IBM Language Environment for VSE/ESA

C Run-Time Programming Guide

Version 1 Release 4 Modification Level 4

SC33-6688-05

IBM Language Environment for VSE/ESA

C Run-Time Programming Guide

Version 1 Release 4 Modification Level 4

SC33-6688-05

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

Sixth Edition (March 2005)

This edition applies to Version 1 Release 4 Modification Level 4 of IBM Language Environment for VSE/ESA, 5686-CF7, and to any subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses 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 Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

You may also send your comments by FAX or via the Internet:

```
Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456
```

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

Figures

Tables

Notices

Programming Interface Information

This book is intended to help the customer program for LE/VSE C Run-Time. This book documents General-Use Programming Interface and associated guidance information provided by the IBM Language Environment for VSE/ESA (LE/VSE) and LE/VSE C Run-Time.

General-Use Programming Interfaces allow the customer to write programs that obtain the services of the C/VSE compiler and LE/VSE.

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.

Trademarks

About This Book

z/VSE is the successor to IBM's VSE/ESA product. Many products and functions supported on z/VSE may continue to use VSE/ESA in their names.

z/VSE can execute in 31-bit mode only. It does not implement z/Architecture, and specifically does not implement 64-bit mode capabilities.

z/VSE is designed to exploit select features of IBM eServer zSeries hardware.

This book provides information about implementing programs written for IBM Language Environment for VSE/ESA (LE/VSE). This book contains guidelines and information for advanced programming topics for developing C language programs to run under the VSE operating system with LE/VSE.

To use this book, or any other books in the library of LE/VSE C Run-Time publications, you must have a working knowledge of the C programming language, the operating system, and where appropriate, the related products. This edition of the *Programming Guide* is intended for users of the C language.

What Is LE/VSE?

LE/VSE is a set of common services and language-specific routines that provide a single run-time environment for applications written in *LE/VSE-conforming* versions of the C, COBOL, and PL/I high level languages (HLLs), and for many applications written in previous versions of COBOL. (For a list of LE/VSE-conforming languages, and a description of compatibility with previous versions of COBOL, see "LE/VSE-Conforming Languages" on page xvi.) LE/VSE also supports applications written in assembler language using LE/VSE-provided macros and assembled using High Level Assembler (HLASM).

Prior to LE/VSE, each programming language provided its own separate run-time environment. LE/VSE combines essential and commonly-used run-time services—such as message handling, condition handling, storage management, date and time services, and math functions—and makes them available through a set of interfaces that are consistent across programming languages. With LE/VSE, you can use one run-time environment for your applications, regardless of the application's programming language or system resource needs, because most system dependencies have been removed.

Services that work with only one language are available within language-specific portions of LE/VSE.

LE/VSE consists of:

- Basic routines for starting and stopping programs, allocating storage, communicating with programs written in different languages, and indicating and handling error conditions.
- Common library services, such as math services and date and time services, that are commonly needed by programs running on the system. These functions are supported through a library of callable services.

- Language-specific portions of the common run-time library.

LE/VSE is the implementation of Language Environment on the VSE platform. Language Environment is also offered on platforms z/OS and VM, and on OS/400 as Integrated Language Environment.

LE/VSE-Conforming Languages

An LE/VSE-conforming language is any HLL that adheres to the LE/VSE common interface. Table 1 lists the LE/VSE-conforming language compiler products you can use to generate applications that run with LE/VSE Release 4.

Table 1. LE/VSE-Conforming Languages

Language	LE/VSE-Conforming Language	Minimum Release
C	IBM C for VSE/ESA	Release 1
COBOL	IBM COBOL for VSE/ESA	Release 1
PL/I	IBM PL/I for VSE/ESA	Release 1

Any HLL not listed in Table 1 is known as a *non-LE/VSE-conforming* or, alternatively, a *pre-LE/VSE-conforming* language. Some examples of non-LE/VSE-conforming languages are:
- C/370
- DOS/VS COBOL
- VS COBOL II
- DOS PL/I
- DOS/VS RPG II

Only the following products can generate applications that run with LE/VSE:
- LE/VSE-conforming languages
- HLASM using LE/VSE-provided macros (for details, see *LE/VSE Programming Guide*)
- DOS/VS COBOL and VS COBOL II, with some restrictions (see LE/VSE Compatibility with Previous Versions of COBOL below).

LE/VSE Compatibility with Previous Versions of COBOL

Although DOS/VS COBOL and VS COBOL II are non-LE/VSE-conforming languages, many applications generated with these compilers can run with LE/VSE without recompiling. For details about compatibility, see *LE/VSE Run-Time Migration Guide*.

However relinking under LE/VSE is the *minimum* effort in order to migrate run-time, and involve LE/VSE COBOL-compatibility routines (rather than the old and unsupported library routines of non-LE/VSE conforming COBOL compilers). This particularily applies to NORES-compiled units or applications that involve former initialization techniques such as ILBDSET0. There are even restrictions with this approach, such as:
- No use of 4-digit dates.
- No exploitation of LE/VSE functionality.
- Interlanguage communication capabilities, and so on.

Therefore you are *strongly recommended* to carry out a (subsequent) full migration to a higher ANSI standard and LE/VSE-conforming COBOL compiler (COBOL for VSE/ESA).

VS COBOL II can also dynamically call some LE/VSE date and time callable services. For details, see *LE/VSE Programming Reference*.

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 language supported by LE/VSE C Run-Time, 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, z/OS, and VM operating systems.

The elements of the LE/VSE C Run-Time 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 "LE/VSE C Run-Time Support for Internationalization" on page 318.)
- 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)

Softcopy Examples

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

- *LE/VSE C Run-Time Library Reference*, SC33-6689
- *LE/VSE C Run-Time Programming Guide*, SC33-6688

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

- B is the *LE/VSE C Run-Time Library Reference*
- G is the *LE/VSE C Run-Time Programming Guide*

Softcopy examples are installed on your system along with the LE/VSE C Run-Time component, in the sublibrary `PRD2.SCEEBASE`.

Example member names are the same as the labels indicated in the book, with a "`.C`" extension.

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

How to Read the Syntax Diagrams

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

- Read the syntax diagrams from left to right, top to bottom following the path of the line.
- Each syntax diagram begins with a double arrowhead (►►).
- An arrow (─►) at the end of a line indicates that the option, service, or macro syntax continues on the next line. A continuation line begins with an arrow (►─).
- If a syntax diagram contains too many items or groups to fit in the diagram, the syntax is shown by a main syntax diagram and one or more syntax fragments. A syntax fragment is referred to in the main diagram by its fragment name between two vertical bars (|).

  Each syntax fragment appears below the main syntax diagram, and begins and ends with a vertical bar (|). A heading above the fragment indicates the name of the fragment.

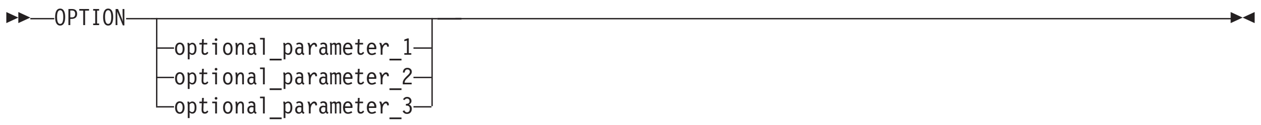
  Read each syntax fragment as though it were imbedded in the main syntax diagram.
- IBM-supplied default keywords appear **above** the main path or options path (see the sample on page xix). In the parameter list, IBM-supplied default choices are underlined.
- Keywords appear in nonitalic capital letters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

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

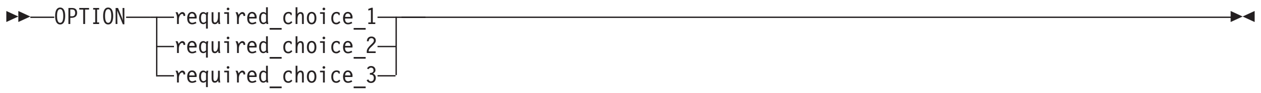
```
►►──OPTION──required_parameter────────────────────────────────────────────────────►◄
```

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

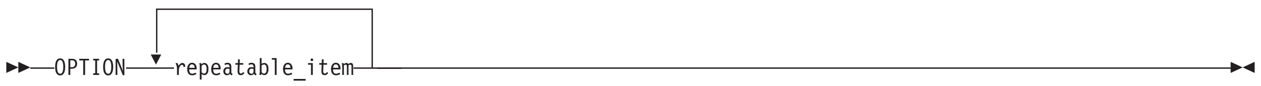
  If choosing one of the items is optional, the entire stack appears below the main line.



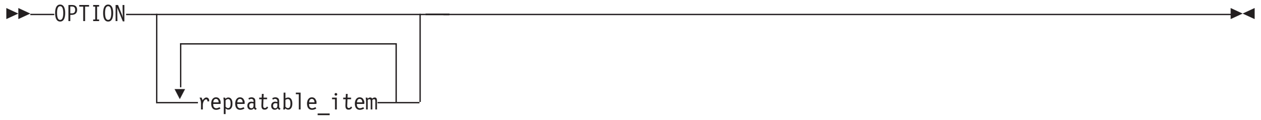
If you *must* choose one of the items, one item of the stack appears on the main path:



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

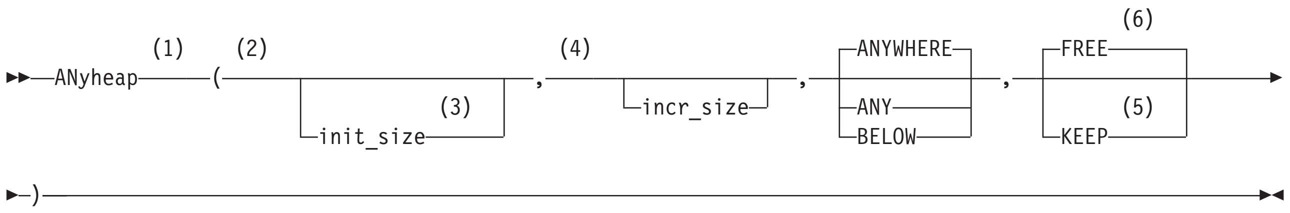


OR



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

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



Notes:

1 Keyword with minimum unambiguous truncation shown in capital letters

2 Opening parenthesis (must be specified if any parameters are specified)

3 Optional parameter

4 Comma (must be specified if there are parameters that follow)

5 Optional keyword

6 Optional keyword (IBM-supplied default)

Where to Find More Information

These are the manuals that describe LE/VSE:

Table 2. LE/VSE Publications

Publication	Form Number
LE/VSE Fact Sheet	GC33-6679
LE/VSE Concepts Guide	GC33-6680
LE/VSE Customization Guide	SC33-6682
LE/VSE Programming Guide	SC33-6684
LE/VSE Programming Reference	SC33-6685
LE/VSE C Run-Time Programming Guide	SC33-6688
LE/VSE C Run-Time Library Reference	SC33-6689
LE/VSE Debugging Guide and Run-Time Messages	SC33-6681
LE/VSE Writing Interlanguage Communication Applications	SC33-6686
LE/VSE Run-Time Migration Guide	SC33-6687
LE/VSE Licensed Program Specifications	GC33-6683

These are the z/VSE manuals to which you might need to refer:

Table 3. z/VSE Publications

Publication	Form Number
z/VSE Administration	SC33-8224
z/VSE Messages and Codes, Volume 1	SC33-8226
z/VSE Messages and Codes, Volume 2	SC33-8227
z/VSE Messages and Codes, Volume 3	SC33-8228
z/VSE Planning	SC33-8221
z/VSE System Control Statements	SC33-8225
z/VSE System Macros Reference	SC33-8230
z/VSE System Macros User's Guide	SC33-8236
z/VSE System Upgrade and Service	SC33-8223
VSE/VSAM User's Guide and Application Programming	SC33-8246
VSE/VSAM Commands	SC33-8245
TCP/IP for VSE/ESA IBM Program Setup and Supplementary Information	SC33-6601

These are the manuals that describe IBM C for VSE/ESA:

Table 4. IBM C for VSE/ESA Publications

Publication	Form Number
Licensed Program Specifications	GC09-2421
Installation and Customization Guide	GC09-2422
Migration Guide	SC09-2423

Table 4. IBM C for VSE/ESA Publications (continued)

Publication	Form Number
User's Guide	SC09-2424
Language Reference	SC09-2425
Diagnosis Guide	GC09-2426

These are the manuals that describe IBM COBOL for VSE/ESA:

Table 5. IBM COBOL for VSE/ESA Publications

Publication	Form Number
General Information	GC33-6679
Licensed Program Specifications	GC33-6680
Migration Guide	SC33-6682
Installation and Customization Guide	GC33-6680
Programming Guide	SC33-6684
Language Reference	SC33-6685
Diagnosis Guide	SC33-6684
Reference Summary	SX26-3834

These are the manuals that describe IBM PL/I for VSE/ESA:

Table 6. IBM PL/I for VSE/ESA Publications

Publication	Form Number
Fact Sheet	GC26-8052
Programming Guide	SC26-8053
Language Reference	SC26-8054
Licensed Program Specifications	GC26-8055
Migration Guide	SC33-6684
Installation and Customization Guide	SC26-8057
Diagnosis Guide	SC26-8058
Compile-Time Messages and Codes	SC26-8059
Reference Summary	SX26-3836

These are the manuals that describe Debug Tool for VSE/ESA:

Table 7. Debug Tool for VSE/ESA Publications

Publication	Form Number
User's Guide and Reference	SC26-8797
Installation and Customization Guide	SC26-8798
Fact Sheet	GC26-8925

You might also refer to the ...

z/VSE Home Page

z/VSE has a home page on the World Wide Web, which offers up-to-date information about VSE-related products and services, new z/VSE functions, and other items of interest to VSE users.

You can find the z/VSE home page at:

`http://www.ibm.com/servers/eserver/zseries/zvse/`

Softcopy Publications

The following collection kit contains the LE/VSE and LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

Summary of Changes

This section describes the changes introduced with the sixth edition and the previous three editions of the manual.

Changes Introduced With Sixth Edition (March 2005)

These are the changes included in the sixth edition of this manual (for LE/VSE 1.4.4):

- The name **VSE/ESA** has now changed to **z/VSE**. However, the names of many features and programs related to z/VSE remain unchanged (such as IBM Language Environment for VSE/ESA, IBM COBOL for VSE/ESA, or Debug Tool for VSE/ESA).
- New entries have been made to these tables because of the ongoing implementation of Euro support:
  - Table 46 on page 354 ("Supported Language-Territory Names and LT Codes")
  - Table 47 on page 356 ("Supported Codeset Names and CC Codes")
  - Table 48 on page 378 ("Coded Character Set Conversion Table")
  - Table 49 on page 385 ("UCS-2 Converters")
  - Table 52 on page 431 ("Compiled Locales Supplied With LE/VSE C Run-Time")
  - Table 53 on page 437 ("Locale Source Files Supplied With LE/VSE C Run-Time")
  - Table 54 on page 443 ("Coded Character Set Names and Corresponding National Languages")
- DSECT utility options `DECIMAL|NODECIMAL` and `UNIQUE|NOUNIQUE` have been added. See "DECIMAL | NODECIMAL" on page 471 and "UNIQUE | NOUNIQUE" on page 479 respectively.

