a program during execution. In Language Environment, environment is normally a reference to the run-time environment of HLLs at the enclave level.

exception. The original event such as a hardware signal, software detected event, or user-signaled event which is a potential condition. This action may or may not include an alteration in a program's normal flow. See also *condition*.

execution time. Synonym for *run time*.

execution environment. Synonym for *run-time environment*.

exponent-overflow exception. The program interruption that occurs when an overflow occurs during the execution of a floating-point instruction, that is, when the result value from the instruction has a characteristic that is larger than the floating-point data format can handle.

exponent-underflow exception. The program interruption that occurs when the result value from executing a floating-point instruction has a nonzero fraction and a characteristic is smaller than the floating-point data format can handle. This program interruption can be disabled through a program mask bit setting.

extended error handling facility. The VS FORTRAN facility that provided automatic error correction and control over both the handling of the errors and the printing of error messages.

external data. Data that persists over the lifetime of an enclave and maintains last-used values whenever a routine within the enclave is reentered. Within an enclave consisting of a single load module, it is equivalent to C writable static data, a Fortran common block, and COBOL EXTERNAL data.

external reference. In an object module, a reference to a symbol, such as an entry point name, defined in another program or module.

F

feedback code (fc). A condition token value. If you specify *fc* in a call to a callable service, a condition token indicating whether the service completed successfully is returned to the calling routine.

file. A named collection of related data records that is stored and retrieved by an assigned name. Equivalent to an MVS *data set*.

fix-up and resume. The correction of a condition either by changing the argument or parameter and running the routine again or by providing a specific value for the result.

fixed-point overflow exception. A program interruption caused by an overflow during signed binary arithmetic or signed left-shift operations. This program interruption can be disabled through a program mask bit setting.

Fortran. A high-level language used primarily for applications involving numeric computations. In previous usage, the name of the language was written in all capital letters, that is, FORTRAN.

Fortran signature CSECT. The resident routine that indicates that the load module in which it is present contains a Fortran routine.

FORTRAN 66. The FORTRAN language standard formally known as *American National Standard FORTRAN, ANSI X3.9-1966*. This language standard specifies the form and establishes the interpretation of programs written to conform to it.

FORTRAN 77. The FORTRAN language standard formally known as *American National Standard FORTRAN, ANSI X3.9-1978*. This language standard specifies the form and establishes the interpretation of programs written to conform to it.

fullword. A sequence of bits or characters that comprises four bytes (one word) and is referenced as a unit.

fullword boundary. A storage location whose address is evenly divisible by 4.

function. A routine that is invoked by coding its name in an expression. The routine passes a result back to the invoker through the routine name.

G

global error table (GET). A method employed by some HLLs, for example, C and VS FORTRAN, to determine actions for handling conditions. Whereas Language Environment condition handling actions are defined at the stack frame level, actions defined using the global error table apply to an entire application until explicitly changed. See also *extended error handling facility*.

H

handle cursor. A pointer used by the condition manager as it traverses the stack. The handle cursor points to the condition handler currently being invoked in the stack frame, whether it be a user-written condition handler or an HLL-specific condition handler.

handled condition. A condition that either a user-written condition handler or the HLL-specific condition handler has processed and for which the condition handler has specified that execution should continue.

hexadecimal. A base 16 numbering system. Hexadecimal digits range from 0 through 9 and A through F, giving values of 0 through 15.

high-level language (HLL). A programming language above the level of assembler language and below that of program generators and query languages. Examples are C, C++, COBOL, Fortran, and PL/I.

I

ILC. Interlanguage communication.

Initial process thread (IPT). See *initial thread*.

initial thread. In terms of POSIX, either the thread established by the `fork()` that created the *process*, or the first thread that calls `main()` after an `exec`. If the initial thread returns from `main()`, the effect is identical to having called `exit()`. Also known as *initial process thread (IPT)*. [POSIX.1]

instance-specific information (ISI). Located within the Language Environment condition token, information used by a condition handler or the condition manager to interpret and react to a specific occurrence of a condition. Qualifying data is an example of instance-specific information.

integer. A positive or negative whole number or zero.

interactive. Pertaining to a program or system that alternately accepts input and responds. In an interactive system, a constant dialog exists between user and system. Contrast with *batch*.

interface validation exit. A routine that, when used with the binder, automatically resolves conflicting references within Fortran routines.

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

interrupt. A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed.

interruption. Synonym for *interrupt*.

ISI. Instance specific information.

J

JCL. Job control language.

job control language (JCL). A sequence of commands used to identify a job to an operating system and to describe a job's requirements.

job step. The job control (JCL) statements that request and control execution of a program and that specify the resources needed to run the program. The JCL statements for a job step include one EXEC statement, which specifies the program or procedure to be invoked, followed by one or more DD statements, which specify the data sets or I/O devices that might be needed by the program.

L

Language Environment. Short form of Language Environment for MVS & VM.

A set of architectural constructs and interfaces that provides a common run-time environment and run-time services to applications compiled by Language Environment-conforming compilers.

Language Environment for MVS & VM. An IBM software product that provides a common run-time environment and common run-time services for conforming high-level language compilers.

Language Environment–conforming. Adhering to Language Environment's common interface conventions.

library. A collection of functions, subroutines, or other data.

LIFO. Last in, first out method of access. A queuing technique in which the next item to be retrieved is the item most recently placed in the queue.

link pack area (LPA). In MVS, an area of main storage containing reenterable routines from system libraries. Their presence in main storage saves loading time when a reenterable routine is needed.

link-edit. To create a loadable computer program by means of a linkage editor or binder.

linkage editor. An operating system component that resolves cross-references between separately compiled or assembled modules and then assigns final addresses to create a single relocatable load module. The linkage editor then stores the load module in a load library on disk.

load module. A collection of one or more routines that have been stored in a library by the linkage or binder after having been compiled or assembled. External references have usually been—but are not necessarily—resolved. When the external references have been resolved, the load module is in a form suitable for execution.

