IBM C for VSE/ESA

User's Guide

Release 1

SC09-2424-00

IBM C for VSE/ESA

User's Guide

Release 1

SC09-2424-00

Note!

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

First Edition (December 1996)

This edition applies to Version 1, Release 1, Modification Level 0, of IBM C for VSE/ESA (Program 5686-A01); Version 1, Release 4, Modification Level 0, of IBM Language Environment for VSE/ESA (Program 5686-094); the VSE C Language Run-Time Support feature of VSE/ESA Version 2 Release 2 (Program 5690-VSE); and to all subsequent releases and modifications until otherwise indicated in new editions. 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 readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See “Communicating Your Comments to IBM” for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this publication.

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.

Contents

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. 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 the customer program with the C/VSE language. This book documents General-Use Programming Interfaces and associated guidance information provided by the IBM C for VSE/ESA and IBM Language Environment for VSE/ESA products.

General-Use Programming Interfaces allow the customer to write programs that obtain the services of the IBM C for VSE/ESA compiler and IBM Language Environment for VSE/ESA.

Standards

Extracts are reprinted from *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C*

language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization, ISO, and the International Electrotechnical Commission, IEC. The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case Postal, 1211 Geneva 20, Switzerland. Copyright remains with ISO and IEC.

Portions of this book are extracted from *X/Open Specification, Programming Languages, Issue 3*, copyright 1988, 1989, February 1992, by the X/Open Company Limited, with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK.

Trademarks and Service Marks

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

AIX/6000
C/370
CICS
IBM
Language Environment
OS/2
OS/390
OS/400
System/370
SAA
VSE/ESA

The following terms are trademarks of other companies:

ANSI	American National Standards Institute
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organization for Standardization
POSIX	Institute of Electrical and Electronic Engineers
X/Open	X/Open Company Ltd.

About This Book

This edition of the *User's Guide* is intended for users of IBM C for VSE/ESA (C/VSE) as implemented for the IBM Language Environment for VSE/ESA (LE/VSE) environment. It contains guidelines for preparing C programs to run under the VSE operating system.

To use this book, or any other C/VSE book, you must have a working knowledge of the C programming language, the operating system, and where appropriate, the related products.

Note: References to LE/VSE also apply to the VSE C Language Run-Time Support feature of VSE/ESA Version 2 Release 2.

The C Language

The C language is a general purpose, function-oriented programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages, and it also provides many of the benefits of a low-level language. Using the C/VSE language, you can write portable code conforming to the ANSI standard.

IBM offers the C language on other platforms, such as the OS/2, AIX/6000, OS/400, OS/390, and VM operating systems.

The elements of the C/VSE implementation include:

- All elements of the joint ISO and IEC standard: ISO/IEC 9899:1990 (E)
- ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
- Locale based internationalization support as defined in: ISO/IEC DIS 9945-2:1992/IEEE POSIX 1003.2-1992 Draft 12 (There are some limitations to fully-compliant behavior as noted in the *LE/VSE C Run-Time Programming Guide*.)
- Extended multibyte and wide character utilities as defined by a subset of the Programming Language C Amendment 1, which will be ISO/IEC 9899:1990/Amendment 1:1994(E)

IBM Language Environment for VSE/ESA

C/VSE exploits the C run-time environment and library of run-time callable services provided by IBM Language Environment for VSE/ESA (LE/VSE).

LE/VSE establishes a common run-time environment and common run-time callable services for language products, user programs, and other products.

The common execution environment is made up of data items and services performed by library routines available to a particular application running in the environment. The services that LE/VSE provides to your application may include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, support for interlanguage communication (ILC), and condition handling.
- Extended services often needed by applications. These functions are contained within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help the execution, performance tuning, performance, and diagnosis of your application.
- Access to language-specific library routines, such as the C functions.

Utilities

The following C/VSE-related utilities are provided with LE/VSE:

- Locale definition utility to generate locales for use with C/VSE applications. A locale is a definition of those aspects of each country or culture that a program must recognize, such as: coded character sets (codepage), national language options, rules and symbols for monetary information, and time zones.
- `iconv` and `genxlt` code set conversion utilities to increase the portability of code that is passed between systems and locales.
- DSECT conversion utility to convert descriptive data produced by High Level Assembler into C/VSE data structures, for C/VSE programs that interface with assembler programs.

For more information about these utilities, see the *LE/VSE C Run-Time Programming Guide*.

Using Your Documentation

The publications in the C/VSE and LE/VSE libraries are designed to help you develop C/VSE applications that run with LE/VSE. Each publication helps you perform a different task. For a complete list of publications you might need, see “Bibliography” on page 115. Table 1 lists the publications in the C/VSE library.

Table 1. How to Use C/VSE Publications

To...	Use...	
Plan for, install, customize, and maintain C/VSE	*Installation and Customization Guide*	GC09-2422
Migrate VSE applications from C/370 to C/VSE	*Migration Guide*	SC09-2423
Get details on C/VSE syntax and specifications of language elements	*Language Reference*	SC09-2425
Find syntax for compile-time options; compile your C/VSE applications; get details on compile-time messages	*User's Guide*	SC09-2424
Diagnose compiler problems and report them to IBM	*Diagnosis Guide*	GC09-2426
Understand warranty information	*Licensed Program Specifications*	GC09-2421

Table 2 on page xi lists the publications in the LE/VSE library. These include publications designed to help you develop and debug your C/VSE applications, diagnose run-time problems that occur in your C/VSE applications, and use C/VSE-related utilities.

Table 2. How to Use LE/VSE Publications

To...	Use...	
Evaluate LE/VSE	*Fact Sheet* *Concepts Guide*	GC33-6679 GC33-6680
Plan for, install, customize, and maintain LE/VSE	*Installation and Customization Guide*	SC33-6682
Understand the LE/VSE program models and concepts	*Concepts Guide* *Programming Guide*	GC33-6680 SC33-6684
Find syntax for LE/VSE run-time options and callable services	*Programming Reference*	SC33-6685
Develop your C/VSE applications	*Programming Guide* *C Run-Time Programming Guide* *C Run-Time Library Reference*	SC33-6684 SC33-6688 SC33-6689
Develop interlanguage communication (ILC) applications	*Writing Interlanguage Communication Applications*	SC33-6686
Debug your C/VSE applications and get details on run-time messages	*Debugging Guide and Run-Time Messages*	SC33-6681
Migrate applications to LE/VSE	*Run-Time Migration Guide*	SC33-6687
Diagnose run-time problems that occur in your C/VSE applications	*Debugging Guide and Run-Time Messages*	SC33-6681
Use C/VSE-related utilities	*C Run-Time Programming Guide*	SC33-6688
Understand warranty information	*Licensed Program Specifications*	GC33-6683

Softcopy Examples

Most examples in the following books are available in machine-readable form:

- *C/VSE Installation and Customization Guide*, GC09-2422
- *C/VSE User's Guide*, SC09-2424
- *C/VSE Language Reference*, SC09-2425

Softcopy examples are indicated in the book by a label in the form, `EDCX`*bnnn*. The *b* refers to the book:

- `I` is the *C/VSE Installation and Customization Guide*
- `U` is the *C/VSE User's Guide*
- `R` is the *C/VSE Language Reference*

Softcopy examples are installed on your system along with C/VSE, in the sublibrary `PRD2.DBASE`.

Example member names are the same as the labels indicated in the book.

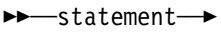
Contact your system programmer if the default names are not used at your installation.

How to Read the Syntax Diagrams

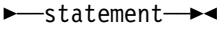
In this book, syntax for commands, directives, and statements is described using the following structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

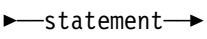
  A double right-arrowhead indicates the beginning of a command, directive, or statement; the single right-arrowhead indicates that it is continued on the next line. (In the following diagrams, `statement` is used to represent a command, directive, or statement.)

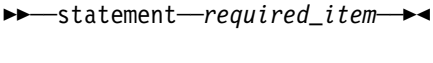
  The following indicates a continuation; the opposing arrowheads indicate the end of a command, directive, or statement.

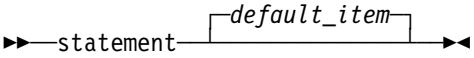
  Diagrams of syntactical units other than complete commands, directives, or statements look like this:

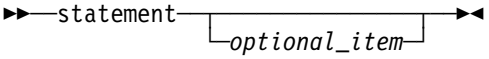
- Required items are on the horizontal line (the main path).

- IBM-supplied default items are above the main path.

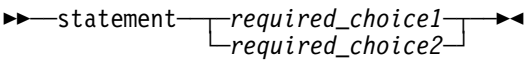
  

- Optional items are below the main path.

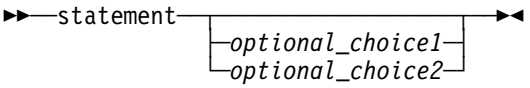
  

- If you can choose from two or more items, they are vertical in a stack.

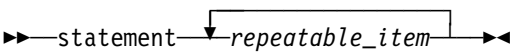
  If you *must* choose one of the items, one item of the stack is on the main path.

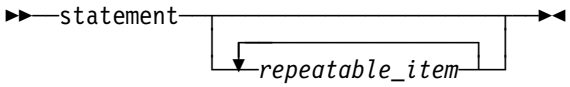
  If choosing one of the items is optional, the entire stack is below the main path.

- An arrow returning to the left above a line indicates an item that you can repeat.

  or

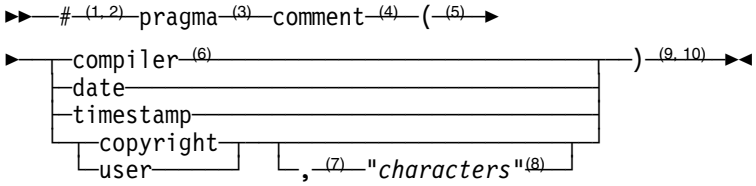


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords are in non-italic letters and should be entered exactly as shown (for example, `pragma`). They must be spelled exactly as shown. Variables are in italics and lowercase letters (for example, *identifier*). They represent user-supplied names or values.
- Keywords that appear in mixed-case letters (for example, `AGGregate`) indicate that the keyword can be abbreviated (`AGG`) or entered in full (`AGGREGATE`).
- If punctuation marks, parentheses, arithmetic operators, or other non-alphanumeric characters are shown, you must enter them as part of the syntax.

Note: The white space is not always required between tokens but you should include at least one blank space between tokens unless otherwise specified.

The following syntax diagram example shows the syntax for the `#pragma comment` directive.



Notes:

1 This is the start of the syntax diagram.

2 The symbol `#` must appear first.

3 The keyword `pragma` must follow the `#` symbol.

4 The keyword `comment` must follow the keyword `pragma`.

5 An opening parenthesis must follow the keyword `comment`.

6 The comment type must be entered only as one of the following: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the `#pragma comment` directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Chapter 1. Compiling with C/VSE

This chapter describes how to compile your program using C/VSE.

C/VSE analyzes the C source program and translates the source code into machine instructions known as *object code*.

To compile your C source program using C/VSE, you must have access to LE/VSE because the compiler itself is written in C and calls run-time library functions to compile code.

Invoking the C/VSE Compiler

When you invoke C/VSE, the operating system automatically tries to locate and execute the compiler. LE/VSE is also required to execute the compiler. The location of the compiler is determined by the system programmer who installed the product. If the compiler and run-time library phases are loaded in the *Shared Virtual Area* (SVA), or the sublibraries are defined in the permanent `LIBDEF` chain, no additional user action is necessary; Otherwise, a `LIBDEF SEARCH PHASE` statement in the Job Control Language (JCL) must specify the sublibrary.

To compile your C source program in batch, you will need to write your own JCL statements, or use a cataloged procedure.

You use JCL to define your jobs and job steps to the operating system. You can describe the steps you want the operating system to perform, and specify the resources that are required by the job. The JCL statements that are essential to run a C job are:

- A `JOB` statement that identifies the start of the job
- `DLBL`, `EXTENT`, and `ASSGN` statements that identify the input/output facilities required by the program executed in the job step
- `OPTION` statement to indicate the options to be used
- An `EXEC` statement that identifies a job step and the program to be executed either directly or by means of a cataloged procedure

For more information about JCL, refer to “Related Publications” on page 115.

Regularly used sets of JCL can be prepared once, and stored in a VSE Librarian sublibrary. Such a set of statements is called a *cataloged procedure*. A cataloged procedure consists of one or more job steps. You include the cataloged procedure in a job by specifying `PROC=`*`name`* on an `EXEC` statement, where *`name`* is the name of the cataloged procedure.

Writing Your Own Job Control Language Statements

The following example shows a general JCL stream to compile a C program:

EDCXUAAD

```
// JOB EDCXUAAD
// LIBDEF PHASE,SEARCH=(user.sublib,PRD2.DBASE,PRD2.SCEEBASE)
// LIBDEF SOURCE,SEARCH=(user.sublib,PRD2.DBASE,PRD2.SCEEBASE)
// OPTION LINK
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='SOURCE,XREF'
#include <stdio.h>
  ⋮
main()
{
 /*  comment   */
  ⋮
}
/*
/&
```

Figure 1. Sample JCL to Compile a C Program

Notes:

1. Additional JCL statements for compiler workfiles (`IJSYS01-IJSYS07`) and `SYSLNK` are required if the labels for these files have not been placed in the System Standard or Partition Standard label areas.
2. Both C/VSE and the run-time library functions use `GETVIS` storage. The `SIZE` parameter on the `EXEC` statement specifies the name of the C/VSE main phase. This will set the size of program storage to the size of the root phase of the compiler and will make all other storage in the partition available as `GETVIS` storage.
3. If the compiler fails with a return code of 16 and no message, increase the partition size and re-run the job.

Specifying the Input Files

Input for the compiler consists of:

- Your C source program
- LE/VSE run-time library standard header files
- Your header files

A list of the files used by C/VSE, along with a brief description, is given in Table 14 on page 99.

To assist you in migrating existing applications from other operating systems to VSE, filename conversions are performed automatically by C/VSE. These conversions will affect filenames specified on `#include` preprocessor directives, and in file I/O library functions such as `fopen()`. See the *C/VSE Language Reference* for general information on the `#include` directive and the available I/O library functions.

Primary Input

The primary input to the compiler is the file containing your C source program. C/VSE will read your C source program from `SYSIPT`, or from the file specified using the `INFILE` compile-time option.

If reading from a logical unit assigned to SYSRDR (for example, SYSIPT), the C source program should immediately follow the EXEC statement used to invoke C/VSE.

If the primary input file is SYSIPT (source statements imbedded in your JCL), the input is terminated by the first /* in column 1. Therefore, C comment statements in the primary input file should not start in column 1.

Secondary Input

Secondary input to the compiler consists of sequential files or VSE Librarian sublibrary members referenced by #include directives.

Specify the sublibraries containing LE/VSE standard header files and user header files with the JCL // LIBDEF statement.

Note: Secondary input to the compiler may contain comments that start in column 1. However, if the secondary input contains comments beginning in column 1 and the PPONLY compile-time option is used, the PPONLY output cannot be used as input to the compiler if it is read back via SYSIPT.

Specifying the Output Files

The compiler will write the listing to SYSLST. SYSLST may be directed to a printer, a direct access device, or a magnetic-tape device. The listing will include the results of the default or specified options of the PARM parameter (that is, the diagnostic messages and the object code listing). For example:

```
// ASSGN SYSLST,PRT1
```

To create an object module and store it on disk or tape, you can use either the OBJECT or DECK compile-time options.

With the OBJECT compile-time option, the compiler writes the object to SYSLNK for input to the linkage editor. With the DECK compile-time option, the compiler writes the object to SYSPCH.

SYSLNK does not need to be assigned if the NOOBJ option is in effect. SYSPCH does not need to be assigned if the NODECK option is in effect.

If the object deck produced by the compiler is to be cataloged in a VSE Librarian sublibrary, the DECK and NAME options should be used as follows:

EDCXUAAE

```
* $$ JOB JNM=EDCXUAAE,PDEST=(*,uid),LDEST=(*,uid),PRI=pri,CLASS=class
* $$ PUN DISP=I,CLASS=class
// JOB EDCXUAAE
// LIBDEF *,SEARCH=(PRD2.DBASE,PRD2.SCEEBASE,...)
// EXEC IESINSRT
$ $$ LST DISP=D
// JOB jobname
// OPTION CATAL
// EXEC LIBR,PARM='MSHP;ACCESS SUBLIB=user.sublib'
* $$ END
// OPTION DECK
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='NAME(name)'
#include <stdio.h>
  ⋮
main()
{
 /*  comment  */
  ⋮
}
/*
// EXEC IESINSRT
$ $$ EOJ
* $$ END
/&
* $$ EOJ
```

Figure 2. Sample JCL to Compile and Catalog Object Deck in a VSE Librarian Sublibrary

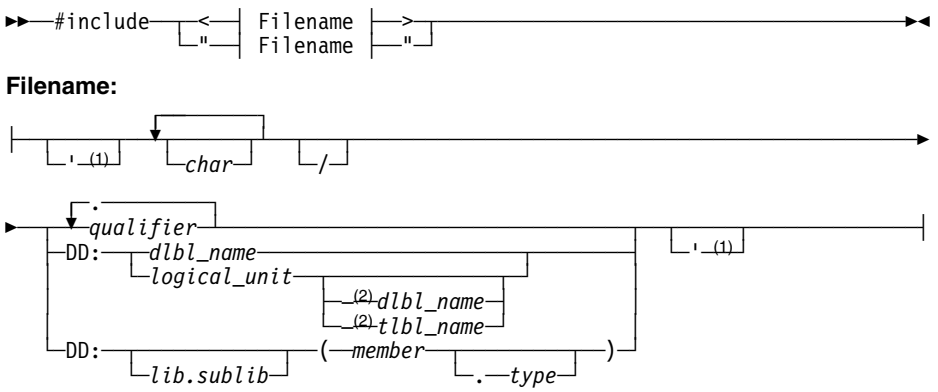
Notes:

1. The JCL information for the compiler workfiles (`IJSYS01`-`IJSYS07`) and `SYSLNK`, if previously added to the System Standard or Partition Standard label area, need not be specified in the job stream.
2. The `DECK` option is specified to indicate to the compiler to write the object deck to `SYSPCH`. `SYSPCH` is assigned to a direct-access device.
3. The `NAME` option is specified to indicate to the compiler to generate a `CATALOG` control statement.

Using Include Files

The `#include` preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your C program.

A description of the `#include` directive is given in the *C/VSE Language Reference*. Its syntax is:



Notes:

1. The single quotation marks must be matched; if you use one, you must use the other.
2. If both *logical_unit* and *dlbl_name* or *tlbl_name* are specified, a – (dash) must be used as separator.

where:

char
: Specifies any alphanumeric character.

dlbl_name
: Specifies the name of a JCL `DLBL` statement.

lib.sublib
: Specifies a valid VSE Librarian sublibrary name.

logical_unit
: Specifies either a system logical unit (for example, `SYSPCH`) or a programmer logical unit (`SYS000` to `SYS254`).

member
: Specifies a VSE Librarian sublibrary member name.

qualifier
: Specifies a 1- to 8-character name. One or more qualifiers joined by periods make up a VSE file ID.

tlbl_name
: Specifies the name of a JCL `TLBL` statement.

type
: Specifies a VSE Librarian sublibrary member type.

No spaces are allowed within the filename specification.

The angle brackets are used to specify system include files, and double quotation marks are used to specify user include files. The difference between using these two formats is in the way that the files are searched for. See “Search Sequences for Include Files” on page 7 for additional information.

When you use the #include directive, you must be aware of:

- The file-naming conversions performed by C/VSE as detailed in "Include Filename Conversion."
- The search order used by C/VSE to locate the file (known as the *library search sequence*). See "Search Sequences for Include Files" on page 7 for more information on the library search sequence.
- The area of the input record containing sequence numbers when you are including files with different record formats. See the *C/VSE Language Reference* for more information on #pragma sequence.

Include Filename Conversion

C/VSE performs filename conversions in the following order:

1. If the filename specification does not appear within single quotation marks, the compiler does the following:
   a. Filenames are stripped from left to right of all characters up to and including the rightmost / (slash).
   b. All underscores (_) that appear in filename specifications are replaced by an @.
   c. All lowercase letters are converted to uppercase.
   d. If a DD: format of the #include directive is used, the compiler attempts to use the file associated with the specified sublibrary member, or the DLBL/TLBL-name and/or the logical unit specified.
   e. If a DD: format of the #include directive is not used, the following is performed:
      1) If only one qualifier is specified, the second qualifier defaults to H.
      2) If a VSE Librarian sublibrary is indicated by the SEARCH or LSEARCH options, the first two qualifiers are used as the member name and member type, while the remaining qualifiers are ignored. Otherwise, if a sequential file is indicated by the SEARCH or LSEARCH options, all qualifiers are used as the sequential filename.

      Notes:

      a) If you specify the CHECKOUT(PPTRACE) compile-time option, a message will be issued stating what include files the preprocessor is looking for.
      b) Member types for the #include directive should not use system names (for example, PHASE or OBJ).
2. If the filename specification appears within single quotation marks, the compiler does the following:
   - None of the VSE Librarian sublibraries are searched.
   - If a filename is given, the compiler attempts to open the file using the filename exactly as specified.

Table 3 on page 7 gives the format of the filename as specified on a #include directive in a source file, and the actual filename used when C/VSE attempts to locate and open the file.

Note: When you use single quotation marks inside the #include directive, or use the DLBL/TLBL-name and/or logical unit format, a library search will not be performed. If the file is not found, the compiler will make no attempt to locate it in the VSE Librarian sublibraries, or any sequential files, specified by the SEARCH or LSEARCH compile-time options, or in any of the sublibraries specified on the // LIBDEF SOURCE JCL statement.

Table 3. Examples of #include Filename Specifications

Example #	#include Directive	Resulting Filename
	Comments	
1	`#include "'USER1.SRC.MYINCS'"`	USER1.SRC.MYINCS
	A library search will not be performed when single quotation marks are used.	
2	`#include <'COMIC/BOOK.OLDIES.K'>`	COMIC/BOOK.OLDIES.K
	C/VSE will attempt to open a file called COMIC/BOOK.OLDIES and will fail because it is not a valid VSE filename. A library search will not be performed when single quotation marks are used.	
3	`#include "sys/abc/xx"`	XX.H
4	`#include "Sys/ABC/xx.h"`	XX.H
5	`#include <sys/name_1>`	NAME@1.H
6	`#include <Name2/App1.App2>`	APP1.APP2
7	`#include <DD:PLAN.SUBLIB(YEAREND.H)>`	YEAREND.H in sublibrary PLAN.SUBLIB
	The VSE Librarian member named YEAREND.H of the sublibrary PLAN.SUBLIB will be used. A library search will not be performed when a DD: format is used.	

Search Sequences for Include Files

With C/VSE, you can specify a search path for locating secondary input files. The two methods for specifying the search path are:

1. Using the SEARCH and LSEARCH compile-time options
2. Using the // LIBDEF statement in your JCL stream

You can use either of these methods to search any VSE Librarian sublibrary. When both of these methods are used at the same time, the sublibraries specified in the compile-time option are searched first.

If a system include file is not found in the sublibraries specified by the SEARCH option, the sublibraries on the // LIBDEF SOURCE JCL statement are searched.

If a user include file is not found in the sublibraries specified by the LSEARCH option, the sublibraries on the // LIBDEF SOURCE JCL statement are searched. If the user include file is not found in any of the sublibraries, the compiler searches the sublibraries specified by the SEARCH option.

The example below shows an excerpt from a JCL stream that compiles a C program:

```
  ⋮
// LIBDEF SOURCE,SEARCH=(JONES.ABC,JONES.DEF)
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='SEARCH(BB.D,BB.F),LSEARCH(CC.X)'
  ⋮
/*
```

The search sequence resulting from the preceding JCL statements is as follows:

Order of Search	For System Include Files	For User Include Files
First	BB.D	CC.X
Second	BB.F	JONES.ABC
Third	JONES.ABC	JONES.DEF
Fourth	JONES.DEF	BB.D
Fifth		BB.F

Because the topic of JCL statements goes beyond the scope of this book, only simple examples will be used. For complete details on JCL, refer to “Related Publications” on page 115.

Chapter 2. Prelinking

This chapter describes when to use the LE/VSE prelinker. The LE/VSE prelinker combines the object modules that form a C application into a single object module that can be link-edited or loaded for execution. For information about how to use the prelinker, including prelinker options, see the *LE/VSE Programming Guide*. For information on link-editing object modules, refer to Chapter 3, “Linking and Running” on page 11.

Prelinking a C Application

The LE/VSE prelinker combines the object modules that form a C application and produces a single object module that can then be link-edited or loaded for execution. The prelinker must be used when:

- C source is compiled with the `RENT` compile-time option
- C source is compiled with the `LONGNAME` compile-time option

For more information about the `RENT` and `LONGNAME` compile-time options, see Chapter 4, “Compile-Time Options” on page 17.

For object modules from applications compiled with the `RENT` compile-time option, the prelinker:

- Combines writable static initialization descriptors
- Maps writable static storage
- Removes writable static name and relocation information

For object modules from applications compiled with the `LONGNAME` compile-time option, the prelinker maps L-names to S-names on output. L-names are mixed-case external names, of up to 255 characters in length, put out by the compiler when compiling with the `LONGNAME` option. S-names are eight character, single-case external names, put out by the compiler when compiling with the `NOLONGNAME` option.

Note: You can exclude object modules that do not refer to writable static or L-names during the prelink step. LE/VSE run-time library functions are not included as part of automatic library calls. This omission can result in warning messages about unresolved references to C library functions or C library objects, but any unresolved C library functions or objects will be resolved in a following link-edit step.

The following shows an excerpt from a compile-prelink-link job which shows the JCL required to execute the prelinker (EDCPRLK):

```
  ⋮
// OPTION LINK
 PHASE phase_name,*
// EXEC PGM=EDCCOMP,SIZE=EDCCOMP,PARM='RENT'
#include <stdio.h>

main() {
  ⋮
 /* C source code */
  ⋮
}
/*
// EXEC PGM=EDCPRLK,SIZE=EDCPRLK,PARM='prelinker_options'
/*
// EXEC PGM=LNKEDT
  ⋮
```

Figure 3. Sample JCL to Execute the Prelinker

For additional information about the prelinker, see the *LE/VSE Programming Guide*.

Chapter 3. Linking and Running

This chapter gives an overview of how to link and run a program in batch. Running a C program in a CICS environment is described in the *LE/VSE C Run-Time Programming Guide*.

If your application is compiled with the RENT or LONGNAME compile-time options, you will have to use the LE/VSE prelinker before linking your application. For information on the LE/VSE prelinker, refer to the *LE/VSE C Run-Time Programming Guide*.

LE/VSE provides a common run-time environment for C, COBOL, and PL/I. For detailed instructions on linking and running existing and new C/VSE programs under LE/VSE, refer to the *LE/VSE Programming Guide*.

The following example shows how to link and run an existing object module using the default run-time options. See the *LE/VSE Programming Guide* for additional information, including how to override the default run-time options.

EDCXUAAF

```
// JOB     EDCXUAAF
// LIBDEF  OBJ,SEARCH=(user.sublib,PRD2.SCEEBASE)
// LIBDEF  PHASE,SEARCH=(PRD2.SCEEBASE)
// OPTION  LINK
   INCLUDE objmod
// EXEC    LNKEDT
// EXEC
/*
/&
```

Figure 4. Linking and Running Using Default Run-Time Options

Note: Information on LE/VSE is reproduced here for convenience only. For detailed information on LE/VSE, please refer to your LE/VSE manuals.

Library Routine Considerations

LE/VSE consists of one run-time component that contains all LE-enabled languages, such as C, COBOL, and PL/I.

LE/VSE is said to be *dynamic*. That is, many of the functions available in C/VSE are not physically stored as a part of your executable program. Instead, only a small portion of code known as a stub routine is actually stored with your executable program and this results in a smaller executable module size. The stub routines contain code that branches to the dynamically loaded LE/VSE routine.

Creating an Executable Program

The linkage editor processes your compiled program (object module) and readies it for execution. The processed object module becomes an executable phase which is stored in a VSE Librarian sublibrary and can be retrieved for execution at any time.

The input to the linkage editor can include object modules and control statements that specify how the input is to be processed. The output from the linkage editor can be a single phase, or multiple phases (generated by using the `PHASE` linkage editor control statement).

The following diagram shows the basic link-editing process for a C program. For more information on using linkage editor control statements, see "Related Publications" on page 115.

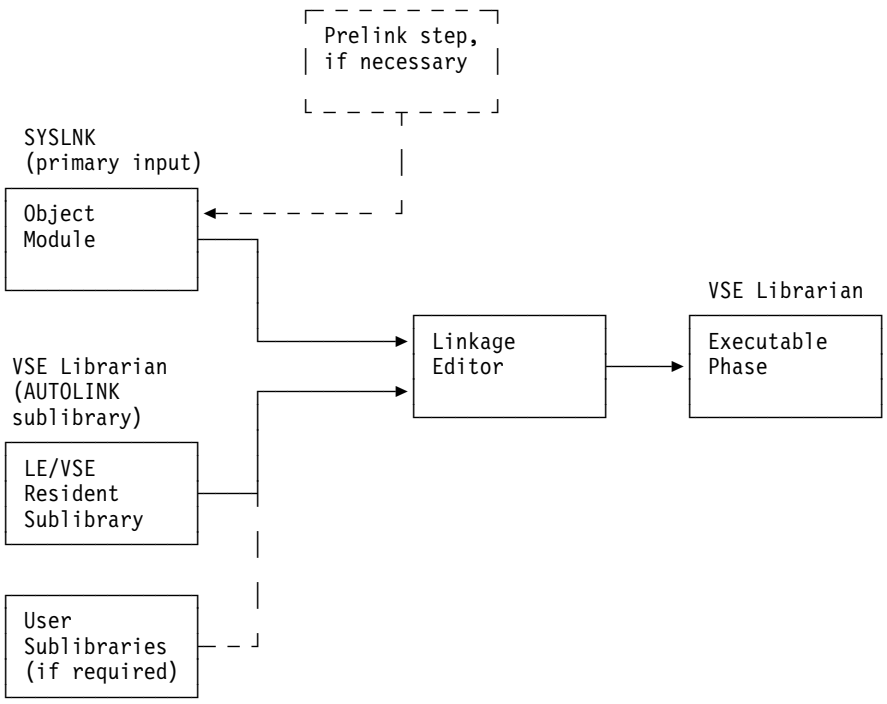


Figure 5. Basic Linkage Editor Processing

Note: The *LE/VSE resident sublibrary* shown in Figure 5 represents the library (or libraries) containing the *resident* category of LE/VSE library routines. These are linked with your application and include such things as initialization routines and callable service stubs. For more information on *resident* library routines, see the *LE/VSE C Run-Time Programming Guide*.

A typical sequence of job control statements for link-editing an object module into a phase is shown in Figure 6 on page 13. The `PHASE` linkage editor control statement in the figure specifies that the link-edited phase is to have the name PROGRAM1. The `// LIBDEF PHASE` and `// OPTION CATAL` job control statements specify that the link-edited phase is to be cataloged in the sublibrary *`user.runlib`*.

EDCXUAAG

```
// JOB     EDCXUAAG
// LIBDEF  OBJ,SEARCH=(user.objlib,PRD2.SCEEBASE)
// LIBDEF  PHASE,CATALOG=(user.runlib)
// OPTION  CATAL
   ACTION  MAP
   PHASE   PROGRAM1,*
   INCLUDE PROGRAM1
// EXEC    LNKEDT,PARM='MSHP'
/*
/&
```

Figure 6. Creating a Phase

Reentrancy in C/VSE

Reentrancy allows more than one job to share a single copy of a phase or to repeatedly use a phase without reloading it.