Changes Introduced With Fifth Edition (March 2003)

The fifth edition of the *C Run-Time Programming Guide*, SC33-6688-04 (March 2003), contained these changes:

- For functions `fopen()` and `freopen()`, the maximum blocksize was increased from 32760 to 65528. See:
  - "LRECL and BLKSIZE Defaults" on page 19.
  - Table 17 on page 50 ("Parameters for the`fopen()` and `freopen()` Functions for SAM I/O").
  - Table 19 on page 74 ("Parameters for the `fopen()` and `freopen()` Functions for VSE/Librarian I/O").
- For functions `fseek()` and `ftell()`, the maximum blocksize was also increased from 32760 to 65528. See "Using `fseek()` and `ftell()` in Text Files (ASA and Non-ASA)" on page 68.

Changes Introduced With Fourth Edition (December 2001)

The fourth edition of the *C Run-Time Programming Guide*, SC33-6688-03 (December 2001), contained minor changes and corrections only.

Part 1. Input and Output

This part describes the models of input and output available with IBM Language Environment for VSE/ESA.

Chapter 1. Introduction to C Input and Output

This chapter provides you with a general introduction to C input and output (I/O). The different types of C input and output are discussed in this chapter: text streams, binary streams, and record I/O.

Types of C Input and Output

A stream is a continuous flow of data elements that are transmitted or intended for transmission in a defined format. A record is a set of data elements treated as a unit, and a file is a named set of records that is stored or processed as a unit.

LE/VSE C Run-Time supports three types of input and output: text streams, binary streams, and record I/O. Text and binary streams are both ANSI standards; record I/O is an LE/VSE C Run-Time extension, initially provided with C/370.

Note: If you have written data in one of these three types and try to read it as another type (for example, reading a binary file in text mode), you may not get the behavior that you expect.

Text streams
contain printable characters and, depending on the type of file, control characters. Text streams are organized into lines. Each line ends with a control character, usually a newline. The last record in a text file may or may not end with a control character, depending on what kind of file you are using. Text files recognize the following control characters:

`\a`	Alarm.
`\b`	Backspace.
`\f`	Form feed.
`\n`	Newline.
`\r`	Carriage return.
`\t`	Horizontal tab character.
`\v`	Vertical tab character.
`\x0E`	DBCS shift out character. Indicates the beginning of a DBCS string, if `MB_CUR_MAX > 1` in the definition of the locale that is in effect. For more information about `MB_CUR_MAX`, see Chapter 6, “LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS),” on page 29.
`\x0F`	DBCS shift in character. Indicates the end of a DBCS string, if `MB_CUR_MAX > 1` in the definition of the locale that is in effect. For more information about `MB_CUR_MAX`, see Chapter 6, “LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS),” on page 29.

Control characters behave differently in ASA files (see Chapter 5, “ASA Text Files,” on page 25).

Binary streams
contain an ordered sequence of bytes. For binary streams, the library does not translate any characters on input or output. It treats them as a continuous stream of bytes, and ignores any record boundaries.

Record I/O
is an IBM C Run-Time extension to the ANSI standard. For files opened in record format, LE/VSE C Run-Time reads and writes one record at a time. If you try to write more data to a record than the record can hold, the data is truncated. For record I/O, LE/VSE C Run-Time only allows the use of `fread()` and `fwrite()` to read from and write to files. Any other functions (such as `fprintf()`, `fscanf()`, `getc()`, and `putc()`) fail. For record-oriented files, records do not change size when you update them. If the new data has fewer characters than the original record, the new data fills the first *n* characters, where *n* is the number of characters of the new data. The record will remain the same size, and the old characters (those after *n*) are left unchanged. A subsequent update begins at the next boundary. For example, if you have the string `"abcdefgh"`:

a	b	c	d	e	f	g	h

and you overwrite it with the string `"1234"`, the record will look like this:

1	2	3	4	e	f	g	h

LE/VSE C Run-Time record I/O is binary. That is, it does not interpret any of the data in a record file and therefore does not recognize control characters. The only exception is for file categories that do not support records. For these files, LE/VSE C Run-Time uses newline characters as record boundaries.

Chapter 2. Models of C I/O

This chapter describes LE/VSE C Run-Time's support for the major models of C I/O, the *record model* and the *byte stream model*.

The Record Model for C I/O

Almost all the kinds of I/O that LE/VSE C Run-Time supports use this model. The only one that does not is memory file I/O.

The record model consists of:

- A *record*, which is the unit of data transmitted to and from a program.
- A *block*, which is the unit of data transmitted to and from a device. Each block may contain one or more records.

In the record model of I/O, records and blocks have the following attributes:

RECFM	Specifies the format of the data or how the data is organized on the physical device.
LRECL	Specifies the length of logical records (as opposed to physical ones).
BLKSIZE	Specifies the length of physical records (blocks on the physical device).

Record Formats

Use the RECFM attribute to specify the record format. The records in a file using the record model have one of the following formats:

`F`	Fixed-length
`V`	Variable-length
`U`	Undefined-length

Note: LE/VSE C Run-Time does not support ISCII/ASCII format-`D` files.

These formats support the following additional options for RECFM:

`A`	Specifies that the file contains ASA print-control characters.
`B`	Specifies that a file is blocked. A blocked file can have more than one record in each block.
`M`	Specifies that the file contains machine control codes.
`S`	Specifies that a file is either in standard format (if it is fixed) or spanned (if it is variable). In a standard file, every block must be full before another one starts. In a spanned file, a record can be longer than a block. If it is, the record is divided into segments and stored in consecutive blocks.

The record formats and the additional options associated with them are discussed in the following sections.

Not all the I/O categories (listed in Table 9 on page 15) support all of these attributes. Depending on what category you are using, LE/VSE C Run-Time ignores or simulates attributes that do not apply. For more information, on the record formats and the options supported for each I/O category, see the section called "Opening Files" in the chapter pertaining to the category.

Fixed-Format Records

Record Format (RECFM): These are the formats you can specify for RECFM if you want to use a fixed-format file:

`F`	Fixed-length, unblocked
`FA`	Fixed-length, unblocked, ASA print-control characters
`FB`	Fixed-length, blocked
`FM`	Fixed-length, unblocked, machine control codes
`FS`	Fixed-length, unblocked, standard
`FBA`	Fixed-length, blocked, ASA print-control characters
`FBM`	Fixed-length, blocked, machine control codes
`FBS`	Fixed-length, blocked, standard
`FSA`	Fixed-length, unblocked, standard, ASA print-control characters
`FSM`	Fixed-length, unblocked, standard, machine control codes
`FBSM`	Fixed-length, blocked, standard, machine control codes
`FBSA`	Fixed-length, blocked, standard, ASA print-control characters

Note: In general, all references in this guide to files with record format `FB` also refer to `FBM` and `FBA`. The specific behavior of ASA files (such as `FBA`) is explained in Chapter 5, "ASA Text Files," on page 25.

Attention: LE/VSE C Run-Time distinguishes between `FB` and `FBS` formats, because an `FBS` file contains no embedded short blocks (the last block may be short). `FBS` files give you much better performance if file repositioning is used. The use of standard (`S`) blocks optimizes the sequential processing of a file on a direct-access device. With a standard format file, the file pointer can be directly repositioned by calculating the exact position in that file of a given record rather than reading through the entire file.

If the records are `FB`, some blocks may contain fewer records than others, as shown in Figure 1.

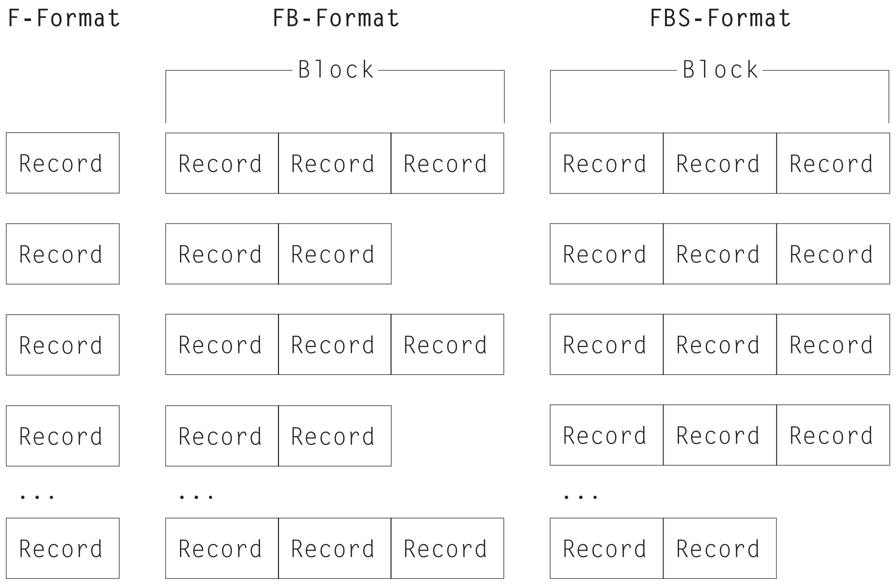


Figure 1. Blocking Fixed-Length Records

Mapping C Types to Fixed Format:

Binary

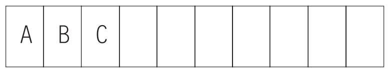
On binary input and output, data flows over record boundaries. Because all fixed-format records must be full, LE/VSE C Run-Time completes any incomplete output record by padding it with nulls (`'\0'`) when you close the file. Incomplete *blocks* are not padded. On input, nulls are visible and are treated as part of the data.

For example, if LRECL is set to 10 and you are writing 25 characters of data, LE/VSE C Run-Time will write two full records, each containing 10 characters, and then an incomplete record containing 5 characters. If you then close the file, LE/VSE C Run-Time will complete the last record with 5 nulls. If you open the file for reading, LE/VSE C Run-Time will read the records in order; it will not strip off the nulls at the end of the last record.

Text (non-ASA)

When writing in a text stream, you indicate the end of the data for a record by writing a newline (`'\n'`) or carriage return (`'\r'`) to the stream. In a fixed-format file, the newline or carriage return will not appear in the external file, and the record will be padded with blanks from the position of the newline or carriage return to LRECL. (A carriage return is considered the same as a newline because the `'\r'` is not written to the file.)

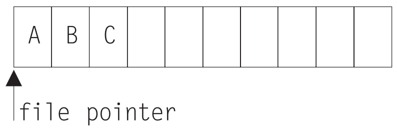
For example, if you have set LRECL to 10, and you write the string "`ABC\n`" to a fixed-format text file, LE/VSE C Run-Time will write this to the physical file:



A record containing only a newline is written to the file as LRECL blanks.

When reading in a text stream, the I/O functions place a newline character ('\n') in the buffer to indicate the end of data for the record. In a fixed-format file, the newline character is placed at the start of the blank padding at the end of the data.

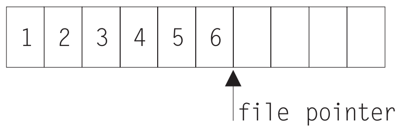
For example, if your file position points to the start of the following record in a fixed-format file opened as a text stream
and you call `fgets()` to read the line of text, `fgets()` places the string



"`ABC\n`" in your input buffer.

Attention: Any blanks written immediately before a newline or carriage return will be considered blank padding when the record is read back from the file. You cannot change the padding character.

When you are updating a fixed-format file opened as a text stream, you can update the amount of data in a record. The maximum length of the updated data is LRECL bytes plus the newline character; the minimum length is zero data bytes plus the newline character. Writing new data into an existing record replaces the old data. If the new data is longer or shorter than the old data, the number of blank padding characters in the record in the external file is changed. When you extend a record, thereby writing over the old newline, there will be a newline character implied after the new characters. For instance, if you were to overwrite the record mentioned in the previous example with the string "123456", the records in the physical file would then look like this:



The blanks at the end of the record imply a newline at position 7. You can see this newline by calling `fflush()` and then performing a read. The implied newline is the first character returned from this read.

A fixed record can hold only LRECL characters. If you try to write more than that, LE/VSE C Run-Time truncates the data unless you are using a standard stream. In this case, the output is split across multiple records. If truncation occurs, LE/VSE C Run-Time raises `SIGIOERR` and sets both `errno` and the error flag.

Text (ASA)

For ASA files, the first character of each record is reserved for the ASA print-control character that represents a newline, a carriage return, or a form feed. This control character represents what should happen before the record is written.

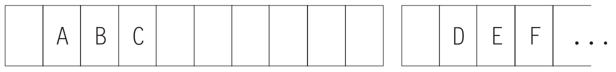
Table 8. C Control to ASA Characters

C Control Character	ASA Character	Description
`'\n'`	`' '`	Skip one line
`'\n\n'`	`'0'`	Skip two lines
`'\n\n\n'`	`'-'`	Skip three lines
`'\f'`	`'1'`	New page
`'\r'`	`'+'`	Overstrike

A control character that ends a logical record is represented at the beginning of the following record in the external file. Since the ASA print-control character is in the first byte of each record, a record can hold only LRECL - 1 bytes of data. As with non-ASA text files described above, LE/VSE C Run-Time adds blank padding to complete any record shorter than LRECL - 1 when it writes the record to the file. On input, LE/VSE C Run-Time removes all trailing blanks. For example, if LRECL is 10, and you enter the string:

```
\nABC\nDEF
```

the record in the physical file will look like this:



On input, this string is read as follows:

```
\nABC\nDEF
```

You can lengthen and shorten records the same way as you can for non-ASA files. For more information about ASA, refer to Chapter 5, “ASA Text Files,” on page 25.

Record
: As with fixed-format text files, a record can hold LRECL characters. Every call to `fwrite()` is considered to be writing a full record. If you write fewer than LRECL characters, LE/VSE C Run-Time completes the record with enough nulls to make it LRECL characters long. If you try to write more than that, LE/VSE C Run-Time truncates the data.

Variable-Format Records

In a file with variable-length records, each record may be a different length. The variable length formats permit both variable-length records and variable-length blocks. The first 4 bytes of each block are reserved for the Block Descriptor Word (BDW); the first 4 bytes of each record are reserved for the Record Descriptor Word (RDW), or, if you are using spanned files, the Segment Descriptor Word (SDW). Illustrations of variable-length records are shown in Figure 2 on page 10.