LPA. Link pack area.

M

main program. The first routine in an enclave to gain control from the invoker. In Fortran, a main program does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It could have a PROGRAM statement as its first statement. Contrast with *subprogram*.

MTF. Multitasking Facility.

Multitasking Facility (MTF). Facility provided separately by C and by Fortran to improve turnaround time on multiprocessor configurations by using MVS multitasking facilities. MTF is provided by C library functions or by Fortran callable services.

multitasking. See *multithreading*.

multithreading. Mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks, or threads.

MVS. Multiple Virtual Storage operating system.

N

nested condition. A condition that occurs during the handling of another, previous condition. Language Environment by default permits 10 levels of nested conditions. This setting may be changed by altering the DEPTHCONDLMT run-time option.

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

next sequential instruction. The next instruction to be executed in the absence of any branch or transfer of control.

nonreentrant. A type of program that cannot be shared by multiple users.

O

object module. A collection of one or more control sections produced by an assembler or compiler and used as input to the linkage editor or binder. Synonym for *text deck* or *object deck*.

OpenEdition MVS. MVS/ESA services that support an environment within which operating systems, servers, distributed systems, and workstations share common interfaces. OpenEdition MVS supports standard application development across multivendor systems. It is required if you want to create and use MVS/ESA applications that conform to the POSIX standard.

operating system. Software that controls the running of programs; in addition, an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

overflow. Exceeding the capacity of the intended unit of storage. See also *fixed-point overflow exception* and *exponent-overflow exception*.

P

parallel program. In the context of the Fortran parallel facility (not MTF), a program that uses parallel language constructs, invokes any of the parallel callable services, or was compiled with the PARALLEL compile-time option.

parallel subroutine. In the context of MVS multitasking and the Fortran Multitasking Facility, those portions of a program that can run independently of the main task program and each other. The parallel subroutines run in MVS subtasks.

parameter. The term used in certain other languages for the Fortran term *dummy argument*. See *argument*, *actual argument*, and *dummy argument*.

parent enclave. The enclave that issues a call to system services or language constructs to create a nested (child) enclave. See also *child enclave* and *nested enclave*.

percolate. The action taken by the condition manager when the returned value from a condition handler indicates that the handler could not handle the condition, and the condition will be transferred to the next handler.

pointer. A data element that indicates the location of another data element.

POSIX. Portable Operating System Interface.

precision. A measure of the ability to distinguish between nearly equal values, usually with data of different lengths. See also *single-precision* and *double-precision*.

preinitialization. A facility that allows a routine to initialize the run-time environment once, perform multiple executions within the environment, then explicitly terminate the environment.

pre–Language Environment–conforming. Any HLL program that does not adhere to Language Environment's common interface. For example, VS COBOL II Application Programming: Language Reference, OS/VS COBOL, OS PL/I, C/370 Version 1 and Version 2, VS FORTRAN Version 1, VS FORTRAN Version 2, FORTRAN IV G1, and FORTRAN IV H Extended are all pre–Language Environment–conforming HLLs.

program. See *enclave*.

program interruption. The interruption of the execution of a program due to some event such as an operation exception, an exponent-overflow exception, or an addressing exception.

program mask. In bits 20 through 23 of the program status word (PSW), a 4-bit structure that controls whether each of the fixed-point overflow, decimal overflow, exponent-overflow, and significance exceptions should cause a program interruption. The bits of the program mask can be manipulated to enable or disable the occurrence of a program interruption.

program status word (PSW). A 64-bit structure that includes the instruction address, program mask, and other information used to control instruction sequencing and to determine the state of the CPU. See also *program mask*.

program unit. Synonym for *compilation unit*.

promote. To change a condition to a different one by a condition handler. A condition handler routine promotes a condition because the error needs to be handled in a way other than that suggested by the original condition.

PSW. Program status word.

Q

q_data. Qualifying data. Information that a user-written condition handler can use to identify and react to a given instance of a condition.

q_data descriptor. A qualifying datum that contains the data type and length of the immediately following qualifying datum associated with a condition token.

q_data_token. An optional 32-bit data object that is placed in the ISI. It is used to access the qualifying data associated with a given instance of a condition.

qualifying data. *q_data*. Unique information associated through a condition token with a given instance of a condition. A user-written condition handler uses qualifying data to identify and react to the condition.

qualifying datum. A single element of qualifying data associated with a condition. See *qualifying data*.

R

reenterable. reentrant

reentrant. The attribute of a routine or application that allows more than one user to share a single copy of a load module.

resident routines. The Language Environment library routines linked with your application. They include such things as initialization routines and *callable service stubs*.

resume. To continue execution in an application at the point immediately after which a condition occurred. This occurs when a condition handler determines that a condition has been handled and normal application execution should continue.

resume cursor. The point in an application at which execution should continue if a condition handler requests the resume action for a condition it is processing. When a condition is signaled, the resume cursor is at the location at which the error occurred or at which the condition was first reported to the condition manager. The resume cursor can be moved with the CEEMRCE or CEEMRCR callable service.

return code. A code produced by a routine to indicate its success or failure. It may be used to influence the execution of succeeding instructions or programs.

return_code_modifier. A value set by Language Environment routines to indicate the severity of an unhandled condition. The return_code_modifier is a component of the return code that indicates the status of the execution of an enclave.

RMODE. Residence mode. Provided by the linkage editor, the attribute of a load module that specifies whether the module, when loaded, must reside below the 16MB virtual storage line or may reside anywhere in virtual storage.

routine. In this book, used as an exact equivalent of a COBOL *program*, a Fortran *main program* or *subprogram*, a PL/I *procedure*, or a C *function* or *program*, and means a named external routine, with or without named entry points, and with or without internal routines or nested programs.

run. To cause a program, utility, or other machine function to be performed.

run time. Any instant at which a program is being executed. Synonymous with *execution time*.

run-time environment. A set of resources that are used to support the execution of a program. Synonymous with *execution environment*.