Reentrant programs can be categorized by their:

- Natural reentrancy; programs that contain no writable static and do not require additional processing to make them reentrant
- Constructed reentrancy; programs that contain writable static and require additional processing to make them reentrant

Writable static is storage that changes and is maintained throughout program execution. It is made up of:

- All program variables with the `static` storage class
- All program variables receiving the `extern` storage class
- All writable strings

Note: If your program contains no writable strings and none of your `static` or `extern` variables are updated in your application (that is, they are read-only), your program is naturally reentrant.

If your program contains writable static, you must use the LE/VSE prelinker to make your program reentrant. This utility concatenates compile-time initialization information (for writable static) from one or more object modules into a single initialization unit. In the process, the writable static part is mapped. If your program does not contain any writable static, you may not need to use the prelinker. For more information on the prelinker, refer to the *LE/VSE C Run-Time Programming Guide*.

To generate a reentrant phase, you must follow these steps:

1. If your program contains writable static, you must compile all your C source files using the `RENT` compile-time option.

   If you are unsure about whether your program contains writable static, compile it with the `RENT` option. Invoking the prelinker with the `MAP` option and the object module as input produces a prelinker map. Any writable static data in the object module appears in the writable static section of the map. For more

information on using the prelinker, refer to the *LE/VSE C Run-Time Programming Guide*.

2. Use the prelinker to combine all input object modules into a single output object module.

   Note: The output object module cannot be used as further input to the prelink utility.

3. Link the program in the usual manner.
4. Optionally, install the program in the SVA area of the system.

   You do not need to install your program in the SVA to run it but if you do not, you will not gain all the benefits of reentrancy.

Linking Modules for Interlanguage Calls

For information on link-editing modules for interlanguage calls, refer to the *LE/VSE Programming Guide*.

Running a C/VSE Program

The following sections describe how to run a program in batch using LE/VSE. Running a C program in a CICS environment is described in the *LE/VSE C Run-Time Programming Guide*.

Note: Information on LE/VSE is reproduced here for convenience only. For detailed information on LE/VSE, please refer to your LE/VSE manuals.

Running an Application

You can request the execution of a phase in an `EXEC` statement in your JCL. The general form of the `EXEC` statement is:

```
// EXEC [PGM=]program_name,SIZE=program_size
```

where *program_name* is the name of the phase of the application to be executed and *program_size* is the amount of program storage required to run the application.

Note: The amount of program storage required to run an application does not include the storage required for LE/VSE heap and stack storage, LE/VSE library routines, or dynamically loaded application routines. Program storage is generally only used to load the main program and, if required, the SORT product (and its work areas). LE/VSE uses partition `GETVIS` storage for all other storage requirements. LE/VSE requires a minimum of 1200KB below-the-line `GETVIS` storage.

Specifying Run-Time Options

Each time your application runs, a set of run-time options must be established. These options determine many of the properties of how the application runs, including its performance, error handling characteristics, storage management, and production of debugging information. You can specify run-time options in any of the following places:

- In the CEEDOPT CSECT, where the installation default options are located (see the *LE/VSE Programming Guide* for more information)
- In the CEEUOPT CSECT where user-supplied default options are located (see the *LE/VSE Programming Guide* for more information)
- `#pragma runopts` in C source code (see the *LE/VSE Programming Guide* for more information)
- In the `PARM` parameter of the `EXEC` statement in your JCL (see below)
- In the assembler user exit (see the *LE/VSE Programming Guide* for more information)

Specifying Run-Time Options in the EXEC Statement

You can override installation default run-time options specified as overrideable, and any application default run-time options, by specifying run-time options in the `PARM` parameter of the `EXEC` statement. The general form for specifying run-time options in the `PARM` parameter of the `EXEC` statement is:

```
// EXEC [PGM=]program_name,                                          X
               PARM='[run-time options/][program parameters]'
```

For example, if you want to generate a storage report and run-time options report for the application PROGRAM1, specify the following:

```
// EXEC PROGRAM1,PARM='RPTSTG(ON),RPTOPTS(ON)/'
```

The run-time options that are passed to the main routine must be followed by a slash (/) to separate them from program parameters. For HLL considerations to keep in mind when specifying run-time options, see the *LE/VSE Programming Guide*.

The `EXECOPS` option for C is used to specify that run-time options passed as parameters at execution time are to be processed by LE/VSE. The option `NOEXECOPS` specifies that run-time options are not to be processed from execution parameters and are to be treated as program parameters. You can specify either option in a `#pragma runopts` statement in your C program. You can also specify either option as a C/VSE compile-time option. `EXECOPS` is the default. When `EXECOPS` is in effect, you can pass run-time options in the `EXEC` statement in your JCL.

VSE normally limits the size of the string you can specify in the `PARM` parameter of the `EXEC` statement to 100 characters. However, if you are running VSE/ESA Version 2 Release 2, or a previous supported release of VSE/ESA with the appropriate PTF applied, you can use the following technique to specify a parameter string of up to 300 characters:

```
// EXEC [PGM=]program_name,                                          X
               PARM='parameter_string_segment',                      X
               PARM='parameter_string_segment',                      X
               PARM='parameter_string_segment'
```

Using this technique, up to three instances of the `PARM` parameter can be specified on an `EXEC` statement, and each *parameter_string_segment* can be up to 100 characters in length.

The following example shows how you might code an EXEC statement and pass a 140-character string consisting of run-time options and program arguments:

```
// EXEC TESTPGM,PARM='ABTERMENC(ABEND) ALL31(ON) NATLANG(ENU) RPTOPTS(OX
               N) RPTSTG(ON) TERMTHDACT(MSG)/RUNDATE=19961213 DBNA',   X
               PARM='ME=ODBMST TRANFLE=OTRNFLE RPTFLE=ORPTFLE'
```

Chapter 4. Compile-Time Options

This chapter describes the options that you can use to control the compilation of your program using C/VSE.

Specifying Compile-Time Options

Whether or not default settings supplied by IBM have been changed during installation, you can override them when you compile your C program by specifying an option in the following ways:

- In the PARM parameter of the EXEC JCL statement that invokes the compiler.
- In a #pragma options preprocessor directive within your source file. See "Specifying Compile-Time Options Using #pragma options" for details.

  If you use a compile-time option in the PARM parameter of the JCL statement that contradicts the options specified on the #pragma options directive, the compile-time option in the PARM parameter of the JCL statement overrides the options on the #pragma options directive.

If two contradictory options are specified, the last one specified is accepted and the first ignored.

If you use one of the following compile-time options, the option name is inserted at the bottom of your object module indicating that it was used:

GONUMBER
INLINE
NAME
OPTIMIZE (all levels)
RENT
START
TARGET (all levels)
TEST
UPCONV

Specifying Compile-Time Options Using #pragma options

You can use the #pragma options preprocessor directive to override the *default* values for compile-time options. Remember that compile-time options specified in the PARM parameter of the EXEC JCL statement can override compile-time options used in #pragma options. For complete details on the #pragma options preprocessor directive, see the *C/VSE Language Reference*.

The #pragma options preprocessor directive must appear before the first C statement in your input source file. Only comments and other preprocessor directives can precede the #pragma options directive, and only the options listed below can be specified. If you specify a compile-time option that is not given in the following list, the compiler generates a warning message and the option is ignored.

AGGREGATE|NOAGGREGATE
CHECKOUT|NOCHECKOUT
GONUMBER|NOGONUMBER
HWOPTS|NOHWOPTS
INLINE|NOINLINE
NAME|NONAME
OPTIMIZE|NOOPTIMIZE
RENT|NORENT
SPILL
START
TEST|NOTEST
UPCONV|NOUPCONV
XREF|NOXREF

Notes:

1. When you specify conflicting attributes either explicitly or implicitly by the specification of other options, the last explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden.
2. When you specify the SOURCE compile-time option in the PARM parameter of the EXEC JCL statement, your listing will contain an options list indicating the options in effect at invocation. The values in the list are the options specified in the PARM parameter of the EXEC JCL statement or the default options specified at installation. These values do not reflect any options specified in a #pragma options preprocessor directive.

Compile-Time Option Defaults

You can alter the compilation of your program by specifying compile-time options in the PARM parameter of the EXEC JCL statement when you invoke the compiler or when you use the preprocessor directive, #pragma options, in your source program. Options that you specify when you invoke the compiler override installation defaults and can override compile-time options specified in a #pragma options directive.

The compile-time default options supplied by IBM can be changed when C/VSE is installed. To determine the current defaults, submit an empty program for compilation with the SOURCE compile-time option specified. In the listing generated, you can view the options that are in effect at invocation; that is, the settings that result from the interaction of the options specified in the PARM parameter of the EXEC JCL statement and the defaults that were specified at installation. The listing does not reflect options specified in #pragma options in the source file being compiled.

Summary of Compile-Time Options

Most compile-time options have a positive and a negative form. The negative form is the positive with NO before it (as in XREF and NOXREF). Table 4 lists the compile-time options in alphabetical order, with their abbreviations and the IBM-supplied defaults. Suboptions inside square brackets are optional. Table 5 on page 20 summarizes the compile-time options by function.

Table 4 (Page 1 of 3). Compile-Time Options, Abbreviations, and IBM Supplied Defaults

Compile-Time Option	Abbreviated Name	IBM Supplied Default
AGGREGATE|NOAGGREGATE	AGG|NOAGG	NOAGG
CHECKOUT[(*options*)]| NOCHECKOUT[(*options*)]	CHE[(*options*)]|NOCHE[(*options*)]	NOCHE(*options*)
CSECT|NOCSECT	CSE|NOCSE	NOCSE
DECK|NODECK[1]	Not applicable	None (Determined by VSE job control option DECK)
DEFINE(*name1*[=*def1*], *name2*[=*def2*],...)	DEF(*name1*[=*def1*], *name2*[=*def2*],...)	No definitions
EXECOPS|NOEXECOPS	EXEC|NOEXEC	EXEC
EXPMAC|NOEXPMAC	EXP|NOEXP	NOEXP
FLAG(*severity*)|NOFLAG	FL(*severity*)|NOFL	FL(I)

Table 4 (Page 2 of 3). Compile-Time Options, Abbreviations, and IBM Supplied Defaults

Compile-Time Option	Abbreviated Name	IBM Supplied Default
GONUMBER|NOGONUMBER	GONUM|NOGONUM	NOGONUM
HWOPTS(STRing|NOSTRing)|NOHWOPTS	HWO(STR|NOSTR)|NOHWO	NOHWO
INLINE[(*options*)]| NOINLINE[(*options*)]	INL[(*options*)]|NOINL[(*options*)]	NOINL(AUTO,NOREPORT,250,1000)
INFILE(*filename*)|NOINFILE	INF(*filename*)|NOINF	NOINF
LANGLVL(*level*)	LANG(*level*)	LANG(EXTENDED)
LIST|NOLIST	LIS|NOLIS	None (Determined by VSE job control option LISTX)
LOCALE(*name*)|NOLOCALE	LOC(*name*)|NOLOC	NOLOC
LONGNAME|NOLONGNAME	LO|NOLO	NOLO
LSEARCH(*opt1*,*opt2*,...)|NOLSEARCH	LSE(*opt1*,*opt2*,...)|NOLSE	NOLSE (Standard library and/or disk search sequence)
MARGINS(*first*,*last*)|NOMARGINS	MAR(*first*,*last*)|NOMAR	F-format: MAR(1,72) V-format: NOMAR
MEMORY|NOMEMORY	MEM|NOMEM	NOMEM
NAME([*name*])|NONAME	NA([*name*])|NONA	NONA
NESTINC(*limit*)|NONESTINC	NEST(*limit*)|NONEST	NEST(16)
OBJECT|NOOBJECT[1]	Not applicable	None (Determined by VSE job control options LINK or CATAL)
OFFSET|NOOFFSET	OFFSET|NOOF	NOOF
OPTIMIZE[(*n*)]|NOOPTIMIZE	OPT[(*n*)]|NOOPT	NOOPT
PPONLY[(*n*|*)]|NOPPONLY	PP[(*n*|*)]|NOPP	NOPP
RENT|NORENT	RENT|NORENT	NORENT
SEARCH(*opt1*,*opt2*,...)|NOSEARCH	SE(*opt1*,*opt2*,...)|NOSE	NOSE (Standard library and/or disk search sequence)
SEQUENCE(*left*,*right*)|NOSEQUENCE	SEQ(*left*,*right*)|NOSEQ	F-format: SEQ(73,80) V-format: NOSEQ
SHOWINC|NOSHOWINC	SHOW|NOSHOW	NOSHOW
SOURCE|NOSOURCE	SO|NOSO	None (Determined by VSE job control option LIST)
SPILL(*size*)|NOSPILL	SP(*size*)|NOSP	SP(128)
SSCOMM|NOSSCOMM	SS|NOSS	NOSS
START	STA	STA
TARGET([*option*])	TARG([*option*])	TARG()
TERMINAL|NOTERMINAL	TERM|NOTERM	None (Determined by VSE job control option TERM)
TEST[(*options*)]|NOTEST[(*options*)]	TEST[(*options*)]|NOTEST[(*options*)]	NOTEST(SYM,BLOCK,LINE,NOPATH)
UPCONV|NOUPCONV	UPC|NOUPC	NOUPC

Table 4 (Page 3 of 3). Compile-Time Options, Abbreviations, and IBM Supplied Defaults

Compile-Time Option	Abbreviated Name	IBM Supplied Default
XREF|NOXREF	XR|NOXR	None (Determined by VSE job control option XREF)

Notes:

1 These options are specified using the // OPTION JCL statement, the // STDOPT system command in the BG partition, or at IPL of the system only—they cannot be specified in the PARM parameter of the EXEC JCL statement or the #pragma options directive. See the detailed description of each option in the following for the correct specifications.

Table 5 lists the compile-time options according to their function.

Table 5 (Page 1 of 2). Summary of Compile-Time Options by Function

Type of Option	Option	Description
File Management	DECK	Produces an object module associated with SYSPCH.
	INFILE	Specifies the main input source file.
	LSEARCH	Specifies the sublibraries to be scanned for user include files.
	MEMORY	Improves compile-time performance by using a memory file in place of a workfile, if possible.
	OBJECT	Produces an object module associated with SYSLNK.
	SEARCH	Specifies the sublibraries to be scanned for system include files.
Listing	AGGREGATE	Lists structures and unions, and their size.
	EXPMAC	Lists all expanded macros.
	LIST	Lists assembler-like code produced by the compiler.
	OFFSET	Lists offset addresses relative to entry points
	SHOWINC	Lists include files if SOURCE option specified.
	SOURCE	Lists source file.
	XREF	Generates a cross reference listing.
Debug and Diagnostic	CHECKOUT	Gives informational messages for possible programming errors.
	FLAG	Specifies the lowest severity level to be listed.
	GONUMBER	Generates line number tables for Debug Tool for VSE/ESA and error tracebacks.
	TEST	Generates information that is suitable for Debug Tool for VSE/ESA.
	TERMINAL	Directs error messages to SYSLOG.

Table 5 (Page 2 of 2). Summary of Compile-Time Options by Function

Type of Option	Option	Description
Source Code Control	CSECT	Checks for the #pragma csect directives in the source.
	LANGLVL	Specifies the C language standard to be used (ANSI, SAA Level 1, SAA Level 2, or Extended).
	MARGINS	Identifies position of C source to be scanned by the compiler.
	NESTINC	Specifies the number of nested include files to be allowed.
	SEQUENCE	Specifies the columns used for sequence numbers.
	SSCOMM	Allows comments to be specified by two slashes (//).
	UPCONV	Preserves *unsignedness* during C type conversions.
Object Code Control	EXECOPS	Allows run-time options to be passed to your program.
	HWOPTS	Generates code for different hardware features.
	INLINE	Inlines user functions into source and helps maximize optimizations.
	LONGNAME	Provides support for external names of mixed case and up to 255 characters long.
	NAME	Generates either a linkage editor PHASE statement or a VSE Librarian CATALOG statement, depending on specification of JCL // OPTION.
	OPTIMIZE	Improves run-time performance.
	RENT	Generates reentrant code.
	SPILL	Specifies the maximum spill area.
	START	Generates an ENTRY CEESTART control statement.
	TARGET	Generates an object module for the targeted operating system.
Preprocessor	DEFINE	Defines preprocessor macro names.
	LOCALE	Specifies a locale to be used at compile time.
	PPONLY	Specifies that only the preprocessor is to be run and not the compiler.

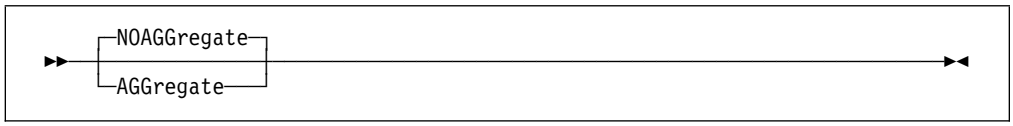
Description of Compile-Time Options

The following sections list the compile-time options alphabetically and describe the compile-time options and their usage.

AGGREGATE|NOAGGREGATE

The AGGREGATE option specifies whether the compiler is to include in the compiler listing a layout of all variables of the type `struct` or `union`.

Syntax for the AGGREGATE option is:

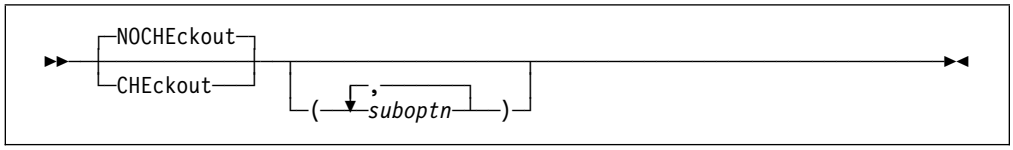


When AGGREGATE is specified, two maps are created: one contains the packed layout and the other contains the unpacked layout. Each layout map contains the offsets and lengths of the structure and union members. One map is generated for each `struct` or `union` tag, or if no tag is specified, a map is generated for the variable name specified on the `struct` or `union` declaration.

CHECKOUT|NOCHECKOUT

The CHECKOUT option specifies that the compiler is to give informational error messages indicating possible programming errors. They can help C programmers in debugging their programs.

Syntax for the CHECKOUT option is:



where *suboptn* is one of the suboptions shown in Table 6.

You can specify CHECKOUT with or without suboptions. If you include suboptions, you can specify any number with commas between them. However, if you do not include suboptions, the defaults are obtained from your installation defaults. The diagnostic messages put out from CHECKOUT start at EDC0800 with the addition of some compile-time messages, such as, EDC0464 and EDC0244, listed in Appendix A, “C/VSE Return Codes and Messages” on page 65.

Table 6 (Page 1 of 2). CHECKOUT *Suboptions, Abbreviations, and Descriptions*

CHECKOUT **Suboption**	**Abbreviated Name**	**Description**
ACCURACY|NOACCURACY	AC|NOAC	Assignments of `long` values to variables that are not `long`
ENUM|NOENUM	EN|NOEN	Usage of enumerations
EXTERN|NOEXTERN	EX|NOEX	Unused variables that have external declarations
GENERAL|NOGENERAL	GE|NOGE	General checkout messages
GOTO|NOGOTO	GO|NOGO	The appearance and usage of `goto` statements
INIT|NOINIT	I|NOI	Variables that are not explicitly initialized
PARM|NOPARM	PAR|NOPAR	Function parameters that are not used

Table 6 (Page 2 of 2). CHECKOUT *Suboptions, Abbreviations, and Descriptions*

CHECKOUT **Suboption**	**Abbreviated Name**	**Description**
PORT|NOPORT	POR|NOPOR	Nonportable usage of the C language
PPCHECK|NOPPCHECK	PPC|NOPPC	All preprocessor directives
PPTRACE|NOPPTRACE	PPT|NOPPT	The tracing of include files by the preprocessor
TRUNC|NOTRUNC	TRU|NOTRU	Variable names that are truncated by the compiler
ALL	ALL	Turns on all of the suboptions for CHECKOUT
NONE	NONE	Turns off all of the suboptions for CHECKOUT

Note: Underscoring indicates the default CHECKOUT suboptions.

You can also turn the CHECKOUT option off for certain files or statements of your source program by using a #pragma checkout(suspend) directive. See the *C/VSE Language Reference* for more information regarding this #pragma directive.

The CHECKOUT|NOCHECKOUT option can be specified in the PARM parameter of the EXEC JCL statement and in the #pragma options preprocessor directive. When both methods are used concurrently, the options are merged. If an option in the PARM parameter of the EXEC JCL statement conflicts with an option in the #pragma options directive, the option in the PARM parameter of the EXEC JCL statement takes precedence. The examples below illustrate these rules:

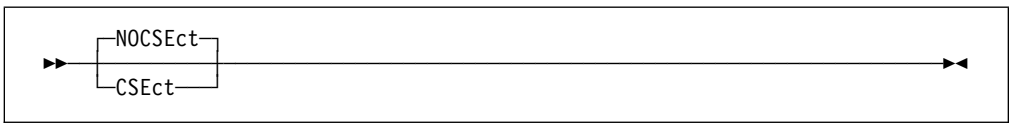
```
Source file:   #pragma options(NOCHECKOUT(NONE,ENUM))
EXEC PARM:     CHECKOUT(GOTO)
Result:        CHECKOUT(NONE,ENUM,GOTO)

Source file:   #pragma options(NOCHECKOUT(NONE,ENUM))
EXEC PARM:     CHECKOUT(ALL,NOENUM)
Result:        CHECKOUT(ALL,NOENUM)
```

The NOCHECKOUT option specifies that the compiler is not to generate informational error messages. Suboptions specified in a #pragma options(NOCHECKOUT(*subopts*)) directive will apply if CHECKOUT is specified in the PARM parameter of the EXEC JCL statement.

CSECT|NOCSECT

The CSECT option ensures that the object module contains named CSECTs. The compiler checks that both the #pragma csect(CODE,...) and #pragma csect(STATIC,...) directives are present in the source code. If you specify the CSECT option, an error message is given if either directive is missing.

Syntax for the CSECT option is:



DECK|NODECK

The DECK option specifies whether or not the compiler is to produce an object module to SYSPCH. The object module produced is written to both SYSPCH and SYSLNK if both DECK and OBJECT are specified.

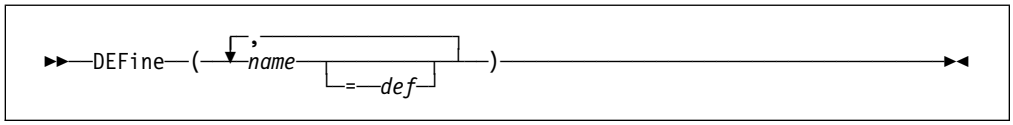
This option cannot be specified in the PARM parameter of the EXEC JCL statement or in a #pragma options processor directive. The DECK compile-time option is specified using the DECK job control option on the JCL // OPTION statement.

If you use the DECK option, ensure that SYSPCH is assigned in the JCL for your compilation.

DEFINE

The DEFINE option defines preprocessor macros that take effect before the file is processed by the compiler.

Syntax for the DEFINE option is:



The option can be used repeatedly. Each definition has the form *name = def*. Defining a macro in this way is equivalent to writing the following preprocessor directive in your C source program:

```
#define name def
```

Notice that

```
DEFINE(name)
```

with no definition is equivalent to

```
#define name 1
```

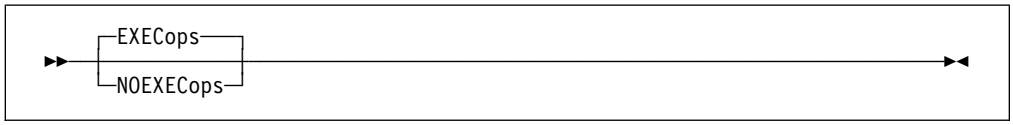
Note: There is no DEFINE compile-time option equivalent of function-like macros that take parameters such as:

```
#define max(a,b) ((a)>(b)?(a):(b))
```

EXECOPS|NOEXECOPS

The EXECOPS option allows you to control whether run-time options will be recognized at run time without changing your source code. It is equivalent to including a #pragma runopts(EXECOPS) directive in your source code. You can control whether run-time options are recognized at run time without changing your source code.

Syntax for the EXECOPS option is:

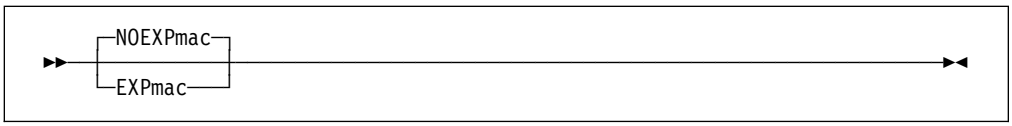


If this option is specified in both the PARM parameter of the EXEC JCL statement and in a #pragma runopts directive, the option in the PARM parameter of the EXEC JCL statement takes precedence.

EXPMAC|NOEXPMAC

The EXPMAC option specifies whether the compiler is to show all expanded macros in the source listing.

Syntax for the EXPMAC option is:

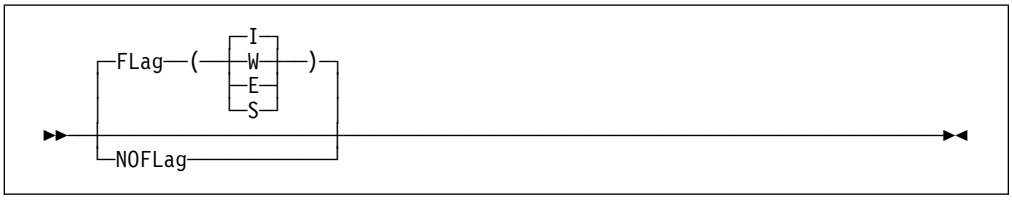


If you want to use the EXPMAC option, you must also specify the SOURCE compile-time option to generate a source listing. If you specify the EXPMAC option but omit the SOURCE option, a warning message is generated and no source listing is produced.

FLAG|NOFLAG

The FLAG option specifies the minimum severity level for which you want notification.

Syntax for the FLAG option is:



You specify the minimum severity level using the FLAG(*severity*) compile-time option, where *severity* is one of the following letters:

I An informational message that may be of interest to you is generated by the compiler. This is the default.

Note: When FLAG(I) is specified, all messages with severity I, W, E, and S are included in the compiler listing.

W A warning message that calls attention to a possible error, although the statement to which it refers is syntactically valid.

Note: When FLAG(W) is specified, all messages with severity W, E, and S are included in the compiler listing.

E An error message that shows that the compiler has detected an error and cannot produce an object deck.

Note: When FLAG(E) is specified, all messages with severity E and S are included in the compiler listing.

S A severe error message describing an error that forces termination of the compilation.

Note: When FLAG(S) is specified, only messages with severity S are included in the compiler listing.

Messages generated by the compiler appear at the end of the compiler listing as well as immediately following the source line in error. See Appendix A, "C/VSE Return Codes and Messages" on page 65 for a list of the messages. Table 7 explains the relationship between the type of messages, return code, and severity level.

Table 7. Selecting the Lowest Severity of Messages to Be Printed Using the FLAG Option

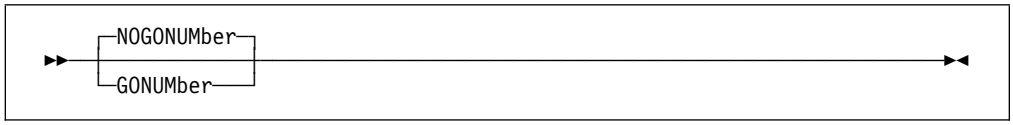
Type of Message	Compile-Time Option	Numeric Severity Level	Return Code
Information	FLAG(I)	00	0
Warning	FLAG(W)	10	4
Error	FLAG(E)	30	12
Severe error	FLAG(S)	> 30	16

The NOFLAG option is equivalent to FLAG(I) and specifies that the compiler is to show all messages.

GONUMBER|NOGONUMBER

The GONUMBER option specifies whether the compiler is to generate line number tables corresponding to the input source file. These tables are for use by Debug Tool for VSE/ESA and for error tracebacks.

Syntax for the GONUMBER option is:



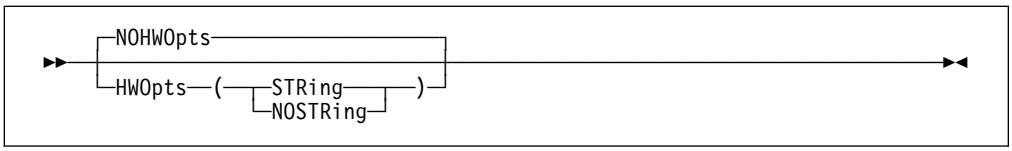
This option is implicitly turned on when the TEST compile-time option is used.

Note: Whenever you specify the GONUMBER compile-time option, a comment noting its use is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an END card at the end of the object module.

HWOPTS|NOHWOPTS

The HWOPTS option specifies whether the compiler is to generate code to take advantage of different hardware.

Syntax for the HWOPTS option is:



The suboptions are:

STRING Create code for hardware that has Logical String Assist (LSA). On such hardware, built-in functions will have better performance if you select this option.

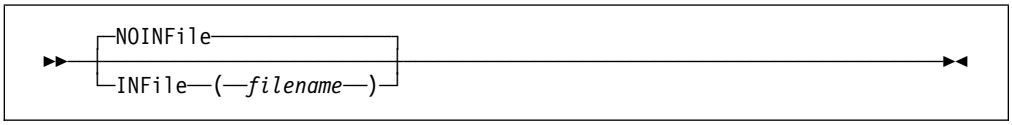
NOSTRING Create code for hardware that does not have LSA.

Note: Whenever you specify the HWOPTS compile-time option, a comment is put in your object module. The comment generated consists of the specified compile-time option on an END card at the end of the object module.

INFILE|NOINFILE

The INFILE option specifies the primary input source file to the compiler. If the option is omitted, the primary input source file is SYSIPT.

Syntax for the INFILE option is:



For details on the specification of *filename*, refer to the sections on using fopen() or freopen() in the chapters dealing with the various types of file I/O in the *LE/VSE C Run-Time Programming Guide*.

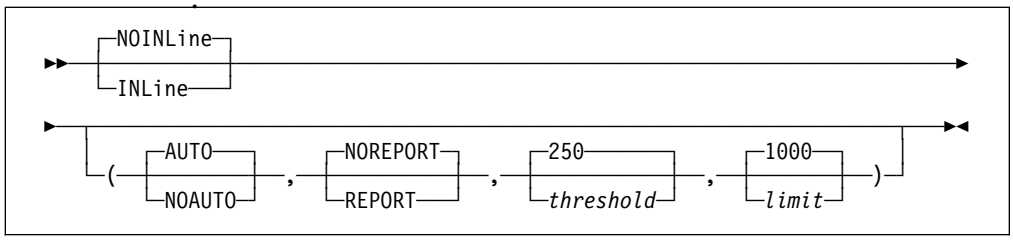
If the input file is a SAM file (excluding VSAM-managed SAM), the file must contain fixed-length 80-byte records and the block size must be less than or equal to 8000.

If the input file is a VSAM file (including VSAM-managed SAM), the file can be any record format.

INLINE|NOINLINE

The INLINE compile-time option allows you to specify that the compiler should place the code for the function at the point of call; this is called *inlining* and it eliminates the linkage overhead. Only use the INLINE compile-time option when you are in the final stages of preparing your application for production. For more information on optimization and the INLINE option, refer to the section “Optimizing Code” in the *LE/VSE C Run-Time Programming Guide*.