Once you have set the LRECL for a variable-format file, you can write up to LRECL minus 4 characters in each record. LE/VSE C Run-Time does not let you see RDWs, BDWs, or SDWs when you open a file as variable-format. To see the RDWs or SDWs and BDWs, open the variable file as undefined-format, as described in “Undefined-Format Records” on page 12.

The value of LRECL must be greater than 4 to accommodate the RDW or SDW. The value of BLKSIZE must be greater than or equal to the value of LRECL plus 4. You should not use a BLKSIZE greater than the maximum logical record length plus 4 for an unblocked file. Doing so results in buffers that are larger than they need to be. The largest amount of data that any one record can hold is LRECL bytes minus 4.

Record Format (RECFM): You can specify the following formats for variable-length records:

`V`	Variable-length, unblocked
`VA`	Variable-length, unblocked, ASA print-control characters
`VB`	Variable-length, blocked
`VM`	Variable-length, unblocked, machine control codes
`VS`	Variable-length, unblocked, spanned
`VBA`	Variable-length, blocked, ASA print-control characters
`VBM`	Variable-length, blocked, machine control codes
`VBS`	Variable-length, blocked, spanned
`VSA`	Variable-length, unblocked, spanned, ASA print-control characters
`VSM`	Variable-length, unblocked, spanned, machine control codes
`VBSA`	Variable-length, blocked, spanned, ASA print-control characters
`VBSM`	Variable-length, blocked, spanned, machine control codes

Note: In general, all references in this guide to files with record format `VB` also refer to `VBM` and `VBA`. The specific behavior of ASA files (such as `VBA`) is explained in Chapter 5, “ASA Text Files,” on page 25.

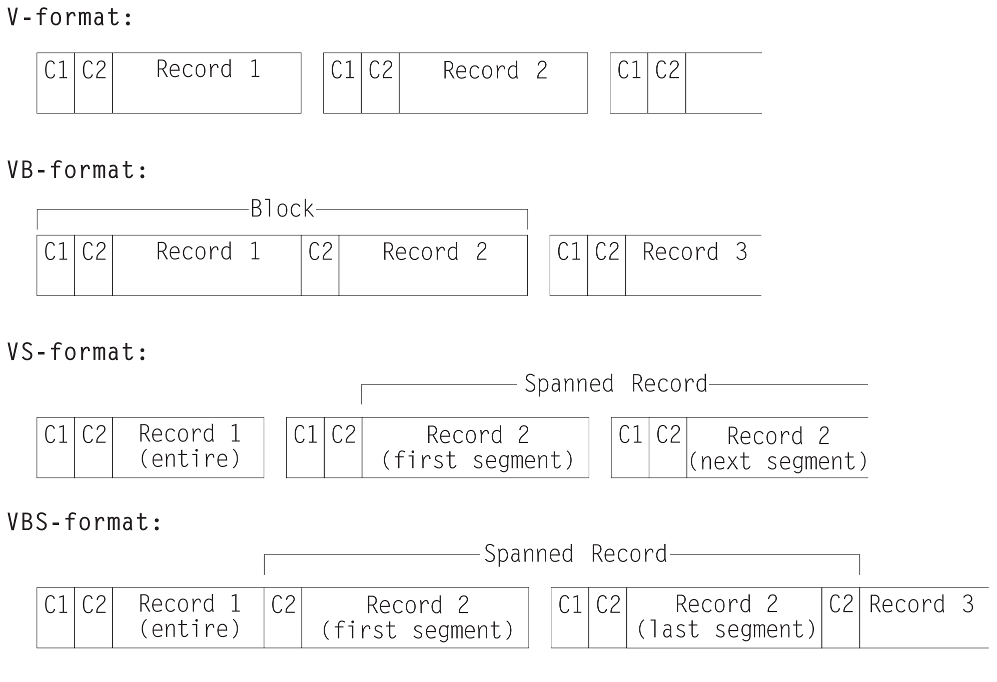
`V`-format signifies unblocked variable-length records. Each record is treated as a block containing only one record.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate.

Spanned Records: A spanned record is opened using both V and S in the format specifier. A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If it does, the record is divided into segments and accommodated in two or more consecutive blocks. The use of spanned records allows you to select a block size, independent of record length, that will combine optimum use of auxiliary storage with the maximum efficiency of transmission.

VS-format specifies that each block contains only one record or segment of a record. The first 4 bytes of a block describe the block control information. The second 4 bytes contain record or segment control information, including an indication of whether the record is complete or is a first, intermediate, or last segment.

VBS-format differs from VS-format in that each block in VBS-format contains as many complete records or segments as it can accommodate, while each block in VS-format contains at most one record per block.


V-format:
C1 C2 Record 1
C1 C2 Record 2
C1 C2
VB-format:
Block
C1 C2 Record 1 C2 Record 2
C1 C2 Record 3
VS-format:
Spanned Record
C1 C2 Record 1 (entire)
C1 C2 Record 2 (first segment)
C1 C2 Record 2 (next segment)
VBS-format:
Spanned Record
C1 C2 Record 1 (entire) C2 Record 2 (first segment)
C1 C2 Record 2 (last segment) C2 Record 3

C1: Block control information
C2: Record or segment control information

Figure 2. Variable-Length Records on VSE

Mapping C Types to Variable Format:

Binary
: On input and output, data flows over record boundaries. Any record will hold up to LRECL minus 4 characters of data. If you try to write more than that, your data will go to the next record, after the RDW or SDW. You will not be able to see the descriptor words when you read the file.

Note: If you need to see the BDWs, RDWs, or SDWs, you can open and read a V-format file as a U-format file. See "Undefined-Format Records" on page 12 for more information.

LE/VSE C Run-Time never creates empty binary records for files opened in V-format. See "Writing to Binary Files" on page 59 for more information. An empty binary record is one that contains only an RDW, which is 4 bytes long. On input, empty records are ignored.

Text (non-ASA)

Record boundaries are used in the physical file to represent the position of the newline character. You can indicate the end of a record by including a newline or carriage return character in your data. In variable-format files, LE/VSE C Run-Time treats the carriage return character as if it were a newline. LE/VSE C Run-Time does not write either of these characters to the physical file; instead, it creates a record boundary. When you read the file back, boundaries are read as newlines.

If a record only contains a newline character, the default behavior of LE/VSE C Run-Time is to write a record containing a single blank to the file. Therefore, the string `' \n'` is treated the same way as the string `'\n'`; both are read back as `'\n'`. All other blanks in your output are read back as is. Any empty (zero-length) record is ignored on input. However, if the environment variable `_EDC_ZERO_RECLEN` was set to `Y` at the time the file was opened, a single newline is written to the file as an empty record, and a single blank represents `' \n'`. On input, an empty record is treated as a single newline and is not ignored.

After a record has been written to a file, you cannot change its length. If you try to shorten a logical record by writing a new, smaller amount of data into it, the C I/O library will add blank characters until the record is full. Writing more data to a record than it can hold causes your data to be truncated unless you are writing to a standard stream. In this case, your output is split across multiple records. If truncation occurs, LE/VSE C Run-Time raises `SIGIOERR` and sets both `errno` and the error flag.

Note: If you did not explicitly set the `_EDC_ZERO_RECLEN` environment variable when you opened the file, you can update a record that contains a single blank to contain a nonblank character, thereby lengthening the logical record from `'\n'` to `'`*x*`\n'`), where *x* is the nonblank character.

Text (ASA)

LE/VSE C Run-Time treats variable-format ASA text files similarly to the way it treats fixed-format ones. Empty records are always ignored in ASA variable-format files; for a record to be recognized, it must contain at least one character as the ASA print-control character.

For more information about ASA, refer to Chapter 5, "ASA Text Files," on page 25.

Record

Each call to `fwrite()` creates a record that must be less than or equal to the size established by LRECL. If you try to write more than LRECL bytes on one call to `fwrite()`, LE/VSE C Run-Time will truncate your data. LE/VSE C Run-Time never creates empty records using record I/O. On input, empty records are ignored unless you have set the `_EDC_ZERO_RECLEN` environment variable to `Y`. In this case, empty records are treated as records with length `0`.

If your application sets `_EDC_ZERO_RECLEN` to `Y`, bear in mind that `fread()` returns back `0` bytes read, but does not set `errno`, and that both `feof()` and `ferror()` return `0` as well.

Undefined-Format Records

Everything in an undefined-format file is treated as data, including control characters and record boundaries. Blocks in undefined-format records are variable-length; each block is considered a record.

It is impossible to have an empty record. Whatever you specify for LRECL has no effect on your data, but the value of LRECL must be less than or equal to the value you specify for BLKSIZE. Regardless of what you specify, LE/VSE C Run-Time sets LRECL to zero when it creates an undefined-format file.

Reading a file in `U`-format enables you to read an entire block at once.

Record Format (RECFM): You can specify the following formats for undefined-length records:

`U`	Undefined-length
`UA`	Undefined-length, ASA print-control characters
`UM`	Undefined-length, machine control codes

`U`, `UA`, and `UM` formats permit the processing of records that do not conform to `F`- and `V`-formats. The operating system treats each block as a record; your program must perform any additional blocking or deblocking.

You can read any file in `U`-format. This is useful if, for example, you want to see the BDWs and RDWs of a file that you have written in `V`-format.

Mapping C Types to Undefined Format:

Binary

When you are writing to an undefined-format file, binary data fills a block and then begins a new block.

Text (non-ASA)

Record boundaries (that is, block boundaries) are used in the physical file to represent the position of the newline character. You can indicate the end of a record by including a newline or carriage return character in your data. In undefined-format files, LE/VSE C Run-Time treats the carriage return character as if it were a newline. LE/VSE C Run-Time does not write either of these characters to the physical file; instead, it creates a record boundary. When you read the file back, these boundaries are read as newlines.

If a record contains only a newline character, LE/VSE C Run-Time writes a record containing a single blank to the file regardless of the setting of the `_EDC_ZERO_RECLEN` environment variable. Therefore, the string `' \n'` (a single blank followed by a newline character) is treated the same way as `'\n'`: both are written out as a single blank. On input, both are read as `'\n'`. All other blank characters are written and read as you intended. After a record has been written to a file, you cannot change its length. If you try to shorten a logical record by writing a new, smaller amount of data into it, the C I/O library adds blank characters until the record is full. Writing more data to a record than it can hold will cause your data to be truncated unless you are writing to a standard stream. In this case, your output is split across multiple records. If truncation occurs, LE/VSE C Run-Time raises `SIGIOERR` and sets both `errno` and the error flag.

Note: You can update a record that contains a single blank to contain a nonblank character, thereby lengthening the logical record from `'\n'` to `'x\n'`), where *x* is the nonblank character.

Text (ASA)
For a record to be recognized, it must contain at least one character as the ASA print-control character.

For more information about ASA, refer to Chapter 5, “ASA Text Files,” on page 25.

Record
Each call to `fwrite()` creates a record that must be shorter than or equal to the size established by BLKSIZE. If you try to write more than BLKSIZE bytes on one call to `fwrite()`, LE/VSE C Run-Time truncates your data.

The Byte Stream Model for C I/O

The byte stream model differs from the record I/O model. In the byte stream model, a file is just a stream of bytes, with no record boundaries. Newline characters written to the stream appear in the external file.

If the file is opened in binary mode, any newline characters previously written to the file are visible on input. LE/VSE C Run-Time memory file I/O is based on the byte stream model (see Chapter 11, “Performing Memory File I/O Operations,” on page 117 for more information).

Mapping the C Types of I/O to the Byte Stream Model

Binary
In the byte stream model, files opened in binary mode do not contain any record boundaries. Data is written as is to the file.

Text The byte stream model does not support ASA. Newlines, carriage returns, and other control characters are written as is to the file.

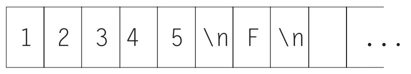
Record
If record I/O is supported by the kind of file you are using, LE/VSE C Run-Time simulates it by treating newline characters as record boundaries. Newlines are not treated as part of the record. A record written out with a newline inside it is not read back as it was written, because LE/VSE C Run-Time treats the newline as a record boundary instead of data.

Memory files do not support record I/O.

As with all other record I/O, you can use only `fread()` and `fwrite()` to read from and write to files. Each call to `fwrite()` inserts a newline in the byte stream; each call to `fread()` strips it off. For example, if you use one `fwrite()` statement to write the string `ABC` and the next to write `DEF`, the byte stream will look like this:

A	B	C	\n	D	E	F	\n		...

There are no limitations on lengthening and shortening records. If you then rewind the file and write new data into it, LE/VSE C Run-Time will replace the old data. For example, if you used the `rewind()` function on the stream in the previous example and then called `fwrite()` to place the string `12345` into it, the stream would look like this:



If you are using files with this model, do not use newline characters in your output. If you do, they will create extra record boundaries. If you are unsure about the data being written or are writing numeric data, use binary instead of text to avoid writing a byte that has the hex value of a newline.

Chapter 3. Opening Files

This chapter describes how to open I/O files. You can open files using the standard C `fopen()` and `freopen()` library functions. The formats of these functions are:

C library functions

```
FILE    *fopen(const char *filename,
               const char *mode);

FILE    *freopen(const char *filename,
               const char *mode, FILE *stream);
```

The C library functions are described in more detail in *LE/VSE C Run-Time Library Reference*.

Depending on the type of file being opened, the *filename* that you specify on these function calls can be a system logical unit, a programmer logical unit, a `DLBL`-name, a `TLBL`-name, a member of a VSE/Librarian sublibrary, a SAM file, or a VSAM data set.

In the following, whenever `fopen()` function call parameters are discussed, it is implied that these are applicable to both the `fopen()` and the `freopen()` function calls, unless otherwise stated.

Categories of I/O

The following table lists the categories of I/O that LE/VSE C Run-Time supports and points to the page where each category is described.

Table 9. Kinds of I/O Supported by LE/VSE C Run-Time

Type of I/O	Suggested uses and supported devices	Model	Page
SAM I/O	Used for dealing with the following kinds of files: • Sequential disk files (SAM and VSAM-managed SAM) • Tapes • Printers • Punch files • Card reader files	Record	47
VSE/Librarian I/O	Used for working with members of VSE/Librarian sublibraries.	Record	73
VSAM I/O	Used for working with VSAM data sets. Supports direct access to records by key, relative record number, or relative byte address. Supports entry-sequenced, relative record, and key-sequenced data sets.	Record	79
Memory Files	Used for applications requiring temporary I/O files without the overhead of system data sets. Fast and efficient.	Byte stream	117
CICS Data Queues	Used under the Customer Information Control System (CICS). CICS data queues are automatically selected under CICS for the standard streams `stdout` and `stderr`. The CICS I/O commands are supported through the Command Level interface. The standard stream `stdin` is treated as an empty file under CICS.	Record	127

Table 9. Kinds of I/O Supported by LE/VSE C Run-Time (continued)

Type of I/O	Suggested uses and supported devices	Model	Page
LE/VSE Message File	Used when you are running with LE/VSE. The message file is automatically selected for `stderr` under LE/VSE.	Record	129

The following table lists the environments that LE/VSE C Run-Time supports, and which categories of I/O work in which environment.