S

SBCS. Single-byte character set.

scalar. A quantity characterized by a single value. Contrast with *aggregate*.

scalar instruction. An instruction, such as a load, store, arithmetic, or logical instruction, that operates on a scalar. Contrast with *vector instruction*.

signal. To make the condition manager aware of a condition for processing.

signature CSECT. The resident routine that indicates that the load module in which it is present contains a routine written in a particular language.

significance exception. The program interruption that occurs when the resulting fraction in a floating-point addition or subtraction instruction is zero. This program interruption can be disabled through a program mask bit setting.

single-byte character set (SBCS). A collection of characters represented by a 1-byte code.

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. See also *precision* and *double-precision*.

source code. The input to a compiler or assembler, written in a source language.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated on a LIFO stack and contains various pieces of information including a save area, condition handling routines, fields to assist the acquisition of a stack frame from the stack, and the local, automatic variables for the routine. In Language Environment, a stack frame is synonymous with *DSA*.

static call. A call that results in the resolution of the called program during the link-edit of the application. Contrast with *dynamic call*.

suboption. A value that can be provided as part of a compile-time or run-time option to further specify the meaning of the option.

subprogram. A program unit that is invoked or used by another program unit. In Fortran, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. Contrast with *main program*.

symbolic feedback code. The symbolic representation of the first 8 bytes of the 12-byte condition tokens. Symbolic feedback codes are provided so that in a condition handling routine you don't have to code the condition token in hexadecimal form.

syntax. The rules governing the structure of a programming language and the construction of a statement in a programming language.

T

task. In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer.

termination imminent step. The final step of the 3-step Language Environment condition handling model. In the termination imminent step, user-written condition handlers and PL/I ON-units are given one last chance to handle a condition or perform cleanup before the thread is terminated. See also *condition step* and *enablement step*.

thread. The basic run-time path within the Language Environment program management model. It is dispatched by the system with its own instruction counter and registers. The thread is where actual code resides.

traceback. A section of a dump that provides information about the stack frame (DSA), the program unit address, the entry point of the routine, the statement number, and status of the routines on the call-chain at the time the traceback was produced.

transient routines. The Language Environment library routines that are loaded at run time. Contrast with *resident routines*.

TSO. TSO/E.

TSO/E. Time Sharing Option Extensions. An MVS component that permits interactive compiling, link-editing, executing, and debugging of programs.

U

underflow. See *exponent-underflow exception*.

unhandled condition. A condition that isn't handled by any condition handler for any stack frame in the call chain. Contrast with *handled condition*.

user-written condition handler. A routine that analyzes and possibly takes action on conditions presented to it by the condition manager. The condition handler is registered either by calling the CEEHDLR callable service or by specifying the USRHDLR run-time option.

V

vector. A linearly ordered collection of scalars of the same type. Each scalar is said to be an *element* of the vector. See also *array*. Contrast with *scalar*.

vector instruction. An instruction, such as a load, store, arithmetic, or logical instruction, that operates on vectors residing in storage or in a vector register in the vector facility. Contrast with *scalar instruction*.

Index

A

- abend (completion) codes
 - ABTERMENC(ABEND) run-time option 41
- abnormal termination (abend)
 - ABTERMENC(ABEND) run-time option 41, 44
 - CEEBXITA assembler user exit 44
- ABS intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- ABTERMENC run-time option
 - ABTERMENC(ABEND) 44
 - ABTERMENC(RETCODE) 42
 - choosing normal or abnormal termination 41
 - choosing return codes or abend codes 41
- ACOS intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- actions requested by a user-written condition handler
 - about 52—54
 - fix-up and resume action 53—54, 55, 56
 - resume with Fortran-specific correction action 54, 56
 - resume with new input value action 53, 56
 - resume with new output value action 54, 56
 - percolate 53, 55
 - promote 53, 55
 - resume 52, 55
 - specifying 55—56
- AFBUOPT—VS FORTRAN error option table 58
- AFHCEEFC callable service 63, 89
- AFHCEEN callable service 63, 90
- AFHCQDSB—type declarations for qualifying data functions 61
- AFHVLPRM—default run-time options, VS FORTRAN 40
- AFHWG—link-edit Fortran only 13
- AFHWLG—link-edit and run Fortran only 13
- AFHWN—link-edit with NCAL
 - description 13
 - for removing conflicting references 31
 - See *also* conflicting references
- AFHWNCH—conflicting references removal tool
 - about 30
 - creating executable load module 33—34
 - creating module in non-executable form 31—33
- AFHWRL—separate and link-edit 13
- AFHWRLG—separate, link-edit, and run 13
- AFHWRLK—library module removal tool 17
- alignment of vector instruction operands 80
- alternative mathematical routines
 - VS FORTRAN 7
- AMBLIST service aid
 - for identifying conflicting references 23
 - See *also* conflicting references
- ARGSTR callable service
 - for retrieving program arguments 82
- ASIN intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- assembler language routine
 - initializing run-time environment 82—86
 - retrieving program arguments 82
 - VFEIN# callable service 82—86
 - with preinitialization services 9—10
- assembler user exit (CEEBXITA)
 - specifies normal or abnormal termination 41
 - specifies return code or completion (abend) code 41
- assembling programs
 - eliminating conflicting references 24—25
 - See *also* conflicting references
- AT statement
 - static debug 8
- ATAN intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- ATAN2 intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- automatic error correction, VS FORTRAN 59, 113—127

B

- BACKSPACE statement
 - error message unit 78

C

- C/C++ library routines
 - duplicate names in Fortran and C libraries
 - See conflicting references