Syntax for the INLINE option is:



You can specify just INLINE if you want to use the defaults. You must include a comma between each suboption even if you want to use the default for one of the suboptions. The suboptions must be specified in the following order:

AUTO|NOAUTO
: The inliner will run in automatic mode and will inline functions within the *threshold* and *limit*.

  If NOAUTO is specified, the inliner will only inline those functions specified with the #pragma inline directive. The #pragma inline and noinline directives allow you to determine which functions are to be inlined and which are not when the INLINE option is specified. These #pragmas have no effect if NOINLINE is specified. See the *C/VSE Language Reference* for more information on #pragma directives.

REPORT|NOREPORT
: An inline report is part of the listing file, consisting of:

  - An inline summary
  - A detailed call structure

  For more information on the inline report, see "Inline Report" on page 55.

threshold
: The maximum relative size of a function to inline. The default for *threshold* is 250 Abstract Code Units (ACU) instructions. ACUs are proportional in size to the executable code in the function; your C code is translated into ACUs by C/VSE. The maximum *threshold* is INT_MAX, as defined in the header file limits.h. Specifying a threshold of 0 is equivalent to specifying NOAUTO.

limit
: The maximum relative size a function can grow before auto-inlining stops. The default for *limit* is 1000 ACUs for that function. The maximum for *limit* is INT_MAX, as defined in the header file limits.h. Specifying a limit of 0 is equivalent to specifying NOAUTO.

The INLINE|NOINLINE option can be specified in the PARM parameter of the EXEC JCL statement and in the #pragma options preprocessor directive. When both methods are used concurrently, the options are merged. If an option in the PARM parameter of the EXEC JCL statement conflicts with an option in the #pragma options directive, the one in the PARM parameter of the EXEC JCL statement takes precedence. For example, because you typically do not want to inline functions while developing a program, you can specify the NOINLINE option in a #pragma options preprocessor directive. When you do want to inline functions, you can override the NOINLINE option by specifying INLINE in the PARM parameter of the EXEC JCL statement rather than by editing the source program.

The example below illustrates these rules:

```
Source file:   #pragma options(NOINLINE(NOAUTO,NOREPORT,,2000))
EXEC PARM:     INLINE(AUTO)
Result:        INLINE(AUTO,NOREPORT,250,2000)
```

Note: Whenever you specify the `INLINE` compile-time option, a comment, with the values of the suboptions, is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an `END` card at the end of the object module.

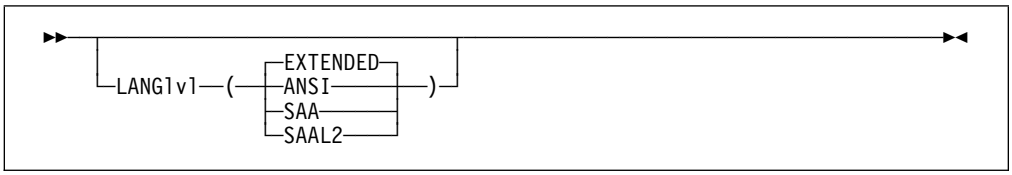
The compile-time option `MEMORY` is ignored if specified with `INLINE`. If you specify `TEST` with `INLINE`, the `INLINE` option is ignored.

If you specify `NOINLINE`, no functions are inlined even if you have `#pragma inline` directives in your code.

LANGLVL

The `LANGLVL` option defines a macro that specifies a language level. You must then include this macro in your code to force conditional compilation (for example, with the use of `#ifdef` directives). In this way, you can write portable code if you correctly code the different parts of your program according to the language level. The macro is used in preprocessor directives in header files.

Syntax for the `LANGLVL` option is:



where:

`LANGLVL(ANSI)` Indicates language constructs defined by ANSI. See the *ANSI C Standard* (X3.159-1989) for more information.

Some non-ANSI stub routines will exist even if you specify `LANGLVL(ANSI)` for compatibility with previous releases. See the *LE/VSE C Run-Time Library Reference* for more information on ANSI compatibility.

`LANGLVL(SAA)` Indicates language constructs defined by SAA. See the *C/VSE Language Reference* for more information.

`LANGLVL(SAAL2)` Indicates language constructs defined by SAA Level 2. See the *C/VSE Language Reference* for more information.

`LANGLVL(EXTENDED)` Indicates all language constructs available with C/VSE. This is the default.

Note: Even if `LANGLVL(ANSI)` is specified, the compiler is still able to read and analyze the `_Packed` keyword. If you want to make your code purely ANSI, then you should redefine `_Packed` in a header file like this:

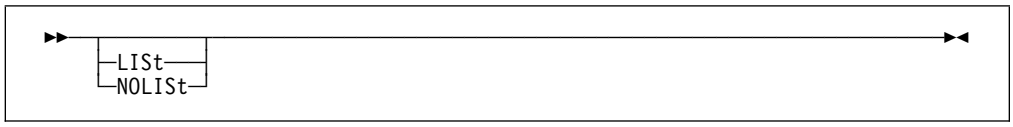
```
#ifdef __ANSI__
  #define _Packed
#endif
```

The compiler will then see the _Packed attribute as a blank when LANGLVL(ANSI) is specified at compile time, and the language level of the code will be ANSI.

LIST|NOLIST

The LIST option specifies whether the compiler is to include a pseudo-assembler language representation of the object module in the compiler listing.

Syntax for the LIST option is:



The LIST option may also be specified using the LISTX job control option on the JCL // OPTION statement.

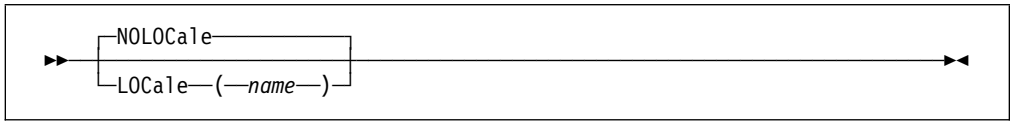
The default is determined by the LISTX job control option.

Note: Usage of information such as registers, pointers, data areas, and control blocks shown in the assembly listing are not programming interface information.

LOCALE|NOLOCALE

The LOCALE option indicates the locale to be used by the compiler as the current locale throughout the compilation unit.

Syntax for the LOCALE option is:



The suboption *name* indicates the name of the locale to be used by the compiler. If the suboption is omitted, the default current locale in the environment is used. If the suboption does not represent a valid locale name, the LOCALE option is ignored and NOLOCALE is assumed.

The negative form NOLOCALE shows that the compiler only recognizes and uses the default code page, which is IBM-1047.

The LOCALE|NOLOCALE option cannot be used in the #pragma options directive. It can only be specified in the PARM parameter of the EXEC JCL statement.

If the LOCALE option is selected, the locale name and the associated code set appear in the header of the listing. A locale name is also generated in the object module.

The formats of the time and the date in the compiler-generated listing file are controlled by the LC_TIME category of the current locale. The identifiers appearing in the Cross Reference table and the Inline report in the listing file produced by the compiler when the XREF, AGGREGATE, and INLINE options are in effect, are sorted as specified by the LC_COLLATE category of the locale specified in the option.

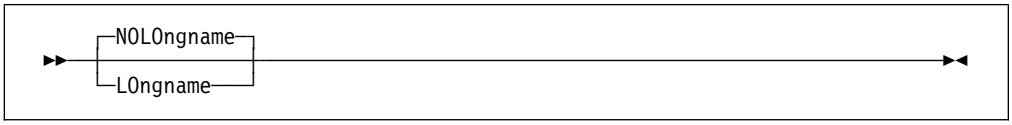
Note: The formats of the predefined macros __DATE__, __TIME__, and __TIMESTAMP__ are not locale sensitive.

For more information on locales, refer to the *LE/VSE C Run-Time Programming Guide*.

LONGNAME|NOLONGNAME

The LONGNAME option specifies that the compiler is to generate untruncated (long) and mixed case external names in the object module produced by the compiler. These names may be up to 255 characters in length. The format of long external names in object modules is not recognized by the system linkage editor. If you use the LONGNAME option, you must use the prelinker, described in the *LE/VSE Programming Guide*.

Syntax for the LONGNAME option is:



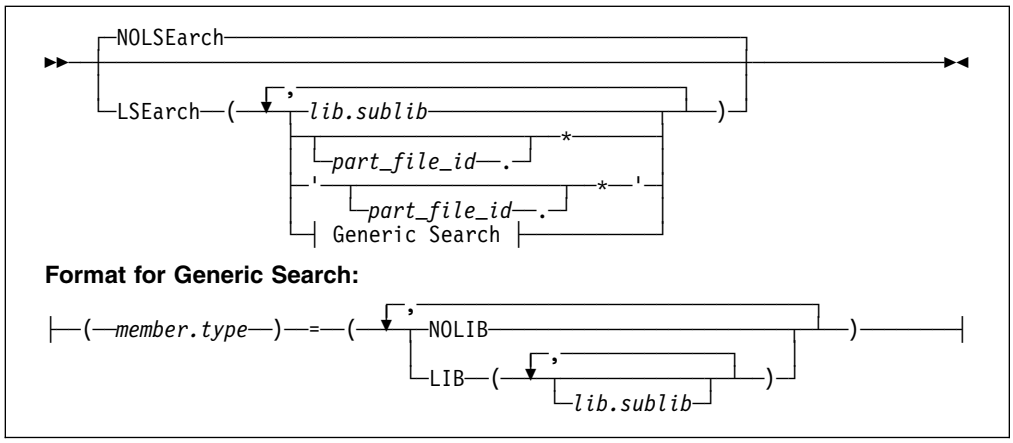
If #pragma map is used to associate an external name with an identifier, the external name is generated in the object module. That is, #pragma map will have the same behavior for the LONGNAME and NOLONGNAME compile-time option. Also, #pragma csect will have the same behavior for the LONGNAME and NOLONGNAME compile-time option.

LSEARCH|NOLSEARCH

The LSEARCH option directs the preprocessor to look for the user include files in the VSE Librarian sublibraries, or in the sequential disk files specified. User include files are files associated with the #include "*filename*" format of the #include C preprocessor directive. See “Using Include Files” on page 4 for a description of the #include preprocessor directive.

For further information on sublibrary search sequences, see “Search Sequences for Include Files” on page 7.

Syntax for the LSEARCH option is:



Format for Generic Search:

where:

lib.sublib
Specifies the name of a VSE Librarian sublibrary containing user include files. The library name (*lib*) is from 1 to 7 characters and the sublibrary name (*sublib*) is from 1 to 8 characters.

part_file_id
Specifies a partial file ID of a sequential file containing a user include file. The full file ID is the 1 to 44 character external name for the sequential file as coded on the `// DLBL` job control statement that defines the file. A period and asterisk (.*) must be appended immediately after *part_file_id*. If no partial file ID is required, specify an asterisk only as the option.

The full file ID is made up as follows:

- For a partial file ID not within single quotes, the jobname followed by a period is added to the start of the partial file ID. For example:

  ```
  LSEARCH(ABC.DEF.*)
  ```

  will create a full file ID of:

  jobname`.ABC.DEF.`*filename*

  where *filename* is the file name on the `#include` directive.

- For a partial file ID specified within single quotes, the jobname is not added to the start of the partial file ID. For example:

  ```
  LSEARCH('ABC.DEF.*')
  ```

  will create a full file ID of:

  `ABC.DEF.`*filename*

  where *filename* is the file name on the `#include` directive.

member.type
Specifies a selection mask which identifies the name or partial name matching the include files that are located in the VSE Librarian sublibraries identified by the `LIB` suboption.

For example, if *member.type* contains `abc*.h`, this indicates that only include files starting with `abc` and having a file type of `h` can be found in the sublibraries indicated by the `LIB` suboption.

LIB(*lib.sublib*)
Specifies the VSE Librarian sublibraries to be searched for the include files that match the selection mask of *member.type*.

lib.sublib is a VSE Librarian sublibrary name which is searched. The order of the sublibrary search is the same order as in the LIB list.

lib.sublib may be omitted completely, in which case no search sublibraries are defined. A warning message is issued in this situation.

NOLIB
Specifies that all LIB(*lib.sublib*) options previously specified for this selection mask should be ignored.

For example:

```
LSEARCH((*.h)=(LIB(TEST.CINC),NOLIB,LIB(PROD.CINC)))
```

is equivalent to:

```
LSEARCH((*.h)=(LIB(PROD.CINC)))
```

The NOLIB format may be useful, for example, if you need to preserve the name of test libraries when your JCL is migrated into production.

The following table shows you how to specify a VSE Librarian sublibrary name or a partial file ID (using the LSEARCH option) for the #include "myincl.h" directive.

Table 8. Specifying VSE Librarian Sublibrary and Partial File ID Using the LSEARCH Option

Option	**For** #include "myincl.h"
LSEARCH(A.B)	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary A.B.
LSEARCH(A.B.*)	The compiler looks for the sequential file *jobname*.A.B.MYINCL.H.
LSEARCH('A.B.*')	The compiler looks for the sequential file A.B.MYINCL.H.
LSEARCH(*)	The compiler looks for the sequential file *jobname*.MYINCL.H.
LSEARCH('*')	The compiler looks for the sequential file MYINCL.H.
LSEARCH((*.h)=(LIB(X.Y)))	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary X.Y.
LSEARCH((*.j)=(LIB(X.Y)))	The compiler will not search sublibrary X.Y because the include filename does not match the pattern of *.j.
LSEARCH((*.h)=(LIB(X1.Y1,X2.Y2)))	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary X1.Y1. If it is not found in this sublibrary, the compiler looks for it in the sublibrary X2.Y2.
LSEARCH((*.*)=(LIB(X1.Y1),NOLIB,LIB(X2.Y2)))	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary X2.Y2. NOLIB causes sublibrary X1.Y1 to be ignored.

For examples of the use of both SEARCH and LSEARCH, see “SEARCH|NOSEARCH” on page 39.

If an include file is not found in any of the LSEARCH sublibraries, the source sublibrary chain specified using the // LIBDEF job control statement is searched. If not found in the LIBDEF chain either, the sublibraries specified using the SEARCH compile-time option, if specified, are searched.

The NOLSEARCH option instructs the preprocessor to search only those sublibraries in the source sublibrary chain specified using the job control // LIBDEF statement.

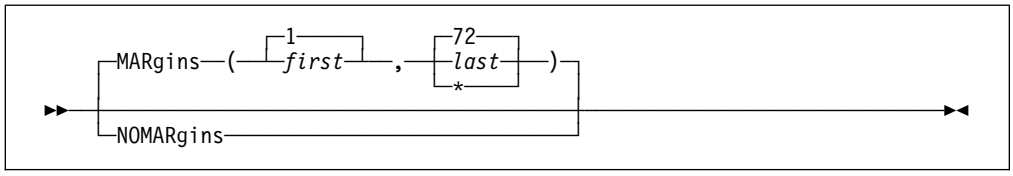
Note: When you use single quotation marks inside the #include directive, or use the DLBL/TLBL-name and/or logical unit format, the file is read directly and no library search is performed.

MARGINS|NOMARGINS

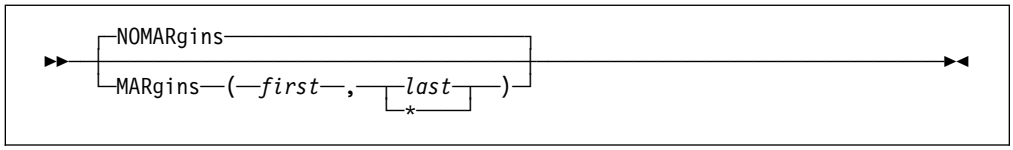
The MARGINS option specifies the columns in the input record that are to be scanned for input to the compiler. The compiler ignores any text in the source input that does not fall within the range specified on the MARGINS option.

Syntax for the MARGINS option is:

For F-format input files:



For V-format input files:



where:

first Specifies the first column of the source input containing valid C. The value of *first* must be greater than 0 and less than 32768.

last Specifies the last column of the source input containing valid C. The value of *last* must be greater than *first* and less than 32768. An asterisk (*) can be assigned to *last* indicating the last column of the input record. Thus, if MARGINS(9,*) is specified, the compiler scans from column 9 to the end of the record for input source statements.

If the MARGINS option is specified along with the SOURCE option, only the range specified on the MARGINS option is shown in the compiler source listing.

The MARGINS and SEQUENCE options can be used together. The MARGINS option is applied first to determine which columns are to be scanned. The SEQUENCE option is then applied to determine which of these columns are not to be scanned. If the SEQUENCE settings do not fall within the MARGINS settings, the SEQUENCE option has no effect.

When a source (or include) file is opened, it initially gets the margins and sequence values specified in the PARM parameter of the EXEC JCL statement (or the defaults if none was specified). These settings can be reset by using the #pragma margins or #pragma sequence directive at any point in the file. When processing has

completed for an #include file, the margins and sequence settings are restored to their values from before the #include directive was encountered.

The NOMARGINS option specifies that the entire input source record is to be scanned for input to the compiler.

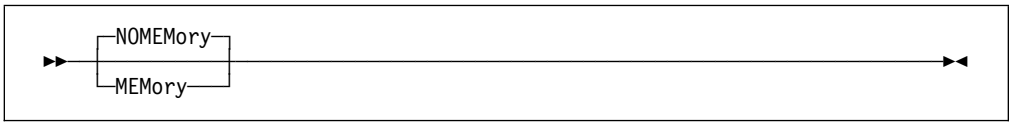
Notes:

1. The MARGINS option does not reformat listings.
2. If your program uses the #include preprocessor directive to include LE/VSE run-time library header files *and* you want to use the MARGINS option, you must ensure that the specifications on the MARGINS option does not exclude columns 20 through 50. That is, the value of *first* must be less than 20, and the value of *last* must be greater than 50. If your program does not include any LE/VSE run-time library header files, you can specify any setting you want on the MARGINS option when the setting is consistent with your own include files.

MEMORY|NOMEMORY

The MEMORY option specifies that the compiler is to use a memory file in place of a workfile if possible. See the *LE/VSE C Run-Time Programming Guide* for more information on memory files.

Syntax for the MEMORY option is:



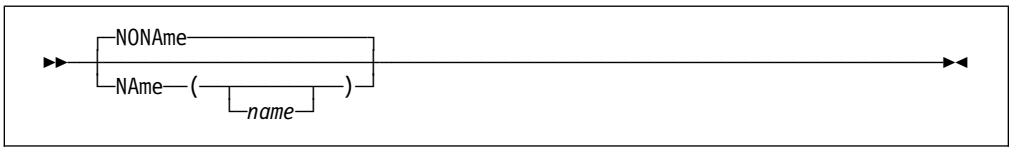
This option is used to increase compilation speed, but to use this option, you may require additional memory. If you use this option and the compilation fails because of a storage error, you must increase your storage size or recompile your program without the MEMORY option.

Do not specify this option with INLINE.

NAME|NONAME

The NAME option is used to generate a linkage editor PHASE statement or a VSE Librarian CATALOG statement.

Syntax for the NAME option is:



When the NAME option is used with the OBJECT option, the compiler writes a PHASE statement to SYSLNK for input to the linkage editor. The format of the phase statement generated is as follows:

```
PHASE name,*
```

When the NAME option is used with the DECK option, the compiler writes a VSE Librarian CATALOG statement and a terminating /* to SYSPCH. This can be used as input to the VSE Librarian to catalog the object code in a VSE Librarian sublibrary. The format of the catalog statement generated is as follows:

```
CATALOG name.OBJ REPLACE=YES
```

To specify a name, use the following format:

```
NAME(name)
```

where *name* is the name to be assigned to the PHASE or CATALOG control statement.

To specify that a PHASE or CATALOG control statement is not to be generated, use the following format:

```
NAME()
```

Note that an empty set of parentheses indicates that a PHASE or CATALOG control statement should not be generated. This is equivalent to specifying NONAME.

The NONAME option specifies that the compiler is not to generate a PHASE or CATALOG control statement.

The NAME|NONAME option can be specified in the PARM parameter of the EXEC JCL statement or in the #pragma options preprocessor directive.

NESTINC|NONESTINC

The NESTINC option allows you to specify the number of nested include files to be allowed in your source program.

Syntax for the NESTINC option is:

You can specify a limit *n* of any integer from 0 to SHRT_MAX, which indicates the maximum limit, as defined in the limits.h header file. To specify the maximum limit, use an asterisk (*).

Note: If you use heavily nested include files, more storage is required for the compilation.

If you specify NONESTINC or an invalid value, the default limit is used.

OBJECT|NOOBJECT

The OBJECT option specifies whether or not the compiler is to produce an object module to SYSLNK for input to the linkage editor.

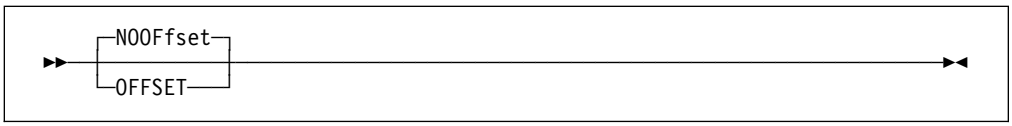
This option cannot be specified in the PARM parameter of the EXEC JCL statement or in a #pragma options processor directive. The OBJECT compile-time option is specified using the LINK or CATAL job control option on the JCL // OPTION statement.

If you use the OBJECT option, ensure that SYSLNK is assigned in the JCL for your compilation.

OFFSET|NOOFFSET

The OFFSET option specifies that the compiler is to display, in the pseudo-assembly listing generated by the LIST option, the offset addresses relative to the entry point or start of each function.

Syntax for the OFFSET option is:



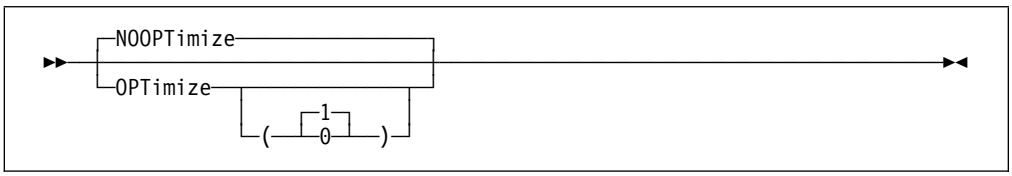
If you use the OFFSET option, you must also specify the LIST option to generate the pseudo-assembly listing. If you specify the OFFSET option but omit the LIST option, a warning message is generated and a pseudo-assembly listing is not produced.

The NOOFFSET option specifies that the compiler is to display, in the pseudo-assembly listing generated by the LIST option, the offset addresses relative to the beginning of the generated code and not the entry point.

OPTIMIZE|NOOPTIMIZE

The OPTIMIZE option instructs the compiler to optimize the machine instructions generated to produce a faster running object code. This type of optimization can also reduce the amount of main storage required for the object module. Using OPTIMIZE will increase compile time over NOOPTIMIZE and may have greater storage requirements. During optimization, the compiler may move code to increase run-time efficiency; as a result, statement numbers in the program listing may not correspond to the statement numbers used in run-time messages.

Syntax for the OPTIMIZE option is:



where:

0 Indicates no optimization is to be done; this is equivalent to the NOOPTIMIZE option. This option should be used in the early stages of your application development since the compilation is efficient but the execution is not.

1 Indicates local optimizations are to be performed. This is equivalent to specifying OPTIMIZE.

If you specify OPTIMIZE with TEST, you will only be able to set breakpoints at function entry points regardless of the suboptions you set.

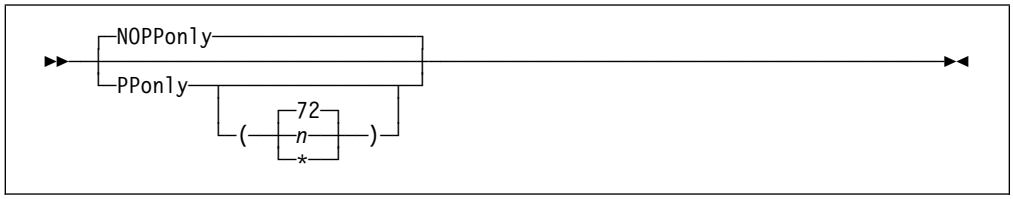
You should specify the option INLINE for optimal run-time performance. See “INLINE|NOINLINE” on page 27 for more information about the INLINE option and the *C/VSE Language Reference* for more information about optimization.

Note: A comment noting the level of optimization is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an END card at the end of the object module.

PPONLY|NOPPONLY

The PPONLY option specifies that only the preprocessor is to be run against the source file. This output of the preprocessor consists of the original source file with all macros expanded and all include files inserted. It is in a format that can be compiled.

Syntax for the PPONLY option is:



If a parameter *n*, an integer value between 2 and 80 inclusive, is specified, all lines are folded at column *n*. If an asterisk (*) is specified, the lines are folded at the maximum record length of 80. Otherwise, all lines are folded at column 72.

All #line and #pragma preprocessor directives (except for #pragma margins and #pragma sequence directives) remain. When PPONLY(*) is specified, #line directives are generated to keep the line numbers generated for the output file from the preprocessor similar to the line numbers generated for the source file. All consecutive blank lines are suppressed.

This output from the preprocessor is written to SYSPCH.

If the PPONLY option is specified, TERMINAL is implicitly turned on. If you specify any of the options SHOWINC, XREF, AGGREGATE, or EXPMAC with PPONLY, a warning is issued and they are ignored.

The NOPPONLY option specifies that both the preprocessor and the compiler are to be run against the source file.

If the PPONLY and LOCALE options are specified, all the #pragma filetag directives in the source file are suppressed. The compiler generates its #pragma filetag directive at the first line in the preprocessed output file in the following format:

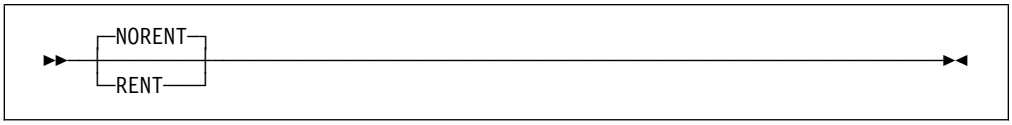
```
??=pragma filetag ("locale_code_page")
```

The *locale_code_page* in the #pragma is the code set of the locale specified in the LOCALE option. For more information on locales, refer to the *LE/VSE C Run-Time Programming Guide*.

RENT|NORENT

The RENT option specifies that the compiler is to take code that is not naturally reentrant and make it reentrant. See the *LE/VSE Programming Guide* for a detailed description of reentrancy. If you use the RENT option, you must use the prelinker, also described in the *LE/VSE Programming Guide*.

Syntax for the RENT option is:



Note: Whenever you specify the RENT compile-time option, a comment noting its use is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an END card at the end of the object module.

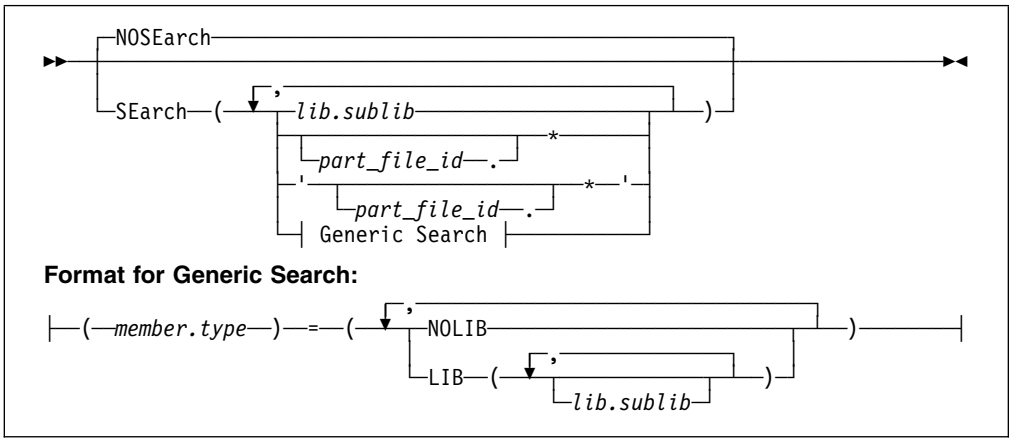
The NORENT option specifies that the compiler is not to specifically generate reentrant code from non-reentrant code. Any naturally reentrant code remains reentrant.

SEARCH|NOSEARCH

The SEARCH option directs the preprocessor to look for the system include files in the VSE Librarian sublibraries, or in the sequential disk files specified. System include files are files associated with the `#include <filename>` format of the `#include` C preprocessor directive. See “Using Include Files” on page 4 for a description of the `#include` preprocessor directive.

For further information on sublibrary search sequences, see “Search Sequences for Include Files” on page 7.

Syntax for the SEARCH option is:



where:

lib.sublib
Specifies the name of a VSE Librarian sublibrary containing system include files. The library name (*lib*) is from 1 to 7 characters and the sublibrary name (*sublib*) is from 1 to 8 characters.

part_file_id
Specifies a partial file ID of a sequential file containing a system include file. The full file ID is the 1 to 44 character external name for the sequential file as coded on the `// DLBL` job control statement that defines the file. A period and asterisk (.*) must be appended immediately after *part_file_id*. If no partial file ID is required, specify an asterisk only as the option.

The full file ID is made up as follows:

- For a partial file ID not within single quotes, the jobname followed by a period is added to the start of the partial file ID. For example:

  ```
  SEARCH(ABC.DEF.*)
  ```

  will create a full file ID of:

  jobname.ABC.DEF.*filename*

  where *filename* is the file name on the `#include` directive.

- For a partial file ID specified within single quotes, the jobname is not added to the start of the partial file ID. For example:

  ```
  SEARCH('ABC.DEF.*')
  ```

  will create a full file ID of:

  ABC.DEF.*filename*

  where *filename* is the file name on the `#include` directive.

member.type
Specifies a selection mask which identifies the name or partial name matching the include files that are located in the VSE Librarian sublibraries identified by the `LIB` suboption.

For example, if *member.type* contains `abc*.h`, this indicates that only include files starting with `abc` and having a file type of `h` can be found in the sublibraries indicated by the `LIB` suboption.

`LIB(`*lib.sublib*`)`
Specifies the VSE Librarian sublibraries to be searched for the include files that match the selection mask of *member.type*.

lib.sublib is a VSE Librarian sublibrary name which is searched. The order of the sublibrary search is the same order as in the `LIB` list.

lib.sublib may be omitted completely, in which case no search sublibraries are defined. A warning message is issued in this situation.

`NOLIB`
Specifies that all `LIB(`*lib.sublib*`)` options previously specified for this selection mask should be ignored.

For example:

```
SEARCH((*.h)=(LIB(TEST.CINC),NOLIB,LIB(PROD.CINC)))
```

is equivalent to:

```
SEARCH((*.h)=(LIB(PROD.CINC)))
```

The NOLIB format may be useful if you need to preserve the name of the test libraries once the JCL is migrated into production.

The following table shows you how to specify a VSE Librarian sublibrary name or a partial file ID (using the SEARCH option) for the #include <myincl.h> directive.

Table 9. Specifying VSE Librarian Sublibrary and Partial File ID Using the SEARCH Option

Option	**For** #include <myincl.h>
SEARCH(A.B)	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary A.B.
SEARCH(A.B.*)	The compiler looks for the sequential file *jobname*.A.B.MYINCL.H.
SEARCH('A.B.*')	The compiler looks for the sequential file A.B.MYINCL.H.
SEARCH(*)	The compiler looks for the sequential file *jobname*.MYINCL.H.
SEARCH('*')	The compiler looks for the sequential file MYINCL.H.
SEARCH((*.h)=(LIB(X.Y)))	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary X.Y.
SEARCH((*.j)=(LIB(X.Y)))	The compiler will not search sublibrary X.Y because the include filename does not match the pattern of *.j.
SEARCH((*.h)=(LIB(X1.Y1,X2.Y2)))	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary X1.Y1. If it is not found in this sublibrary, the compiler looks for it in the sublibrary X2.Y2.
SEARCH((*.*)=(LIB(X1.Y1),NOLIB,LIB(X2.Y2)))	The compiler looks for the VSE Librarian member MYINCL.H in the sublibrary X2.Y2. NOLIB causes sublibrary X1.Y1 to be ignored.