Table 10. I/O Categories and Environments Supported

	Environment	
Type of I/O	**VSE batch**	**CICS**
SAM I/O	Yes	No
VSE/Librarian I/O	Yes	No
VSAM I/O	Yes	No
Memory Files	Yes	Yes
CICS Data Queues	No	Yes
LE/VSE Message File	Yes	No

Specifying What Kind of File to Use

SAM Files

SAM files include sequential disk files (including SAM ESDS files, but excluding other VSAM data sets), as well as non-disk files, such as non-VSAM tape files, printer files, etc.

VSE/Librarian Members

Members of VSE/Librarian sublibraries are identified by their member name and type. The name of the library and sublibrary is optional.

VSAM Data Sets

LE/VSE C Run-Time recognizes a VSAM data set if the file exists and has been defined as a VSAM cluster before the call to `fopen()`. Under VSE, `fopen()` may be called with either the data set name or a referencing `DLBL/TLBL`-name.

Memory Files

You can use regular memory files on all the systems that LE/VSE C Run-Time supports. To create one, specify `type=memory` on the `fopen()` call that creates the file. A memory file, once created, exists until either of the following happens:
- You explicitly remove it with `remove()` or `clrmemf()`.
- The root program is terminated.

While a memory file exists, you can just use another `fopen()` that specifies the memory file's name; you do not have to specify `type=memory`. For example:

EDCXGOF1

```
 /* EDCXGOF1
    This example shows how fopen() may be used with memory files
  */

#include <stdio.h>
char text[3], *result;
FILE * fp;

int main(void)
   {
   fp = fopen("a.b", "w, type=memory");  /* Opens a memory file */
   fprintf(fp, "%d\n",10);               /* Writes to the file  */
   fclose(fp);                           /* Closes the file     */
   fp = fopen("a.b", "r");               /* Reopens the same    */
                                         /*  file (already      */
                                         /*  a memory file)     */
   if ((result=fgets(text,3,fp)) !=NULL) /* Retrieves results   */
      printf("value retrieved is %s\n",result);
   fclose(fp);                           /* Closes the file     */

   return(0);
   }
```

Figure 3. Example of Using `fopen()` with Memory Files

A valid memory *filename* will match current file restrictions on a real file.

CICS Data Queues

A CICS transient data queue is a pathway to a single predefined destination. The destination can be a `DLBL`-name, another transient data queue, a VSAM data set, a terminal, or another CICS environment. The CICS system administrator defines the queues that are active during execution of CICS. (For DCT customization, see *LE/VSE Customization Guide*.) All users who direct data to a given queue will be placing data in the same location, in order of occurrence.

You cannot use `fopen()` to specify this kind of I/O. It is the category selected automatically when you call any ANSI functions that reference `stdout` and `stderr` under CICS. If you reference either of these in a C program under CICS, LE/VSE C Run-Time attempts to open the CESO (`stdout`) or CESE (`stderr`) queue. If you want to write to any other queue, you should use the CICS-provided interface.

LE/VSE Message File

The LE/VSE message file is managed by LE/VSE and may not be directly opened or closed with `fopen()`, `freopen()` or `fclose()` within a C application. In LE/VSE, output from `stderr` is directed to the LE/VSE message file by default. You can use `freopen()` and `fclose()` to manage `stderr`, or you can redirect it to another destination. There are application writer interfaces (AWIs) that enable you to access the LE/VSE message file directly. These are documented in *LE/VSE Programming Guide*.

See Chapter 13, “Performing LE/VSE Message File Operations,” on page 129 for more information on LE/VSE message files.

How to Specify RECFM, LRECL, and BLKSIZE

The values of the RECFM, LRECL, and BLKSIZE attributes are significant under VSE. When you open a non-VSAM file, LE/VSE C Run-Time searches for these file attributes in the following places:

1. The `fopen()` or `freopen()` function call that opens the file
2. The `DLBL` statement if one exists (described in "File Definition Statements" on page 21)
3. The VSAM catalog (predefined SAM ESDS files only—see also "VSAM Catalog Information for SAM ESDS Files" on page 19.)
4. The default values for `fopen()` or `freopen()`

File attributes for VSAM data sets are determined from the VSAM catalog only.

Note: RECFM, LRECL and BLKSIZE can be provided by the Vendor Exit through third party products.

When you call `fopen()` and specify a write mode (`w`, `wb`, `w+`, `wb+`, or `w+b`) for an existing non-VSAM file, LE/VSE C Run-Time uses the supplied or default values for `fopen()`, not the values for the existing file. These defaults are listed in Table 11 on page 20.

Certain categories of I/O may ignore or simulate some attributes such as BLKSIZE or RECFM that are not physically supported on the device. Table 9 on page 15 lists all the categories of I/O that LE/VSE C Run-Time supports and directs you to where you can find more information about them.

Specifying RECFM

The record format can only be specified in the `recfm` parameter on the call to the `fopen()` library function for non-VSAM files. However, for VSAM data sets and predefined SAM ESDS files, RECFM is determined from the VSAM catalog as the value set by the IDCAMS `DEFINE` command (for SAM ESDS files, see "VSAM Catalog Information for SAM ESDS Files" on page 19).

If you are creating a file and you do not select a record format, LE/VSE C Run-Time might use a default depending on the file type. See "`fopen()` Defaults" on page 19 for additional information.

Specifying LRECL

The logical record length can only be specified in the `lrecl` parameter on the call to the `fopen()` library function for non-VSAM files. However, for VSAM data sets and predefined SAM ESDS files, LRECL is determined from the VSAM catalog as the value set by the IDCAMS `DEFINE` command (for SAM ESDS files, see "VSAM Catalog Information for SAM ESDS Files" on page 19).

If you are creating a file and you do not select a logical record length, LE/VSE C Run-Time uses a default. See "`fopen()` Defaults" on page 19 for details on how defaults are determined.

Specifying BLKSIZE

You can specify the block size for non-VSAM data sets in:

- The `BLKSIZE` parameter of the JCL `DLBL` statement for non-FBA SAM files using `noseek`
- The `blksize` parameter on a call to the `fopen()` or `freopen()` library function

For VSAM data sets, however, BLKSIZE is determined from the VSAM catalog only as the value set by the IDCAMS `DEFINE` command.

If you are creating a file and you do not select a block size, LE/VSE C Run-Time uses a default as described in “`fopen()` Defaults.”

VSAM Catalog Information for SAM ESDS Files

The information contained in the VSAM catalog for SAM ESDS files is sometimes different than expected:

- To create a SAM ESDS file with VSAM catalog information that correctly reflects the `lrecl`, `blksize`, and `recfm` parameters from `fopen()`, you should first remove the file with `remove()` and then open the file with the `noseek` parameter.
- When a SAM ESDS file is created and `noseek` has not been specified, LE/VSE C Run-Time will provide the expected behaviour for the record format specified on `fopen()`. However, the internal VSAM access will be performed using `recfm=U`. This causes RECFM=U and the BLKSIZE to be recorded in the VSAM catalog. Subsequent opens for read can not rely on the catalog information if the original file was opened with fixed or variable record formats.
- If an existing non-empty SAM ESDS file is opened for write, the catalog LRECL, BLKSIZE, and RECFM information is updated only if the block size specified on `fopen()` is greater than that of the existing file. This is a VSE/VSAM restriction.

`fopen()` Defaults

The file attributes RECFM, LRECL, and BLKSIZE for VSAM data sets are determined from the VSAM catalog only. Any specification of these attributes on the `fopen()` function call is ignored by LE/VSE C Run-Time.

For memory files, no defaults are discussed as file attributes are not applicable to this type of file.

You cannot specify a file attribute more than once on a call to `fopen()`. If you do, the function call fails. If a file is opened for append or update and the file attributes specified on the call to `fopen()` differ from the actual file attributes of an existing file, `fopen()` might fail or file data might be corrupted.

In calls to `fopen()`, the `lrecl`, `blksize`, and `recfm` parameters are optional. (If you are opening a file for read or append, any attributes that you specify must match the existing attributes in the VSAM catalog if these are available, or if they are not, those used when the file was created.)

If you do not specify file attributes for `fopen()` you get the following defaults:

RECFM Defaults

If `recfm` is not specified in the call to `fopen()` for system logical units (for example, `SYSLST`), or for card and printer devices, it will default to `recfm=F`.

If `recfm` is not specified in the call to `fopen()` for VSE/Librarian members, and the member does not exist or it is being opened for write, it will default to `recfm=FB`.

No default `recfm` is supported for file types other than the above.

LRECL and BLKSIZE Defaults

If `blksize` is not specified in the call to `fopen()` for VSE/Librarian members, it will default to `blksize=4000`. For VSE/Librarian members, `blksize` specifies the logical blocking only and has no impact on the physical blocking of the data. Larger blocksizes will result in a better performance for sequential processing.

LRECL and BLKSIZE defaults for system logical units, and for card and printer devices, differ from other file types. For example, the default LRECL may be obtained from the `blksize` parameter. Refer to "Parameters Supported by File Type" on page 52 for additional information about LRECL and BLKSIZE defaults for system logical units, and for card and printer devices.

For other file types (SAM disk and tape files, including SAM ESDS files), the following defaults apply:

Table 11. `fopen()` Defaults for LRECL and BLKSIZE

lrecl specified?	blksize specified?	RECFM	LRECL	BLKSIZE
Note: For exceptions to this table, see above.				
no	no	All `F`	80	80
		All `FB`	80	maximum integral multiple of 80 less than or equal to 6080 if disk or tape; 80 otherwise
		All `V` or all `VB`	1028 if disk or tape	6144 if disk or tape and blocked; 1032 if disk or tape and unblocked
		All `U`	0	6144
yes	no	All `F`	*lrecl*	*lrecl*
		All `FB`	*lrecl*	maximum integral multiple of *lrecl* less than or equal to 6144 if disk or tape and *lrecl* less than 6144; *lrecl* if disk or tape and *lrecl* greater than or equal to 6144
		All `V`	*lrecl*	*lrecl + 4*
		All `VB`	*lrecl*	6144 if disk or tape and *lrecl* less than 6140; *lrecl*+4 if disk or tape and *lrecl* greater than or equal to 6140
		All `U`	0	*lrecl*
no	yes	All `F` or all `FB`	*blksize*	*blksize*
		All `V` or all `VB`	minimum of 1028 or *blksize - 4* if *blksize* greater than 0; 0 otherwise	*blksize*
		All `U`	0	*blksize*
Note: "All" includes the standard (`S`) specifier for fixed formats, the spanned (`S`) specifier for variable formats, the ASA print-control character (`A`) specifier, and the machine control code (`M`) specifier.				

It is possible to have conflicting LRECL and BLKSIZE attributes. The restrictions are:

- For a `V` file, LRECL must be greater than 4 bytes and must be at least 4 bytes smaller than BLKSIZE.
- For an `F` file, LRECL must be equal to BLKSIZE, and must be at least 1 for non-tape files and 18 for tape files.
- For an `FB` file, BLKSIZE must be an integer multiple of LRECL.
- For a `U` file, LRECL must be less than or equal to BLKSIZE and must be greater than or equal to 0. BLKSIZE must be at least 1.
- In spanned files, LRECL and BLKSIZE must both be greater than 4.

The maximum LRECL supported is 32760. The maximum BLKSIZE supported is 65528. Use of a BLKSIZE greater than 32760 requires the appropriate hardware-device support. To determine the maximum LRECL and BLKSIZE values for the various file types and devices available on your operating system, refer to the publications listed in “Where to Find More Information” on page xxi.

LE/VSE C Run-Time cannot always check to ensure that the blocks read from a disk file are not larger than the block size specified on the call to `fopen()`. If the block size specified on the call to `fopen()` is smaller than the block size specified when the file was created, storage will be overwritten. No message will be returned. Therefore, you must ensure that the block size specified on the call to `fopen()` matches the block size specified when the file was created.

File Definition Statements

The `DLBL`, `TLBL`, `ASSGN` and `EXTENT` job control statements are used to define a file to the operating system, and is a request to the operating system for the allocation of input/output resources. Each job step must include a `DLBL`, `TLBL` and/or `ASSGN` statement for each file that is opened by `DLBL`/`TLBL`-name, system logical unit or programmer logical unit.

Your *System Control Statements* manual describes the syntax of job control statements. To define a file to the operating system, the following JCL statements are used:

- For a disk device, use the `DLBL` JCL statement. The *file-ID* parameter specifies the name of the file. BLKSIZE can be specified for a SAM file. (See “How to Specify RECFM, LRECL, and BLKSIZE” on page 18 for restrictions regarding the `BLKSIZE` parameter.) The amount of space can be specified for a SAM ESDS file using the `RECORDS` and `RECSIZE` parameters.

  The `EXTENT` and `ASSGN` statements specify the volume(s) on which a non-VSAM file will reside.
- For a magnetic-tape device, use the `TLBL` and `ASSGN` JCL statements for a *labeled* tape.

  For an *unlabeled* tape, use the `ASSGN` statement. The `TLBL` statement must not be present.
- For a unit-record device, use the `ASSGN` JCL statement.

The `DLBL`, `TLBL` and `ASSGN` statements enable you to write C source programs that are independent of the files and input/output devices they will use. You can modify the parameters of a file or process different files without recompiling your program.

Chapter 4. Buffering of C Streams

This chapter describes buffering modes used by LE/VSE C Run-Time, library functions available to control buffering and methods of flushing buffers.

LE/VSE C Run-Time uses buffers to map C I/O to system-level I/O. When LE/VSE C Run-Time performs I/O operations, it uses one of the following buffering modes:

Line buffering
: Characters are transmitted to the system as a block when a newline character is encountered. Line buffering is meaningful only for text streams.

Full buffering
: Characters are transmitted to the system as a block when a buffer is filled.

No buffering
: Characters are transmitted to the system as they are written to a memory file.