- C/C++ routines
 - duplicate names in Fortran and C libraries
 - See conflicting references
 - requirement to link-edit 4
- callable services
 - for Language Environment callable services
 - AFHCEE—invoke callable service passing feedback code 63, 89
 - AFHCEEN—invoke callable service without feedback code 63, 90
 - linkage conventions and 9
 - qualifying data (q_data)
 - See qualifying data (q_data), callable services
 - restrictions on calling from Fortran 9
- cataloged procedures
 - AFHWG—link-edit Fortran only 13
 - AFHWLG—link-edit and run Fortran only 13
 - AFHWN—link-edit with NCAL 13
 - AFHWRL—separate and link-edit 13
 - AFHWRLG—separate, link-edit, and run 13
 - CEEWG—link-edit 13
 - CEEWG—link-edit and run 13
 - CEEWG—load and run 13
 - for compiling 14
 - for link-editing and running 13–14
- CDUMP callable service
 - output to the message file 78
 - return code set by 43
- CEE.V1R5M0.SAFHFORT data set
 - description 12
 - resolving conflicting library routine references 26–27
 - See *also* conflicting references
- CEE.V1R5M0.SCEELKED data set
 - description 12
 - duplicate names in Fortran and C libraries
 - See conflicting references
- CEE.V1R5M0.SCEERUN data set
 - description 13
 - specifying under TSO 15
- CEE.V1R5M0.SCEESAMP data set
 - AFHCQDSB—type declarations for qualifying data functions 61
 - AFHWNCH—conflicting references removal tool 30
 - AFHWRLK—library module removal tool 17
 - symbolic feedback code files
 - CEEFORCT—for common conditions 51
 - FORFORCT—for Fortran conditions 51
- CEE0CE—resume with new input value 54, 56
- CEE0CF—resume with new output value 54, 56
- CEE3SRC callable service
 - user return code set by 43
- CEE3SRP callable service 49
- CEEBXITA assembler user exit
 - normal or abnormal termination 44
 - return code or completion (abend) code 44
- CEEBXITA assembler user exit (*continued*)
 - specifies normal or abnormal termination 41
 - specifies return code or completion (abend) code 41
- CEEFORCT—symbolic feedback codes for common conditions 52
- CEEHDLR callable service 50
- CEEHDLU callable service 50
- CEEMRCE callable service 49
- CEEMRCR callable service
 - about 49
 - example 73
- CEEPINTV—interface validation exit
 - resolving conflicting references 27–29
- CEESG007—Fortran signature CSECT 18–20
- CEEUOPT—default run-time options 40
- CEEWG—link-edit 13
- CEEWG—link-edit and run 13
- CEEWG—load and run 13
- CEEXOPT macro 40
- CLOCK callable service
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- CLOSE statement, STATUS='DELETE'
 - error message unit 78
- COBOL routines with Fortran
 - requirement to link-edit 4
- code, restricted source 7
- compatibility
 - IFYVRENT facility (reentrant I/O library) 4
 - incompatibilities with VS FORTRAN
 - summary of 3, 5
 - load module 4
 - non-Fortran routines 4
 - object module
 - incompatibilities, summary of 3, 5
 - supported Fortran compilers 3
 - unsupported VS FORTRAN facilities 3, 5–8
- compile-time options
 - EC—extended common block 6
 - PARALLEL—parallel program 6
 - RENT—create reentrant program 9
- compilers, Fortran
 - supported with Language Environment 3
- compiling programs
 - eliminating conflicting references 24–25
 - See *also* conflicting references
- completion (abend) codes
 - ABTERMENC(ABEND) run-time option 41, 44
 - CEEBXITA assembler user exit 44
 - values 44
- condition
 - about 48
 - severity 50

- condition (*continued*)
 - unhandled 56—57
- condition handler, user-written
 - See user-written condition handler
- condition handling model 48—57
 - actions requested by a user-written condition handler
 - about 52—54
 - fix-up and resume action 53—54, 55, 56
 - percolate 53, 55
 - promote 53, 55
 - resume 52, 55
 - resume with Fortran-specific correction
 - action 54, 56
 - resume with new input value action 53, 56
 - resume with new output value action 54, 56
 - specifying 55—56
 - CEE3SRP callable service 49
 - CEEHDLR callable service 50
 - CEEHDLU callable service 50
 - CEEMRCE callable service 49
 - CEEMRCR callable service
 - about 49
 - example 73
- condition
 - about 48
 - enabled, enablement 49
 - severity 50
 - signaling 48
 - unhandled 56—57
- condition handler, user-written
 - See condition handling model, user-written condition handler
- condition manager 48
- condition step 50
- condition token
 - about 50
 - as feedback code 51
 - CEE0CE—resume with new input value 56
 - CEE0CF—resume with new output value 56
 - FOR0070—resume with Fortran-specific correction 56
 - instance-specific information (ISI) 50
 - symbolic feedback code 51
- cursors
 - handle cursor 49
 - resume cursor 49
- enablement routine, Fortran-specific 49
- enablement routine, language-specific 49
- enablement step 49
- facility IDs
 - about 50
 - list of 52
- feedback code 51
 - See *also* condition handling model, symbolic feedback code
- handle cursor 49
- condition handling model (*continued*)
 - move resume cursor explicit (CEEMRCE) 49
 - move resume cursor relative (CEEMRCR)
 - about 49
 - example 73
 - qualifying data (q_data) 50
 - registering a user-written condition handler (CEEHDLR) 50
 - responses requested by a user-written condition handler
 - See actions requested by a user-written condition handler
 - resume cursor 49
 - resume with Fortran-specific correction action 54
 - resume with new input value action 53
 - resume with new output value action 54
 - set resume point (CEE3SRP) 49
 - severity of a condition 50
 - stack frame 49
 - symbolic feedback code
 - See *also* condition handling model, feedback code
 - about 51
 - CEE0CE—resume with new input value 54
 - CEE0CF—resume with new output value 54
 - FOR0070—resume with Fortran-specific correction 54
 - symbolic feedback code files
 - about 51
 - CEEFORCT—for common conditions 52
 - FORFORCT—for Fortran conditions 52
 - unregistering a user-written condition handler (CEEHDLU) 50
 - user-written condition handler
 - about 50
 - actions 52—54
 - Fortran declarations for 54—56
 - Fortran examples of 63—73
 - Fortran-specific services for writing 59—63
 - writing 52—56
- condition manager 48
- condition step 50
- condition token
 - about 50
 - as feedback code 51
 - CEE0CE—resume with new input value 56
 - CEE0CF—resume with new output value 56
 - FOR0070—resume with Fortran-specific correction 56
 - instance-specific information (ISI) 50
 - symbolic feedback code 51
- conflicting library routine references
 - See conflicting references
- conflicting references 21—35
 - about 21
 - eliminating
 - by recompiling programs 24