If an include file is not found in any of the SEARCH sublibraries, the source sublibrary chain specified using the // LIBDEF job control statement is searched.

The NOSEARCH option instructs the preprocessor to search only those sublibraries in the source sublibrary chain specified using the job control // LIBDEF statement.

Note: When you use single quotation marks inside the #include directive, or use the DLBL/TLBL-name and/or logical unit format, the file is read directly and no library search is performed.

Table 10 on page 42 shows the search performed when both the SEARCH and the LSEARCH option is used. The following SEARCH and LSEARCH options are in effect:

```
LSEARCH(lib1.sublib1,(*.j)=(LIB(lib2.sublib2)))
SEARCH((*.h)=(LIB(lib3.sublib3)),seq.*)
```

For detailed information about the specification of #include filenames, see “Using Include Files” on page 4.

Table 10. Libraries Searched When Both SEARCH and LSEARCH Are Specified

#include	Libraries Searched
"'a.b.c.d.e.f.g'"	The compiler will attempt to open a sequential file 'a.b.c.d.e.f.g'. No search is performed.
"dd:*dlbl*"	The compiler will attempt to open a file with the DLBL-name of *dlbl*. No search is performed.
"dd:*lib.sublib*(*mem.type*)"	The compiler will attempt to open member *mem.type* in sublibrary *lib.sublib*. No search is performed.
<head.h>	The compiler will search for the following: 1. Member HEAD.H in sublibrary LIB3.SUBLIB3 2. Sequential file *jobname*.SEQ.HEAD.H 3. Member HEAD.H in sublibraries in source search chain specified by the // LIBDEF JCL statement
"aa/bb/cc/dd/head"	The compiler will search for the following: 1. Member HEAD.H in sublibrary LIB1.SUBLIB1 2. Member HEAD.H in sublibraries in source search chain specified by the // LIBDEF JCL statement 3. Member HEAD.H in sublibrary LIB3.SUBLIB3 4. Sequential file *jobname*.SEQ.HEAD.H
"head.j.k.l"	The compiler will search for the following: 1. Member HEAD.J in sublibrary LIB1.SUBLIB1 2. Member HEAD.J in sublibrary LIB2.SUBLIB2 3. Member HEAD.J in sublibraries in source search chain specified by the // LIBDEF JCL statement 4. Sequential file *jobname*.SEQ.HEAD.J.K.L

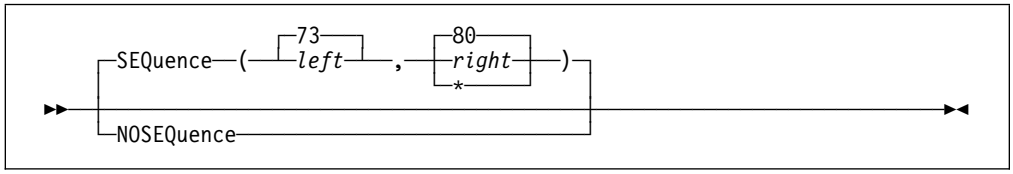
For further information on library search sequences, see "Search Sequences for Include Files" on page 7.

SEQUENCE|NOSEQUENCE

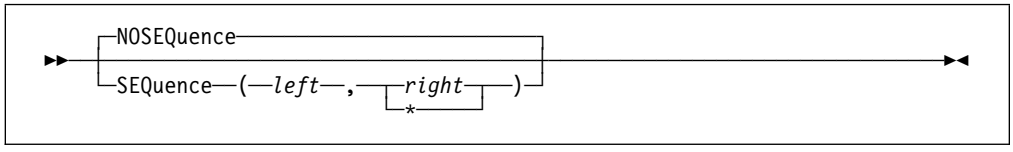
The SEQUENCE option defines the section of the input record that is to contain sequence numbers. No attempt is made to sort the input lines or records into the specified sequence or to report records out of sequence.

Syntax for the SEQUENCE option is:

For F-format input files:



For V-format input files:



where:

left Specifies the column number of the left-hand margin. The value of *m* must be greater than 0 and less than 32768.

right Specifies the column number of the right-hand margin. The value of *right* must be greater than *left* and less than 32768. An asterisk (*) can be assigned to *right* indicating the last column of the input record. Thus, SEQUENCE(74,*) shows that sequence numbers are between column 74 and the end of the input record.

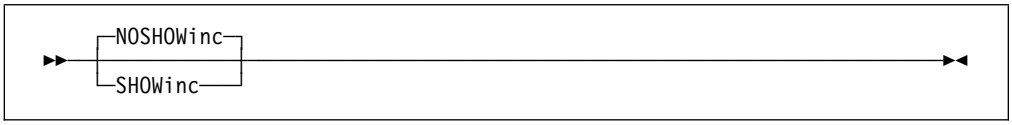
The IBM-supplied default for fixed-length records is SEQ(73,80), and the default for variable-length and undefined-length records is NOSEQ.

Note: If your program uses the #include preprocessor directive to include LE/VSE run-time library header files *and* you want to use the SEQUENCE option, you must ensure that the specifications on the SEQUENCE option do not include any columns between 20 and 50, both inclusive. That is, both *left* and *right* must be less than 20, or both must be greater than 50. If your program does not include any LE/VSE run-time library header files, you can specify any setting you want on the SEQUENCE option when the setting is consistent with your own include files.

SHOWINC|NOSHOWINC

The SHOWINC option specifies that the compiler is to show, in both the compiler listing and the pseudo-assembly listing, all include files processed.

Syntax for the SHOWINC option is:

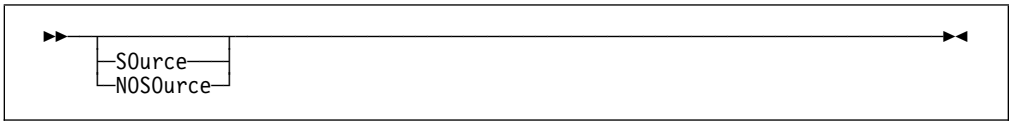


In the listing, all #include preprocessor directives are replaced with the source contained in the include file. The filename and the first letter of the file type of the include file are shown on the listing under the heading “INCLUDE.” This option only applies if the SOURCE option is also specified.

SOURCE|NOSOURCE

The SOURCE option specifies that the compiler is to generate a listing that shows the original source input statements plus any diagnostic messages.

Syntax for the SOURCE option is:



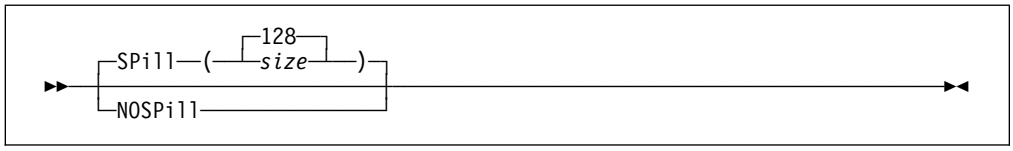
The SOURCE option may also be specified using the LIST job control option on the JCL // OPTION statement.

The default is determined by the LIST job control option.

SPILL|NOSPILL

The SPILL option allows you to specify the size of the spill area to be used for the compilation. When too many registers are in use at once, the compiler dumps some of them into the temporary storage called the spill area.

Syntax for the SPILL option is:



You may have to expand the spill area; you will receive a compiler message telling you the size to which you should increase the spill area. (Once you know the size of the spill area required for your source program, you can then add a `#pragma options(SPILL(size))` directive to your source.) The maximum spill area size is 3900.

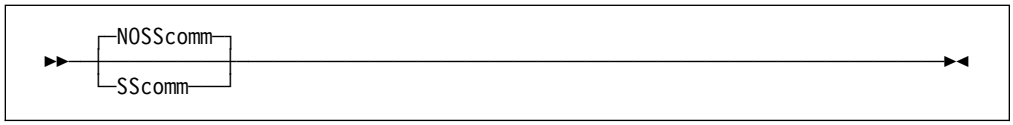
Note: There is an upper limit of 4096 bytes for the combined area for your spill area, local variables and arguments passed to called functions. For best use of your stack, do not pass large arguments, such as structures, by value.

If you specify NOSPILL, you will get the default value for SPILL.

SSCOMM|NOSSCOMM

The SSCOMM option specifies that the compiler is to recognize two slashes (//) as the beginning of a comment that will terminate at the end of the line. It will also recognize /* */ as comments for compatibility with code in which // is accepted as the beginning of comments.

Syntax for the SSCOMM option is:

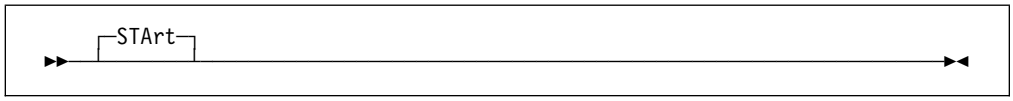


NOSSCOMM indicates that /* */ is the only valid comment format.

START

The START option specifies that an ENTRY CEESTART control statement is to be generated.

Syntax for the START option is:



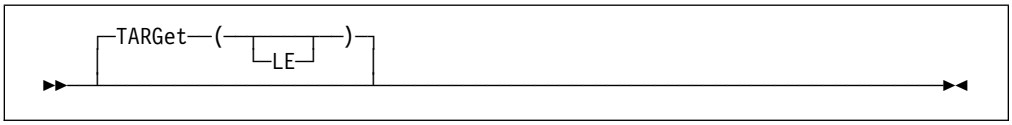
Notes:

1. Whenever you specify the START compile-time option, a comment noting its use is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an END card at the end of the object module.
2. There is no negative form of the START option. That is, specifying NOSTART is not supported.

TARGET

The TARGET option specifies the run-time environment for which the object module is to be generated.

Syntax for the TARGET option is:



where:

TARGET() Generates object code to run under LE/VSE. It is the same as TARGET(LE). This is the default.

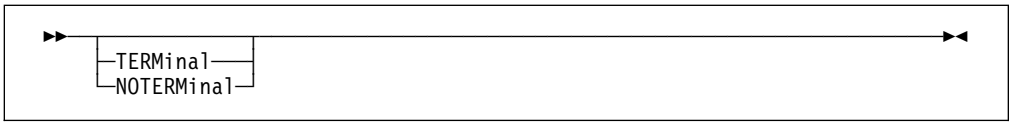
TARGET(LE) Generates object code to run under LE/VSE.

Note: A comment noting the value of TARGET used is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an END card at the end of the object module.

TERMINAL|NOTERMINAL

The TERMINAL option specifies that all error messages from the compiler are to be directed to SYSLOG.

Syntax for the TERMINAL option is:



If the PPONLY option is specified, TERM is implicitly turned on.

The TERMINAL option may also be specified using the TERM job control option on the JCL // OPTION statement.

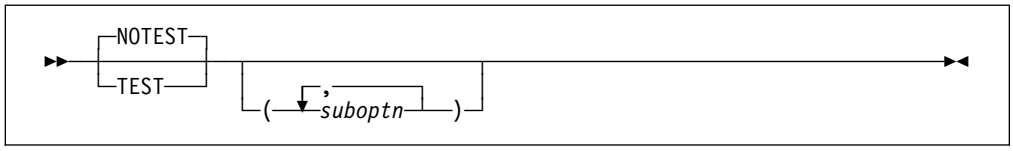
If you specify NOTERMINAL, then no separate log of error messages is produced. If you specify the SOURCE option to generate a listing, the error messages are interspersed in the listing.

The default is determined by the TERM job control option.

TEST|NOTEST

The TEST option specifies that the compiler is to generate information for Debug Tool for VSE/ESA. If the NOTEST option is chosen, debugging information is not generated.

Syntax for the TEST option is:



where *suboptn* is one of the suboptions shown in Table 11. (Default suboptions are indicated by underscoring.)

Table 11. TEST *Suboptions*

TEST **Suboption**	**Description**
SYM|NOSYM	Generates symbol table information
BLOCK|NOBLOCK	Generates symbol information for nested blocks
LINE|NOLINE	Generates line number hooks
PATH|NOPATH	Generates path breakpoints
ALL	Is equivalent to TEST(SYM,BLOCK,LINE,PATH)
NONE	Is equivalent to TEST(NOSYM,NOBLOCK,NOLINE,NOPATH)

The TEST|NOTEST option can be specified on the invocation line and in the #pragma options preprocessor directive. When both methods are used concurrently, the options are merged. If an option in the PARM parameter of the EXEC JCL statement conflicts with an option in the #pragma options directive, the one in the PARM parameter of the EXEC JCL statement takes precedence. For example, if you do not want to generate debugging information when you compile a program, you can specify the NOTEST option in a #pragma options preprocessor directive. When you do want to generate debugging information, you can then override the NOTEST option by specifying TEST in the PARM parameter of the EXEC JCL statement rather than editing your source program. The example below illustrates these rules:

```
Source file:   #pragma options(notest(block,line))
EXEC PARM:     TEST(SYM,NOBLOCK)
Result:        TEST(SYM,NOBLOCK,LINE)
```

The NOTEST option specifies that the compiler is not to generate information for the debugger. Options specified in a #pragma options(NOTEST) directive or NOTEST apply if TEST is specified in the PARM parameter of the EXEC JCL statement.

Any #line directives are ignored when the TEST option is active. Do not use any #line directives before a #pragma options directive that contains the TEST option.

The compile-time option INLINE is ignored if specified with TEST.

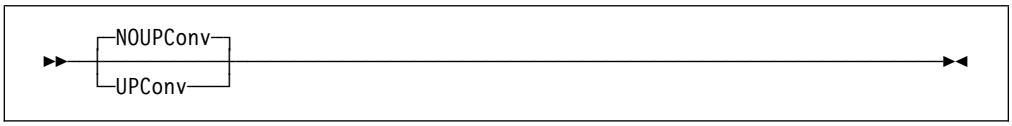
If the TEST option is specified, GONUMBER is implicitly turned on.

If you specify OPTIMIZE with TEST, you will only be able to set breakpoints at function entry points regardless of the suboptions you set.

UPCONV|NOUPCONV

The UPCONV option specifies that the compiler is to follow unsignedness preserving rules when doing C type conversions; that is, when widening all integral types (char, short, int, and long). Use this option when compiling older C programs that depend on the older conversion rules.

Syntax for the UPCONV option is:

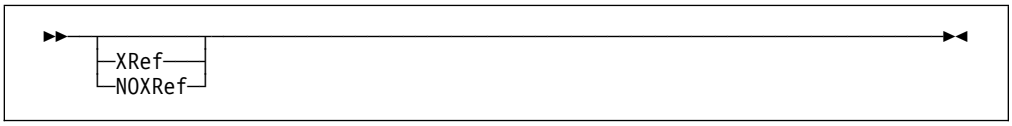


Note: Whenever you specify the UPCONV compile-time option, a comment noting its use is generated in your object module to aid you in diagnosing problems with your program. The comment generated consists of the specified compile-time option on an END card at the end of the object module.

XREF|NOXREF

The XREF option specifies that the compiler is to include in the source listing a cross reference table of names used in the program, together with the numbers of the lines where they are declared or referenced.

Syntax for the XREF option is:



A separate offset listing of the variables will appear after the cross reference table.

The XREF option may also be specified using the XREF job control option on the JCL // OPTION statement.

The default is determined by the XREF job control option.

Using the Compiler Listing

During compilation, if requested, the compiler creates a listing that contains information about the source program and the compilation.

The listing is written to SYSLST.

If compilation terminates before reaching a particular stage of processing, the corresponding parts of listings are generated.

The listing contains standard information that always appears, together with optional information supplied by default or specified through compile-time options.

If you use the TERMINAL option, all error messages issued by the compiler are directed to SYSLOG as well as being interspersed with the compiler listing.

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

Example of a C/VSE Compiler Listing

Figure 7 shows an example of a compiler listing.

```
5686A01 V1 R1 M00 IBM C/VSE                                DD:SYSIPT                  09/04/1996 17:05:52   PAGE     1

                                 * * * * *  P R O L O G  * * * * *

 COMPILE TIME LIBRARY . . . . . : 11040000
 COMMAND OPTIONS:
    PROGRAM NAME. . . . . . . . : DD:SYSIPT
    COMPILER OPTIONS. . . . . . : *NOGONUMBER *NONAME     *NODECK     *NORENT     *TERMINAL   *NOUPCONV   *SOURCE     *LIST
                                : *XREF       *AGGR       *NOPPONLY   *NOEXPMAC   *NOSHOWINC  *NOOFFSET   *NOMEMORY   *NOSSCOMM
                                : *NOCSECT    *NOLONGNAME *START      *EXECOPS    *NOEVENTS   *NOINFILE
                                : *TARGET()   *FLAG(I)    *NOTEST(SYM,BLOCK,LINE,NOPATH)      *OPTIMIZE(0)*SPILL(128)
                                : *INLINE(AUTO,REPORT,250,1000)       *NESTINC(16)
                                : *NOCHECKOUT(NOPPTRACE,PPCHECK,GOTO,ACCURACY,PARM,NOENUM,
                                : *           NOEXTERNAL,TRUNC,INIT,NOPORT,GENERAL)
                                : *NOSEARCH
                                : *NOLSEARCH
                                : *OBJECT     *NOHWOPTS   *NOLOCALE
    LANGUAGE LEVEL. . . . . . . : *EXTENDED
    SOURCE MARGINS. . . . . . . :
      VARYING LENGTH. . . . . . : 1 - 32767
      FIXED LENGTH  . . . . . . : 1 - 72
    SEQUENCE COLUMNS. . . . . . :
      VARYING LENGTH. . . . . . : NONE
      FIXED LENGTH. . . . . . . : 73 - 80

                                      * * * * *  S O U R C E  * * * * *

 LINE  STMT                                                                                                    SEQNBR INCNO
            *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.......*
    1       | /* EDCXUAAA                                                                                    |      1
    2       |    This is an example of a C/VSE program                                                       |      2
    3       |  */                                                                                            |      3
    4       |                                                                                                |      4
    5       |#include <stdio.h>                                                                              |      5
    6       |                                                                                                |      6
    7       |#include "edcxuaab.h"                                                                           |      7
    8       |                                                                                                |      8
    9       |void convert(double);                                                                           |      9
   10       |                                                                                                |     10
   11       |int main(int argc, char **argv)                                                                 |     11
   12       |{                                                                                               |     12
   13       |    double c_temp;                                                                              |     13
   14       |    int ch, i;                                                                                  |     14
   15       |                                                                                                |     15
   16     1 |    if (argc == 1) {  /* get Celsius value from stdin */                                        |     16
   17       |                                                                                                |     17
   18     2 |       printf("Enter Celsius temperature: \n");                                                 |     18
   19       |                                                                                                |     19
   20     3 |       if (scanf("%f", &c_temp) != 1) {                                                         |     20
   21     4 |          printf("You must enter a valid temperature\n");                                       |     21
   22       |       }                                                                                        |     22
   23       |       else {                                                                                   |     23
   24     5 |          convert(c_temp);                                                                      |     24
   25       |       }                                                                                        |     25
   26       |    }                                                                                           |     26
```

Figure 7 (Part 1 of 7). Example of a Compiler Listing

```
  27       |    else {  /* convert the invocation arguments to Fahrenheit */                                    |    27
  28       |                                                                                                    |    28
  29     6 |       for (i = 1; i < argc; ++i) {                                                                 |    29
  30     7 |           if (sscanf(argv[i], "%f", &c_temp) != 1)                                                 |    30
  31     8 |              printf("%s is not a valid temperature\n",argv[i]);                                    |    31
  32       |           else                                                                                     |    32
  33     9 |              convert(c_temp);                                                                      |    33
  34       |       }                                                                                            |    34
  35       |    }                                                                                               |    35
  36       |}                                                                                                   |    36
  37       |                                                                                                    |    37
  38       |void convert(double c_temp) {                                                                       |    38
  39       |   double f_temp = (c_temp * CONV + OFFSET);                                                        |    39
  40    10 |   printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);                                    |    40
  41       |}                                                                                                   |    41
                              * * * * *  E N D  O F  S O U R C E  * * * * *

                         * * * * *  I N C L U D E S  * * * * *

INCLUDE FILES  ---  FILE#   NAME

                        1   DD:(STDIO.H)
                        2   DD:(FEATURES.H)
                        3   DD:(EDCXUAAB.H)

                         * * * * *  E N D  O F  I N C L U D E S  * * * * *

                               * * * * *  C R O S S  R E F E R E N C E  L I S T I N G  * * * * *

IDENTIFIER          DEFINITION      ATTRIBUTES
                                    <SEQNBR>-<FILE NO>:<FILE LINE NO>

___valist           5-1:110         TYPEDEF
                                    TYPE = ARRAY[2] OF
                                             POINTER TO UNSIGNED CHARACTER

__amrc_ptr          5-1:435         TYPEDEF
                                    TYPE = POINTER TO STRUCTURE __amrctype

__amrc_type         5-1:431         TYPEDEF
                                    TYPE = STRUCTURE __amrctype

__amrctype          5-1:385         CLASS = TAG, TYPE = STRUCTURE

__amrc2_ptr         5-1:448         TYPEDEF
                                    TYPE = POINTER TO STRUCTURE __amrc2type
.
.
.
scanf               5-1:201         CLASS = EXTERNAL REFERENCE,
                                    TYPE = C FUNCTION RETURNING SIGNED INTEGER
                                    20-0:20

size_t              5-1:39          TYPEDEF
                                    TYPE = UNSIGNED INTEGER

sscanf              5-1:209         CLASS = EXTERNAL REFERENCE,
                                    TYPE = C FUNCTION RETURNING SIGNED INTEGER
                                    30-0:30

FILE                5-1:62          TYPEDEF
                                    TYPE = STRUCTURE __ffile

                          * * * * *  E N D  O F  C R O S S  R E F E R E N C E  L I S T I N G  * * * * *
```

Figure 7 (Part 2 of 7). Example of a Compiler Listing

```
                    * * * * *  S T R U C T U R E  M A P S  * * * * *

 AGGREGATE MAP FOR: STRUCTURE __amrctype                                    TOTAL SIZE: 220 BYTES

      OFFSET          LENGTH        MEMBER NAME
    BYTES(BITS)     BYTES(BITS)
       0               4            __code
       0               4              __error
       0               4              __abend
       0               2                __syscode
       2               2                __rc
       0               4              __feedback
       0               1                __fdbk_fill
       1               1                __rc
       2               1                __ftncd
       3               1                __fdbk
       0               4              __alloc
       0               2                __svc99_info
       2               2                __svc99_error
       4               4            __RBA
       8               4            __last_op
      12             208            __msg
      12               4              __len_fill
      16               4              __len
      20             120              __str[120]
     140               4              __parmr0
     144               4              __parmr1
     148               8              __fill2[2]
     156              64              __str2[64]

 AGGREGATE MAP FOR: _PACKED STRUCTURE __amrctype                            TOTAL SIZE: 220 BYTES

      OFFSET          LENGTH        MEMBER NAME
    BYTES(BITS)     BYTES(BITS)
       0               4            __code
       0               4              __error
       0               4              __abend
       0               2                __syscode
       2               2                __rc
       0               4              __feedback
       0               1                __fdbk_fill
       1               1                __rc
       2               1                __ftncd
       3               1                __fdbk
       0               4              __alloc
       0               2                __svc99_info
       2               2                __svc99_error
       4               4            __RBA
       8               4            __last_op
      12             208            __msg
      12               4              __len_fill
      16               4              __len
      20             120              __str[120]
     140               4              __parmr0
     144               4              __parmr1
     148               8              __fill2[2]
     156              64              __str2[64]

.
.
.
```

Figure 7 (Part 3 of 7). Example of a Compiler Listing

```
=================================================================================================================================
| AGGREGATE MAP FOR: _PACKED STRUCTURE __fpos_t                                                    TOTAL SIZE: 32 BYTES          |
|===============================================================================================================================|
|      OFFSET       |      LENGTH       | MEMBER NAME                                                                           |
|    BYTES(BITS)    |    BYTES(BITS)    |                                                                                       |
|===================|===================|=======================================================================================|
|       0           |      32           |  __fpos_elem[8]                                                                       |
=================================================================================================================================

                        * * * * *  E N D  O F  S T R U C T U R E  M A P S  * * * * *

                          * * * * *  M E S S A G E  S U M M A R Y  * * * * *

       TOTAL            INFORMATIONAL(00)         WARNING(10)         ERROR(30)           SEVERE ERROR(40)

         0                   0                     0                   0                   0
                      * * * * *  E N D  O F  M E S S A G E  S U M M A R Y  * * * * *

                  INLINE REPORT (SUMMARY)

 REASON:    P : #PRAGMA NOINLINE WAS SPECIFIED FOR THIS ROUTINE
            F : #PRAGMA INLINE WAS SPECIFIED FOR THIS ROUTINE
            A : AUTOMATIC INLINING
            - : NO REASON
 ACTION:    I : ROUTINE IS INLINED AT LEAST ONCE
            L : ROUTINE IS INITIALLY TOO LARGE TO BE INLINED
            T : ROUTINE EXPANDS TOO LARGE TO BE INLINED
            C : CANDIDATE FOR INLINING BUT NOT INLINED
            N : NO DIRECT CALLS TO ROUTINE ARE FOUND IN FILE (NO ACTION)
            U : SOME CALLS NOT INLINED DUE TO RECURSION OR PARAMETER MISMATCH
            - : NO ACTION
 STATUS:    D : INTERNAL ROUTINE IS DISCARDED
            R : A DIRECT CALL REMAINS TO INTERNAL ROUTINE (CANNOT DISCARD)
            A : ROUTINE HAS ITS ADDRESS TAKEN (CANNOT DISCARD)
            E : EXTERNAL ROUTINE (CANNOT DISCARD)
            - : STATUS UNCHANGED
 CALLS/I      : NUMBER OF CALLS TO DEFINED ROUTINES / NUMBER INLINE
 CALLED/I     : NUMBER OF TIMES CALLED / NUMBER OF TIMES INLINED

 REASON  ACTION  STATUS   SIZE (INIT)    CALLS/I   CALLED/I     NAME

    A      I       E      13              0         2/2        convert
    A      N       E      101   (67)      2/2       0          main

 MODE = AUTO    INLINING THRESHOLD = 250    EXPANSION LIMIT = 1000

                  INLINE REPORT (CALL STRUCTURE)

 DEFINED FUNCTION      : convert
     CALLS TO          : 0
  CALLED FROM(2,2)     : main(2,2)

 DEFINED FUNCTION      : main
     CALLS TO(2,2)     : convert(2,2)
  CALLED FROM          : 0
```

Figure 7 (Part 4 of 7). Example of a Compiler Listing

```
OFFSET OBJECT CODE  LINE#         P S E U D O   A S S E M B L Y   L I S T I N G

000000                          5  main     DS    0F
000000  47F0  F024              6           B     36(,r15)
00001A  41E0  F038              7           LA    r14,56(,r15)
00001E  58F0  C074              8           L     r15,116(,r12)
000022  07FF                    9           BR    r15
000024  90E6  D00C             10           STM   r14,r6,12(r13)
000028  5820  D04C             11           L     r2,76(,r13)
00002C  4100  20C8             12           LA    r0,200(,r2)
000030  5500  C00C             13           CL    r0,12(,r12)
000034  4720  F01A             14           BH    26(,r15)
000038  58F0  D048             15           L     r15,72(,r13)
00003C  90F0  2048             16           STM   r15,r0,72(r2)
000040  9210  2000             17           MVI   0(r2),16
000044  50D0  2004             18           ST    r13,4(,r2)
000048  18D2                   19           LR    r13,r2
00004A  0530                   20           BALR  r3,r0
00004C                    END OF PROLOG
00004C  5840  ****             22           L     r4,=A(@STATICC)
000050  5010  D088             23           ST    r1,136(,r13)
                                *   /* EDCXUAAA
                                *      This is an example of a C/VSE program
                                *    */
                                *
                                *  #include <stdio.h>
                                *
                                *  #include "edcxuaab.h"
                                *
                                *  void convert(double);
                                *
                    00011 |     *  int main(int argc, char **argv)
                                *  {
                                *      double c_temp;
                                *      int ch, i;
                                *
                    00016 |     *      if (argc == 1) {  /* get Celsius value from stdin */
000054  5860  D088             40           L     r6,136(,r13)
000058  4150  0001             41           LA    r5,1
00005C  5950  6000             42           C     r5,0(,r6)
000060  4770  ****             43           BNE   @8L2
                                *
.
.
.
                    00041 |     +
000194                         156 @8L10     DS    0F
000194  4150  0001             157           LA    r5,1
000198  5A50  D090             158           A     r5,144(,r13)
00019C  5050  D090             159           ST    r5,144(,r13)
0001A0  5850  D088             160           L     r5,136(,r13)
0001A4  5850  5000             161           L     r5,0(,r5)
0001A8  5950  D090             162           C     r5,144(,r13)
0001AC  4720  30AC             163           BH    @8L7
0001B0                         164 @8L5      DS    0D
                                *             }
                                *      }
                    00036 |     *  }
0001B0  41F0  0000             168           LA    r15,0
```

Figure 7 (Part 5 of 7). Example of a Compiler Listing

```
0001B4                     START OF EPILOG
0001B4  180D                  170           LR    r0,r13
0001B6  58D0  D004            171           L     r13,4(,r13)
0001BA  58E0  D00C            172           L     r14,12(,r13)
0001BE  9826  D01C            173           LM    r2,r6,28(r13)
0001C2  051E                  174           BALR  r1,r14
0001C4  0707                  175           NOPR
0001C8                     START OF LITERALS
0001C8  4220  0000  0000      177                 =D'32'
0001CE  0000
0001D0  411C  CCCC  CCCC      179                 =D'1.8'
0001D6  CCCC
0001D8  0000  0001            185                 =F'1'
0001DC  ****  ****            186                 =A(@STATICC)
0001E0  0000  0000            187                 =V(printf)
0001E4  0000  0000            188                 =V(scanf)
0001E8  0000  0000            189                 =V(sscanf)
0001EC                     END OF LITERALS

                           PPA2: COMPILE UNIT BLOCK
0001EC  0300  2200            193                 =F'50340352'       Flags
0001F0  0000  0000            194                 =A(CEESTART)
0001F4  0000  0000            195                 =F'0'              No PPA4
0001F8  0000  01FC            196                 =A(TIMESTMP)
                           PPA2 END

                           PPA1: ENTRY POINT CONSTANTS
000004  14CE  A106            200                 =F'349085958'      Flags
000008  0000  01EC            201                 =A(PPA2)
00000C  0000  0000            202                 =F'0'              No PPA3
000010  0000  00C8            203                 =F'200'            DSA Size
000014  ****                  204                 AL2(4),C'main'
                           PPA1 END

                           ***   GENERAL PURPOSE REGISTERS USED: 1100111000000011
                           ***   FLOATING POINT REGISTERS USED: 1111
                           ***   SIZE OF DYNAMIC STORAGE: 200
                           ***   SIZE OF REGISTER SPILL AREA: 1000000(MAX) 0(USED)
                           ***   SIZE OF EXECUTABLE CODE: 452

                              I B M / 3 7 0   I N S T R U C T I O N   U S A G E
          OP CODE  NUM   %            OP CODE  NUM   %            OP CODE  NUM   %            OP CODE  NUM   %

          L        22  20.00          LA       20  18.18          ST       12  10.91          BC       11  10.00
          BALR      9   8.18          MVC       9   8.18          A         3   2.73          C         3   2.73
          CH        2   1.82          BCR       2   1.82          STD       2   1.82          LD        2   1.82
          AD        2   1.82          MD        2   1.82          SLL       2   1.82          STM       2   1.82
          LR        2   1.82          MVI       1   0.91          LM        1   0.91          CL        1   0.91

.
.
.
                                    E X T E R N A L   S Y M B O L   D I C T I O N A R Y

          NAME        TYPE  ID  ADDR    LENGTH          NAME        TYPE  ID  ADDR    LENGTH

                       PC    1 000000   0002C8          CONVERT      LD    0 000210   000001
          MAIN         LD    0 000000   000001                       PC    2 000000   00008C
          PRINTF       ER    3 000000                   SCANF        ER    4 000000
          SSCANF       ER    5 000000                   CEESG003     ER    6 000000
          CEESTART     ER    7 000000                   CEEMAIN      SD    8 000000   00000C
          MAIN         ER    9 000000                   EDCINPL      ER   10 000000
```

Figure 7 (Part 6 of 7). Example of a Compiler Listing

```
                          *****  S T O R A G E  O F F S E T  L I S T I N G  *****

IDENTIFIER          DEFINITION      ATTRIBUTES
                                    <SEQNBR>-<FILE NO>:<FILE LINE NO>

argc                11-0:11         CLASS = PARAMETER,          OFFSET = 0,         LENGTH = 4

argv                11-0:11         CLASS = PARAMETER,          OFFSET = 4,         LENGTH = 4

c_temp              13-0:13         CLASS = AUTOMATIC,          OFFSET = 152,       LENGTH = 8

c_temp              38-0:38         CLASS = PARAMETER,          OFFSET = 0,         LENGTH = 8

f_temp              39-0:39         CLASS = AUTOMATIC,          OFFSET = 144,       LENGTH = 8

i                   14-0:14         CLASS = AUTOMATIC,          OFFSET = 144,       LENGTH = 4

                    *****  E N D  O F  S T O R A G E  O F F S E T  L I S T I N G  *****

                           *****  E N D  O F  C O M P I L A T I O N  *****
```

Figure 7 (Part 7 of 7). Example of a Compiler Listing

Compiler Listing Components

The following sections describes the components of a C/VSE compiler listing.