The buffer mode affects the way the buffer is flushed. You can use the `setvbuf()` and `setbuf()` library functions to control buffering, but you cannot change the buffering mode after an I/O operation has used the buffer, as all read, write, and reposition operations do. In some circumstances, acquiring a position alters the contents of the buffer. It is strongly recommended that you only use `setbuf()` and `setvbuf()` before *any* I/O, to conform with ANSI, and to avoid any dependency on the current implementation. If you use `setvbuf()`, LE/VSE C Run-Time may or may not accept your buffer for its internal use.

Full buffering is the default except in the following cases:
- If you are running under CICS, LE/VSE C Run-Time also uses line buffering.
- `stderr` is line-buffered by default.
- If you are using a memory file, LE/VSE C Run-Time does not use any buffering.

For record I/O files, buffering is meaningful only for blocked files. For unblocked files, the buffer is full after every write and is therefore written immediately, leaving nothing to flush. For blocked files, however, the buffer can contain one or more records that have not been flushed and that require a flush operation for them to go to the system.

You can flush buffers to the system in several different ways.
- If you are using full buffering, LE/VSE C Run-Time automatically flushes a buffer when it is filled.
- If you are using line buffering for a text file, LE/VSE C Run-Time flushes a buffer when you complete it with a control character. Specifying line buffering for a record I/O or binary file has no effect; LE/VSE C Run-Time treats the file as if you had specified full buffering.
- LE/VSE C Run-Time flushes buffers to the system when you close a file or end a program.
- LE/VSE C Run-Time flushes buffers to the system when you call the `fflush()` library function, with the following restrictions:

  - A file opened in text mode does not flush data if a record has not been completed with a newline.
  - A file opened in fixed format does not flush incomplete records to the file.
  - An `FBS` file does not flush out a short block unless it is a disk file opened without the `noseek` parameter.
- All streams are flushed across `system()` calls.

If you are reading a record that another user is writing to at the same time, you can see the new data if you call `fflush()` to refresh the contents of the input buffer.

Note: This is not supported for VSAM data sets.

You may not see output if a program that is using input and output fails, and the error handling routines cannot close all the open files.

Chapter 5. ASA Text Files

This chapter describes ASA text files, the print-control characters used in ASA files, how LE/VSE C Run-Time translates the print-control characters, and how LE/VSE C Run-Time treats ASA files during input and output. The first column of each record in an ASA file contains a print-control character (' ', '0', '-', '1', or '+') when it appears in the external medium.

LE/VSE C Run-Time translates print-control characters in ASA files opened for text processing (`r`, `w`, `a`, `r+`, `w+`, or `a+`). On input, LE/VSE C Run-Time translates ASA characters to sequences of control characters, as shown in Table 12. On output, LE/VSE C Run-Time performs the reverse translation. The following sequences of control characters are translated, and the resultant ASA character becomes the first character of the following record:

Table 12. C Control to ASA Characters Translation Table

C Control Character Sequence	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines
\n\n\n	'-'	skip three lines
\f	'1'	new page
\r	'+'	overstrike

If you are writing to the first record or byte of the file and the output data does not start with a translatable sequence of C control characters, the ' ' ASA print-control character is written to the file before the specified data.

LE/VSE C Run-Time does not translate or verify control characters when you open an ASA file for binary or record I/O.

Example of Writing to an ASA File

EDCXGAS1

```
 /* EDCXGAS1
    This example shows how to write to an ASA file
  */

#include <stdio.h>
#define MAX_LEN 80

int main(void) {
   FILE *fp;
   char s[MAX_LEN+1];

   fp = fopen("asa.file", "w,recfm=fba");
   if (fp != NULL) {
      fputs("\n\nabcdef\f\r345\n\n", fp);
      fputs("\n\n9034\n", fp);
      fclose(fp);
   }
   fp = fopen("asa.file", "rb,recfm=fb,type=record");
   if (fp != NULL) {
      fread(s, 1, MAX_LEN, fp);
      while (!feof(fp)) {
         printf("record=%s\n", s);
         fread(s, 1, MAX_LEN, fp);
      }
      fclose(fp);
   }
}
```

Figure 4. ASA Example

This program writes five records to the file `asa.file`, as follows:

```
record=0abcdef
record=1
record=+345
record=-
record= 9034
```

Note that the last record is `" 9034"`. The last single '\n' does not create a record with a single control character (' '). If this same file is opened for read, and the `getc()` function is called to read the file 1 byte at a time, the same characters as those that were written out by `fputs()` in the first program are read.

ASA files are treated as follows:

- If the first record written does not begin with a control character, then a single newline is written and then followed by data; that is, the ASA character defaults to a space when none is specified.
- In ASA files, control characters are treated the same way that they are treated in other text files, with the following exceptions:

  '\f' — form feed
  : Defines a record boundary and determines the ASA character of the following record. Refer to Table 12 on page 25.

  '\n' — newline
  : Does either of these:

  - Defines a record boundary and determines the ASA character of the following record (see translation table above).
  - Modifies the preceding ASA character if the current position is directly after an ASA character of ' ' or '0' (see translation table above).

'\r' — carriage return
: Defines a record boundary and determines the ASA character of the following record (see translation table above).

- Records are terminated by writing a newline ('\n'), carriage return ('\r'), or form feed ('\f') character.
- An ASA character can be updated to any other ASA character.

  Updates made to any of the C control characters that make up an ASA cause the ASA character to change.

  If the file is positioned directly after a ' ' or '0' ASA character, writing a '\n' character changes the ASA character to a '0' or '-' respectively. However, if the ASA character is a '-', '1' or '+', the '\n' truncates the record (that is, it adds blank padding to the end of the record), and causes the following record's ASA character to be written as a ' '. Writing a '\f' or '\r' terminates the record and start a new one, but writing a normal data character simply overwrites the first data character of the record.
- You cannot overwrite the ASA character with a normal data character. The position at the start of a record (at the ASA character) is the logical end of the previous record. If you write normal data there, you are writing to the end of the previous record. LE/VSE C Run-Time truncates data for the following files, except when they are standard streams:
  - Variable-format files
  - Undefined-format files
  - Fixed-format files in which the previous record is full of data

  When truncation occurs, LE/VSE C Run-Time raises SIGIOERR and sets both errno and the error flag.
- Even when you update an ASA print-control character, seeking to a previously recorded position still succeeds. If the recorded position was at a control character that no longer exists (because of an update), the reposition is to the next character. Often, this is the first data character of the record. For example, if you have the following string:

```
\n\n\nHELLO WORLD
     ↑
 x = ftell()
```

  you have saved the position of the third newline. If you then update the ASA character to a form feed ('\f'), the logical ASA position x no longer exists.

```
   \fHELLO WORLD
```

  If you call fseek() with the logical position x, it repositions to the next valid character, which is the letter 'H'.

```
\fHELLO WORLD
  ↑
 fseek() to pos x
```

- If you try to shorten a record when you are updating it, LE/VSE C Run-Time adds enough blank padding to fill the record.
- The ASA character can represent up to three newlines, which can increase the logical record length by 1 or 2 bytes.
- Extending a fixed logical record on update implies that the logical end of the line follows the last written non-blank character.

- If an undefined text record is updated, the length of the physical records does not change. If the replacement record is:
  - *Longer* - data characters beyond the record boundary are truncated. At the point of truncation, the User error flag is set and `SIGIOERR` is raised (if the signal is not set up to be ignored). Truncation continues until you do one of these:
    1. write a newline character, carriage return, or form feed to complete the current record
    2. close the file explicitly or implicitly at termination
    3. reposition to another position in the file.
  - *Shorter* - the blank character is used to overwrite the rest of the record.
- If you close an ASA file that has a newline as its last character, LE/VSE C Run-Time does not write the newline to the physical file. The next time you read from the file or update it, LE/VSE C Run-Time returns the newline to the end of the file. An exception to this rule happens when you write only a newline to a new file. In this case, LE/VSE C Run-Time does not truncate the newline; it writes a single blank to the file. On input, however, you will read two newlines.
- Using ASA format to read a file that contains zero-length records results in undefined behavior.
- You may have trouble updating a file if two ASA characters are next to each other in the file. For example, if there is a single-byte record (containing only an ASA character) immediately followed by the ASA character of the next record, you are positioned at or within the first ASA character. If you then write a sequence of '\n' characters intended to update both ASA characters, the '\n's will be absorbed by the first ASA character before overflowing to the next record. This absorption may affect the crossing of record boundaries and cause truncation or corruption of data.

  At least one normal intervening data character (for example, a space) is required between '\n' and '\n' to differentiate record boundaries.

  Note: Be careful when you update an ASA file with data containing more than one consecutive newline—the result of the update depends on how the original ASA records were structured.
- If you are writing data to a non-blocked file without intervening flush or reposition requests, each record is written to the system on completion (that is, when a `'\n'`, `'\r'` or `'\f'` character is written or when the file is closed).

  If you are writing data to a blocked file without intervening flush or reposition requests, and the file is opened in full buffering mode, the block is written to the system on completion of the record that fills the block. If the blocked file is line buffered, each record is written to the system on completion.

  If you are writing data to a spanned file without intervening flush or reposition requests, and the record spans multiple blocks, each block is written to the system once it is full and the user writes an additional byte of data.
- If a flush occurs while an ASA character indicating more than one newline is being updated, the remaining newlines will be discarded and a read will continue at the first data character. For example, if '\n\n\n' is updated to be '\n\n' and a flush occurs, then a '`0`' will be written out in the ASA character position.

Chapter 6. LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)

The number of characters in some languages such as Japanese or Korean is larger than 256, the number of distinct values that can be encoded in a single byte. The characters in such languages are represented in computers by a sequence of bytes, and are called multibyte characters. This chapter explains how the LE/VSE C Run-Time supports multibyte characters.

LE/VSE C Run-Time supports the IBM EBCDIC encoding of multibyte characters, in which each natural language character is uniquely represented by one to four bytes. The number of bytes that encode a single character depend on the *global shift-state information.* If a stream is in initial shift state, one multibyte character is represented by a sequence of bytes that has the following characteristics:
- It starts with the byte containing the shift-out (`0x0e`) character.
- The shift-out character is followed by 2 bytes that encode the decimal value of the character.
- These bytes may be followed by a byte containing the shift-in (`0x0f`) character.

If the sequence of bytes ends with the shift-in character, the state remains initial, making this sequence represent a 4-byte multibyte character. Multibyte characters of various lengths can be normalized by the set of LE/VSE C Run-Time library functions and encoded in units of one length. Such normalized characters are called wide characters; in LE/VSE C Run-Time they are represented by two bytes. Conversions between multibyte format and wide character format can be performed by string conversion functions such as `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well by the family of the wide character I/O functions. `MB_CUR_MAX` is defined in the `stdlib.h` header file. Depending on its value, either of the following happens:
- When `MB_CUR_MAX` is 1, all bytes are considered single-byte characters; shift-out and shift-in characters are treated as data as well.
- When `MB_CUR_MAX` is 4:
  - On input, the wide character I/O functions read the multibyte character from the streams, and convert them to the wide characters.
  - On output, they convert wide characters to multibyte characters and write them to the output streams.

Both binary and text streams have *orientation*. Streams opened with `type=record` do not. There are three possible orientations of a stream:

Non-oriented
: A stream that has been associated with an open file before any operation other than `setbuf()` or `setvbuf()` is performed. Subsequent operations on a non-oriented stream change the orientation of the stream. You can use the `setbuf()` and `setvbuf()` functions only on a non-oriented stream. When you use these functions, the stream remains non-oriented. When you perform one of the wide character input/output operations on a non-oriented stream, the stream becomes *wide-oriented*. When you perform one of the byte input/output operations on a non-oriented stream, the stream becomes *byte-oriented*.

Wide-oriented
: A stream on which any wide character input/output functions are

guaranteed to operate correctly. Conceptually, wide-oriented streams are sequences of wide characters. The external file associated with a wide-oriented stream is a sequence of *multibyte* characters. Using byte I/O functions on a wide-oriented stream results in undefined behavior. A stream opened for record I/O cannot be wide-oriented.

Byte-oriented
A stream on which any byte input/output functions are guaranteed to operate properly. Using wide character I/O functions on a byte input/output stream results in undefined behavior. Byte-oriented streams have minimal support for multibyte characters.

Calls to the `clearerr()`, `feof()`, `ferror()`, `fflush()`, `fgetpos()`, or `ftell()` functions do not change the orientation.

Once you have established a stream's orientation, the only way to change it is to make a successful call to the `freopen()` function, which removes a stream's orientation.

The `wchar.h` header file declares the `WEOF` macro and the functions that support wide character input and output. The macro expands to a constant expression of type `wint_t`. Certain functions return `WEOF` type when the end-of-file is reached on the stream.

Note: The behavior of the wide character I/O functions is affected by the LC_CTYPE category of the current locale, and the setting of `MB_CUR_MAX`. Wide-character input and output should be performed under the same LC_CTYPE setting. If you change the setting between when you read from a file and when you write to it, or vice versa, you may get undefined behavior. If you change it back to the original setting, however, you will get the behavior that is documented. See the introduction of this chapter for a discussion of the effects of `MB_CUR_MAX`.

Opening Files

You can use the `fopen()` or `freopen()` library functions to open I/O files that contain wide characters. You do not need to specify any special parameters on these functions for wide character I/O.

Reading Streams and Files

Wide character input functions read multibyte characters from the stream and convert them to wide characters. The conversion process is performed in the same way that the `mbrtowc()` function performs conversions.

The following LE/VSE C Run-Time library functions support wide character input:
- `fgetwc()`
- `fgetws()`
- `getwc()`
- `getwchar()`
- `swscanf()`

In addition, the following byte-oriented functions support handling multibyte characters by providing conversion specifiers to handle the `wchar_t` data type:
- `scanf()`
- `fscanf()`

- `sscanf()`

All other byte-oriented input functions treat input as single-byte.

For a detailed description of unformatted and formatted I/O functions, refer to *LE/VSE C Run-Time Library Reference*.

The wide-character input/output functions perform the conversions between multibyte and single-byte states that the `mbrtowc()` and `wcrtomb()` functions do.

When you are using wide-oriented input functions, multibyte characters are converted to wide characters according to the current shift state. Invalid double-byte character sequences cause conversion errors on input. As LE/VSE C Run-Time uses wide-oriented functions to read a stream, it updates the shift state when it encounters shift-out and shift-in characters. Wide-oriented functions always read complete multibyte characters. Byte-oriented functions do not check for complete multibyte characters, nor do they maintain information about the shift state. Therefore, they are not to be used with wide character I/O.

For binary streams, no validation is performed to ensure that records start or end in initial shift state. For text streams, however, all records must start and end in initial shift state.

Writing Streams and Files

Wide character output functions convert wide characters to multibyte characters and write the result to the stream. The conversion process is performed in the same way that the `wcrtomb()` function performs conversions.