- conflicting references (*continued*)
 - identifying 22—23
 - list of names 22
 - resolving automatically 25—29
 - summary 25
 - using DFSMS/MVS 27—29
 - using interface validation exit 27—29
 - with no C/C++ routines 26—27
 - resolving manually 30—35
 - conflicting references removal tool (AFHWNCH) 30
 - including specific library routines 34—35
 - removing conflicting references 30—34
- conflicting references removal tool (AFHWNCH)
 - about 30
 - creating executable load module 33—34
 - creating module in non-executable form 31—33
- correction, error
 - VS FORTRAN 59, 113—127
- COS intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- COSH intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- CPDUMP callable service
 - output to the message file 78
- cursors
 - handle cursor 49
 - resume cursor 49

D

- data spaces
 - EC compile-time option 6
- data, qualifying
 - See qualifying data (*q_data*)
- datum, qualifying
 - See qualifying data (*q_data*)
- DEBUG statement
 - static debug 8
- Debug, Interactive 3
- debug, static 8
- descriptor, *q_data*
 - format of 105
 - needed by Fortran functions 62
- DFSMS/MVS, interface validation exit
 - resolving conflicting references 27—29
- DISPLAY statement
 - static debug 8
- DUMP callable service
 - output to the message file 78

- DUMP callable service (*continued*)
 - return code set by 43
- duplicate names in Fortran and C libraries
 - See *also* conflicting references
 - list of 22
- DVCHK subroutine 81
- Dynamic STEPLIB Facility
 - for run-time libraries 16
 - invoking interface validation exit 28
- dynamically loading Fortran routines
 - Fortran signature CSECT (CEESG007) 19—20

E

- EC compile-time option 6
- eliminating conflicting references 24—25
 - See *also* conflicting references
 - by recompiling 24—25
- enabled, definition 49
- enablement routine, Fortran-specific 49
- enablement routine, language-specific 49
- enablement step 49
- END statement
 - return code set by 43
- ENDFILE statement
 - error message unit 78
- entry point names, library routine
 - duplicate names in Fortran and C libraries
 - See conflicting references
- environment, run-time, initializing
 - assembler language routine 82—86
 - VFEIN# callable service 82—86
- ERF intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- ERFC intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- ERRMON subroutine 59
- error correction, VS FORTRAN 59, 113—127
- error handling, VS FORTRAN
 - See extended error handling facility, VS FORTRAN
- error message unit
 - permissible I/O statements 77—78
- error messages, run-time
 - message numbers
 - changed from VS FORTRAN 76
 - impact on IOSTAT specifier values 77
 - Language Environment to VS FORTRAN mapping 110—112
 - VS FORTRAN to Language Environment mapping 107—109

error messages, run-time (*continued*)
 message texts 76

error option table (AFBUOPT), VS FORTRAN 58

errors, run-time, handling 48—73
 See *also* condition handling model
 Fortran-specific services for 59—63
 See *also* qualifying data (q_data), callable services
 See *also* qualifying data (q_data), functions in Fortran routines
 exponent-overflow exception 68—70
 fixed-point overflow exception 66—68
 invalid character in numeric field 70—73
 square-root exception 64—66

ERRSAV subroutine 59

ERRSET subroutine 59

ERRSTR subroutine 59

ERRTRA subroutine 59

ERRUNIT run-time option 77

EXIT callable service
 duplicate name in Fortran and C libraries
 See *also* conflicting references
 as a conflicting reference 22
 unique, alternate name 24

EXP intrinsic function
 duplicate name in Fortran and C libraries
 See *also* conflicting references
 as a conflicting reference 22
 unique, alternate name 24

exponent-overflow exception
 OVERFL subroutine 81
 qualifying data 98—100
 sample condition handler for 68—70

exponent-underflow exception
 qualifying data 98—100

extended common block 6

extended error handling facility, VS FORTRAN
 about 57
 automatic error correction 59, 113—127
 ERRMON subroutine 59
 error correction 59, 113—127
 error option table (AFBUOPT), VS FORTRAN 58
 ERRSAV subroutine 59
 ERRSET subroutine 59
 ERRSTR subroutine 59
 ERRTRA subroutine 59

F

facility IDs
 about 50
 list of 52

feedback code 51
 See *also* symbolic feedback code

fix-up actions
 VS FORTRAN 59, 113—127

fix-up and resume action 53—54, 55, 56
 resume with Fortran-specific correction action 54, 56
 resume with new input value action 53, 56
 resume with new output value action 54, 56

fixed-point divide exception
 DVCHK subroutine 81
 qualifying data 98—100

fixed-point overflow exception
 OVERFL subroutine 81
 program mask bit allows 80
 qualifying data 98—100
 sample condition handler for 66—68

floating-point divide exception
 DVCHK subroutine 81
 qualifying data 98—100

FOR0070—resume with Fortran-specific correction 54, 56

FORFORCT—symbolic feedback codes for Fortran conditions 52

Fortran compilers
 supported with Language Environment 3

FORTTRAN IV G1
 object module compatibility 3

FORTTRAN IV H Extended
 object module compatibility 3

Fortran signature CSECT (CEESG007) 18—20

Fortran-specific enablement routine 49

functions
 duplicate names in Fortran and C libraries
 See conflicting references
 qualifying data (q_data)
 See qualifying data (q_data), functions