Heading Information

The first page of the listing is identified by the product number, the compiler version and release numbers, the date and time when the compilation began, and the name of the file containing the source code.

The first page and subsequent pages are numbered. At the very end of the compiler listing is the "`END OF COMPILATION`" message.

Prolog Section

The Prolog section of the listing provides information about the compile-time library, compile-time options, and other items in effect when the compiler was invoked.

Note: The compile-time options specified in a `#pragma options` directive, as well as options that are implicitly turned on (such as `GONUMBER` when `TEST` is specified), are not identified in the listing.

The following sections describe the optional parts of the listing in the order in which they appear, and the compile-time options that generate them.

Source Program

If the option `SOURCE` is specified, the input to the compiler is included in the listing file.

Note: If the `SHOWINC` option is specified, the source listing shows the included source after the `#include` directives have been processed.

Includes Section

The Includes section is generated when you use `#include` files and specify the `SOURCE` or `LIST` option.

Cross-Reference Table

If the option XREF is specified, the listing file includes a cross-reference table containing a list of the identifiers in the source program together with the numbers of the lines in which they appear.

Structure and Union Maps

You obtain structure and union maps by using the AGGREGATE option. The table generated shows how each structure and union in the program is mapped. It contains the following information:

- The name of the structure or union and the elements within the structure or union
- The byte offset of each element from the beginning of the structure or union; the bit offset for unaligned bit data is also given
- The length of each element
- The total length of each structure, union, and substructure

Messages

If the preprocessor or the compiler detects an error, or the possibility of an error, it generates messages. If you specify the SOURCE compile-time option, preprocessor error messages appear immediately after the source statement in error. You can generate your own messages in the preprocessing stage by using #error. For information on #error, see the *C/VSE Language Reference*.

If you specify the CHECKOUT or FLAG(I) compile-time option, informational diagnostic messages are generated.

For more information on the compiler messages, see "FLAG|NOFLAG" on page 25, and Appendix A, "C/VSE Return Codes and Messages" on page 65.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

Inline Report

If the INLINE(,REPORT,,) option is specified, an inline report is included in the listing. This report contains an inline summary and a detailed call structure.

Note: No report is produced when your source file contains only one defined function.

The summary contains:

- Name of each defined function. Function names are sorted in alphabetical order.
- Reason for action on a function:
  - A #pragma noinline was specified for that function
  - A #pragma inline was specified for that function
  - Auto-inlining acted on that function
  - There was no reason to inline the function
- Action on a function:
  - Function was inlined at least once
  - Function was not inlined because of initial size constraints

  - Function was not inlined because of expansion beyond size constraint
  - Function was a candidate for inlining but was not inlined
  - Function was a candidate for inlining but was not referenced
  - This function is directly recursive, or some calls have mismatching parameters
- Status of original function after inlining:
  - Function is discarded because it is no longer referenced and was defined as `static` internal
  - Function was not discarded for various reasons:
    - Function is external (It can be called from outside the compilation unit)
    - Some call to this function remains
    - Function has its address taken
- Initial relative size of function (in Abstract Code Units (ACU))
- Final relative size of function (in ACUs) after inlining
- Number of calls within the function and the number of these calls that were inlined into the function
- Number of times the function is called by others in the compile unit and the number of times this function was inlined
- Mode selected and the value of *`threshold`* and *`limit`* specified for this compilation

The detailed call structure contains specific information of each function such as:

- What functions it calls
- What functions call it
- In which functions it is inlined

The information can help you to better analyze your program if you want to use the inliner in selective mode.

There may be additional messages as a result of the inlining. For example, if inlining a function with automatic storage will increase the automatic storage of the function it is being inlined into by more than 4K, a message is written.

Pseudo-Assembly Listing

The option `LIST` generates a listing of the machine instructions in the object module, including any compiler-generated functions, in a form similar to assembler language.

The source statement line numbers and the line number of any inlined code is displayed within this Pseudo-Assembly listing to aid you in debugging inlined code.

External Symbol Dictionary

The External Symbol Dictionary lists the names that the compiler generates for the output object module. It includes address and size information about each symbol.

External Symbol Cross Reference Listing

The External Symbol Cross Reference listing is generated if you specify the `XREF` compile-time option. It shows the original name and corresponding mangled name for each symbol.

Storage Offset Listing

If you specify the `XREF` option, the listing file includes offset information for identifiers.

Chapter 5. Run-Time Options

This chapter describes the specification of run-time options and the `#pragma runopts` preprocessor directives available to you. For a detailed description of the LE/VSE run-time options, their abbreviations and IBM-supplied defaults, see the *LE/VSE Programming Reference*.

Specifying Run-Time Options

This section describes the run-time options that you can specify:

- On the `PARM` option of the `EXEC` JCL statement when the `#pragma runopts(execops)` directive is specified in your source program (default).
- On a `#pragma runopts` preprocessor directive in your `main()` program. For more information on the `#pragma runopts` prepossessor directive, see "Specifying Run-Time Options Using the #pragma runopts Preprocessor Directive."

The precedence of run-time options for applications compiled with LE/VSE is described in the *LE/VSE Programming Reference*.

If two contradictory options are specified, the last option specified is accepted and the first ignored.

The values of all parameters are filled in successively from the system defaults and the `PARM` parameter of the `EXEC` statement.

For more information on the LE/VSE run-time options, see the *LE/VSE Programming Reference*.

Specifying Run-Time Options Using the #pragma runopts Preprocessor Directive

You can use the `#pragma runopts` preprocessor directive to override the default values for run-time options or to specify the run-time options: `ARGPARSE`, `ENV`, `PLIST`, `REDIR`, and `EXECOPS`. If the run-time option `EXECOPS` is in effect, run-time options specified in the `PARM` parameter of the `EXEC` statement override options specified on the `#pragma runopts` preprocessor directive.

If the run-time option `EXECOPS` is not in effect, all run-time options specified on the `PARM` parameter of the `EXEC` statement are treated as arguments to be passed to your program at execution time. For LE/VSE, `EXECOPS` is the default.

The `#pragma runopts` directive can appear in any file: main, include, or source, if it is before any other C source code. You may specify multiple run-time options per directive or multiple directives per compilation unit. If you wish to specify the `REDIR` or `ARGPARSE` options, the `#pragma runopts` directive must be in the same compilation unit as `main()`. When you specify multiple instances of `#pragma runopts` in separate compilation units, be aware that the compiler generates a CSECT. When you link multiple compilation units that specify `#pragma runopts`, the linker takes only the first CSECT, thereby ignoring your other option statements. Therefore, you should always keep your `#pragma runopts` specification in the same source as your `main()` program.

For more information on the `#pragma runopts` preprocessor directive, see the *C/VSE Language Reference*.

Chapter 6. C/VSE Example

This chapter contains an example of the basic steps for compiling, linking, and running a C/VSE program.

Example of a C/VSE Program

The following example shows a simple C program that converts temperatures in Celsius to Fahrenheit. You can either enter the temperatures on the `PARM` parameter of the `EXEC` JCL statement, or supply the temperature to be converted in the JCL input stream (`SYSIPT`).

In this example, the main program calls the function `convert()` to perform the conversion of the Celsius temperature to a Fahrenheit temperature and to print the result.

EDCXUAAA

```
 /* EDCXUAAA
    This is an example of a C/VSE program
  */

#include <stdio.h>                1

#include "edcxuaab.h"             2

void convert(double);             3

int main(int argc, char **argv)   4
{
    double c_temp;                5
    int ch, i;

    if (argc == 1) {  /* get Celsius value from stdin */

       printf("Enter Celsius temperature: \n");   6

       if (scanf("%f", &c_temp) != 1) {
          printf("You must enter a valid temperature\n");
       }
       else {
          convert(c_temp);        7
       }
    }
```

Figure 8 (Part 1 of 2). Celsius to Fahrenheit Conversion

```
    else {  /* convert the invocation arguments to Fahrenheit */

       for (i = 1; i < argc; ++i) {
           if (sscanf(argv[i], "%f", &c_temp) != 1)
              printf("%s is not a valid temperature\n",argv[i]);
           else
              convert(c_temp);    7
       }
    }
}

void convert(double c_temp) {     8
   double f_temp = (c_temp * CONV + OFFSET);
   printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);
}
```

Figure 8 (Part 2 of 2). Celsius to Fahrenheit Conversion

EDCXUAAB

```
/*************************************************************
 *  User include file:  EDCXUAAB.H                           *
 *************************************************************/  9

#define CONV   (9./5.)
#define OFFSET 32
```

Figure 9. User #include File for Conversion Program

1 This preprocessor directive includes the system file that contains the declarations of standard library functions, such as the `printf()` library function used by this program.

The compiler searches for the VSE Librarian sublibrary member `STDIO.H`. See “Search Sequences for Include Files” on page 7 for a description of the sublibraries used in the search.

2 This preprocessor directive includes a user file that defines constants that are used by the program.

The compiler searches for the VSE Librarian sublibrary member `EDCXUAAB.H`. See “Search Sequences for Include Files” on page 7 for a description of the sublibraries used in the search.

3 This is a function prototype declaration. This statement declares `convert()` as an external function having one parameter.

4 The program begins execution at this entry point.

5 This is the automatic (local) data definition to `main()`.

6 This `printf()` statement is a call to a C library function that allows you to format your output and print it on the standard output device. The `printf()` library function is declared in the LE/VSE run-time library standard I/O header file `stdio.h` included at the beginning of the program.

7 This statement contains a call to the function `convert()`. It was declared earlier in the program as receiving one double value, and not returning a value.

8 This is a function definition. In this example, the declaration for this function appears immediately before the definition of the `main()` function. The C code for the function is in the same file as the code for the `main()` function.

9 This is the user include file containing the definitions for `CONV` and `OFFSET`.

If you need more details on the constructs of the C language, see the *C/VSE Language Reference* and the *LE/VSE C Run-Time Library Reference*.

Compiling, Linking, and Running the C/VSE Example

If the sample C program shown on page 61 were stored in member `CTOF.C` in sublibrary `SHARON.SOURCE`, and your header file `EDCXUAAB.H` was found in sublibrary `SHARON.INCL`, you could use the following JCL to compile, link, and execute the source code:

EDCXUAAC

```
// JOB EDCXUAAC
// LIBDEF *,SEARCH=(PRD2.DBASE,PRD2.SCEEBASE) 1
// OPTION LINK
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='INFILE(DD:SHARON.SOURCE(CTOF.C)),   X
               LSEARCH(SHARON.INCL)'    2
/*
// EXEC LNKEDT
/*
// EXEC
27 3
/*
/&
```

Figure 10. JCL to Compile, Link, and Run the Conversion Program

1 You must have both C/VSE and the LE/VSE C run-time library in your `PHASE` sublibrary search chain.

You must have both C/VSE and the LE/VSE C run-time library in your `SOURCE` sublibrary search chain:

- The PRD2.SCEEBASE sublibrary contains the standard header files required for the compilation. Alternately, you can use the `SEARCH` compile-time option to specify the sublibrary containing the standard header files.
- The PRD2.DBASE sublibrary contains message files required for the compilation.

You must have the LE/VSE C run-time library in your `OBJ` sublibrary search chain.

Notes:

1. The above example uses the generic search chain "*" for simplicity.
2. The sublibrary names shown are the default installation sublibrary names. Contact your system programmer if the default names are not used at your installation.

2 The `INFILE` compile-time option specifies the primary input source file as a member of a VSE Librarian sublibrary.

The `LSEARCH` compile-time option specifies the sublibrary where the user include file is found.

3 This is the input data provided for program execution. Alternatively, this input can be provided through the `PARM` parameter of the `EXEC` JCL statement.

Note: The JCL for the compiler workfiles (`IJSYS01`-`IJSYS07`) and `SYSLNK` must be provided if these are not in the Partition Standard or System Standard label areas.

Appendix A. C/VSE Return Codes and Messages

This appendix contains information about the compile-time messages and should not be used as programming interface information.

Return Codes

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved:

Table 12. Return Codes from Compilation of a C Program

Return Code	Meaning
0	No error detected; informational messages may have been issued, compilation completed; successful execution anticipated.
4	Possible error detected; compilation completed, successful execution probable.
12	Error detected; compilation may have been completed, successful execution impossible.
16	Severe error detected; compilation terminated abnormally, successful execution impossible.
33	A library level prior to V1R4 was used; compilation terminated abnormally, successful execution impossible.

The return code indicates the highest error severity that is detected during a compilation. Therefore, a particular error type entry includes any error types with a lower return code. For example, return code 12 indicates that Severe error messages were issued, but it is also possible that Error, Warning, and Informational messages were issued.

Compiler Messages

Message format:

```
EDCnnnn ss text <&n>
```

where:

nnnn Is the error message number
ss Is the error severity as follows:
- 00 Informational message
- 10 Warning message
- 30 Error message
- 40 Severe error message
- 50 Fatal error message

text Is the message text
&n Is a compiler substitution variable

EDC0001 50 Internal compiler error at procedure *&1*.

Explanation: An error occurred during compilation. See the *C/VSE Diagnosis Guide* for a description of what to do.

EDC0010 50 Cannot load compiler phase *&1*.

Explanation: Fetching of compiler phase *&1* failed.

Recovery: Ensure that you have sufficient `GETVIS` storage available in the partition to run the compiler.

EDC0041 30 Identifier *&1* must be declared before it is used.

Explanation: A C identifier must be declared before it is used in an expression.

Recovery: Declare an identifier of that name in the current scope or in a higher scope.

EDC0042 30 A declaration must declare a variable, a tag, or enum members.

Explanation: A data type cannot be followed by a semicolon or an enum tag without the list of enum constants.

Recovery: Add a variable name to the data type or provide the list of enum constants to complete the declaration.

EDC0043 30 A type must be specified in the declaration of the object *&1*.

Explanation: A declaration must always specify the type of the object declared or the return type of the function declared. A missing type (an object name followed by a semicolon) does not default to int.

Recovery: Declare the object or function with a type.

EDC0044 30 An incomplete struct or union tag cannot be used in a declaration.

Explanation: Only pointer declarations can include incomplete types. A struct or union tag is undefined if the list describing the name and type of its members has not been specified.

Recovery: Define the tag before it is used in the declaration of an identifier or complete the declaration.

EDC0045 30 The enum constants must be specified when the enum tag is declared.

Explanation: When an enumeration tag is declared, the list of the enumeration constants must be included in the declaration.

Recovery: Add the list of enumeration constants in the enum tag declaration.

EDC0047 30 External objects cannot be initialized in a block.

Explanation: A variable has been declared at block scope with storage class extern, and has been given an explicit initializer. This is not permitted.

Recovery: Initialize the external object in the external declaration.

EDC0048 10 The function *&1* is not defined but has #pragma *&2* directive specified.

Explanation: The pragma inline and noinline directives are valid only for functions defined within the current compilation unit.

Recovery: Define the function.

EDC0049 10 The pragma *&1* directive has already been specified for function *&2*.

Explanation: Only one of pragma inline and pragma noinline can be specified for a single function. The second pragma specified will be ignored.

Recovery: Remove one of the conflicting pragmas.

EDC0050 10 A C reserved word cannot appear in a #pragma directive.

Explanation: A reserved word, such as int or break, has been used in a #pragma directive. This can cause unexpected results, so it is not permitted.

Recovery: Replace the reserved word by the name of a function or variable.

EDC0060 30 Width of bit-field *&1* must be less than or equal to 32 bits.

Explanation: The bit-field width must not exceed the maximum bit size of the bit-field type.

Recovery: Define the bit-field width to be less than or equal to 32 bits.

EDC0061 30 Bit-field *&1* must have type signed int or unsigned int.

Recovery: Define the bit-field with a type signed int or unsigned int.

EDC0062 30 A bit-field with a zero width must be unnamed.

Explanation: A named bit-field must have a positive width; a zero width bit-field is used for alignment only, and must not be named.

Recovery: Redefine the bit-field with a width greater than zero or remove the name of the bit-field.

EDC0063 30 Width of bit-field *&1* must be a constant expression.

Recovery: Replace the expression that specifies the width of the bit-field with a constant expression.

EDC0064 30 Width of bit-field *&1* must be positive.

Recovery: Replace the constant expression that specifies the width of the bit-field with a positive value.

EDC0065 30 Width of bit-field *&1* must be a constant integral expression.

Recovery: Replace the constant expression that specifies the width of the bit-field with a constant integral expression.

EDC0067 30 A struct or union member cannot be declared with a storage class.

Explanation: A storage class specifier was found in the declaration of a struct or union member. Members automatically take on the same storage class as their parent structure or union.

Recovery: Remove the storage class specifier from the member of the struct or union.

EDC0068 30 The *&1* definition must specify a member list.

Explanation: The declaration of a struct or a union that includes an empty member list enclosed between braces is not a valid struct or union definition.

Recovery: Specify the members of the struct or union in the definition or remove the empty braces to make it a simple struct or union tag declaration.

EDC0069 30 A member declaration must specify a name.

Explanation: A struct or union member declaration must specify a name. A type cannot be followed by a semicolon.

Recovery: Declare the member with a name.

EDC0080 30 An object cannot be cast to a struct, union or function type.

Explanation: An attempt was made to cast an operand to a struct, union, or function type.

Recovery: Use pointers instead of struct, union or function objects. Cast the pointer operand to a pointer to the struct, union, or function type.

EDC0081 30 A struct or union cannot be cast to another data type.

Recovery: Use a pointer to the struct or union as the operand.

EDC0082 30 The data conversion is not valid.

Explanation: The statement contains an expression that converts data to a type that is not valid. See the *LE/VSE C Run-Time Library Reference* for the table of correct data conversions.

Recovery: Check the type declaration of the indicated operand and the type of the conversion. Ensure the conversion is correct.

EDC0083 30 Only casts to arithmetic types are allowed in an arithmetic constant expression.

Explanation: In an arithmetic constant expression, casts must be to arithmetic types. Valid arithmetic types include: char, signed and unsigned int, enum, float, double, and long double.

Recovery: Remove the cast operator or change the cast to an arithmetic one.

EDC0084 30 The subscript must be a constant integral expression.

Recovery: Replace the expression that specifies the array subscript by a constant integral expression.

EDC0097 30 Pointers to void and pointers to function are not assignment compatible.

Recovery: Ensure that your function declarations are correct.

EDC0098 30 Pointers to void and pointers to function cannot be compared.

Recovery: Check the logic of the comparison.

EDC0099 30 A pointer to an incomplete type cannot be subscripted.

Recovery: Define the type before you reference it.

EDC0100 30 Operand of bitwise complement must have integral type.

Explanation: The operand of the bitwise complement operator does not have an integral type. Valid integral types include: signed and unsigned char; signed and unsigned short, long, and int; and enum.

Recovery: Change the type of the operand, or use a different operand.

EDC0101 30 Operand of unary minus operator must have arithmetic type.

Explanation: The operand of the unary minus operator (-) does not have an arithmetic type. Valid arithmetic types include: signed and unsigned char; signed and unsigned short, long, and int; enum, float, double, and long double.

Recovery: Change the type of the operand, or use a different operand.

EDC0102 30 Operand of logical negation must have scalar type.

Explanation: The operand of the logical negation operator (!) does not have a scalar type. Valid scalar types include: signed and unsigned char; signed and unsigned short, long, and int; enum, float, double, long double, and pointers.

Recovery: Change the type of the operand, or use a different operand.

EDC0103 10 The size of this type is zero.

Explanation: The sizeof operator cannot be used on a void type. If it is, the compiler returns zero for the size of the expression.

Recovery: Ensure that sizeof() is used on a valid type.

EDC0104 30 Only pointers to compatible types can be subtracted.

Explanation: The expression must contain pointers to compatible data types. See the *C/VSE Language Reference* for the rules on compatible types.

Recovery: Ensure that the pointers point to compatible data types.

EDC0105 30 Operand of address operator must be a function or an lvalue.

Explanation: The operand of the address operator (unary &) is not valid. The operand must be either a function designator or an lvalue that designates an object that is not a bit-field and is not declared with register storage class.

Recovery: Change the operand.

EDC0106 30 The sizeof operator cannot be used with a function, void or bit-field.

Explanation: The operand of the sizeof operator is not valid. The sizeof operator cannot be applied to an expression that has a function type or an incomplete type, to the parenthesized name of such a type, or to an lvalue that designates a bit-field object.

Recovery: Change the operand.

EDC0107 30 Operand of indirection operator must be a pointer.

Explanation: The operand of the indirection operator (unary *) is not a pointer.

Recovery: Change the operand.

EDC0108 30 Operand of arrow operator must be a pointer to a struct or union.

Explanation: The left hand operand of the arrow operator (->) must have type "pointer to structure" or "pointer to union".

Recovery: Change the operand.

EDC0109 30 The subscript must be an integral expression.

Explanation: The subscript expression must have integral type. Valid integral types include: char, signed and unsigned int, and enum.

Recovery: Change the subscript expression to have an integral type.

EDC0110 30 Operand of dot operator must be a struct or a union.

Explanation: The left hand operand of the dot (.) operator is not of type struct or union.

Recovery: Change the operand.

EDC0111 30 Identifier *&1* must be a member of the struct or union.

Explanation: The specified member does not belong to the structure or union given. One of the following has occurred:

- The right hand operand of the dot (.) operator is not a member of the structure or union specified on the left hand side of the operator
- The right hand operand of the arrow (->) operator is not a member of the structure or union pointed to by the pointer on the left hand side of the operator

Recovery: Change the identifier.

EDC0112 30 The expression must be a function designator.

Explanation: The expression is followed by an argument list but does not evaluate to a function designator.

Recovery: Change the expression to be a function or a pointer to a function.

EDC0113 30 Operand must have integral type.

Explanation: The operand of the bitwise operators or modulus (%) operator must have integral type. Valid integral types include: char, signed and unsigned int, and enum.

Recovery: Change the operand.

EDC0114 30 Operand must be a modifiable lvalue.

Explanation: See the *C/VSE Language Reference* for a description of lvalue.

Recovery: Change the operand.

EDC0115 30 A struct or union can be assigned only to a compatible type.

Explanation: Two structures have compatible types if both have been declared with the same structure tag. Two unions have compatible types if both have been declared with the same union tag. However, tags are scope sensitive. Even if two tag names and their member lists are identical, if their definitions are located in different scopes, the types associated with these tags are different.

Recovery: Ensure that the structures or unions used in the assignment have been declared with the same tag in the same scope.

EDC0116 30 Identifier *&1* cannot be redeclared as an enum tag.

Explanation: In the declaration, the object is declared to be an enum tag. The object was previously declared to the tag of a struct or union type.

Recovery: Change the name of the tag.

EDC0117 30 The operation between these types is not valid.

Explanation: The identifiers on the left hand side and the right hand side of the operator have types that do not conform to the restrictions of the operator. The operation specified in the expression cannot be performed. See the *C/VSE Language Reference* for the list of operator restrictions.

Recovery: Change the operands.

EDC0118 30 The divisor for the modulus or division operator cannot be zero.

Explanation: The value of the divisor expression cannot be zero.

Recovery: Change the expression used as the divisor.

EDC0119 30 The void pointer must be cast prior to this operation.

Explanation: A void pointer must be cast to a data type before it is used in this operation.

Recovery: Cast the pointer to a type other than void prior to this operation.

EDC0120 30 Operand of unary plus operator must have arithmetic type.

Explanation: The operand of the unary plus operator (+) does not have an arithmetic type. Valid arithmetic types include: signed and unsigned char; signed and unsigned short, long, and int; enum, float, double, and long double.

Recovery: Change the operand.

EDC0121 30 Operand must have scalar type.

Explanation: The operand for this operation does not have scalar type. Valid scalar types include: signed and unsigned char; signed and unsigned short, long, and int; enum, float, double, long double, and pointers.

Recovery: Change the type of the operand, or use a different operand.

EDC0122 30 Operand must have arithmetic type.

Explanation: The operand of this operation does not have arithmetic type. Valid arithmetic types include: signed and unsigned char; signed and unsigned short, long, and int; enum, float, double, and long double.

Recovery: Change the operand.

EDC0123 30 If one operand is void, the other must be void.

Explanation: If one operand in the conditional expression has type void, the other operand must also have type void.

Recovery: Make the operands compatible.

EDC0125 30 Operands of the conditional operator must have compatible types.

Explanation: If one operand of the conditional expression has type struct or union, the other operand must be a struct or union declared using the same tag in the same scope.

Two structures have compatible types if both have been declared with the same structure tag. Two unions have compatible types if both have been declared with the same union tag. However, tags are scope sensitive. Even if two tag names and their member lists are identical, if their definitions are located in different scopes, the types associated with these tags are different.

Recovery: Ensure that the structures or unions used in the conditional expression have been declared/defined with the same tag (in the same scope).

EDC0126 30 If one operand is a pointer, the other must also be a pointer.

Explanation: If one of the result operands of a conditional expression is a pointer, the other result operand must be either a pointer to the same qualified or unqualified type, a NULL pointer, or a pointer to void.

Recovery: Change the operands.

EDC0127 30 If the operands are pointers, they must point to compatible types.

Explanation: If one operand of either the relational or the equality operator is a pointer, the other operand must be either a pointer to the same qualified or unqualified type, a NULL pointer, or a pointer to void.

Recovery: Change the operands.

EDC0128 30 Two pointers cannot be added.

Explanation: The addition operator requires that either both operands have arithmetic type or, if one of the operands is a pointer, the other one must have integral type. Valid integral types include: char, signed and unsigned int, and enum. Valid arithmetic types include: the integral types plus float, double, long double, and bit fields.

Recovery: Change the operands.

EDC0130 30 The operation cannot be performed on an incomplete struct or union.

Explanation: The definition of the operand must be completed prior to this operation. A structure or union type is completed when the definition of its tag is specified. A struct or union tag is defined when the list describing the name and type of its members is specified.

Recovery: Define the tag before using it in an expression.

EDC0131 30 Subtraction between void pointers is not allowed.

Recovery: Cast the pointers to a type other than void or do not subtract them.

EDC0132 30 A pointer to void cannot be subscripted.

Explanation: The subscript operator requires a pointer to a valid address.

Recovery: Cast the pointer to a type other than void before using it with the subscript operator.

EDC0133 30 An identifier cannot be declared in a cast or sizeof expression.

Explanation: Only abstract declarators can appear in cast or sizeof expressions.

Recovery: Remove the identifier from the cast or sizeof expression and replace it with an abstract declarator.

EDC0136 30 The sizeof operator cannot be used with arrays of unknown size.

Recovery: Ensure the array and its size have been declared before using it with the sizeof operator.

EDC0137 30 The indirection operator cannot be applied to a pointer to an incomplete struct or union.

Explanation: Except for pointers, it is not valid to declare an object of incomplete structure or union type. A structure or union is incomplete when the definition of its tag has not been specified. A struct or union tag is undefined when the list describing the name and type of its members has not been specified.

Recovery: Define the tag before using it in the declaration of an identifier.

EDC0138 30 The indirection operator cannot be applied to a void pointer.

Explanation: The indirection operator requires a pointer to a valid address.

Recovery: Cast the pointer to a type other than void before using it with the indirection operator.

EDC0139 10 A translation unit must contain at least one external declaration.

Explanation: A translation unit that does not contain any external declaration will not be linked to. Normally it will not affect the execution of the executable program.

Recovery: Ensure this is what was intended or change the appropriate declarations to the external ones.

EDC0140 00 Operand has type *&1*.

Explanation: An error has occurred due to conflicting operands. This message states the type of the operand used in the expression.

Recovery: No recovery is necessary if this result was intended. Change the type of the operand if necessary.

EDC0141 00 Prototype has type *&1*.

Explanation: An error has occurred due to conflicting function declarations. This message states the type of the prototype declaration.

Recovery: No recovery is necessary if this result was intended. Change the type of the prototype if necessary.

EDC0142 00 Previous declaration has type *&1*.

Explanation: An error has occurred due to conflicting identifier declarations. This message states the type of the identifier in the current declaration.

Recovery: No recovery is necessary if this result was intended. Change the declarations to the same type if necessary.

EDC0143 30 The pre- and post- increment and decrement operators cannot be applied to void pointers.

Explanation: Pointers to void cannot be incremented or decremented.

Recovery: Cast the pointer to a type other than void before using it with any of the increment or decrement operators.

EDC0145 00 Redeclaration has type *&1*.

Explanation: An error has occurred because of conflicting declarations. This message states the type of the identifier in the redeclarations.

Recovery: No recovery is necessary if this result was intended. Change the types in the declarations to be compatible, if necessary.

EDC0146 00 Function has return type *&1*.

Recovery: No recovery is necessary if this result was intended. Change the return type if necessary.

EDC0147 00 Argument has type *&1*.

Explanation: The argument type in the function call conflicts with the parameter in the function prototype. This message states the type of the argument.

Recovery: No recovery is necessary if this result was intended. Change the argument types to be compatible, if necessary.

EDC0148 00 Expression has type *&1*.

Recovery: Informational message. No recovery is necessary if this result was intended.

EDC0149 30 Operation not allowed with enum that is not defined.

Explanation: The sizeof or cast operator cannot be used with an enum that is not defined.

Recovery: Define the enum.

EDC0150 30 The number of initializers cannot be greater than the number of elements.

Explanation: Too many initializers were found in the initializer list for the indicated declaration.

Recovery: Check the number of initializers. Check the closing brace at the end of the initializer list to ensure that it has been positioned correctly.

EDC0151 30 The initializer must be a constant expression.

Explanation: The initializers for identifiers of static storage duration, or for identifiers of an array, structure, or union type must be constant expressions.