The following LE/VSE C Run-Time functions support wide character output:
- `fputwc()`
- `fputws()`
- `swprintf()`
- `vswprintf()`
- `putwc()`
- `putwchar()`

In addition, the following byte-oriented functions support handling multibyte characters by providing conversion specifiers to handle the `wchar_t` data type:
- `printf()`
- `fprintf()`
- `sprintf()`

All other output functions do not support the `wchar_t` data type. However, all of the output functions support multibyte character output for text streams if `MB_CUR_MAX` is 4.

For a detailed description of unformatted and formatted I/O functions, refer to *LE/VSE C Run-Time Library Reference*.

Writing Text Streams

When you are using wide-oriented output functions, wide characters are converted to multibyte characters. For text streams, all records must start and end in initial shift state. The wide-character functions add shift-out and shift-in characters as they are needed. When the file is closed, a shift-in character may be added to complete the file in initial shift state.

When you are using byte-oriented functions to write out multibyte data, LE/VSE C Run-Time starts each record in initial shift state and makes sure you complete each record in initial shift state before moving to the next record. When a string starts with a shift-out, all data written is treated as multibyte, not single-byte. This means that you cannot write a single-byte control character (such as a newline) until you complete the multibyte string with a shift-in character.

Attempting to write a second shift-out character before a shift-in is not allowed. LE/VSE C Run-Time truncates the second shift-out and raises `SIGIOERR` if `SIGIOERR` is not set to `SIG_IGN`.

When you do write a shift-in character to an incomplete multibyte character, LE/VSE C Run-Time completes the multibyte character with a padding character (`0xfe`) before it writes the shift-in. The padding character is not counted as an output character in the total returned by the output function; you will never get a return code indicating that you wrote more characters than you provided. If LE/VSE C Run-Time adds a padding character, however, it does raise `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`.

Control characters written before the shift-in are treated as multibyte data and are not interpreted or validated.

When you close the file, LE/VSE C Run-Time ensures that the file ends in initial shift state. This may require adding a shift-in and possibly a padding character to complete the last multibyte character, if it is not already complete. If padding is needed in this case, LE/VSE C Run-Time does not raise `SIGIOERR`.

Multibyte characters are never split across record boundaries. In addition, all records end and start in initial shift state. When a shift-out is written to the file, either directly or indirectly by wide-oriented functions, LE/VSE C Run-Time calculates the maximum number of complete multibyte characters that can be contained in the record with the accompanying shift-in. If multibyte output (including any required shift-out and shift-in characters) does not fit within the current record, the behavior depends on what type of file it is (a memory file has no record boundaries and so never has this particular problem). For a standard stream, data is wrapped from one record to the next. Shift characters may be added to ensure that the first record ends in initial shift state and that the second record starts in the required shift state.

For files that are not standard streams or memory files, any attempt to write data that does not fit into the current record results in data truncation. In such a case, the output function returns an error code, raises `SIGIOERR`, and sets `errno` and the error flag. Truncation continues until initial state is reached and a newline is written to the file. An entire multibyte stream may be truncated, including the shift-out and shift-in, if there are not at least two bytes in the record. For a wide-oriented stream, truncation stops when a `wchar_t` newline character is written out.

Updating a wide-oriented file or a file containing multibyte characters is strongly discouraged, because your update may overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The data after the shift-out is meaningless when it is treated in initial shift state. Appending new data to the end of the file is safe for a wide-oriented file.

Writing Binary Streams

When you are using wide-oriented output functions, wide characters are converted to multibyte characters. No validation is performed to ensure that records start or end in initial shift state. When the file is closed, any appends are completed with a shift-in character, if it is needed to end the stream in initial shift state. If you are updating a record when the stream is closed, the stream is flushed. See "Flushing Buffers" for more information.

Byte-oriented output functions do not interpret binary data. If you use them for writing multibyte data, ensure that your data is correct and ends in initial shift state.

Updating a wide-oriented file or a file containing multibyte characters is strongly discouraged (see "Writing Text Streams" on page 31 for details.)

If you update a record after you call `fgetpos()`, the shift state may change. Using the `fpos_t` value with the `fsetpos()` function may cause the shift state to be set incorrectly.

Flushing Buffers

You can use the library function `fflush()` to flush streams to the system. For more information about `fflush()`, see *LE/VSE C Run-Time Library Reference*.

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of stream. If you call one LE/VSE C Run-Time program from another LE/VSE C Run-Time program by using the ANSI `system()` function, all open streams are flushed before control is passed to the callee.

Flushing Text Streams

When you call `fflush()` after updating a text stream, `fflush()` calculates your current shift state. If you are not in initial shift state, LE/VSE C Run-Time looks forward in the record to see whether a shift-in character occurs before the end of the record or any shift-out. If not, LE/VSE C Run-Time adds a shift-in to the data if it will not overwrite a shift-out character. The shift-in is placed such that there are complete multibyte characters between it and the shift-out that took the data out of initial state. LE/VSE C Run-Time may accomplish this by skipping over the next byte in order to leave an even number of bytes between the shift-out and the added shift-in.

Updating a wide-oriented or byte-oriented multibyte stream is strongly discouraged. In a byte-oriented stream, you may have written only half of a multibyte character when you call `fflush()`. In such a case, LE/VSE C Run-Time adds a padding byte before the shift-out. For both wide-oriented and byte-oriented streams, the addition of any shift or padding character does not move the current file position.

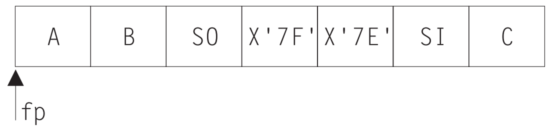
Calling `fflush()` has no effect on the current record when you are writing new data to a wide-oriented or byte-oriented multibyte stream, because the record is incomplete.

Flushing Binary Streams

In a wide-oriented stream, calling `fflush()` causes LE/VSE C Run-Time to add a shift-in character if the stream does not already end in initial shift state. In a byte-oriented stream, calling `fflush()` causes no special behavior beyond what a call to `fflush()` usually does.

`ungetwc()` Considerations

`ungetwc()` pushes wide characters back onto the input stream for binary and text files. You can use it to push one wide character onto the `ungetwc()` buffer. Never use `ungetc()` on a wide-oriented file. After you call `ungetwc()`, calling `fflush()` backs up the file position by one wide character and clears the pushed-back wide character from the stream. Backing up by one wide character skips over shift characters and backs up to the start of the previous character (whether single-byte or double-byte). For text files, LE/VSE C Run-Time counts the newlines added to the records as single-byte characters when it calculates the file position. For example, if you have the following stream:
you can run the following code fragment:



```
fgetwc(fp);      /* Returns X'00C1' (the hexadecimal         */
                 /*    wchar representation of A)            */
fgetwc(fp);      /* Returns X'00C2' (the hexadecimal         */
                 /*    wchar representation of B)            */
fgetwc(fp);      /* Returns X'7FFE' (the hexadecimal         */
                 /*    wchar representation of the DBCS      */
                 /*    character) between the SO and SI      */
                 /*    characters; leaves file position at C  */
ungetwc('Z',fp); /* Logically inserts Z before SI character   */
fflush(fp);      /* Backs up one wchar, leaving position at   */
                 /*    beginning of X'7FFE' DBCS char        */
                 /*    and DBCS state in double-byte mode;    */
                 /*    clears Z from the logical stream      */
```

Figure 5. `ungetwc()` Example

You can set the _EDC_COMPAT environment variable before you open the file, so that `fflush()` ignores any character pushed back with `ungetwc()` or `ungetc()`, and leaves the file position where it was when `ungetwc()` or `ungetc()` was first issued. Any characters pushed back are still cleared. For more information about _EDC_COMPAT, see Chapter 21, "Using Environment Variables," on page 219.

Setting Positions within Files

The following conditions apply to text streams and binary streams.

Repositioning within Text Streams

When you use the `fseek()` or `fsetpos()` function to reposition within files, LE/VSE C Run-Time recalculates the shift state.

If you update a record after a successful call to the `fseek()` function or the `fsetpos()` function, a partial multibyte character can be overwritten. Calling a wide character function for data after the written character can result in undefined behavior.

Use the `fseek()` or `fsetpos()` functions to reposition only to the start of a multibyte character. If you reposition to the middle of a multibyte character, undefined behavior can occur.

Repositioning within Binary Streams

When you are working with a wide-oriented file, keep in mind the state of the file position that you are repositioning to. If you call `ftell()`, you can seek with `SEEK_SET` and the state will be reset correctly. You cannot use such an `ftell()` value across a program boundary unless the stream has been marked wide-oriented. A seek specifying a relative offset (`SEEK_CUR` or `SEEK_END`) will change the state to initial state. Using relative offsets is strongly discouraged, because you may be seeking to a point that is not in initial state, or you may end up in the middle of a multibyte character, causing wide-oriented functions to give you undefined behavior. These functions expect you to be at the beginning or end of a multibyte character in the correct state. Using your own offset with `SEEK_SET` also does the same. For a wide-oriented file, the number of valid bytes or records that `ftell()` supports is cut in half.

When you use the `fsetpos()` function to reposition within a file, the shift state is set to the state saved by the function. Use this function to reposition to a wide character that is not in the initial state.

`ungetwc()` Considerations

For text files, the library functions `fgetpos()` and `ftell()` take into account the character you have pushed back onto the input stream with `ungetwc()`, and move the file position back by one wide character. The starting position for an `fseek()` call with an *`origin`* of `SEEK_CUR` also takes into account this pushed-back wide character. Backing up one wide character means backing up either a single-byte character or a multibyte character, depending on the type of the preceding character. The implicit newlines at the end of each record are counted as wide characters.

For binary files, the library functions `fgetpos()` and `ftell()` also take into account the character you have pushed back onto the input stream with `ungetwc()`, and adjust the file position accordingly. However, the `ungetwc()` must push back the same type of character just read by `fgetwc()`, so that `ftell()` and `fgetpos()` can save the state correctly. An `fseek()` with an *`origin`* of `SEEK_CUR` also accounts for the pushed-back character. Again, the `ungetwc()` must "unget" the same type of character for this to work properly. If the `ungetwc()` pushes back a character in the opposite state, you will get undefined behavior.

You can make only one call to `ungetwc()`. If the current logical file position is already at or before the first `wchar` in the file, a call to `ftell()` or `fgetpos()` after `ungetwc()` fails.

When you are using `fseek()` with an *`origin`* of `SEEK_CUR`, the starting point for the reposition also accounts for the presence of `ungetwc()` characters and compensates as `ftell()` and `fgetpos()` do. Specifying a relative offset other than `0` is not supported and results in undefined behavior.

You can set the _EDC_COMPAT environment variable to specify that `ungetwc()` should not affect `fgetpos()` or `fseek()`. (It will still affect `ftell()`.) If the environment variable is set, `fgetpos()` and `fseek()` ignore any pushed-back wide character. See Chapter 21, “Using Environment Variables,” on page 219 for more information about _EDC_COMPAT.

If a repositioning operation fails, LE/VSE C Run-Time attempts to restore the original file position by treating the operation as a call to `fflush()`. It does not account for the presence of `ungetwc()` characters, which are lost.

Closing Files

LE/VSE C Run-Time expects files to end in initial shift state. For binary byte-oriented files, you must ensure that the ending state of the file is initial state. Failure to do so results in undefined behavior if you reaccess the file again. For wide-oriented streams and byte-oriented text streams, LE/VSE C Run-Time tracks new data that you add. If necessary, LE/VSE C Run-Time adds a padding byte to complete any incomplete multibyte character and a shift-in to end the file in initial state.

Chapter 7. Standard Streams and Redirection

A C program has associated with it *standard streams*. You do not have to open these streams, because they are automatically set up for you by C when you include the stdio.h header file. Table 13 below shows three standard streams for C and the functions that implicitly use them.

Table 13. C Standard Streams

Name of stream	Purpose	Functions that use it
stdin	The input device from which your C program usually retrieves its data.	**getchar() scanf() gets()**
stdout	The output device to which your C program normally directs its output.	**printf() puts() putchar()**
stderr	The output device to which your C program directs its diagnostic messages. LE/VSE C Run-Time uses `stderr` to collect error messages about exceptions that occur.	**perror()**

On I/O operations requiring a file pointer, you can use `stdin`, `stdout`, or `stderr` in the same manner as you would any other file pointer.

The default behavior for the C standard streams is for them to open automatically on first reference. You do not have to call `fopen()` to open them. For example:

```
printf("%d\n",n);
```

with no preceding `fopen()` statement writes the decimal number *n* to the `stdout` stream.

By default, `stdin` interprets the character sequence `/*` as indicating that the end of the file has been reached.

Default Open Modes

The default open modes for the C standard streams are:

stdin	r
stdout	w
stderr	w

Where the streams go depends on what kind of environment you are running under. These are the defaults:

- **Under VSE batch:**
  - `stdin` goes to `SYSIPT`. If `SYSIPT` cannot be opened, all read operations from `stdin` will fail.
  - `stdout` goes first to `SYSLST`; if `SYSLST` cannot be opened, the `stdout` stream is sent to `stderr`.
  - `stderr` will go to the LE/VSE `MSGFILE`.

- **Under CICS:**
  - stdin is not supported under CICS.
  - stdout and stderr are assigned to transient data queues, allocated during CICS initialization. The CICS standard streams can be redirected only to or from memory files. You can do this by using freopen().

You can also redirect the standard streams to other files. See "Using the Redirection Symbols" and sections following.

Using the Redirection Symbols

The following table lists the redirection symbols supported by LE/VSE C Run-Time for redirection of C standard streams from the PARM parameter of the EXEC statement or from a system() call. **0**, **1**, and **2** represent stdin, stdout, and stderr, respectively.

Table 14. LE/VSE C Run-Time Redirection Symbols

Symbol	Description
<*fn*	associates the file specified as *fn* with stdin; reopens *fn* in mode r.
0<*fn*	associates the file specified as *fn* with stdin; reopens *fn* in mode r.
>*fn*	associates the file specified as *fn* with stdout; reopens *fn* in mode w.
1>*fn*	associates the file specified as *fn* with stdout; reopens *fn* in mode w.
>>*fn*	associates the file specified as *fn* with stdout; reopens *fn* in mode a.
2>*fn*	associates the file specified as *fn* with stderr; reopens *fn* in mode w.
2>> *fn*	associates the file specified as *fn* with stderr; reopens *fn* in mode a.
2>**&1**	associate stderr with stdout; same file and mode.
1>**&2**	associate stdout with stderr; same file and mode.

In Table 14, *fn* can be specified as either a DLBL/TLBL-name and/or a logical unit, or as a file ID.