G

GAMMA intrinsic function
 duplicate name in Fortran and C libraries
 See *also* conflicting references
 as a conflicting reference 22
 unique, alternate name 24

GET—global error table
 See extended error handling facility, VS FORTRAN

global error table (GET)
 See extended error handling facility, VS FORTRAN

H

handle cursor 49

handler, condition
 See user-written condition handler

I

I/O statements for the error message unit 77

- IBCOM# callable service
 - initializing run-time environment 82—86
- identifying conflicting references 22—23
 - See also conflicting references
- IFYVRENT facility 4
- incompatibilities with VS FORTRAN
 - summary of 3, 5
- initializing run-time environment
 - assembler language routine 82—86
 - VFEIN# callable service 82—86
- instance-specific information (ISI) 50
- Interactive Debug 3
- interface validation exit
 - resolving conflicting references 27—29
- interfaces, internal run-time library 7
- internal run-time library interfaces 7
- intrinsic functions
 - duplicate names in Fortran and C libraries
 - See conflicting references
- IOSTAT specifier 77
- ISI—instance-specific information 50

J

- JOBLIB ddname
 - CEE.V1R5M0.SCEERUN data set and 13

L

- Language Environment
 - about 2
 - callable services
 - Fortran services to invoke 63
 - incompatibilities with VS FORTRAN
 - summary of 3, 5
 - interface validation exit
 - resolving conflicting references 27—29
 - unsupported VS FORTRAN facilities 3, 5—8
- language-specific enablement routine 49
- library interfaces, internal 7
- library module removal tool (AFHWRLK) 17
- library routines
 - duplicate names in Fortran and C libraries
 - See also conflicting references
 - including specific library routines 34—35
 - removing from load module 17
- library source materials, restricted 7
- link mode (self-contained load modules)
 - availability of 7
 - compatibility 4
- link-edit (CEEWL) 13
- link-edit and run (CEEWLG) 13
- link-edit and run Fortran only (AFHWLG) 13
- link-edit Fortran only (AFHWL) 13
- link-edit with NCAL (AFHWN)
 - description 13

- link-edit with NCAL (AFHWN) (*continued*)
 - for removing conflicting references 31
 - See also conflicting references
- link-editing
 - conflicting references removal tool
 - See conflicting references
 - duplicate names in Fortran and C libraries
 - See conflicting references
 - libraries used for
 - CEE.V1R5M0.SAFHFORT 12
 - CEE.V1R5M0.SCEELKED 12
 - removing library routines from load module 17
 - resolving conflicting library routine references
 - See conflicting references
- load and run (CEEWG) 13
- load libraries
 - availability under TSO 15
 - CEE.V1R5M0.SCEERUN data set 13
- load mode, VS FORTRAN
 - load module compatibility 4
- load modules
 - compatibility with VS FORTRAN
 - link mode 4
 - load mode 4
 - IFYVRENT facility (reentrant I/O library) 4
 - link mode, VS FORTRAN 4
 - load mode, VS FORTRAN 4
 - non-Fortran routines 4
 - removing library routines from 17
- LOG intrinsic function
 - duplicate name in Fortran and C libraries
 - See also conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- LOG10 intrinsic function
 - duplicate name in Fortran and C libraries
 - See also conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24

M

- mathematical routines, alternative
 - VS FORTRAN 7
- message file
 - data set attributes 79
 - ddname 79
 - LRECL—record length 79
 - RECFM—record format 79
 - MSGFILE run-time option 79
 - permissible I/O statements 77—78
- messages, error, run-time
 - message numbers
 - changed from VS FORTRAN 76
 - impact on IOSTAT specifier values 77
 - Language Environment to VS FORTRAN mapping 110—112

- messages, error, run-time (*continued*)
 - message numbers (*continued*)
 - VS FORTRAN to Language Environment mapping 107—109
 - message texts 76
- misaligned vector instruction operands 80
- move resume cursor explicit (CEEMRCE) 49
- move resume cursor relative (CEEMRCR)
 - about 49
 - example 73
- MSGFILE run-time option 79
 - data set attributes 79
- MTF (multitasking facility), support for 6
- multitasking facility (MTF), support for 6
- MVS/TSO Dynamic STEPLIB Facility
 - for run-time libraries 16
 - invoking interface validation exit 28

N

- names, library routine
 - duplicate names in Fortran and C libraries
 - See conflicting references
- normal termination
 - ABTERMENC(RETCODE) run-time option 41, 42

O

- object module compatibility
 - incompatibilities with VS FORTRAN
 - summary of 3, 5
 - supported Fortran compilers 3
 - unsupported VS FORTRAN facilities 3, 5—8
- options, compile-time
 - See compile-time options
- options, run-time
 - See run-time options
- OVERFL subroutine 81

P

- PARALLEL compile-time option 6
- parallel programs 6
- PDUMP callable service
 - output to the message file 78
- percolate action 53, 55
- PL/I routines with Fortran
 - requirement to link-edit 4
- preinitialization services 9—10
- PRINT statement and the message file 78
- program arguments
 - about 36
 - assembler language routine 82
 - where coded in option string 37
- program interruption
 - exponent-overflow exception
 - OVERFL subroutine 81