Recovery: Remove the initialization or change the indicated initializer to a constant expression.

EDC0152 30 A register array may only be used as the operand to sizeof.

Explanation: The only operator that may be applied to a register array is sizeof.

Recovery: Remove the register keyword from the declaration.

EDC0153 30 An initializer for a static identifier cannot have the automatic storage class.

Explanation: The initializer cannot have an automatic storage class if the identifier being initialized has a static storage class.

Recovery: Either change the storage class in the identifier declaration or change the initializer.

EDC0158 00 After widening, previous declaration has type *&1*.

Explanation: An error has occurred due to conflicting identifier declarations. This message states the type of the identifier in the current declaration, after the identifier's type has been widened.

Recovery: No recovery is necessary if this result was intended. Change the declarations to the same type if necessary.

EDC0159 00 After widening, redeclaration has type *&1*.

Explanation: An error has occurred because of conflicting declarations. This message states the type of the identifier in the redeclarations, after the identifier's type has been widened

Recovery: No recovery is necessary if this result was intended. Change the types in the declarations to be compatible, if necessary.

EDC0166 30 A non-lvalue array cannot be subscripted.

Explanation: Subscript operator cannot be used with a non-lvalue array.

Recovery: Change the array to an lvalue one.

EDC0167 30 A non-lvalue array cannot be used in this context.

Explanation: The location of a non-lvalue array may not be referenced.

Recovery: Change the array to an lvalue one.

EDC0168 30 Operation not valid on a function pointer.

Explanation: Subscript operator, additive operator, and prefix and postfix increment and decrement operators cannot be used with an operand of type pointer to function.

Recovery: Change the operator or the operand.

EDC0170 30 A function cannot be initialized.

Explanation: An attempt was made to assign an initial value to a function identifier.

Recovery: Remove the assignment operator and the initializer.

EDC0171 30 A function cannot return a function.

Explanation: A function cannot have a return type of function.

Recovery: Return a pointer to the function or specify a different return type.

EDC0172 30 Function *&1* cannot have a storage class of auto or register.

Recovery: Remove the storage class specifier for the function identifier, or change it to either extern or static.

EDC0173 30 A function cannot be a member of a struct or union.

Recovery: Use a pointer to the function or remove the function from the member list.

EDC0174 30 A function cannot be an element of an array.

Recovery: Use a pointer to the function, or change the type of the element.

EDC0175 30 A function cannot return a *&1* qualified type.

Explanation: The const or volatile qualifier cannot be used to qualify a function's return type.

Recovery: Remove the qualifier or return a pointer to the qualified type.

EDC0176 30 Return type must be compatible with the declaration of function *&1*.

Explanation: The return statement of the function tries to return a structure or a union type that is not compatible with the return type specified in the function declaration/definition. The type of a structure or union is represented by its tag.

Two structures have compatible types if both have been declared with the same structure tag. Two unions have compatible types if both have been declared with the same union tag. However, tags are scope sensitive. Even if two tag names and their member lists are identical, if their definitions are located in different scopes, the types associated with these tags are different.

Recovery: Ensure that the same tag (in the same scope) is used in the function declaration/definition, as well as in the declaration/definition of the value specified on the return statement,

EDC0177 30 A function declared to return void cannot return a value.

Explanation: When a function is declared to have a void return type, the return statement of the function cannot return any value. An attempt was made to return a value in a function that was declared/defined with a void return type.

Recovery: Change the declaration to specify the return type or do not return a value.

EDC0178 30 A function cannot return an array.

Recovery: Return a pointer to the array or specify a different return type.

EDC0179 30 The function *&1* cannot be redefined.

Explanation: It is not valid to define a function more than once. Do not confuse function definitions and function declarations. A declaration describes the return type of the function. A definition is a declaration followed by the code that is to be executed when the function is called (the code portion is called the function body). Only one definition per function is allowed.

Recovery: Remove all extra definitions or change the name of the function.

EDC0180 30 The static function *&1* is referenced but is not defined in this file.

Explanation: A static function was declared and referenced in this file. The definition of the function was not found before the end of the file. When a function is declared to be static, the function definition must appear in the same file.

Recovery: Define the function or remove the static storage class.

EDC0181 30 The struct, union, or enum tag *&1* cannot be redefined.

Explanation: A struct or union tag is defined when it is declared with the list describing the name and type of its members. An enum tag is defined when it is declared with the list of its enumeration constants. It is not valid to define a struct, union, or enum tag more than once in the same scope.

Recovery: Remove all extra definitions or rename the tag.

EDC0183 30 An argument cannot be an incomplete struct or union.

Explanation: The argument has an incomplete struct or union type. A structure or union is incomplete when the definition of the tag (that is, when the number and the type of its members) has not been specified. It is not valid to pass arguments of incomplete type to a function.

Recovery: Use a pointer to the incomplete type or define the type before using it.

EDC0184 30 An argument cannot have type void.

Explanation: The indicated parameter has type void. A parameter of type void cannot be passed on a function call.

Recovery: Use a pointer to void or cast the type of the argument.

EDC0185 10 Function *&1* has not been prototyped prior to use.

Explanation: A prototype declaration of the function specifying the number and type of the parameters was not found before the function was used. Errors may occur if the function call does not respect the function definition.

Recovery: Include a prototype declaration of the function before calling it.

EDC0187 30 The declaration or definition of the function is not valid.

Explanation: The compiler cannot read the declaration. It assumes that the function declaration was not valid. The return type or the parameters may have been specified incorrectly.

Recovery: Check for incorrect spelling or missing parentheses.

EDC0188 30 The return type is not valid for a function of this linkage type.

Recovery: Use a valid return type.

EDC0189 30 The return type of the function main must have type int.

Recovery: Change the return type of function main to int.

EDC0190 30 A switch expression must have integral type.

Explanation: The controlling expression in a switch statement must have integral type. Valid integral types include: char, signed and unsigned int, and enum.

Recovery: Change the expression.

EDC0191 30 A case label must be a constant integral expression.

Explanation: The expression in the case statement must be a constant integral expression. Valid integral expressions are: char, signed and unsigned int, and enum.

Recovery: Change the expression.

EDC0192 30 The case label cannot be a duplicate of the case label on line *&1*.

Explanation: Two case labels in the same switch statement cannot evaluate to the same integer value.

Recovery: Change one of the labels.

EDC0193 30 A default case label cannot be placed outside a switch statement.

Recovery: Remove the default case label, or place it inside a switch statement. Check for misplaced braces on a previous switch statement.

EDC0194 30 A switch statement cannot contain more than one default statement.

Recovery: Remove one of the default statements.

EDC0195 30 A case statement cannot be placed outside a switch statement.

Recovery: Remove the case statement, or place it within a switch statement group. Check for misplaced braces on the previous switch statement.

EDC0196 00 The case label evaluates to integer value *&1*.

Explanation: An error occurred due to conflicting case labels. This message states the value of the case labels.

Recovery: Change the case label if necessary.

EDC0198 30 If the operands are pointers they must point to compatible objects or incomplete types.

Explanation: If both operands of a relational operator are pointers, they must point to qualified or unqualified versions of compatible object or incomplete types. Pointers to functions are not allowed.

Recovery: Change the operands.

EDC0200 30 A break statement cannot be placed outside a while, do, for, or switch statement.

Recovery: Remove the break statement or place it inside a while, do, for or switch statement. Check for misplaced braces on a previous statement.

EDC0201 30 A continue statement cannot be placed outside a while, do, or for loop.

Recovery: Remove the continue statement or place it inside a while, do or for loop. Check for misplaced braces on a previous loop.

EDC0220 30 Only arrays and pointers to object types can be subscripted.

Explanation: An attempt was made to subscript an identifier that was not an array or a pointer to an object type.

Recovery: Remove the subscripts or change the identifier.

EDC0221 30 Array size must be a positive constant integral expression.

Explanation: The array size declared is not valid. If compilation continues, the compiler will assume that the array has size 1.

Recovery: Make the array size a positive constant integral expression.

EDC0222 30 Arrays cannot be redeclared with a different size.

Recovery: Make the size consistent with the previous declaration or remove one of the array declarations.

EDC0223 30 All array dimensions except the first must be specified.

Explanation: Only the first dimension of an initialized array may be unspecified. All the other dimensions must be specified on the declaration.

Recovery: Specify all the other dimensions in the array declaration.

EDC0224 30 All dimensions must be specified for array definitions.

Explanation: All the dimensions of arrays of automatic or static storage class must be specified on the declaration. If the declaration of the automatic or static array provides an initialization, the first dimension may be unspecified because the initialization will determine the size needed.

Recovery: Specify all of the dimensions in the array declaration.

EDC0225 30 Arrays that are members must have all dimensions specified.

Explanation: Arrays that are struct or union members must have all dimensions specified in the array declaration.

Recovery: Specify all of the dimensions in the array declaration.

EDC0226 30 The parameter lists of the function pointers are not compatible.

Explanation: In assignment or initialization of function pointers, the parameter lists of the function pointers must have compatible type.

Recovery: Ensure that the parameter lists of the function pointers are compatible.

EDC0227 30 The return types of the function pointers are not compatible.

Explanation: In assignment or initialization of function pointers, the return types of the function pointers must have compatible type.

Recovery: Ensure that the return types of the function pointers are compatible.

EDC0228 30 The linkage types of the function pointers are not compatible.

Explanation: In assignment or initialization of function pointers, the linkage types of the function pointers must be compatible.

Recovery: Ensure that the linkage types of the function pointers are compatible.

EDC0240 10 Escape sequence is out of range for character representation.

Explanation: Character constants specified in an escape sequence exceeded the decimal value of 255, or the octal equivalent of 377, or the hexadecimal equivalent of FF.

Recovery: Change the escape sequence so that the value does not exceed the maximum value.

EDC0242 40 Nesting cannot exceed the maximum limit *&1*.

Explanation: The internal compiler limit of *&1* nested #include files was exceeded.

Recovery: Remove the nesting by putting all of the #include files at the same level, or reduce the number of nesting levels.

EDC0244 10 External name *&1* has been truncated to *&2*.

Explanation: The external object has a name *&1* which exceeds the limit and has been truncated to the name *&2*.

Recovery: Change the name if necessary.

EDC0246 30 Floating point constant is out of range.

Explanation: The compiler detected a floating-point overflow either in scanning a floating-point constant, or in performing constant arithmetic folding.

Recovery: Change the floating-point constant so that it does not exceed the maximum value.

EDC0247 40 Virtual storage exceeded.

Explanation: The compiler ran out of memory trying to compile the file. This sometimes happens with large files or programs with large functions. Note that very large programs limit the amount of optimization that can be done.

Recovery: Shut down any large processes that are running, ensure your swap path is large enough, turn off optimization, and redefine your virtual storage to a larger size. You can also divide the file into several small sections or shorten the function.

EDC0248 30 External name *&1* cannot be redefined.

Explanation: C/VSE will generate this message if you have used an external identifier in both uppercase and lowercase, such as,

```
extern FUNC(); extern func() {}
```

Because the linker on System/370 does not distinguish between upper and lowercase characters, these two declarations will be mapped to the same name, but C recognizes these two identifiers as different identifiers. If you intended the upper and lowercase external identifiers to map to the same identifier, use the #pragma map directive. For example,

```
#pragma map(FUNC, "func")
```

Recovery: Remove one of the definitions or change one of the names.

EDC0250 10 The maximum number of errors for one line has been exceeded.

Explanation: The compiler is unable to specify the location of each error in the listing because there are too many errors on one line.

Recovery: Correct the errors or split the source line into multiple lines.

EDC0251 30 The physical size of an array is too large.

Explanation: The compiler cannot handle any size which is too large to be represented internally.

Recovery: Reduce the size of the array.

EDC0252 30 The physical size of a struct or union is too large.

Explanation: The compiler cannot handle any size which is too large to be represented internally.

Recovery: Reduce the sizes of the struct or union members.

EDC0260 30 Declaration cannot specify multiple sign type specifiers.

Explanation: A declaration can specify a signed or unsigned type, but not both.

Recovery: Keep only one sign type specifier.

EDC0261 30 Declaration cannot specify multiple length type specifiers.

Explanation: A declaration can specify a long or short type, but not both.

Recovery: Keep only one length type specifier.

EDC0262 30 Declaration cannot specify multiple type specifiers.

Explanation: A declaration can specify only one data type specifier. Valid specifiers include: char, int, float, and double.

Recovery: Keep only one type specifier.

EDC0265 30 Declaration cannot specify multiple storage class specifiers.

Explanation: A declaration can specify only one storage class. Valid storage classes include: auto, static, extern, register, and typedef.

Recovery: Use only one storage class specifier.

EDC0266 30 The *&1* type specifier cannot be used with float or double.

Explanation: The type specifiers signed, unsigned and short cannot be used with type float or double.

Recovery: Ensure that the appropriate type is used, and remove the incorrect type specifier from the declaration. Use type long double if a larger identifier is required.

EDC0268 30 The long type specifier cannot be used with float.

Recovery: Remove the long type specifier or use double instead of float.

EDC0269 30 The long or short type specifier cannot be used with char.

Recovery: Remove the length type specifier. Use type int or short int if a larger identifier is required.

EDC0270 30 The *&1* type specifier cannot be used with void.

Explanation: No other type specifier can be used with type void.

Recovery: Remove the type specifier or the void.

EDC0272 30 The *&1* type specifier cannot be used with struct, union, or enum.

Explanation: No other type specifiers can be used with struct, union or enum.

Recovery: Remove the type specifier.

EDC0274 30 The *&1* type specifier cannot be used for variables declared with a typedef.

Explanation: No other type specifiers can be used for variables declared with a typedef.

Recovery: Remove the type specifier or the typedef.

EDC0277 30 _Packed can only qualify a struct or union.

Recovery: Remove the _Packed specifier from the declaration/definition, or ensure it qualifies a struct or union.

EDC0278 30 Declaration cannot specify multiple *&1* specifiers.

Recovery: Ensure that only one *&1* specifier is used.

EDC0280 30 The predefined macro *&1* cannot be redefined.

Explanation: The macro *&1* is predefined. Predefined macros cannot be redefined.

Recovery: Remove the redefine statement.

EDC0281 30 The identifier *&1* cannot be redeclared.

Explanation: Only external objects can be redeclared.

Recovery: Delete or change the name of the extra declaration.

EDC0282 30 The struct member *&1* cannot be redeclared.

Explanation: The same struct member cannot be redeclared. To redeclare the structure itself, the same tag must be used.

Recovery: Delete or change the name of the extra declaration.

EDC0283 30 The tag *&1* cannot be redefined as a tag of another type.

Explanation: The tag is already associated with another struct, union, or enum type.

Recovery: Delete or rename the tag.

EDC0284 30 The label *&1* cannot be redefined.

Explanation: The label has already been defined in the function (a label of the same name followed by a colon and a section of code already appeared in the same function). It is not valid to redefine a label.

Recovery: Change the name of one label.

EDC0285 30 #undef cannot be used with the predefined macro *&1*.

Explanation: Predefined macros cannot be undefined.

Recovery: Delete the #undef directive.

EDC0286 30 The redeclaration cannot specify a different storage class.

Explanation: The redeclaration, including type qualifiers (const, volatile), must be identical to the first declaration.

Redeclaring basic types: The type (which includes the type specifiers and the length and sign adjectives) and the type qualifiers (const, volatile) must be the same.

Redeclaring functions: The return type with its type qualifiers has to be the same. If the function has been prototyped, the prototyped redeclarations must have an identical parameter list (the number and type of the parameters must be the same).

Redeclaring pointers: They have to point at the same type (including the type qualifiers).

Redeclaring arrays: Their members must be of the same type (including the type qualifiers). The array size must be the same.

Redeclaring enumerations, structures, and unions: They must have the same tag.

Recovery: Ensure that the storage class of the subsequent declarations matches the original declaration or remove one of the declarations.

EDC0287 30 The goto label is not defined in function *&1*.

Explanation: The goto label is referenced but not defined in the function. The label definition (label followed by a colon and a section of code) must appear in the same function that references the label.

Recovery: Define the goto label in the function or remove the reference.

EDC0288 30 The void type can only be used with functions and pointers.

Explanation: The type void can only be used as the return type or parameter list of a function, or with a pointer indicating the type to which it is pointed. No other object can be of type void.

Recovery: Ensure that the declaration uses type void correctly.

EDC0289 30 The typedef name *&1* cannot be redefined.

Explanation: Redefinitions of typedef names are not allowed even if the definitions occur at file scope with identical type specifiers.

Recovery: Remove identical definitions or, for a new definition, rename the typedef.

EDC0291 30 The *&1* storage class cannot be used with external identifier *&2*.

Explanation: Identifiers may only be declared with auto or register storage class if they are declared inside a block.

Recovery: Remove the storage class specifier or change the scope of the identifier so that it is no longer at file scope.

EDC0292 30 The block scope declaration of object *&1* must be compatible with its external declaration.

Explanation: This block scope redeclaration of the external object is incompatible with the previous external declaration.

Recovery: Ensure that the block scope declaration is identical with the file scope declaration, or remove one of the declarations.

EDC0293 30 The static storage class cannot be used with functions declared at block scope.

Recovery: Place the declaration of the static function at file scope, or remove the storage class specifier.

EDC0294 30 The typedef storage class cannot be used on function definitions.

Explanation: The typedef storage class can only be used with function declarations to declare a function type. A typedef name cannot carry the information of a function definition; it cannot specify the part of code to be executed when a function is called.

Recovery: Remove the typedef storage class.

EDC0295 30 The external name *&1* must not conflict with the name in #pragma *&2*.

Explanation: The name specified in the #pragma directive is an external name that must not conflict with any other visible external names.

Recovery: Specify a different name in the external name or in the #pragma directive.

EDC0296 30 The external name *&1* in #pragma *&2* must not conflict with the name in #pragma *&3*.

Explanation: The name specified in the #pragma directive is an external name that must not conflict with any other visible external names.

Recovery: Specify a different name for one of the external names.

EDC0297 30 Only functions or typedefs of functions can be specified on #pragma linkage directive.

Explanation: The parameter specified on the #pragma linkage directive is not a function or typedef of a function.

Recovery: Change the name specified on the #pragma linkage directive or remove the directive.

EDC0298 30 A #pragma *&1* directive was previously specified for the object *&2*.

Explanation: More than one #pragma linkage directive was specified for the same object.

Recovery: Remove the extra #pragma linkage directives.

EDC0299 30 A map name was previously given to the object *&1*.

Explanation: An object can map to only one name. See the *C/VSE Language Reference* for more information on #pragma map.

Recovery: Remove the extra #pragma map directives.

EDC0300 30 The floating point constant is not valid.

Explanation: See the *C/VSE Language Reference* for a description of a floating-point constant.

Recovery: Ensure that the floating-point constant does not contain any characters that are not valid.

EDC0301 30 A const qualified object cannot be modified.

Explanation: The value of a const cannot be changed. Increment/decrement can only be performed on objects that are not constants.

Recovery: Either do not declare the object with the const type qualifier, or do not use the object in an increment/decrement operation.

EDC0305 30 The identifier *&1* in #pragma environment must be a function name.

Recovery: Change the identifier to that of a function name.

EDC0306 10 #pragma linkage *&1* ignored for function *&2*.

Recovery: Remove the directive.

EDC0308 30 An enum constant must be an integral constant expression that has a value representable as an int.

Explanation: If an enum constant is initialized in the definition of an enum tag, the value that the constant is initialized to must be an integral expression that has a value representable as an int.

Recovery: Remove the initial value, or ensure that the initial value is an integral constant expression that has a value representable as an int.

EDC0312 10 Value *&1* specified in #pragma *&2* is out of range.

Explanation: In #pragma margins and #pragma sequence, the value specified for the right margin or sequence column must be greater than or equal to the value specified for the left margin or sequence column. The values specified for the left and right margins or sequence columns must lie in the range 1 to 32767.

Recovery: Change the value specified for the left or right margin or sequence column.

EDC0321 30 Redeclaration has a different number of parameters than the previous declaration.

Explanation: The prototyped redeclaration of the function is not correct. The redeclaration must specify the same number of parameters as the previous declaration.

Recovery: Make the redeclaration consistent with the original declaration.

EDC0322 30 Type of the parameter *&1* cannot conflict with previous declaration of function *&2*.

Explanation: The type of this parameter is incompatible with the type of the corresponding parameter in the previous declaration of the function.

Recovery: Ensure that the subsequent declaration or function call matches the prototype in both the number and type of parameters. If the parameter in the prototype is an incomplete struct or union tag, declare the incomplete tag at file scope before the function is prototyped.

EDC0323 30 Redeclaration cannot specify fewer parameters before ellipsis than the previous declaration.

Explanation: The prototyped redeclaration of the function is not correct. Fewer parameters appear before the ellipsis in this function redeclaration than the previous declaration.

Recovery: Ensure that the redeclaration is consistent with the previous declaration.

EDC0324 30 The void type specifier cannot be used with other type specifiers.

Explanation: When void is used in the parameter list of a prototyped function declaration, it indicates that the function does not expect any parameters. Therefore, if void is used in a prototyped declaration, it must be the only type descriptor in the parameter list and must not appear more than once in the list.

Recovery: If the function does not require any parameters, use void only once in the parameter list. If the function requires parameters, remove void from the parameter prototype list.

EDC0325 30 The type of the parameters must be specified in a prototype.

Explanation: A prototype specifies the number and the type of the parameters that a function requires. A prototype that does not specify the type of the parameters is not correct, for example,

```
fred(a,b);
```

Recovery: Specify the type of the parameters in the function prototype.

EDC0326 30 The only storage class that can be used with parameters is register.

Recovery: Remove the storage class specified in the parameter declaration or use the register storage class.

EDC0327 30 Redeclarations and function calls must be compatible with prototype.

Explanation: The number or the type of the parameters (or both) on the call does not agree with the specification given in the function prototype declaration.

Recovery: Make the call consistent with the declaration.

EDC0328 30 The function call cannot have more arguments than the prototype specifies.

Explanation: The function call is not valid. There are more arguments in this function call than there were parameters specified in the function declaration.

Recovery: Make the call consistent with the declaration.

EDC0329 30 Object *&1* must be specified in the parameter list for function *&2*.

Explanation: For function definitions that do not use the prototype style, a list of parameter names usually appears between the parentheses following the function name. A list of declarations that indicates the type of the parameters follows. In this case, the declaration of an object that was not listed between the parentheses was found in the parameter declaration list.

Recovery: Ensure that the declaration list only specified parameters that appear between the parentheses of the function.

EDC0330 30 A parameter cannot be declared when function *&1* parentheses are empty.

Explanation: For function definitions that do not use the prototype style, a list of parameter names usually appears between parentheses following the function name. A list of declarations that indicates the type of the parameters follows. In this case, objects are declared in the parameter declaration list but no parameter appeared between the function parentheses.

Recovery: Ensure that the declaration list only specifies parameters that were listed between the function parentheses.

EDC0331 30 Parentheses must appear in the declaration of function *&1*.

Explanation: The syntax of the declaration is not correct. The compiler assumes it is the declaration of a function in which the parentheses surrounding the parameters are missing.

Recovery: Check the syntax of the declaration. Ensure the object name and type are properly specified. Check for incorrect spelling or missing parentheses.

EDC0333 30 The parameters in the definition of the function *&1* must be named.

Explanation: For function definitions, all the parameters in the parameter list must be named. It is not valid to specify only the parameter's type in a function definition head.

Recovery: Name the parameters in the parameter list.

EDC0334 30 External identifier *&1* cannot be initialized more than once.

Recovery: Check the previous declarations of the object. Ensure that only one declaration specifies an initializer.

EDC0335 30 The declarations of the function *&1* must be consistent in their use of the ellipsis.

Explanation: If an ellipsis is used in a function declaration, the ellipsis must be present in all the function redeclarations. If no ellipsis is used in a function declaration, the following redeclarations cannot specify an ellipsis. Any redeclaration that does not use the ellipsis consistently is not correct.

Recovery: Make the redeclaration consistent with the previous declaration.

EDC0337 30 Declaration list cannot appear when parameters in parentheses are prototyped.

Explanation: For function definitions that do not use the prototype style, a list of parameter names usually appears between parentheses following the function name. A list of declarations that indicates the type of parameters follows. In this case, the parameters between the parentheses are prototyped. These two styles of declaration cannot be mixed.

Recovery: Remove either the function declaration list or the type given to the parameters in the function parentheses.

EDC0338 30 Prototype *&1* must contain widened types if prototype and nonprototype declarations are mixed.

Explanation: Nonprototype function declarations, popularly known as K&R prototypes, only specify the function return type. The function parentheses are empty; no information about the parameters is given.

Nonprototype function definitions specify a list of parameter names appearing between the function parentheses followed by a list of declarations (located between the parentheses and the opening left brace of the function) that indicates the type of the parameters.

A nonprototype function definition is also known as a K&R function definition.

A prototype function declaration or definition specifies the type and the number of the parameters in the parameter declaration list that appears inside the function parenthesis. A prototype function declaration is better known as an ANSI prototype, and a prototype function definition is better known as an ANSI function definition.

When the nonprototype function declarations/definitions are mixed with prototype declarations, the type of each prototype parameter must be compatible with the type that results from the application of the default argument promotions.

Most types are already compatible with their default argument promotions. The only ones that are not are `char`, `short`, and `float`. Their promoted versions are, respectively, `int`, `int`, and `double`.

This message can occur in several situations. The most common is when mixing ANSI prototypes with K&R function definitions. If a function is defined using a K&R-style header, then its prototype, if present, must specify widened versions of the parameter types. Here is an example.

```
int function( short );
int function( x )
  short x;
{ }
```

This is not valid because the function has a K&R-style definition and the prototype does not specify the widened version of the parameter. To be correct, the prototype should be

```
int function( int );
```

because `int` is the widened version of `short`.

Another possible solution is to change the function definition to use ANSI syntax. This particular example would be changed to

```
int function( short );
int function( short x )
{ }
```

This second solution is preferable, but either solution is equally valid.

Recovery: Give a promoted type to the parameter in the prototype function declaration.

EDC0347 30 Syntax error: possible missing *&1* or *&2*.

Explanation: A syntax error has occurred. This message lists the tokens that the parser expected and did not find.

Recovery: Correct the syntax error and compile again.

EDC0348 30 Syntax error: possible missing *&1*.

Explanation: A syntax error has occurred. This message lists the tokens that the parser expected and did not find.

Recovery: Correct the syntax error and compile again.

EDC0349 30 Unexpected text *&1* ignored.

Explanation: A syntax error has occurred. This message lists the tokens that were discarded by the parser when it tried to recover from the syntax error.

Recovery: Correct the syntax error and compile again.

EDC0350 30 Syntax error.

Explanation: See the *C/VSE Language Reference* for a complete description of C syntax rules.

Recovery: Correct the syntax error and compile again.

EDC0356 30 A constant expression cannot contain a comma operator.

Recovery: Modify the constant expression to remove the comma operator.

EDC0370 30 Operand of offsetof macro must be a struct or a union.

Explanation: The first operand of the offsetof macro must be a structure or union type.

Recovery: Change the operand.

EDC0371 30 The dot operator cannot be applied to an incomplete struct or union.

Explanation: A structure or union is incomplete when the definition of its tag has not been specified. A struct or union tag is undefined when the list describing the name and type of its members has not been specified.

Recovery: Give a definition of the tag before the operator is applied to the structure.

EDC0372 30 The arrow operator cannot be applied to an incomplete struct or union.

Explanation: A structure or union is incomplete when the definition of its tag has not been specified. A struct or union tag is undefined when the list describing the name and type of its members has not been specified.

Recovery: Give a definition of the tag before the operator is applied to the structure.

EDC0396 10 The compiler directive *&1* is ignored.

Explanation: The compiler directive is ignored in certain situations. For example, the #line directive is ignored when the option EVENTS or TEST is in effect.

Recovery: Ensure that the compiler directives and the other options settings are consistent.

EDC0397 30 Macro argument list is not complete; either the arguments are not fully specified or a *&1* is missing.

Recovery: Complete the specification of the macro argument list.

EDC0398 10 The pragma *&1* directive for function *&2* is not valid.

Explanation: The pragma inline and noinline directives must be issued at file scope in order to take effect.

Recovery: Issue the pragma directive at file scope.

EDC0399 30 A character constant must contain at least one character.

Recovery: Put at least one character inside the pair of single quotation marks.

EDC0400 30 String literals must end before the source line unless the continuation symbol is used.

Explanation: String literals must end before the end of the source line. String literals can be constructed which are longer than one line by using the line continuation sequence (backslash (\) at the end of the line) or by using the concatenation of adjacent string literals.

Recovery: Either end the string with a quotation mark or use the continuation sequence.

EDC0401 30 The character is not valid.

Explanation: A character not in the C source character set has been encountered.

Recovery: Remove the character. Check the syntax.

EDC0403 10 The #line directive must specify a string literal or a new-line character.

Explanation: The integer value in the #line directive must be followed by a string literal or the end of the line.

Recovery: Correct the #line directive.

EDC0404 30 End of file was reached before end of comment that started on line *&1*.

Explanation: A comment that was not terminated has been detected. The beginning of the comment was on the specified line.

Recovery: End the comment before the file ends.

EDC0405 10 A new-line character is required.

Explanation: A character sequence was encountered when the preprocessor required a new-line character.

EDC0406 30 Preprocessing token # must be followed by a parameter.

Explanation: The # preprocessor operator may only be applied to a macro parameter.

Recovery: Place a parameter after the # token, or remove the token.

EDC0407 30 The #include directive is not valid.

Explanation: The #include file specifier is missing or not valid.

Recovery: Check the spelling and syntax of the #include file path.

EDC0408 30 A #if, #elif, #ifdef or #ifndef block must end with a #endif.

Recovery: End the conditional preprocessor statements with a #endif.

EDC0409 30 A macro name on *&1* directive is expected.

Recovery: Ensure that a macro name follows the #define, #undef, #ifdef, or #ifndef preprocessor directive.

EDC0410 30 A *&1* can only appear within a #if, #elif, #ifdef or #ifndef block.

Recovery: Delete the #elif or #else statement, or place it within a conditional preprocessor block. Check for misplaced braces.

EDC0412 30 A #endif must follow a #if, #elif, #ifdef or #ifndef block.

Recovery: Delete the #endif statement, or place it after a conditional preprocessor block.

EDC0413 30 #elif cannot follow #else.

Explanation: The #elif directive may not follow an #else directive within an #if, #elif, #ifdef, or #ifndef block.

Recovery: Remove the #elif or the #else.

EDC0414 30 End of file is not expected.

Explanation: The end of the source file has been encountered prematurely.

Recovery: Check for misplaced braces.

EDC0415 30 Text is too long.

Explanation: The specified token is too long to be processed. This condition arises when a numeric literal with many leading zeros or a floating point literal with many trailing digits in the fraction is coded.

Recovery: Create a shorter token.

EDC0416 30 The integer constant suffix is not valid.

Explanation: The integer constant has a suffix letter that is not recognized as a valid suffix.

EDC0417 30 Integer constant is out of range.

Explanation: The specified constant is too large to be represented by an unsigned long int.

Recovery: The constant integer must have a value less than 4294967296.

EDC0418 10 Escape character *&1* is not valid and is ignored.

Explanation: An escape sequence that is not valid has been encountered in a string literal or a character literal. It is replaced by the character following the backslash (\).

Recovery: Change or remove the escape sequence.

EDC0419 30 A character literal must end before the end of a line.

Explanation: Character literals must be terminated before the end of the source line.

Recovery: End the character literal before the end of the line. Check for misplaced quotation marks.

EDC0420 30 The ## operator cannot appear first or last in the macro replacement list.

Explanation: The ## operator must be preceded and followed by valid tokens in the macro replacement list.