Notes:

1. If you use the NOREDIR option on a #pragma runopts directive, you cannot redirect standard streams using the preceding list of symbols.
2. If you want to pass one of the redirection symbols as an argument, you can enclose it in double quotation marks.
3. When two options specifying redirection conflict with each other, or when you redirect a standard stream more than once, the redirection fails. If you do the latter, you will get an abend. For example, if you specify

   ```
   2>&1
   ```

   and then

   ```
   1>&2
   ```

   LE/VSE C Run-Time uses the first redirection and ignores any subsequent ones. If you specify

   ```
   >a.out
   ```

   and then

   ```
   1>&2
   ```

   the redirection fails and the program abends.

4. A failed attempt to redirect a standard stream causes your program to fail in initialization.

Assigning the Standard Streams

You can redirect a C standard stream by assigning a valid file pointer to it, as follows:

```
FILE *stream;
stream = fopen("new.file", "w+");
stdout = stream;
```

This method of redirecting streams is known as *direct assignment*.

You must ensure that the streams are appropriate; for example, do not assign a stream opened for `w` to `stdin`. Doing so would cause a function such as `getchar()` called for the stream to fail, because `getchar()` expects a stream to be opened for read access.

Using the `freopen()` Library Function

You can use the `freopen()` C library function to redirect C standard streams in all environments.

Redirecting Streams with the `MSGFILE` Option

You can redirect `stderr` by specifying a `DLBL`/`TLBL`-name and/or a logical unit on the `MSGFILE` run-time option and not redirecting `stderr` elsewhere (such as on the `PARM` parameter of the `EXEC` statement). The default logical unit for the LE/VSE `MSGFILE` is `SYSLST`. See *LE/VSE Programming Guide* for more information on `MSGFILE`.

`MSGFILE` Considerations

LE/VSE C Run-Time makes a distinction between types of error output according to whether the output is directed to the `MSGFILE`, to `stderr`, or to `stdout`:

Table 15. Output Destinations under LE/VSE C Run-Time

Destination of Output	**Type of Message**	**Produced by**	**Default Destination**
`MSGFILE` output	LE/VSE messages (CEExxxx)	LE/VSE conditions	LE/VSE `MSGFILE`
	LE/VSE C Run-Time language messages (EDCxxxx)	LE/VSE C Run-Time unhandled conditions	LE/VSE `MSGFILE`
`stderr` messages	`perror()` messages (EDCxxxx)	Issued by a call to `perror()`	LE/VSE `MSGFILE`
	User output sent explicitly to `stderr`	Issued by a call to `fprintf()`	LE/VSE `MSGFILE`
`stdout` messages	User output sent explicitly to `stdout`	Issued by a call to `printf()`	`stdout`

Table 16 on page 40 describes the destination of output to `stderr` and `stdout` after redirection has occurred. Whenever `stdout` and `stderr` share a common destination, the output is interleaved. The default case is the one where `stdout` and `stderr` have not been redirected.

Table 16. LE/VSE C Run-Time Interleaved Output

	stderr not redirected	**stderr redirected to destination other than stdout**	**stderr redirected to stdout**
stdout not redirected	stdout to itself stderr to MSGFILE	stdout to itself stderr to its other destination	Both to stdout
stdout redirected to destination other than stderr	stdout to its other destination stderr to MSGFILE	stdout to its other destination stderr to its other destination	Both to the new stdout destination
stdout redirected to stderr	Both to MSGFILE	Both to the new stderr destination	stdout to stderr stderr to stdout

LE/VSE C Run-Time routes error output as follows:

- MSGFILE output
  - LE/VSE messages (messages prefixed with CEE)
  - Language messages (messages prefixed with EDC)
- stderr output
  - perror() messages (messages prefixed with EDC and issued by a call to perror())
  - Output explicitly sent to stderr (for example, by a call to fprintf())

By default, LE/VSE C Run-Time sends all stderr output to the MSGFILE destination and stdout output to its own destination. You can change this by using LE/VSE C Run-Time redirection, which enables you to redirect stdout and stderr to a DLBL/TLBL-name and/or a logical unit, a file ID, or each other. Unless you have redirected stderr, it always uses the MSGFILE destination. When you redirect stderr to stdout, stderr and stdout share the stdout destination. When you redirect stdout to stderr, they share the stderr destination.

Redirecting Streams

This section describes how to redirect C standard streams under VSE.

Under VSE Batch

You can redirect standard streams in the following ways:

- From the freopen() library function call
- On the PARM parameter of the EXEC statement used to invoke your C program
- Using ASSGN statements

Because the topic of JCL statements goes beyond the scope of this book, only simple examples will be shown here.

Using the PARM Parameter of the EXEC Statement

The following example shows an excerpt taken from a job stream. It demonstrates the redirection of stdout using the PARM parameter of the EXEC statement:

```
KNOWN:   - The program name is BATCHPGM
         - The program has 1 required parameter.  In this example, we
           will use 'DEBUG' for the required parameter
         - The output from BATCHPGM is to be directed to a sequential
           file called 'MAINT.LOG.LISTING'

USE THE FOLLOWING JCL statements:
         // JOB jobname
         // EXEC BATCHPGM,SIZE=BATCHPGM,PARM='DEBUG >''MAINT.LOG.LISTING'' '
            .
            .
            .
```

Figure 6. Redirecting `stdout` under VSE Batch

The standard streams can only be redirected to files where the file attributes can be determined by LE/VSE C Run-Time or where defaults are used for the file attributes. The files that standard streams can be redirected to are as follows:
- Unit record devices (for example, card readers/punches, printers, system logical units)
- SAM ESDS files that have been explicitly defined
- VSAM ESDS data sets
- Memory files

The files that standard streams cannot be redirected to are as follows:
- SAM files other than SAM ESDS files
- SAM ESDS files that have not been explicitly defined
- Tape devices
- Members of VSE/Librarian sublibraries

Using `ASSGN` Statements

When you use `ASSGN` statements to redirect standard streams, the standard streams will be associated with system logical units as follows:
- `stdin` will be associated with `SYSIPT`. If `SYSIPT` is not assigned, no characters can be read in from `stdin`.
- `stdout` will be associated with `SYSLST`. If `SYSLST` is not assigned, no characters can be written to `stdout`.
- `stderr` will be associated with `SYSLST` (assuming that the LE/VSE &msgfile is directed to `SYSLST` as `stderr` is directed to the LE/VSE `MSGFILE`). If `SYSLST` is not assigned, no characters can be written to `stderr`.

The following example shows an excerpt taken from a job stream. It demonstrates the redirection of the three standard streams using `DLBL`, `EXTENT` and `ASSGN` statements:

```
KNOWN:   - The program name is MONITOR
         - The input to MONITOR is to be retrieved from a sequential
           file called 'SAFETY.CHEM.LIST'
         - The output of MONITOR is to be directed to a new sequential
           file called 'YEAREND.ACTION.CHEM'
         - Any errors generated by MONITOR are to be directed to a new
           sequential file called 'YEAREND.ACTION.CHEM'

USE THE FOLLOWING JCL statements:
         // JOB jobname
         // DLBL IJSYSIN,'SAFETY.CHEM.LIST',0,SD
         // EXTENT SYSIPT
         ASSGN SYSIPT,DISK,VOL=volser,SHR
         // DLBL IJSYSLS,'YEAREND.ACTION.CHEM',0,SD
         // EXTENT SYSLST,volser,...
         ASSGN SYSLST,DISK,VOL=volser,SHR
           .
           .
           .
         // EXEC MONITOR,SIZE=MONITOR
         CLOSE SYSIPT,SYSRDR
         CLOSE SYSLST,PRT1
           .
           .
           .
```

Figure 7. Redirecting Standard Streams Using `ASSGN` Statements

Redirecting Streams under CICS

There are several ways to redirect C standard streams under CICS:

- You can assign a memory file to the stream (for example, `stdout=myfile`).
- You can use `freopen()` to open a standard stream as a memory file.
- You can use CICS facilities to direct where the stream output goes.

If you assign a file pointer to a stream or use `freopen()` on it, you will not be able to use C functions to direct the information outside or elsewhere in the CICS environment. Once access to a CICS transient data queue has been removed, either by a call to `freopen()` or `fclose()`, or by the assignment of another file pointer to the stream, LE/VSE C Run-Time does not provide a way to regain access. Once C functions have lost access to the transient data queues, you must use the CICS-provided facilities to regain it.

CICS provides a facility that enables you to direct where a given transient data queue, the default standard stream implementation, will go, but you must configure this facility before a CICS cold start.

Passing Standard Streams across a `system()` Call

A `system()` call occurs when one LE/VSE C Run-Time program calls another LE/VSE C Run-Time program by using the `system()` function. Standard streams are inherited across calls to the ANSI `system()` function.

Inheritance includes any redirection of the stream as well as the open mode of the stream. For example, if program A reopens `stdout` as `"A.B"` for `"wb"` and then calls program B, program B inherits the definition of `stdout`. If program B reopens `stdout` as `"C.D"` for `"ab"` and then uses `system()` to call program C, program C inherits `stdout` opened to `"C.D"` for append. Once control returns to the calling program, the definitions of the standard streams from the time of the `system()` call are restored. For example, when program B finally returns control to program A, `stdout` is restored to `"A.B"` opened for `"wb"`.

The file position and the amount of data that is visible in the called and calling programs depend on whether the standard streams are opened for binary, text, or record I/O.

Since the I/O Stream standard streams are implemented in terms of the C standard streams, behavior of the I/O Stream standard streams across a `system()` call is based on the behavior of the C standard streams across `system()`.

Passing Binary Streams

If the standard stream being passed across a `system()` call is opened in binary mode, any reads or writes issued in the called program occur at the next byte in the file. On return, the position of the file is wherever the called program is positioned. This includes any possible repositions made by the called program if the file is enabled for positioning. Because output to binary files is done byte by byte, all bytes are written to `stdout` and `stderr` in the order they are written. This is shown in the following example:

```
printf("123");
printf("456");
system("CHILD");     ──▸     int main(void)  { putc('7',stdout);}
printf("89")
```

The output from this example is:

```
123456789
```

Memory files are always opened in binary mode, even if you specify text. Any standard streams redirected to memory files and passed across `system()` calls will be treated as binary files.

If `freopen()` is applied to a C standard stream, thereby creating a binary stream, the results of I/O to the associated I/O Stream standard stream across a `system()` call are undefined.

Passing Text Streams

If the C standard stream being passed across a `system()` call is opened in text mode (the default), the file position in the called program is placed at the next record boundary, if it is not already at the start of a record. Any data in the current record that is unread is skipped. Here is an example:

```
INPUT FILE          ROOT C PROGRAM            CHILD PROGRAM
----------          int main() {              int main() {
abcdefghijklm           char c[4]                char d[2]
nopqrstuvwxyz           c[0] = getchar();         d[0] = getchar();
0123456789ABC           c[1] = getchar();         d[1] = getchar();
DEFGHIJKLMNOP           system("CHILD");          printf("%.2s\n",
                        c[2] = getchar();         d);
                        c[3] = getchar();     }
                        printf("%.4s\n",c);
                    }

OUTPUT
------
no          ──▸  from the child
ab01        ──▸  from root
```

When you write to a spanned file, the file position moves to the beginning of the next record, if that record exists. If not, the position moves to the end of the incomplete record.

For non-spanned standard streams opened for output, if the caller has created a text record missing an ending control character, the last record is hidden from the called program. The called program can append new data if the stream is open in append mode. Any appends made by the called program will be after the last record that was complete at the time of the `system()` call.

When the called program terminates, it completes any new unfinished text record with a newline; the addition of the newline does not move the file position. Once any incomplete record is completed, the file position moves to the next record boundary, if it is not already on a record boundary or at `EOF`.

When control returns to the original caller, any incomplete record hidden at the time of the `system()` call is restored to the end of the file. If the called program is at `EOF` when it is terminated and the caller was within an incomplete record at the time of the `system()` call, the position upon return is restored to the original record offset at the time of the `system()` call. This position is usually the end of the incomplete record. Generally, if the caller is writing to a standard stream and does not complete the last record before it calls `system()`, writes continue to add to the last record when control returns to the caller. For example:

```
printf("test\n");
printf("abc");
system("hello");    ──▶  int main(void) { printf("hello world\n"); }
printf("def\n");
```

The output from this example is as follows:

```
test
hello world
abcdef
```

If `stdout` had been opened for `"w+"` in this example, and a reposition had been made to the character `'b'` before the `system()` call, upon return, the incomplete record `"abc"` would have been restored and the position would have been at the `'b'`. The subsequent write of `def` would have performed an update to give:

```
test
hello world
adef
```

Passing Record I/O Streams

For record I/O, all reads and writes made by the called program occur at the next record boundary. Since complete records are always read and written, there is no change in the file position across a `system()` call boundary.

In the following example, `stdout` is a variable-length record I/O file.

```
fwrite("test",1,4,stdout);
fwrite("abc",1,3,stdout);
system("hello");    ──▶      int main(void) {
fwrite("def",1,3,stdout);        fwrite("hello world",1,11,stdout);
                             }
```

The output from this code fragment is as follows:

```
test
abc
hello world
def
```

Chapter 8. Performing SAM I/O Operations

This chapter describes using SAM I/O under VSE batch. SAM I/O includes support for the following:
- Sequential disk files (SAM and VSAM-managed SAM)
- Non-disk files, such as tapes, printers, etc.

Notes:
1. LE/VSE C Run-Time does not support BDAM or ISAM files.
2. LE/VSE C Run-Time does not support SAM I/O under CICS. All I/O under CICS must be via the CICS command level interface.

SAM I/O supports text, binary, and record I/O, in three record formats, fixed (`F`), variable (`V`), and undefined (`U`).

See Chapter 6, "LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)," on page 29 for information about using wide-character I/O with LE/VSE C Run-Time.

Opening Files

To open a SAM file, use the standard C `fopen()` or `freopen()` library functions. These are described in general terms in *LE/VSE C Run-Time Library Reference* Details about them specific to all LE/VSE C Run-Time I/O are discussed in Chapter 3, "Opening Files," on page 15. This section describes considerations for using `fopen()` and `freopen()` with SAM files.

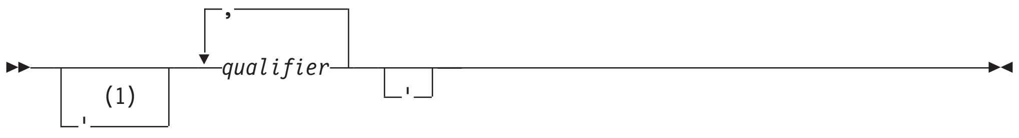
Note: The `freopen()` function cannot be used with SAM ESDS files when `DISP=(,DELETE)` has been specified on the `DLBL` statement.