- program interruption (*continued*)
 - exponent-overflow exception (*continued*)
 - qualifying data 98—100
 - sample condition handler for 68—70
 - exponent-underflow exception
 - qualifying data 98—100
 - fixed-point divide exception
 - DVCHK subroutine 81
 - qualifying data 98—100
 - fixed-point overflow exception
 - OVERFL subroutine 81
 - program mask bit allows 80
 - qualifying data 98—100
 - sample condition handler for 66—68
 - floating-point divide exception
 - DVCHK subroutine 81
 - qualifying data 98—100
 - square-root exception
 - qualifying data 101
 - sample condition handler for 64—66
 - unnormalized-operand exception
 - qualifying data 98—100
- program mask
 - fixed-point overflow exception enabled 80
- promote action 53, 55
- PRTUNIT run-time option 77

Q

- q_data
 - See qualifying data (q_data)
- q_data descriptor
 - format of 105
 - needed by Fortran functions 62
- QDCH1 function 61, 95
- QDCH255 function 61, 95
- QDCH31 function 61, 95
- QDCH6 function 61, 95
- QDCH62 function 61, 95
- QDCH8 function 61, 95
- QDCX16 function 61, 95
- QDCX32 function 61, 95
- QDCX8 function 61, 95
- QDFETCH callable service 60, 91
- QDINT1 function 61, 95
- QDINT2 function 61, 95
- QDINT4 function 61, 95
- QDINT8 function 61, 95
- QDLEN function 62, 92
- QDLOC function 62, 92
- QDR16 function 61, 95
- QDR4 function 61, 95
- QDR8 function 61, 95
- QDSTORE callable service 60, 93
- QDTYPE function 62, 94

QDTYPE_CHARACTER—data type value for character type 95
 QDTYPE_CHAR—data type value for character type 95
 QDTYPE_COMPLEX—data type value for complex type 95
 QDTYPE_FAILURE—data type value for unknown type 95
 QDTYPE_INT—data type value for integer type 95
 QDTYPE_INTEGER—data type value for integer type 95
 QDTYPE_REAL—data type value for real type 95
 QDTYPE_UNSIGNED—data type value for unsigned type 95
 QDUS1 function 61, 95
 qualifying data (q_data)
 about 50
 AFHCQDSB—type declarations for qualifying data functions 61
 callable services
 QDFETCH—retrieve value 60, 91
 QDSTORE—store value 60, 93
 functions
 QDCH1—CHARACTER*1 value 61, 95
 QDCH255—CHARACTER*255 value 61, 95
 QDCH31—CHARACTER*31 value 61, 95
 QDCH62—CHARACTER*62 value 61, 95
 QDCH6—CHARACTER*6 value 61, 95
 QDCH8—CHARACTER*8 value 61, 95
 QDCX16—COMPLEX*16 value 61, 95
 QDCX32—COMPLEX*32 value 61, 95
 QDCX8—COMPLEX*8 value 61, 95
 QDINT1—INTEGER*1 value 61, 95
 QDINT2—INTEGER*2 value 61, 95
 QDINT4—INTEGER*4 value 61, 95
 QDINT8—INTEGER*8 value 61, 95
 QDLEN—length 62, 92
 QDLOC—address 62, 92
 QDR16—REAL*16 value 61, 95
 QDR4—REAL*4 value 61, 95
 QDR8—REAL*8 value 61, 95
 QDTYPE—data type 62, 94
 QDUS1—UNSIGNED*1 value 61, 95
 q_data descriptor
 format of 105
 needed by Fortran functions 62
 q_data structure
 abend 97
 arithmetic program interruptions 98—100
 bit manipulation conditions 102—104
 exponent-overflow exception 98—100
 exponent-underflow exception 98—100
 fixed-point divide exception 98—100
 fixed-point overflow exception 98—100
 floating-point divide exception 98—100
 math conditions 102—104
 square-root exception 101

qualifying data (q_data) (*continued*)
 q_data structure (*continued*)
 unnormalized-operand exception 98—100
 qualifying datum
 See qualifying data (q_data)

R

READ statement
 error message unit 78
 reassembling programs
 eliminating conflicting references 24—25
 See *also* conflicting references
 recompiling programs
 eliminating conflicting references 24—25
 See *also* conflicting references
 reentrant I/O library 4
 reentrant program
 Fortran with non-Fortran 9
 separate and link-edit
 AFHWRL cataloged procedure 13
 separate, link-edit, and run
 AFHWRLG cataloged procedure 13
 references to library routines
 duplicate names in Fortran and C libraries
 See conflicting references
 references, conflicting
 See conflicting references
 registering a user-written condition handler (CEEHDLR) 50
 removing library routines from load module 17
 RENT compile-time option
 reentrant program 9
 replacing library routines in load module 17
 resident routines
 CEE.V1R5M0.SCEELKED and
 CEE.V1R5M0.SAFHFORT data sets 12
 resolving conflicting library routine references
 See conflicting references
 responses requested by a user-written condition handler
 See actions requested by a user-written condition handler
 restricted source materials 7
 result_code
 values set by a condition handler 55
 resume action 52, 55
 resume cursor 49
 resume with Fortran-specific correction action 54, 56
 resume with new input value action 53, 56
 resume with new output value action 54, 56
 return codes
 ABTERMENC(RETCODE) run-time option 41, 42
 detecting 42
 for unhandled conditions 44
 specifying in Fortran 43