EDC0421 30 The macro parameter list is incorrect.

Explanation: The macro parameter list must be empty, contain a single identifier, or contain a list of identifiers separated by commas.

Recovery: Correct the parameter list.

EDC0422 30 Parameter *&1* cannot be redefined in the macro parameter list.

Explanation: The identifiers in the macro parameter list must be distinct.

Recovery: Change the identifier name in the parameter list.

EDC0423 10 Macro name *&1* cannot be redefined.

Explanation: A macro may be defined multiple times only if the definitions are identical except for white space.

Recovery: Change the macro definition to be identical to the preceding one, or remove it.

EDC0424 30 The expression on the #if or #elif directive is not a valid constant expression.

Recovery: Replace the expression that controls #if or #elif by a constant integral expression.

EDC0425 30 Argument list must specify same number of arguments as required by macro definition.

Explanation: The number of arguments specified on a macro invocation is different from the number of arguments required for the macro.

Recovery: Make the number of arguments consistent with the macro definition.

EDC0426 10 The #error text is too long.

Explanation: The text specified for the #error directive is too long to be processed. The maximum length allowed for #error text is 4096 characters.

Recovery: Specify a shorter message.

EDC0427 30 #error *&1*

Explanation: This is the message issued by the #error directive.

Recovery: Because this is a user-created message, the recovery depends on the nature of the error.

EDC0428 30 A preprocessing directive must end before the end of a line.

Explanation: The end of line has been encountered while scanning a preprocessing directive.

EDC0429 30 String literal cannot exceed maximum length of 4096.

Explanation: A string constant of length greater than 4096 characters was encountered.

Recovery: Specify a shorter string literal.

EDC0430 10 The preprocessing directive *&1* is not valid.

Explanation: An unrecognized preprocessing directive has been encountered.

Recovery: Check the spelling and syntax or remove the directive that is not valid.

EDC0431 10 The end of a #include file was encountered before the end of the comment.

Recovery: End the comment before ending the #include file. Check for misplaced or missing punctuation.

EDC0432 10 The end of file was encountered immediately after a continuation line.

Recovery: Remove the continuation character from the last line of the file, or add code after the continuation character.

EDC0433 10 #line value too large.

Recovery: Ensure that the #line value does not exceed the maximum value (32767) for short integers.

EDC0434 10 *&1* value must contain only decimal digits.

Explanation: A non-numeric character was encountered in the *&1* value.

Recovery: Check the syntax of the value given.

EDC0435 10 A valid wide character must either be 0x4040 or have both bytes between 0x41 and 0xfe, inclusive.

Explanation: The value of a double byte character was out of range.

Recovery: Change the value accordingly.

EDC0436 10 Wide character string is not valid.

Explanation: The value given to the wide character string is not valid.

EDC0437 30 A character string literal cannot be concatenated with a wide string literal.

Explanation: A string that has a prefix L cannot be concatenated with a string that is not prefixed.

Recovery: Check the syntax of the value given.

EDC0438 10 An error was detected in #pragma *&1*.

Explanation: For a description of the syntax for #pragma directives, see the *C/VSE Language Reference*.

Recovery: Check the syntax of the #pragma directive. Change the graphics character # to the graphics character associated with code point '7B'X.

EDC0439 10 Option *&1* on #pragma *&2* is not supported.

Explanation: For a list of all valid options for #pragma directives, see the *C/VSE Language Reference*.

Recovery: Ensure the #pragma syntax and options are correct.

EDC0440 10 #pragma *&1* must appear before any C code.

Recovery: Place the #pragma directive before any C code in the file.

EDC0441 10 #pragma *&1* is unrecognized and is ignored.

Explanation: An unrecognized #pragma directive was encountered. See the *C/VSE Language Reference* for the list of valid #pragmas available.

Recovery: Change or remove the #pragma directive.

EDC0442 10 Option on #pragma *&1* is out of range.

Explanation: The specified #pragma option is not within the range of the valid values. See the *C/VSE Language Reference* for more information on the #pragma directives.

Recovery: Change the option or remove the #pragma directive.

EDC0443 10 The #pragma *&1* must appear before any C code or directive, except #pragma filetag.

Recovery: This #pragma must appear before all C code and directives with the exception of #pragma filetag, which may precede it.

EDC0444 10 The #pragma *&1* must appear only once and before any C code.

Recovery: Remove all but one of the specified #pragma directives and place the #pragma directive before any C code.

EDC0446 10 Only one FETCHABLE function may be specified.

Explanation: Only one #pragma linkage directive with option FETCHABLE may be specified in a compilation.

Recovery: Change the linkage specified on the #pragma linkage of the function so that only one function in the file has linkage FETCHABLE.

EDC0447 30 A wide character string or constant must have an even number of bytes.

Recovery: Ensure all wide character constants and character strings have an even number of bytes.

EDC0448 10 A duplicate #pragma *&1* is ignored.

Recovery: Remove the duplicate #pragma.

EDC0449 10 A new-line is not expected before the end of the preprocessing directive.

Explanation: A new-line was encountered before the preprocessor directive was complete.

Recovery: Ensure the preprocessor directive ends before the end of the line.

EDC0450 10 Option *&1* ignored because option *&2* specified.

Explanation: Specifying the second option indicated means the first has no effect. For example, the PPONLY option causes the OPTIMIZE option to be ignored, since no code will be generated.

Recovery: Remove one of the options.

EDC0451 10 An incomplete option has been specified.

Explanation: Refer to “Compile-Time Option Defaults” on page 18 for information on specifying compile-time options.

Recovery: Complete or remove the option.

EDC0452 10 Suboption *&2* of *&1* is not valid.

Explanation: An incorrect suboption of the specified compile-time option has been given. See "Compile-Time Option Defaults" on page 18 for more information on compile-time options.

Recovery: Change or remove the incorrect suboption.

EDC0453 10 *&1* suboptions must be separated by commas.

Recovery: Use commas to delimit your suboptions.

EDC0454 10 Expecting *&1* on *&2* option.

Recovery: Use the correct syntax for specifying the option.

EDC0455 10 Suboption *&2* of *&1* is out of range.

Explanation: A suboption of the specified compile-time option is not within the range of valid values. See "Compile-Time Option Defaults" on page 18 for more information on compile-time options.

Recovery: Change or remove the suboption.

EDC0456 10 Suboptions *&2* and *&3* of option *&1* conflict.

Explanation: Conflicting suboptions of the indicated compile-time option have been specified.

Recovery: Remove one of the conflicting suboptions.

EDC0457 10 Simple name *&1* must begin with an alphabetic character on *&2*.

Recovery: Change the name to start with an alphabetic character.

EDC0458 10 Simple name *&1* on option *&3* is too long.

Recovery: Shorten the name.

EDC0459 10 Data set name *&1* on option *&3* is too long.

Recovery: Use an appropriate data set name.

EDC0460 10 Macro name *&1* must not begin with a numeric character on *&2* option.

Explanation: Macro names must begin with an alphabetic character or an underscore.

Recovery: Change the macro name.

EDC0461 30 *&1* cannot be defined as a macro on the *&2* option.

Recovery: Remove the macro definition.

EDC0462 10 Macro definition on the *&1* option is not valid.

Recovery: Remove the macro definition or change the macro name.

EDC0463 10 Option *&1* is not valid.

Explanation: An incorrect compile-time option has been encountered. See "Compile-Time Option Defaults" on page 18 for valid compile-time options.

Recovery: Change or remove the option.

EDC0464 10 Character constant has more than four bytes.

Explanation: A character constant can only have up to four bytes.

Recovery: Change the character constant to contain four bytes or less.

EDC0465 40 Unable to open the default file for *&1* output.

Recovery: Ensure that you have enough storage for the file.

EDC0466 10 Characters in data set name *&1* on *&2* option are not valid.

Recovery: Change the data set name to a more appropriate name.

EDC0467 10 Data set name *&1* on *&2* option is not valid.

Recovery: Change the data set name to a more appropriate name.

EDC0468 10 Macro name *&1* on *&2* option is already defined.

Explanation: On the DEFINE option a macro may be defined multiple times only if the definitions are identical except for white space.

Recovery: Change the name of the macro.

EDC0469 10 Macro name *&1* has been truncated to *&2* on the *&3* option.

Explanation: The length of the macro name on the DEFINE option is greater than the maximum allowed. The name has been truncated.

Recovery: Change the macro name if necessary.

EDC0470 10 Macro name *&1* contains characters not valid on the *&2* option.

Explanation: Macro names can contain only alphanumeric characters and the underscore character.

Recovery: Change the macro name.

EDC0471 10 Unable to open *&1* data set *&2*.

Recovery: Check that the data set exists and that you have enough disk space.

EDC0474 30 *&1* must be specified in the csect #pragma.

Recovery: Use the appropriate format for the directive.

EDC0475 10 Option *&1* ignored because option *&2* is not specified.

Explanation: The second option must be specified for the first to have an effect.

Recovery: Specify the second option, or remove the first.

EDC0477 10 Missing parameter for option *&1*.

Explanation: The parameter on this option is required.

Recovery: Specify a valid parameter on this option.

EDC0478 30 *&1* is not supported in *&2*.

Explanation: The option or pragma has specified a feature which is not supported.

Recovery: Remove this option or pragma.

EDC0485 30 Cannot declare a pointer to a function with built-in linkage.

Recovery: Remove the #pragma linkage or built-in keyword from the declaration of the function.

EDC0486 30 Cannot explicitly or implicitly take the address of a function with built-in linkage.

Explanation: The address of a built-in function cannot be determined. The compiler does not allow for the declaration of a pointer to a built-in function.

Recovery: Remove the #pragma linkage or built-in keyword from the declaration of the function.

EDC0495 10 The name in option *&1* is not valid. The option is reset to *&2*.

Explanation: The name specified as a suboption of the option is syntactically or semantically incorrect and thus can not be used.

Recovery: Make sure that the suboption represents a valid name. For example, in option LOCALE(localename), the suboption 'localename' must be a valid locale name which exists and can be used. If not, the LOCALE option is reset to NOLOCALE.

EDC0501 40 Unable to open file for compiler intermediate code.

Explanation: Not able to open intermediate listing file.

EDC0502 40 Unable to open source image file.

Explanation: The source image file (IJSYS07) could not be opened. The member was not added to the file because of errors.

Recovery: See previous error messages. Check DLBL for IJSYS07, correct the errors and try the request again.

EDC0503 40 Unable to open listing file *&1*.

Explanation: The source listing file could not be opened.

EDC0504 30 Unable to find #include file *&1*.

Explanation: The file specified on the #include directive could not be found.

Recovery: Ensure the #include file name and the search path are correct.

EDC0505 10 Unable to determine the codepage of source file *&1*.

Explanation: Ensure that the file specified as input to the compiler has a valid codepage associated with it. Codepage 1047 is being used as the default.

EDC0506 30 Unable to find source file *&1*.

Explanation: Ensure that the name of the file specified as primary input to the compiler corresponds to an existing C source file.

EDC0550 30 Macro parameter list must end before the end of the line.

Explanation: The list of parameters for a macro on a #define directive did not end before the end of the line.

Recovery: End the parameter list before the end of the line. Check that all required continuation lines have been coded.

EDC0551 10 The #include file header cannot be empty.

Explanation: The #include file header specified is empty.

Recovery: Remove the #include directive or ensure that the header is not empty.

EDC0552 10 The #include header *&1* is not valid.

Explanation: The name of the file specified on the #include directive is not valid.

Recovery: Remove or correct the #include directive. Check the syntax.

EDC0553 10 Built-in function *&1* is unrecognized. The default linkage convention will be used.

Recovery: Check that you are using the correct function name.

EDC0560 30 The decimal size is outside the range of 1 to *&1*.

Explanation: The specified decimal size should be between 1 and DEC_DIG.

Recovery: Specify the decimal size between 1 and DEC_DIG.

EDC0561 30 The decimal precision is outside the range of 0 to *&1*.

Explanation: The specified decimal precision should be between 0 and DEC_PRECISION.

Recovery: Specify the decimal precision between 0 and DEC_PRECISION.

EDC0562 30 The decimal size is not valid.

Explanation: The decimal size must be a positive constant integral expression.

Recovery: Specify the decimal size as a positive constant integral expression.

EDC0563 30 The decimal precision is not valid.

Explanation: The decimal precision must be a constant integral expression.

Recovery: Specify the decimal precision as a constant integral expression.

EDC0564 30 The decimal precision is bigger than the decimal size.

Explanation: The specified decimal precision should be less than or equal to the decimal size.

Recovery: Specify the decimal precision less than or equal to the decimal size.

EDC0565 30 The decimal constant is out of range.

Explanation: The compiler detected a decimal overflow in scanning a decimal constant.

Recovery: Change the decimal constant so that it does not exceed the maximum value.

EDC0566 30 The *&1* type specifier cannot be used with decimal types.

Explanation: No other type specifier (such as long, short, unsigned, signed) can be used with type decimal.

Recovery: Remove the type specifier.

EDC0567 10 The fraction part of the result was truncated.

Explanation: Due to limitations on the number of digits representable, the calculated intermediate result may result in truncation in the decimal places after the operation is performed.

Recovery: Check to make sure that no significant digit is lost.

EDC0570 10 The pre- and post- increment and decrement operators cannot be applied to type *&1*.

Explanation: The decimal types with no integral part cannot be incremented or decremented.

Recovery: Reserve at least one digit in the integral part of the decimal types.

EDC0571 30 Only decimal types can be used with the *&1* operator.

Explanation: The operand of the digitsof or precisionof operator is not valid. The digitsof and precisionof operators can only be applied to decimal types.

Recovery: Change the operand.

EDC0573 10 Whole-number-part digits in the result may have been lost.

Explanation: Due to limitations on the number of digits representable, the calculated intermediate result may result in loss of digits in the integer portion after the operation is performed.

Recovery: Check to make sure that no significant digit is lost.

EDC0574 30 The function *&1* can only have one decimal type argument.

Explanation: The function call is not valid. The functions decabs, decchk and decfix have one decimal type as their argument.

Recovery: Only pass one decimal type as argument when these functions are called.

EDC0575 30 Digits have been lost in the whole-number part.

Explanation: In performing the operation, some non-zero digits in the whole-number part of the result are lost.

EDC0576 10 Digits may have been lost in the whole-number part.

Explanation: In performing the operation, some digits in the whole-number part of the result may have been lost.

Recovery: Check to make sure that no significant digit is lost.

EDC0670 10 The string *&1* was found where a delimiter was expected following a quoted suboption for the run-time option *&2*.

Explanation: A quoted suboption must be followed by either a comma, right parenthesis, or space.

Recovery: Correct the run-time options string.

EDC0671 10 An end quote delimiter did not occur before the end of the run-time option string.

Explanation: Quotes, either single or double, must be in pairs.

Recovery: Correct the run-time options string.

EDC0672 10 The character '*&1*' is not a valid run-time option delimiter.

Explanation: Options must be separated by either a space or a comma.

Recovery: Correct the run-time options string.

EDC0673 10 The character '*&1*' is not a valid suboption delimiter for run-time options.

Explanation: Suboptions must be delimited by a comma.

Recovery: Correct the run-time options string.

EDC0674 10 The string '*&1*' was found where a delimiter was expected following the suboptions for the run-time option *&2*.

Explanation: Suboptions which are enclosed within parenthesis must be followed by either a space or a comma.

Recovery: Correct the run-time options string.

EDC0675 10 The string *&1* was too long and was ignored.

Explanation: The maximum string length for an option or suboption was exceeded.

Recovery: Correct the run-time options string.

EDC0676 10 The end of the suboption string did not contain a right parenthesis.

Explanation: A left parenthesis did not have a matching right parenthesis.

Recovery: Correct the run-time options string.

EDC0677 10 The run-time option *&1* is not supported.

Explanation: *&1* is an option from a previous release that is not supported or mapped by LE.

Recovery: Correct the run-time options string.

EDC0678 10 The run-time option *&1* was mapped to the run-time option *&2&3*.

Explanation: *&1* is an option from a previous release that is supported by LE for compatibility.

Recovery: Change the run-time options string to use the *&2* and *&3* instead.

EDC0679 10 The run-time option *&1* was an invalid run-time option.

Explanation: *&1* is not an LE option or an option that is recognized by LE for previous release language compatibility.

Recovery: Correct the run-time options string.

EDC0680 10 Too many suboptions were specified for the run-time option *&1*.

Explanation: The number of suboptions specified for *&1* exceeded that defined for the option.

Recovery: Correct the run-time options string.

EDC0681 10 The run-time option *&1* appeared in the options string.

Explanation: *&1* is an option from a previous release that is supported by LE for compatibility, but ignored if TRAP is specified.

Recovery: Change the run-time options string to use the TRAP option instead.

EDC0682 10 An invalid character occurred in the numeric string '*&1*' of the run-time option *&2*.

Explanation: *&1* did not contain all decimal numeric characters.

Recovery: Correct the run-time options string.

EDC0683 10 The installation default for the run-time option *&1* could not be overridden.

Explanation: *&1* has been defined as non-overrideable at installation time.

Recovery: Correct the run-time options string.

EDC0684 10 The string *&1* was not a valid suboption of the run-time option *&2*.

Explanation: *&1* is not in the set of recognized values.

Recovery: Correct the run-time options string.

EDC0685 10 The number *&1* of the run-time option *&2* exceeded the range of -2147483648 to 2147483647.

Explanation: *&1* exceeded the range of -2147483648 to 2147483647.

Recovery: Correct the run-time options string.

EDC0686 10 The value *&1* was not a valid MSGQ number.

Explanation: *&1* must be greater than zero.

Recovery: Correct the run-time options string.

EDC0687 10 The STORAGE option quoted suboption string *&1* was not one character long.

Explanation: The only acceptable length for STORAGE suboptions within quotes is one.

Recovery: Correct the run-time options string.

EDC0688 10 The UPSI option suboption string *&1* was not eight characters long.

Explanation: The only acceptable length for the UPSI suboption is eight.

Recovery: Correct the run-time options string.

EDC0689 10 The run-time option *&1* was partially mapped to the run-time option *&2*.

Explanation: *&1* is an old language option that is being supported by LE for compatibility. The user should use *&2* instead.

Recovery: Change the run-time options string to use the *&2* instead.

EDC0690 10 A punctuation error was detected in #pragma runopts.

Recovery: Use the appropriate syntax for the #pragma runopts directive.

EDC0695 10 Unable to load Language Environment run-time options processing services.

Recovery: Ensure that you have access to all the LE/VSE library files.

EDC0697 10 One or more settings of the run-time options STAE or SPIE were ignored.

Explanation: STAE, SPIE, NOSTAE, and NOSPIE are options from a previous release that are ignored when the TRAP options is specified.

Recovery: Change the run-time options string to use the TRAP option instead.

EDC0698 10 One or more settings of the run-time options STAE or SPIE were mapped to TRAP.

Explanation: STAE, SPIE, NOSTAE, and NOSPIE are options from a previous release that are supported by LE for compatibility.

Recovery: Change the run-time options string to use the TRAP option instead.

EDC0750 10 #pragma *&1* is ignored because the LOCALE compile-time option is not specified.

Explanation: The LOCALE compile-time option is required for #pragma *&1*

Recovery: Remove all the #pragma *&1* directives or specify the LOCALE compile-time option.

EDC0751 10 The #pragma filetag directive cannot be empty.

Explanation: The #pragma filetag directive is empty.

Recovery: Remove the #pragma filetag directive or ensure that the pragma is not empty.

EDC0752 10 #pragma filetag is ignored because the conversion table from *&1* to *&2* cannot be opened.

Explanation: During compilation, source code is converted from the code set specified by #pragma filetag to the code set specified by the LOCALE compile-time option, if they are different. A conversion table form *&1* to *&2* must be loaded prior to the conversion. No conversion is done when the conversion table is not found.

Recovery: Create the conversion table from *&1* to *&2* and ensure it is accessible from the compile-time. If message files are used in the application to read and write data, a conversion table from *&2* to *&1* must also be created to convert data from run-time locale to the compile-time locale.

EDC0753 10 Error messages are not converted because the conversion table from *&1* to *&2* cannot be opened.

Explanation: Error messages issued by C/VSE are written in code page 1047. These messages must be converted to the code set specified by the locale compile-time option because they may contain variant characters, such as #. Before doing the conversion, a conversion table from *&1* to *&2* must be loaded. The error messages are not converted because the conversion table cannot be found.

Recovery: Make sure the conversion table from *&1* to *&2* is accessible from the compiler.

EDC0754 10 No conversion on character *&1* because it does not belong to the input code set *&2*.

Explanation: No conversion has be done for the character because it does not belong to the input code set.

Recovery: Remove or change the character to the appropriate character in the input code set.

EDC0755 10 Incomplete character or shift sequence was encountered during the conversion of the source line.

Explanation: Conversion stops because an incomplete character or shift sequence was encountered at the end of the source line.

Recovery: Remove or complete the incomplete character or shift sequence at the end of the source line.

EDC0756 10 Only conversion tables that map single byte characters to single byte characters are supported.

Explanation: The compiler expected a single byte to single byte character mapping during conversion. Conversion stops when there is insufficient space in the conversion buffer.

Recovery: Ensure that the conversion table is in single byte to single byte mapping.

EDC0757 10 Invalid conversion descriptor was encountered during the conversion of the source line.

Explanation: No conversion was preformed because conversion descriptor is not valid.

EDC0758 10 #pragma *&1* must appear on the first directive before any C code.

Recovery: Put this #pragma as the first directive before any C code.

EDC0759 10 This character will not be supported in the future release.

Explanation: C/VSE has migrated to support code page 1047 only if the NOLOCALE compile-time option is specified. The support for this character will be dropped in the future release because it is not a valid 1047 code point.

Recovery: Rewrite this character in code page 1047 or use LOCALE compile-time option.

EDC0800 10 Parameter *&1* is not referenced.

Explanation: The identified variable has been declared in a function parameter list, but never referenced within the function body.

Recovery: Remove the parameter declaration if it is not needed.

EDC0801 10 Automatic variable *&1* is not referenced.

Explanation: The identified variable has been declared at block scope, but never referenced.

Recovery: Remove the variable declaration if it is not needed.

EDC0802 10 Static variable *&1* is not referenced.

Explanation: The identified static variable has been declared, but never referenced.

Recovery: Remove the variable declaration if it is not needed.

EDC0803 10 External variable *&1* is not referenced.

Explanation: The identified variable has been declared either at file scope or extern at block scope, and was never referenced.

Recovery: Remove the variable declaration if it is not needed.

EDC0804 10 Function *&1* is not referenced.

Explanation: The identified function has been declared, but never referenced.

Recovery: Remove the function declaration if the function is not needed.

EDC0805 10 Automatic variable *&1* is set but not referenced.

Explanation: The identified variable has been declared and initialized, but never referenced. Variables of type array, struct, or union are not checked for this condition.

Recovery: Remove the variable declaration and initialization if they are not needed.

EDC0806 10 Static variable *&1* is set but not referenced.

Explanation: The identified variable has been declared and initialized, but never referenced. Variables of type array, struct, or union are not checked for this condition.

Recovery: Remove the variable declaration and initialization if they are not needed.

EDC0807 10 Variable *&1* may not have been set before it is referenced.

Explanation: The compiler encountered an attempt to access the value of the identified variable before the variable was explicitly initialized.

Recovery: Ensure the variable is explicitly initialized before its value is accessed.

EDC0808 10 Variable *&1* was not explicitly initialized.

Explanation: If not explicitly initialized, variables with storage class auto or register contain indeterminate values.

Recovery: Initialize the variable.

EDC0809 10 *&1* redefinition hides earlier one.

Explanation: A typedef was defined at an inner scope with the same name as a previous typedef definition made at an outer scope. The inner scope definition overrides the previous one.

Recovery: Ensure this is what was intended or use different names for the two typedefs.

EDC0810 10 External variable *&1* is set but not referenced.

Explanation: The identified variable has been declared and initialized, but never referenced. Variables of type array, struct, or union are not checked for this condition.

Recovery: Remove the variable declaration and initialization if they are not needed.

EDC0811 10 Statement has no effect.

Explanation: The statement does not cause any storage to be changed or functions to be called.

Recovery: Change or delete the statement.

EDC0812 10 Expression has no effect.

Explanation: An expression with no effect has been discovered where expressions with side effects are usually expected.

Recovery: Change or delete the expression.

EDC0813 10 if-statement is empty.

Explanation: The statement body for an if statement contains no executable code.

Recovery: Change the statement body to contain executable code or delete the if statement.

EDC0814 10 else-statement is empty.

Explanation: The statement body for an else statement contains no executable code.

Recovery: Change the statement body to contain executable code or delete the else statement.

EDC0815 10 Loop body is empty.

Explanation: The statement body for a loop statement contains no executable code.

Recovery: Change the statement body to contain executable code or remove the loop statement.

EDC0816 10 Assignment found in a control expression.

Explanation: The control expression for a switch, if, for, or while statement contains an unparenthesized assignment statement. A common programming problem is the substitution of an assignment statement (i = 3) for what should be a comparison statement (i == 3).

Recovery: Verify whether the statement should be an assignment or a comparison.

EDC0817 10 Type conversion may result in lost precision.

Explanation: The required type conversion may cause lost precision. See the *C/VSE Language Reference* for more information on type conversions.

Recovery: If precision is important in the operation, eliminate the type conversion.

EDC0818 10 Pointer type conversion found.

Explanation: Conversion of pointer types may change the pointer values.

Recovery: None, if the conversion was intended. Otherwise, declare the pointer to void instead of to another type, and then cast it.

EDC0819 10 Bitwise operator applied to a signed type.

Explanation: Bitwise operators may change the value of a signed type by shifting the bit used to indicate the sign of the value.

Recovery: Change the operand to an unsigned type or remove the bitwise operation.

EDC0820 10 Right shift operator applied to a signed type.

Explanation: A right shift operator may change the value of a signed type by shifting the bit used to indicate the sign of the value.

Recovery: Change the operand to an unsigned type or remove the shift operation.

EDC0821 10 Relational expression is always true.

Explanation: The control expression of a switch, if, for, or while statement has a constant value, and the result is always true. This may not be effective code.

Recovery: Verify if this result was intended. Change the control expression if necessary.

EDC0822 10 Relational expression is always false.

Explanation: The control expression of a switch, if, for, or while statement has a constant value, and the result is always false. This may not be effective code.

Recovery: Verify if this result was intended. Change the control expression if necessary.

EDC0823 10 Expression contains division by zero.

Explanation: An expression containing division by zero was found.

Recovery: Eliminate the division by zero if it was not intended.

EDC0824 10 Expression contains modulus by zero.

Explanation: An expression containing modulus by zero was found.

Recovery: Eliminate the modulus by zero if it was not intended.

EDC0825 10 Code cannot be reached.

Explanation: A statement without a label has been found after an unconditional transfer of control, such as a goto.

Recovery: If the statement should be executed, make the transfer of control conditional, or label the statement. If not, remove the statement.

EDC0826 10 Execution fall-through within a switch statement.

Explanation: A case label has been encountered that was not preceded by either a break or return statement.

Recovery: Precede the case label with a break or return statement.

EDC0827 10 Nonprototype function declaration encountered.

Explanation: A nonprototype function declaration was found. For example,

```
int addnum();
```

Function declarations should include the return type of the function and the types of its parameters. Calls to nonprototype functions get no type checking or type conversions on parameters.

Recovery: Change the nonprototype declarations to prototype declarations such as the following:

```
int addnum(int, int);
```

EDC0828 10 The return type of the function main should have type int, not void.

Explanation: If main is declared to return void, the exit code from the program will be indeterminate.

EDC0829 10 Possibly ambiguous operator usage encountered.

Explanation: Expressions consisting of traditional mathematical symbols sometimes have bugs created by misunderstanding of operator precedence. Nonparenthesized expressions containing shift operators, relationals, and bitwise operators may have precedence that is counterintuitive. The identified operator has at least one operand that may have this property.

Recovery: Use the appropriate parentheses to eliminate the ambiguity.

EDC0830 10 Value is not a member of the enumeration.

Explanation: Variables of type enum are not expected to be used in situations other than assignment and comparison, and can only be assigned proper members of their enumeration, either directly, from function return values, or from another variable of the same type.

Recovery: Ensure operations involving variables of type enum are valid.

EDC0831 10 Case label is not a member of the enumeration.

Explanation: In a switch statement where the switch control expression is an enum, the case label values must be members of the enumeration.

Recovery: Ensure the case label is a member of the enumeration.

EDC0832 10 Unstructured goto statement encountered.

Explanation: The target label of a goto statement should not be located in an inner block such as a loop.

Recovery: Ensure the target label of the goto statement is not located in an inner block.

EDC0833 10 Implicit return statement encountered.

Explanation: C allows returns from a function call without specifying a return statement. However, if a function is to return a value, a return statement must be included.

Recovery: Add a return statement to the called function if you want it to return a value.

EDC0834 10 Missing function return value.

Explanation: The function was declared to return a value, and a return statement with no value has been encountered. If return statement is not included in the function, it will return an indeterminate value to the caller.

Recovery: Add a return value to the return statement.

EDC0835 10 Structure or union remapping will be performed for this copy operation.

Explanation: A struct or union assignment has been encountered which requires an implicit pack or unpack operation. This form of assignment is often less efficient than assignments that have identical pack characteristics.

Recovery: Revise the statements to avoid unnecessary pack and unpack operations.

EDC0836 10 The same #pragma *&1* directive was previously specified for the object *&2*.

Explanation: The function was already declared using the same #pragma directive.

Recovery: Remove one of the #pragma linkage directives.

EDC0837 10 goto statement encountered.

Explanation: A goto statement was found.

Recovery: No recovery necessary.

EDC0838 10 Comparison is not valid because the numeric constant is out of range.

Explanation: A comparison between a variable and a constant that is not in the variable's range of possible values has been detected.

Recovery: Delete the comparison, or use a constant that is in the variable's range of possible values.

EDC0839 10 Unary minus applied to an unsigned type.

Explanation: An unsigned type cannot have a sign.

Recovery: Remove the unary minus or change the type to be signed.

EDC0841 10 File *&1* has already been included.

Explanation: The file specified was included by a previous #include directive.

Recovery: Remove one of the #include directives.

EDC0842 10 Macro name *&1* on #undef not defined.

Explanation: The specified macro name has never been defined or has already been removed by a previous #undef directive.

Recovery: Define the macro name, or remove the #undef directive.

EDC0843 10 Macro name *&1* on #define is also an identifier.

Explanation: The specified macro definition will override an existing identifier definition.

Recovery: Rename or remove the macro or the identifier.

EDC0844 10 Macro name *&1* on #define is also a keyword.

Explanation: The specified macro definition will override an existing keyword definition.

Recovery: Rename the macro or remove the definition.

EDC0845 10 Identifier *&1* assigned default value of 0.

Explanation: The indicated identifier in an #if or #elif expression was assigned the default value of zero. The identifier may have been intended to be expanded as a macro.

Recovery: Assign the identifier a value if necessary.

EDC0846 10 Expanding trigraph *&1* in string literal.

Explanation: A trigraph has been expanded in a string literal. This may not be the intended behavior.

Recovery: Ensure this is the intended behavior. If not, use escape sequences to represent characters, for example '\?' for the character '?'.

EDC0847 10 Expanding trigraph *&1* in character literal.

Explanation: A trigraph has been expanded in a character literal. This may not be the intended behavior.

Recovery: Ensure this is the intended behavior. If not, use escape sequences to represent characters, for example '\?' for the character '?'.

EDC0848 10 Some program text not scanned due to *&1* option.

Explanation: The setting of the margins and/or sequence options has resulted in some program text not being scanned.