- REWIND statement
 - error message unit 78
- run-time environment, initializing
 - assembler language routine 82—86
 - VFEIN# callable service 82—86
- run-time error messages
 - message numbers
 - changed from VS FORTRAN 76
 - impact on IOSTAT specifier values 77
 - Language Environment to VS FORTRAN mapping 110—112
 - VS FORTRAN to Language Environment mapping 107—109
 - message texts 76
- run-time errors, handling 48—73
 - See *also* condition handling model
 - Fortran-specific services for 59—63
 - See *also* qualifying data (q_data), callable services
 - See *also* qualifying data (q_data), functions
 - in Fortran routines
 - exponent-overflow exception 68—70
 - fixed-point overflow exception 66—68
 - invalid character in numeric field 70—73
 - square-root exception 64—66
- run-time library interfaces, internal 7
- run-time library source materials, restricted 7
- run-time options 36—40
 - AFHVLPRM—default values, VS FORTRAN 40
 - CEEUOPT—default values 40
 - CEEXOPT macro 40
 - coding the option string 36—37
 - default values 40
 - list of 37—39
 - USRHDLR—register user-written condition handler 55, 56
 - VSF2PARAM macro, VS FORTRAN 40
- running your program
 - library used for
 - CEE.V1R5M0.SCEERUN 13
 - load module compatibility 4

S

- SAFHFORT data set
 - description 12
 - resolving conflicting library routine references 26—27
 - See *also* conflicting references
- SCEELKED data set
 - description 12
 - duplicate names in Fortran and C libraries
 - See conflicting references
- SCEERUN data set
 - description 13
 - specifying under TSO 15
- SCEESAMP data set
 - AFHCQDSB—type declarations for qualifying data functions 61
 - AFHWNCH—conflicting references removal tool 30
 - AFHWRLK—library module removal tool 17
 - symbolic feedback code files
 - CEEFORCT—for common conditions 51
 - FORFORCT—for Fortran conditions 51
- SDUMP callable service
 - output to the message file 78
- self-contained load modules
 - compatibility 4
- self-contained load modules (link mode)
 - availability of 7
- separate and link-edit (AFHWRL) 13
- separate, link-edit, and run (AFHWRLG) 13
- set resume point (CEE3SRP) 49
- severity of a condition 50
- signaling a condition 48
- signature CSECT, Fortran (CEESG007) 18—20
- SIN intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- SINH intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- source materials, restricted 7
- SQRT intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- square-root exception
 - qualifying data 101
 - sample condition handler for 64—66
- stack frame 49
- static debug 8
- STEPLIB ddname
 - CEE.V1R5M0.SCEERUN data set and 13
 - specifying under TSO 15
- STOP statement
 - return code set by 43
- symbolic feedback code
 - See *also* feedback code
 - about 51
 - CEE0CE—resume with new input value 54
 - CEE0CF—resume with new output value 54
 - FOR0070—resume with Fortran-specific correction 54
- symbolic feedback code files
 - about 51
 - CEEFORCT—for common conditions 52

- symbolic feedback code files (*continued*)
 - FORFORCT—for Fortran conditions 52
- SYS1.VSF2FORT data set
 - CEE.V1R5M0.SCEELKED and
 - CEE.V1R5M0.SAFHFORT data sets 12
- SYS1.VSF2LOAD data set
 - CEE.V1R5M0.SCEERUN data set 13
- SYSLIB ddname
 - for linkage editor
 - CEE.V1R5M0.SAFHFORT data set 12
 - CEE.V1R5M0.SCEELKED data set 12
- SYSRCS callable service
 - user return code set by 43
- SYSRCX callable service
 - return code set by 43
- system completion (abend) codes
 - See completion (abend) codes
- system link list
 - CEE.V1R5M0.SCEERUN data set and 13

T

- TAN intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- TANH intrinsic function
 - duplicate name in Fortran and C libraries
 - See *also* conflicting references
 - as a conflicting reference 22
 - unique, alternate name 24
- termination with completion (abend) code
 - ABTERMENC(ABEND) run-time option 41, 44
 - CEEBXITA assembler user exit 44
- termination with return code
 - ABTERMENC(RETCODE) run-time option 41
- termination, abnormal
 - ABTERMENC(ABEND) run-time option 41, 44
 - CEEBXITA assembler user exit 44
- termination, normal
 - ABTERMENC(RETCODE) run-time option 41
- texts of run-time error messages 76
- TRACE statement
 - static debug 8
- transient routines
 - availability under TSO 15
 - CEE.V1R5M0.SCEERUN data set 13
- TSO
 - invoking interface validation exit 28
 - specifying run-time libraries 16

U

- unhandled condition 56—57

- unnormalized-operand exception
 - qualifying data 98—100
- unregistering a user-written condition handler (CEEHDLU) 50
- unsupported VS FORTRAN facilities 3, 5—8
- user completion (abend) codes
 - See completion (abend) codes
- user-written condition handler
 - about 50
 - actions 52—54
 - Fortran declarations for 54—56
 - Fortran examples of 63—73
 - Fortran-specific services for writing 59—63
 - writing 52—56
- USRHDLR run-time option 55, 56

V

- vector instruction operands, misaligned 80
- VFEIN# callable service
 - initializing run-time environment 82—86
- VM, Fortran support on 3
- VS FORTRAN
 - alternative mathematical routines 7
 - automatic error correction 59, 113—127
 - duplicate names in Fortran and C libraries
 - See conflicting references
 - error correction 59, 113—127
 - error message numbers
 - Language Environment to VS FORTRAN mapping 110—112
 - VS FORTRAN to Language Environment mapping 107—109
 - error option table (AFBUOPT), VS FORTRAN 58
 - extended error handling facility 57—59
 - incompatibilities, summary of 3, 5
 - library routines
 - removing from load module 17
 - load module compatibility 4
 - mathematical routines, alternative 7
 - object module compatibility 3
 - unsupported facilities 3, 5—8
- VSCOM# callable service
 - initializing run-time environment 82—86
- VSF2FORT data set
 - CEE.V1R5M0.SCEELKED and
 - CEE.V1R5M0.SAFHFORT data sets 12
- VSF2LOAD data set
 - CEE.V1R5M0.SCEERUN data set 13
- VSF2PARAM macro, VS FORTRAN 40

W

- WRITE statement and the message file 78