Recovery: Reset the margins and/or sequence options if necessary.

EDC0849 10 #include header *&1* transformed into *&2*.

Explanation: The message indicates the transformation performed on the #include file before the search for the file begins.

Recovery: No recovery necessary if the result is what was intended.

EDC0850 10 #include searching for file *&1*.

Explanation: The message indicates the transformation performed on the #include file before the search for the file begins.

Recovery: No recovery necessary if the result is what was intended.

EDC0851 10 #include found file *&1*.

Explanation: The message indicates the actual file found for the #include directive.

Recovery: No recovery necessary if the result is what was intended.

EDC0852 10 #undef undefining macro name *&1*.

Explanation: This message traces the execution of the #undef directive.

Recovery: No recovery necessary if the result is what was intended.

EDC0853 10 Macro name *&1* on #define has a previous identical definition.

Explanation: The macro has already been identically defined. This may indicate that a file has been #included more than once.

Recovery: Remove one of the definitions or rename one of the macros.

EDC0854 10 #line directive changing line to *&1* and file to *&2*.

Explanation: This message traces the execution of the #line directive.

Recovery: No recovery necessary if the result is what was intended.

EDC0855 10 *&1* condition evaluates to *&2*.

Explanation: This message traces the evaluation of the test condition of an #if, #ifdef, or #elif directive.

Recovery: No recovery necessary if the result is what was intended.

EDC0856 10 defined(*&1*) evaluates to *&2*.

Explanation: This message traces the evaluation of the defined(*&1*) construct on an #if or #elif expression.

Recovery: No recovery necessary if the result is what was intended.

EDC0857 10 Begin skipping tokens.

Explanation: This message traces the execution of conditional compilation directives, for example indicating that code is skipped after an #if with a condition that evaluates to false.

Recovery: Ensure the appropriate tokens were skipped.

EDC0858 10 Stop skipping tokens.

Explanation: This message traces the execution of conditional compilation directives, for example, indicating that an #endif marked the end of a block of skipped code.

Recovery: Ensure the appropriate tokens were skipped.

EDC0859 10 *&1* nesting level is *&2*.

Explanation: This message traces the nesting level of conditional compilation directives.

Recovery: No recovery necessary if the result is what was intended.

EDC0860 10 String literals concatenated.

Explanation: This message traces the concatenation of two string literals.

Recovery: Ensure the concatenation is what was intended.

EDC0861 10 Optional brace encountered.

Explanation: A optional brace was found.

Recovery: No recovery necessary.

EDC0862 10 Matching optional brace encountered.

Explanation: A matching optional brace was found.

Recovery: No recovery necessary.

EDC0863 10 Incompletely bracketed initializer encountered, *&1* left brace(s) assumed.

Explanation: An initializer for an aggregate type was missing a left brace or braces. The compiler assumes the brace is meant to be there.

Recovery: Ensure this is what was intended.

EDC0864 10 Incompletely bracketed initializer encountered, *&1* right brace(s) assumed.

Explanation: An initializer for an aggregate type was missing a right brace or braces. The compiler assumes the brace is meant to be there.

Recovery: Ensure this is what was intended.

EDC0865 10 Floating-point constant is out of range.

Explanation: Refer to the float.h header file for the valid range for floating-point constants.

Recovery: Ensure the floating-point constant is within the valid range.

EDC0868 10 The incomplete struct or union tag *&1* was introduced in a parameter list.

Explanation: The incomplete struct or union tag introduced in the parameter list will not be compatible with subsequent uses of the tag.

Recovery: Declare the incomplete struct or union tag at file scope before the function declaration.

EDC0869 10 The incomplete struct or union tag *&1* was not completed before going out of scope.

Explanation: An incomplete struct or union tag introduced at block scope was not completed before the end of the scope.

Recovery: Provide a complete declaration for the struct or union tag.

EDC0870 10 #line directive changing line to *&1*.

Explanation: This message traces the execution of the #line directive.

EDC0900 40 Unable to open *&1*.

Recovery: Ensure file exists.

EDC0901 40 Unable to read *&1*.

Explanation: The compiler encountered an error while reading from the specified file.

EDC0902 40 Unable to write to *&1*.

Recovery: Ensure that the disk drive is not in an error mode and that there is enough disk space left.

EDC0903 30 Read/write pointer initialization of read-only object *&1* is not valid.
EDC1300 30 Maximum spill size of *&2* is exceeded in function *&1*..

Explanation: The maximum allowable spill area size has been exceeded.

Recovery: Reduce the complexity of the named function by breaking it up into smaller functions.

EDC1301 30 Spill size for function *&1* is not sufficient. Recompile specifying option SPILL(n) where *&2* < n <= *&3*.

Explanation: The specified spill area size has been exceeded.

Recovery: Recompile using the SPILL(n) option *&2* < n <= *&3* or with a different OPT level.

Note:

The following error messages may be produced by the compiler if the message file is itself invalid.

SEVERE ERROR EDC0090: Unable to open message file *&1*.
SEVERE ERROR EDC0091: Invalid offset table in message file *&1*.
SEVERE ERROR EDC0092: Message component *&1* not found.
SEVERE ERROR EDC0093: Message file *&1* corrupted.
SEVERE ERROR EDC0094: Integrity check failure on msg *&1*
SEVERE ERROR EDC0095: Bad substitution number in message *&1*
SEVERE ERROR EDC0096: Virtual storage exceeded
ERROR: Failed to open message file. Reason *&1*.
ERROR: Unable to read message file. Reason *&1*.
ERROR: Invalid offset table in message file *&1*.
ERROR: Message component *&1* not found.
ERROR: Message file *&1* corrupted.
ERROR: Integrity check failure on msg *&1* — retrieved *&2*.
ERROR: Message retrieval disabled. Cannot retrieve *&1*.
INTERNAL ERROR: Bad substitution number in message *&1*.

Appendix B. Other Return Codes and Messages

See the *LE/VSE Debugging Guide and Run-Time Messages* for messages and return codes for the following:

- LE/VSE prelinker
- LE/VSE run-time library
- `localedef` utility
- `genxlt` utility
- `iconv` utility
- `DSECT` utility

perror() Messages

When a call to the LE/VSE run-time library function `perror()` or `strerror()` is made, a message is printed or retrieved respectively, which corresponds to the current value of `errno`. For additional information about these messages, refer to the *LE/VSE Debugging Guide and Run-Time Messages* for the EDC*nnnn* message, where *nnnn* is the `errno` value plus 5000. For example, to find the message description for an `errno` value of 57, look up the message EDC5057.

To determine the `errno` value for a given message text, locate the message in the range EDC5000 to EDC5224 in the *LE/VSE Debugging Guide and Run-Time Messages*. When found, subtract 5000 from the message number to calculate the `errno` value. For example, to determine the `errno` value related to the message "An invalid argument was passed," scan the messages from EDC5000 to EDC5224 until the message text is found (EDC5030). The `errno` value is determined by subtracting 5000 from the message number—in this case 30.

Note: The run-time messages and the `errno` values are subject to change and are not general-use programming interface information. It is not good programming practice to write portable code that relies on these messages or `errno` values.

Appendix C. Files Used during Compile, Prelink, Link-Edit, and Execution

This appendix describes the files used in a compile/prelink/link-edit/run job.

Cross-Reference of Files Used

The following table provides a cross-reference of the files required for each job step and a description of how the files are used.

Table 13. Cross Reference of File and Job Step

File	Compile	Prelink	Link-Edit	Run
Phase Sublibrary	X	X	X	X
SYSIPT	X	X	X	X
SYSLST	X	X	X	X
Source Sublibrary	X			
Object Sublibrary		X	X	
SYSPCH	X	X		
SYSLNK	X	X	X	
SYSLOG		X		
IJSYS01	X		X	
IJSYS02-IJSYS07	X			

Description of Files Used

The following table lists the files used during compile, prelink, link-edit, and execution. A description of what the file is used for is also given.

Table 14 (Page 1 of 2). File Descriptions for Compilation, Prelink, Link-Edit, and Execution

File	In Job Step	Description
Phase Sublibrary	Compile	VSE Librarian sublibrary for C/VSE modules and messages
SYSIPT	Compile	Input file containing the C source program
SYSLST	Compile	Output file for compiler listing
Source Sublibrary	Compile	VSE Librarian sublibrary for C standard header files
SYSPCH	Compile	File for object module or PPONLY output
SYSLNK	Compile	File for object module
IJSYS01-IJSYS07	Compile	Workfiles
Phase Sublibrary	Prelink	VSE Librarian sublibrary containing prelinker modules and messages
SYSIPT	Prelink	File containing object module for the prelinker
Object Sublibrary	Prelink	VSE Librarian sublibrary for prelinkage autocall library

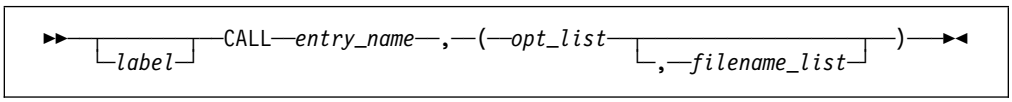
Table 14 (Page 2 of 2). File Descriptions for Compilation, Prelink, Link-Edit, and Execution

File	In Job Step	Description
SYSPCH	Prelink	File for output of the prelinker
SYSLNK	Prelink	File for output of the prelinker
SYSLST	Prelink	File for listing of prelink listing
SYSLOG	Prelink	File for listing of prelinker diagnostic messages
Object Sublibrary	Link-Edit	VSE Librarian sublibrary for autocall library
SYSLNK	Link-Edit	Primary input file for linkage editor
SYSIPT	Link-Edit	Primary input file for linkage editor (the linkage editor switches between SYSLNK and SYSIPT)
Phase Sublibrary	Link-Edit	VSE Librarian sublibrary to contain output phase
SYSLST	Link-Edit	File for listings and diagnostics produced by the linkage editor
IJSYS01	Link-Edit	Workfile
Phase Sublibrary	Run	VSE Librarian sublibrary containing LE/VSE run-time library functions
SYSLST	Run	File for listings and diagnostics from user program

Appendix D. Invoking C/VSE from Assembler

To compile your C source program dynamically, you can use the VSE macro instruction CALL in an assembler language program. For complete information on this macro instruction, refer to “Related Publications” on page 115.

The syntax of the CALL macro instruction follows:



where:

entry_name Specifies the name of the C/VSE main phase, EDCCOMP.

opt_list Specifies the address of a list containing the options to be specified for the compilation.

The option list must begin on a halfword boundary and the first 2 bytes must contain a count of the number of bytes in the remainder of the list; you specify the options in the same manner as you would on the PARM parameter of the EXEC JCL statement. If you do not want to specify any options, the count must be zero.

filename_list Specifies the address of a list containing alternative filenames (DLBL-names) for the files used during the compiler processing. If standard filenames are to be used, you can omit this parameter.

The filename list must begin on a halfword boundary and the first two bytes must contain a count of the number of bytes in the remainder of the list. Each name should be left-justified and padded with blanks to a length of 8 bytes.

The sequence of filenames in the list is:

- SYSIPT
- SYSLNK
- (3rd entry is not used)
- (4th entry is not used)
- (5th entry is not used)
- SYSPRINT
- SYSLST
- SYSPCH
- IJSYS01
- IJSYS02
- IJSYS03
- IJSYS04
- IJSYS05
- IJSYS06
- IJSYS07
- SYSPCH

An alternative filename can be omitted from the list by entering binary zeros in its 8-byte entry, or if it is at the end of the list, by shortening the list. If a filename is omitted, the standard filename will be assumed.

The return code from the compiler will be returned in register 15.

If the macro instruction is coded incorrectly, the compiler will not be invoked and the return code will be 32. A possible cause of that error is that the count of bytes in the alternative filenames list is not a multiple of 8 or is not between 0 to 128.

If an alternative filename for SYSLST is specified, the stdout stream will be redirected to the alternate filename.

The following example shows the use of the CALL assembler macro to do partial file renaming. The example is followed by the JCL used to invoke it.

EDCXUAAH

```
*********************************************************************
*                                                                   *
*  EDCXUAAH                                                         *
*                                                                   *
*  This assembler routine demonstrates file renaming (dynamic       *
*  compilation) using the assembler CALL macro.                     *
*                                                                   *
*  In this specific scenario a subset of all the files are renamed. *
*  This renaming is accomplished by shortening the list of          *
*  filenames.                                                       *
*                                                                   *
*  The compiler and run-time sublibraries must either be in the SVA *
*  or be specified on the LIBDEF SEARCH statement in your JCL.      *
*                                                                   *
*********************************************************************
*
CALL     CSECT
         STM   14,12,12(13)
         USING CALL,15
         LA    3,MODE31
         O     3,=X'80000000'
         DC    X'0B03'
MODE31   DS    0H
         USING *,3
         LR    12,15
         ST    13,SAVE+4
         LA    15,SAVE
         ST    15,8(,13)
         LR    13,15
*
*   Invoke the compiler using CALL macro
*
         CDLOAD EDCCOMP
         LR    15,1
         CALL  (15),(OPTIONS,FILENMES)
         L     13,4(,13)
         LM    14,12,12(13)
         SR    15,15
         BR    14
```

Figure 11 (Part 1 of 2). Using the Assembler CALL Macro

```
*
*   Constant and save area
*
SAVE     DC    18F'0'
OPTIONS  DC    H'21',C'SOURCE,LIST,TERM,TEST'
FILENMES DC    H'96'
         DC    CL8'SRCFILE'
         DC    XL8'00'
         DC    XL8'00'
         DC    XL8'00'
         DC    XL8'00'
         DC    XL8'00'
         DC    CL8'LISFILE'
         DC    XL8'00'
         DC    XL8'00'
         DC    XL8'00'
         DC    XL8'00'
         DC    XL8'00'
         END
```

Figure 11 (Part 2 of 2). Using the Assembler CALL Macro

EDCXUAAI

```
// JOB EDCXUAAI
/* ------------------------------------------------------------------
/*
/*  EDCXUAAI: Standard file renaming using the assembler CALL macro.
/*
/*  This job:
/*    - Compiles using primary input from DLBL SRCFILE
/*    - Copies the compiler listing from the DLBL LISFILE to SYSLST
/*
/*  Header files come from the sublibrary MYHDR.LIB.
/*
/*  Compilation is controlled by the assembler phase named CALLDD which
/*  is stored as MYTST.LIB(CALLDD.PHASE).
/*
/*  CALLDD was created by assembling the example EDCXUAAH.
/*
/* ------------------------------------------------------------------
// LIBDEF *,SEARCH=(MYHDR.LIB,MYTST.LIB,PRD2.DBASE,PRD2.SCEBASE)
// DLBL SRCFILE,'file-id',0,SD
// EXTENT unit,volser,,,start,end
ASSGN unit,DISK,VOL=volser,SHR
// DLBL LISFILE,'file-id',0,SD
// EXTENT unit,volser,1,,start,end
ASSGN unit,DISK,VOL=volser,SHR
// EXEC  PGM=CALLDD
/*
// UPSI 1
// EXEC DITTO
$$DITTO  SDP FILEIN=LISFILE
/*
/&
```

Figure 12. JCL for the Assembler CALL Macro

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 *IBM Dictionary of Computing*, SC20-1699.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

absolute value. The magnitude of a real number regardless of its algebraic sign.

abstract code unit (ACU). A measurement used by the C/VSE compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

ACU. Abstract code unit.

address. A name, label, or number identifying a location in storage, a device in a system or network, or any other data source.

aggregate. An array or a structure. Also, a compile-time option to show the layout of a structure or union in the listing.

alias. An alternate label used to refer to the same data element or point in a computer program.

alignment. See *boundary alignment*.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

Note: IBM has defined an extension to ASCII code (characters 128-255).

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

anonymous union. A union that is declared within a structure and that does not have a name.

ANSI. American National Standards Institute.

API. Application program interface.

application. The use to which an information processing system is put, for example, a payroll application, an airline reservation application, a network application.

application program interface (API). The formally defined programming language interface between an IBM system control program or a licensed program and the user of the program.

argument. In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called a parameter.

arithmetic object. An integral object, a bit field, or floating-point object.

array. A variable that contains an ordered group of data objects. All objects in an array have the same data type.

array element. A single data item in an array.

ASCII. American National Standard Code for Information Interchange.

assembly language. A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond directly to the storage and registers of the machine.

assignment conversion. A change to the form of the right operand that makes the right operand have the same data type as the left operand.

assignment expression. An operation that stores the value of the right operand in the storage location specified by the left operand.

associativity. The order for grouping operands with an operator (either left-to-right or right-to-left).

automatic calling. Calling in which the elements of the selection signal are entered into the data network contiguously at the full data signalling rate.

B

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

binary expression. An expression containing two operands and one operator.

binary stream. An ordered sequence of untranslated characters.

bit field. A member of a structure or union that contains a specified number of bits.

block. The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

block statement. Any number of data definitions, declarations, and statements that appear between the symbols { and }. The block statement is considered to be a single C-language statement.

boundary alignment. The position in main storage of a fixed-length field (such as byte or doubleword) on an integral boundary for that unit of information. For example, on System/370 a word boundary is a storage address evenly divisible by two.

break statement. A language control statement that contains the word **break** and a semicolon. It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

buffer. A portion of storage used to hold input or output data temporarily.

buffer flush. A process that removes the contents of a buffer. After a buffer flush, the buffer is empty.

built-in. A function which the compiler will automatically inline instead of the function call unless the programmer specifies not to.

C

C language. A general-purpose high-level programming language.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions.

C language statement. A C language statement contains zero or more expressions. All C language statements, except block statements, end with a **;** (semicolon) symbol. A block statement begins with a **{** (left brace) symbol, ends with a **}** (right brace) symbol, and contains any number of statements.

C library. A system library that contains common C language subroutines for file access, memory allocation, and other functions.

call. To transfer control to a procedure, program, routine, or subroutine.

case clause. In a `switch` statement, a `case` label followed by any number of statements.

case label. The word `case` followed by a constant expression and a colon.

cast expression. An expression that converts the type of the operand to a specified scalar data type (the operator).

cast operator. The cast operator is used for explicit type conversions.

cataloged procedures. A set of control statements placed in a library and retrievable by name.

char specifier. A char is a built-in data type. In C, **char**, **signed char**, and **unsigned char** are all distinct data types.

character constant. A character or an escape sequence enclosed in single quotation marks.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character variable. A data object whose value can be changed during program execution and whose data type is **char**, **signed char**, or **unsigned char.**

CICS. Customer Information Control System.

collating sequence. A specified arrangement for the order of characters in a character set.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

compile. To transform a set of programming language statements (source file) into machine instructions (object module).

compiler. A program that translates instructions written in a programming language (such as C language) into machine language.

complex number. A complex number is made up of two parts: a real part and an imaginary part. A complex number can be represented by an ordered pair (*a*, *b*), where *a* is the value of the real part and *b* is the value of the imaginary part. The same complex number could also be represented as *a* + *bi*, where *i* is the square root of -1.

conditional compilation statement. A preprocessor statement that causes the preprocessor to process specified source code in the file depending on the evaluation of a specific condition.

const. An attribute of a data object that declares the object cannot be changed.

constant expression. An expression having a value that can be determined during compilation and that does not change during program execution.

control statement. A statement that changes the path of execution.

conversion. A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values. Because accuracy of data representation varies among different data types, information may be lost in a conversion.

D

data object. A storage area used to hold a value.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.

data type. A category that specifies the interpretation of a data object such as its mathematical qualities and internal representation.

DBCS. (1) See *double-byte character set*. (2) See *ASCII*.

decimal constant. A numerical data type used in standard arithmetic operations.

declaration. A description that makes an external object or function available to a function or a block.

declare. To identify the variable symbols to be used at preassembly time.

default. An attribute, value or option that is used when no alternative is specified by the programmer.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default clause. In a **switch** statement, the keyword **default** followed by a colon, and one or more statements. When the conditions of the specified **case** labels in the **switch** statement do not hold, the **default** clause is chosen.

default initialization. The initial value assigned to a data object by the compiler if no initial value is specified by the programmer. **extern** and **static** variables receive a default initialization of zero, while the default initial value for **auto** and **register** variables is undefined.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. A data description that reserves storage and may provide an initial value.

demangling. The conversion of mangled names back to their original source code names. See also *mangling*.

denormal. Pertaining to a number with a value so close to `0` that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

digit. Any of the numerals from 0 through 9.

domain. All the possible input values for a function.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires

hardware and supporting software that are DBCS capable.

double precision. Pertaining to the use of two computer words to represent a number with greater accuracy. For example, a floating-point number would be stored in the long format.

doubleword. A sequence of bits or characters that comprises two computer words and can be addressed as a unit.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time.

dynamic binding. Binding that occurs at run time.

E

EBCDIC. See *extended binary-coded decimal interchange code*.

E-format. Floating-point format, consisting of a number in scientific notation.

element. The component of an array, subrange, enumeration, or set.

enumeration constant. An identifier that is defined in an enumerator and that has an associated integer value. You can use an enumeration constant anywhere an integer constant is allowed.

enumeration data type. A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

enumeration tag. The identifier that names an enumeration data type.

enumerator. An enumeration constant and its associated value.

EOF. End of file.

escape sequence. A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one to three octal or hexadecimal digits.

exception. In C, any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine.

executable program. A program that can be run on a processor.

expression. A representation for a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A definition appearing outside a function. The defined object is accessible to all functions that follow the definition and are located within the same source file as the definition.

F

fetch control block (FECB). An executable dynamic stub which is created by a `fetch()` function call. The stub transfers control to the true entry point of the module specified in the `fetch` call. The stub also switches the writable static environment thereby giving each instance of the `fetched` routine its own global data.

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

float constant. A constant representing a nonintegral number.

foreground processing. The execution of a computer program that preempts the use of computer facilities.

free store. Dynamically allocates memory. New and delete are used to allocate and deallocate free store.

function. A named group of statements that can be invoked and evaluated and can return a value to the calling statement.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of arguments.

function declarator. The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a ; (semicolon). It is required by the compiler when the function will be declared later so type checking can occur.

function scope. Labels that are declared in a function have function scope and can be used anywhere in that function.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

G

global. Pertaining to information available to more than one program or subroutine.

global scope. See *file scope*.

global variable. A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

H

halfword. A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit.

hard error. An error condition on a network that requires that the network be reconfigured or that the source of the error be removed before the network can resume reliable operation.

header file. A file that contains system-defined control information that precedes user data.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. The special characters are `\x`, `0x`, or `0X`.

I

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. A text file that contains declarations used by a group of functions, programs, or users. Also known as a header file.

initialize. To set the starting value of a data object.

initializer. An expression used to initialize data objects. In C, there are two types of initializers:

- An expression followed by an assignment operator is used to initialize fundamental data type objects.
- An expression enclosed in braces (**{}**) is used to initialize aggregates.

inlined function. Inlining is a hint to the compiler to perform inline expansion of the body of a function member. Functions declared and defined simultaneously in a class definition are inline. You can also explicitly declare a function inline by using the keyword **inline**. Both member and nonmember functions can be inlined. You can direct the compiler to inline a function with the `inline` keyword.

input stream. A sequence of control statements and data submitted to a system from an input unit.

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

integer constant. A decimal, octal, or hexadecimal constant.

integral boundary. A location in main storage at which a fixed-length field, such as a halfword or doubleword, must be positioned. The address of an integral boundary is a multiple of the length of the field, expressed in bytes.

integral object. A character object, an object having an enumeration type, an object having variations of the type `int`, or an object that is a bit field.

internal data definition. A description of a variable appearing at the beginning of a block that causes storage to be allocated for the lifetime of the block.

interrupt. A temporary suspension of a process caused by an external event, performed in such a way that the process can be resumed.

intrinsic function. A function supplied by a program as opposed to a function supplied by the compiler.

IPL. Initial Program Load.

ISA. Initial Storage Area.

J

JCL. Job Control Language.

K

keyword. (1) A predefined word reserved for the C language, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

L

L-name. An external C name in an object module or an external non-C name in an object module produced by compiling with the LONGNAME option.

label. (1) An identifier followed by a colon. It is the target of a goto statement. (2) An identifier within or attached to a set of data elements.

labeled statement. A possibly empty statement immediately preceded by a label.

late binding. See *dynamic binding*.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, function calls, subroutines, or other data. (2) A set of object modules that can be specified in a link command.

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

linkage editor. Synonym for linker.

linker. A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single executable program (phase).

literal. See *constant*.

loader. A routine, commonly a computer program, that reads data into main storage.

local. Pertaining to information that is defined and available in only one function of a computer program.

local scope. A name declared in a block has local scope and can only be used in that block.

long constant. An integer constant followed by the letter L in uppercase or lowercase.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

main function. A function with the identifier `main` that is the first user function to get control when program execution begins. Each C program must have exactly one function named `main`.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. The linker uses these mangled names to ensure type-safe linkage.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

map. A set of values having a defined correspondence with the quantities or values of another set.

map file. A listing file that can be created during the link step and that contains information on the size and mapping of segments and symbols.

mapping. The establishing of correspondences between a given logical structure and a given physical structure.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

member. A data object in a structure or a union.

metalanguage. A language used to specify another language.

migrate. To move to a changed operating environment, usually to a new release or version of a system.

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multiprocessing. Simultaneous or parallel processing of two or more computer programs or sequences by a multiprocessor.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of more than one task or program.

N

newline character. A control character that causes the print or display position to move to the first position on the next line. This control character is represented by `\n` in the C language.

NULL. A pointer guaranteed not to point to a data object.

null character (\0). The ASCII or EBCDIC character with the hex value `00`, all bits turned off.

null value. A parameter position for which no value is specified.

O

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as C language).

object module. A portion of an object program produced by a compiler from a source program, and suitable as input to a linkage editor.

octal. A base eight numbering system.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

operand. An entity on which an operation is performed.

operating system. Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operation. A specific action such as add, multiply, shift.

operator. A symbol (such as +, -, *) that represents an operation (in this case, addition, subtraction, multiplication).

overflow. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overflow condition. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overlay. To write over existing data in storage.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C operators when the functions and operators are used with class types.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

pad. To fill unused positions in a field with data, usually zeros, ones, or blanks.

parameter declaration. A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

persistent environment. A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

pointer. A variable that holds the address of a data object or function.

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

precision. A measure of the ability to distinguish between nearly equal values.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

preprocessor statement. A statement that begins with the symbol # and is interpreted by the preprocessor.

primary expression. An identifier, a parenthesized expression, a function call, an array element specification, or a structure or union member specification.

process. An instance of an executing application and the resources it uses.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters.

R

record. The unit of data transmitted to and from a program.

recoverable error. An error condition that allows continued execution of a program.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

register. A storage area commonly associated with fast-access storage, capable of storing a specified amount of data such as a bit or an address.

reserved word. In programming languages, a keyword that may not be used as an identifier.

rounding. To omit one or more of the least significant digits in a positional representation and to adjust the remaining digits according to a specified rule. The purpose of rounding is usually to limit the precision of a number or to reduce the number of characters in the number.

run-time library. A collection of functions in object code form, whose members can be referred to by an application program during the linking step.

S

S-name. An external non-C name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

SAA. Systems Application Architecture.

scalar. An arithmetic object, or a pointer to an object of any type.

scope. That part of a source program in which an object is defined and recognized.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape.

signal. A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero.

signal handler. A function to be called when the signal is reported.

single-byte character set. A set of characters in which each character is represented by 1 byte of storage.

single precision. Pertaining to the use of one computer word to represent a number, in accordance with the required precision.

software signal. A signal that is explicitly raised by the user (by using the `raise` function).

source file. A file that contains source statements for such items as language programs and data description specifications.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

SQL. Structured Query Language.

stack. An area of storage used for keeping variables associated with each call to a function or block.

stand-alone. Pertaining to operation that is independent of any other device, program, or system.

statement. An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

static binding. Binding that occurs at compilation time based on the resolution of overloaded functions.

storage class specifier. One of: **auto**, **register**, **static**, or **extern**.

stream. See *data stream*.

string constant. Zero or more characters enclosed in double quotation marks.

structure. A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

structure tag. The identifier that names a structure data type.

stub routine. Within run-time libraries, contains the minimum lines of code required to locate a given routine at run time.

subsystem. A secondary or subordinate system, or programming support, usually capable of operating independently of or asynchronously with a controlling system.

swap. To exchange one thing for another.

switch expression. The controlling expression of a **switch** statement.

switch statement. A C language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile.

Systems Application Architecture (SAA). Pertaining to the definition of a common programming interface, conventions, and protocols for designing and developing applications with cross-system consistency.

T

tag. One or more characters attached to a set of data that identifies the set.

task. One or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer.

template function. A function generated by a function template.

thread. A unit of execution within a process.

trap. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred.

trigraph sequence. A combination of three keystrokes used to represent unavailable characters in a C source program. Before preprocessing, each trigraph sequence in a string or a literal is replaced by the single character that it represents.

truncate. To shorten a value to a specified length.

try block. A block in which a known C exception is passed to a handler.

type. The description of the data and the operations that can be performed on or by the data. Also see *data type*.

type balancing. A conversion that makes both operands have the same data type. If the operands do not have the same size data type, the compiler converts the value of the operand with the smaller type to a value having the larger type.

type conversion. See *boundary alignment*.

type definition. A definition of a data type.

type specifier. Used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unary expression. An expression that contains one operand.

underflow. A condition that occurs when the result of an operation is less than the smallest possible nonzero number.

union. A construct that can hold any one of several data types, but only one data type at a time.

union tag. The identifier that names a union data type.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

V

variable. An object that can take different values at different times.

visible. Visibility of identifiers is based on scoping rules and is independent of *access.*

volatile. An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

VSAM. Virtual Storage Access Method.

W

whitespace. Space characters, tab characters, form feed characters, and newline characters.

wide character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

word boundary. The storage position at which data must be aligned for certain processing operations. The halfword boundary must be divisible by 2, the fullword boundary by 4, and the doubleword boundary by 8.

Z

zero suppression. The removal of, or substitution of blanks for, leading zeros in a number. For example, `00057` becomes ` 57` when using zero suppression.

Bibliography

IBM C for VSE/ESA Publications

Licensed Program Specifications, GC09-2421

Installation and Customization Guide, GC09-2422

Migration Guide, SC09-2423

User's Guide, SC09-2424

Language Reference, SC09-2425

Diagnosis Guide, GC09-2426

IBM Language Environment for VSE/ESA Publications

Fact Sheet, GC33-6679

Concepts Guide, GC33-6680

Debugging Guide and Run-Time Messages, SC33-6681

Installation and Customization Guide, SC33-6682

Licensed Program Specifications, GC33-6683

Programming Guide, SC33-6684

Programming Reference, SC33-6685

Run-Time Migration Guide, SC33-6687

Writing Interlanguage Communication Applications, SC33-6686

C Run-Time Programming Guide, SC33-6688

C Run-Time Library Reference, SC33-6689

Related Publications

VSE/ESA Version 1 Release 4

System Control Statements, SC33-6513

System Macros User's Guide, SC33-6515

System Macros Reference, SC33-6516

VSE/ESA Version 2

System Control Statements, SC33-6613

System Macros User's Guide, SC33-6615

System Macros Reference, SC33-6616

Debug Tool for VSE/ESA

User's Guide and Reference, SC26-8797

Softcopy Publications

The following collection kit contains the C/VSE, LE/VSE, and other LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

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

Index

Special Characters

A

B

C

D

E

F

G

H

I

P

R

S

T

U

W

X

Communicating Your Comments to IBM

IBM C for VSE/ESA
User's Guide
Release 1

Publication No. SC09-2424-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@vnet.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

IBM C for VSE/ESA
User's Guide
Release 1

Publication No. SC09-2424-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

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

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SC09-2424-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5686-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2424-00