Using `fopen()` or `freopen()`

Files are opened with a call to `fopen()` or `freopen()` in the format: `fopen("`*`filename`*`", "`*`mode`*`")`.

Filenames for SAM Files

Using a File ID: The syntax for the *filename* argument on your `fopen()` or `freopen()` call when using a file ID is shown in the following diagram:



Notes:

1 The single quotation marks must be matched; if you use one, you must use the other.

A sample construct is:

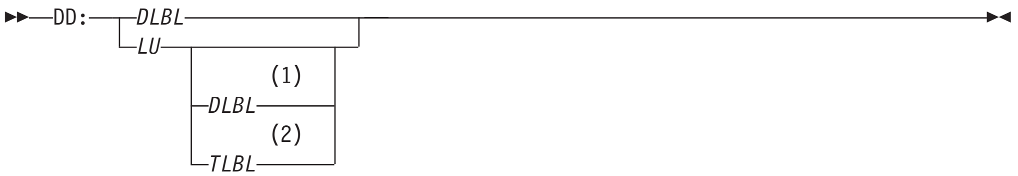
```
'qualifier1.qualifier2'
```

'
: When you enclose a file ID in single quotation marks, the file ID is *fully qualified*. The file opened is the one specified by the file ID inside the quotation marks. If the file ID is not fully qualified, LE/VSE C Run-Time adds the job name to the front of the file ID. For example, the statement `fopen("a.b","w");` opens a file *jobname*.A.B, where *jobname* is the name of the job submitted. If the file ID is fully qualified, LE/VSE C Run-Time does not add a job name.

qualifier
: Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national ($, #, @), the hyphen, or the character X'C0'. The first character should be either alphabetic or national.

  You can join qualifiers with periods. The maximum length of a file ID is 44 characters, including periods.

Using a DLBL/TLBL-name and/or Logical Unit: The syntax for the *filename* argument on your `fopen()` or `freopen()` call when using a `DLBL/TLBL`-name and/or logical unit is shown in the following diagram:



Notes:

1 If both *LU* and *DLBL* or *TLBL* are specified, a dash (-) must be used as separator (no blanks are allowed).

2 If both *LU* and *DLBL* or *TLBL* are specified, a dash (-) must be used as separator (no blanks are allowed).

LU
: Can be either a system logical unit (`SYSIPT`, `SYSLST`, `SYSPCH`, `SYSLNK`, or `SYSLOG`) or a programmer logical unit (`SYS000` to `SYS254`).

DLBL **or** *TLBL*
: A 1- to 8-character name of which at most the first 7 characters are used. These characters may be alphanumeric or national ($, #, @). The first character must be either alphabetic or national.

LE/VSE C Run-Time interprets the above as follows:

1. If the first or only parameter following `DD:` is a system logical unit (for example, `SYSLST`), the system logical unit will be opened.
2. If a `DLBL` JCL statement is present for the *DLBL* specified, a disk file will be opened.
3. If a `TLBL` JCL statement is present for the *TLBL* specified, a *labeled* tape file will be opened. The programmer logical unit must also be specified as shown in the above diagram.
4. If neither a `DLBL` nor a `TLBL` JCL statement is present for the *DLBL* or *TLBL* specified, the programmer logical unit (if specified) will be checked to determine if it is assigned. If it is assigned to a tape device, an *unlabeled* tape file will be opened. If it is assigned to a card or printer device, the card or printer device will be opened.

Specification of logical unit is ignored for a SAM ESDS file which must be opened using a DLBL-name.

Tapes

LE/VSE C Run-Time supports standard label (SL) tapes. If you are creating labeled tape files, you can only open them by TLBL. LE/VSE C Run-Time provides support for opening tapes in read, write, or append mode, but not update. When you open a tape for read or append, any file attributes you specify must match those of the existing file exactly. The only repositioning function supported for tape files is `rewind()`, which is available only when you have opened a tape for read. For tape files opened for write or append, calling `rewind()` has no effect. Calls to any repositioning function other than `rewind()` will fail.

Opening FBS-format tape files with append-only mode will fail.

When you open a tape file for output, the file ID you specify in the JCL must match the file ID specified in the tape label, even if the existing tape file is empty. If this is not the case, you must either change the JCL to specify the correct file ID or write to another tape file, or reinitialize the tape to remove the tape label and the data.

You can append only to an existing file on tape. Attempting to append to a file that does not already exist on a tape will cause an error. You can create an empty file on a tape by opening the file for write and closing it without writing any data to it.

If a TLBL exists for the tape file, the file will be opened with standard labels. Otherwise it will be opened as unlabeled.

Multivolume Files

LE/VSE C Run-Time supports files that span more than one volume of disk or tape. To open a multivolume file for write, you must open it by DLBL/TLBL-name.

You can open multivolume tape files only for read or write. Opening them for update or append is not supported.

You can open multivolume disk files for read, write, or update, but not for append. If you open one in `r+` or `rb+` mode, you can read and update the file, but you cannot extend the file.

The only repositioning function which is supported for multivolume files is `rewind()`; calls to any of the other repositioning functions fail.

Other Devices

LE/VSE C Run-Time supports several other devices for input and output (for example, card and printer). You can open these devices only by DLBL-name.

The only repositioning function supported for card reader devices is `rewind()`. No repositioning functions are supported for output-only devices such as card punch devices and printers.

Although LE/VSE C Run-Time supports I/O operations on the system log using the system logical unit SYSLOG, the actual interface is different from that of other file types. As a result, most `fopen()` parameters are not applicable to this file. If records which are longer than 68 bytes are written to `SYSLOG`, they will be wrapped.

`fopen()` and `freopen()` Parameters

The following table lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are allowed and applicable for SAM I/O, and lists the option values that are valid for the applicable ones. Detailed descriptions of these options follow the table.

Table 17. Parameters for the `fopen()` and `freopen()` Functions for SAM I/O

Parameter	Allowed?	Applicable?	Notes
`recfm=`	Yes	Yes	Any of the 27 record formats available under LE/VSE C Run-Time, plus `A` are valid. See the parameter list below for details.
`lrecl=`	Yes	Yes	`0` or any positive integer up to 32760 is valid. See the parameter list below for details.
`blksize=`	Yes	Yes	`0` or any positive integer up to 65528 is valid. See the parameter list below for details.
`space=`	Yes	Yes	Valid for VSAM-managed SAM only. See the parameter list below for details.
`type=`	Yes	Yes	May be omitted. If you do specify it, `type=record` is the only valid value.
`acc=`	Yes	No	Not used for SAM I/O.
`password=`	Yes	No	Not used for SAM I/O.
`asis`	Yes	No	Ignored.
`byteseek`	Yes	Yes	Used for binary files to specify that the seeking functions should use relative byte offsets instead of encoded offsets. See the parameter list below for details.
`noseek`	Yes	Yes	Used to disable seeking functions for improved performance. See the parameter list below for details.
`OS`	Yes	No	Ignored.
`rewind=`	Yes	Yes	This parameter is supported for tape files only. See the parameter list below for details.
`dsn=`	Yes	No	Ignored.

`recfm=`

LE/VSE C Run-Time allows you to specify any of the 27 possible RECFM types (listed on pages 6, 9, and 12), as well as the LE/VSE C Run-Time RECFM `A`.

When you are opening an existing file for read or append (or for write, if you have specified `DISP=SHR` for a disk file or `DISP=SHR/DISP=MOD` for a tape file), any RECFM that you specify must match that of the existing file, except that you may specify `recfm=U` to open any file for read, and you may specify `recfm=FBS` for a file created as `recfm=FB`. Specifying `recfm=FBS` indicates to LE/VSE C Run-Time that there are no short blocks within the file. If there are, undefined behavior results.

For variable-format SAM files, the RDW, SDW, and BDW contain the length of the record, segment, and block as well as their own lengths. If you open a file for read with `recfm=U`, LE/VSE C Run-Time treats each physical block as an undefined-format record. For files created with `recfm=V`, LE/VSE C Run-Time does not strip off block descriptor words (BDWs) or record descriptor words (RDWs), and for blocked files, it does not deblock records. Using `recfm=U` is helpful for viewing variable-format files or seeing how records are blocked in the file.

Specifying `recfm=A` indicates that the file contains ASA print-control characters. If you create a file by opening it for write or append, the `A` attribute is added to the default RECFM. For more information about ASA, see Chapter 5, "ASA Text Files," on page 25.

Specifying `recfm=*` is not supported under LE/VSE C Run-Time.

The values that may be specified for RECFM for a given file type are subject to any limitations described in "Parameters Supported by File Type" on page 52.

`lrecl=` and `blksize=`
The LRECL that you specify on the `fopen()` call defines the maximum record length that the C library allows. Records longer than the maximum record length are not written to the file. The first 4 bytes of each block and the first 4 bytes of each record of variable-format files are used for control information. For more information, see "Variable-Format Records" on page 9.

The maximum LRECL supported for sequential disk files is 32760. The maximum BLKSIZE supported for sequential disk files is 65528.

When you are opening an existing file for read or append (or for write, if you have specified `DISP=SHR` for a disk file or `DISP=SHR`/`DISP=MOD` for a tape file), any LRECL or BLKSIZE that you specify must match that of the existing file, except when you open an `F` or `FB` format file on a disk device without specifying the `noseek` parameter. In this case, you can specify the `S` attribute to indicate to LE/VSE C Run-Time that the file has no imbedded short blocks. Files without short blocks improve LE/VSE C Run-Time's performance.

The values that may be specified for LRECL and BLKSIZE for a given file type are subject to any limitations described in "Parameters Supported by File Type" on page 52.

`space=(records=`*n*`)` or `space=(records=(`*n*`,`*n1*`))`
This parameter enables you to specify the number of records for the primary allocation (*n*), and optionally, the number of records for the secondary allocation (*n1*) of a SAM ESDS file. It is used in conjunction with the `blksize` parameter to modify the `DLBL` options `RECORDS` and `RECSIZE` respectively.

`type=`
You can omit this parameter. If you specify it, the only valid value for SAM I/O is `type=record`, which opens a file for record I/O.

Note: `type=record` is allowed only when opening a binary file.

`acc=`
This parameter is not valid for SAM I/O. If you specify it, LE/VSE C Run-Time ignores it.

`password=`
This parameter is not valid for SAM I/O. If you specify it, LE/VSE C Run-Time ignores it.

`asis`
If you specify this parameter, LE/VSE C Run-Time ignores it.

byteseek
When you specify this parameter and open a file in binary mode, all repositioning functions (such as `fseek()` and `ftell()`) use relative byte offsets from the beginning of the file instead of encoded offsets. To have the `byteseek` parameter set as the default for all your calls to `fopen()` or `freopen()`, you can set the environment variable `_EDC_BYTE_SEEK` to `Y`. See Chapter 21, "Using Environment Variables," on page 219 for more information.

noseek
Specifying this parameter on the `fopen()` call disables the repositioning functions `ftell()`, `fseek()`, `fgetpos()`, and `fsetpos()` for as long as the file is open. When you have specified `noseek` and have opened a disk file for read only, the only repositioning function allowed on the file is `rewind()`, if the device supports rewinding. Otherwise, a call to `rewind()` sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`. Calls to `ftell()`, `fseek()`, `fsetpos()`, or `fgetpos()` return `EOF`, set `errno`, and set the stream error flag on.

The use of the `noseek` parameter may improve performance when you are reading and writing files.

Note: If you specify the `noseek` parameter when you open a file for writing, you must specify `noseek` on any subsequent `fopen()` call that simultaneously opens the file for reading; otherwise, you will get undefined behavior.

OS
If you specify this parameter, LE/VSE C Run-Time ignores it.

rewind=
This parameter, which is supported for tape files only, determines whether the tape is rewound and/or unloaded as a result of `fclose()` depending on the value specified as follows:
- If `norwd` is specified, the tape will not be rewound.
- If `unload` is specified, the tape will be rewound and unloaded.

If the `rewind` parameter is omitted, the default is to rewind but not unload.

dsn=
This parameter is not valid for SAM I/O. If you specify it, LE/VSE C Run-Time ignores it.

Parameters Supported by File Type

Table 18 shows the supported `fopen()` and `freopen()` `mode`, `blksize`, `lrecl` and `recfm` parameter values for the various file types.

Table 18. `fopen()` and `freopen()` Parameters Supported by File Type

File Type	Parameter	Value
SYSIPT	mode	`r` and `rb`.
	blksize	The value specified is not used.
	lrecl	As required. Default is 80. Maximum is 512.
	recfm	Fixed. If blocked, ASA print-control characters, or machine control codes are specified, they are ignored. Default is `F`.

Table 18. `fopen()` and `freopen()` Parameters Supported by File Type (continued)

File Type	Parameter	Value
SYSPCH	mode	w, wb, a, and ab. If a or ab is specified, it will be treated as if w or wb was specified.
	blksize	The value specified is not used.
	lrecl	As required. Maximum is 512. Default is 80 if neither ASA print-control characters nor machine control codes are specified—otherwise 81.
	recfm	Fixed. If blocked is specified, it is ignored. ASA print-control characters are ignored unless specified with type=record. Machine control codes can be specified. Default is F.
SYSLNK	mode	w, wb, a, and ab. If a or ab is specified, it will be treated as if w or wb was specified.
	blksize	The value specified is not used.
	lrecl	The value specified is not used. (Special format SYSLNK records containing 80 bytes of user data are written.)
	recfm	Fixed unblocked. Default is F.
SYSLST	mode	w, wb, a, and ab. If a or ab is specified, it will be treated as if w or wb was specified.
	blksize	The value specified is not used.
	lrecl	As required. For fixed or undefined record format, the default is 133 if using ASA or 132 if not. For variable record format, the default is 129 if using ASA or 128 if not. If SYSLST is assigned to disk or tape, LRECL is treated as 121. Maximum is 512.
	recfm	Fixed, variable, and undefined. If blocked is specified, it is ignored. ASA print-control characters or machine control codes can be specified. Default is F.
SYSLOG	mode	w, wb, a, ab, r, and rb. If a or ab is specified, it will be treated as if w or wb was specified.
	blksize	The value specified is not used.
	lrecl	The value specified is not used.
	recfm	The value specified is not used.
SAM Disk Files	mode	r, r+, w, w+, a, a+, rb, r+b, wb, w+b, ab, and a+b. The append and update modes are not supported for multivolume files.
	blksize	As required. (Must be a multiple of LRECL for files in fixed blocked record format.) For default values, see Table 11 on page 20.
	lrecl	As required. For default values, see Table 11 on page 20.
	recfm	Fixed blocked and unblocked, variable blocked and unblocked, spanned blocked and unblocked, and undefined. ASA print-control characters or machine control codes can be specified. If FBS (fixed block standard) is specified, fseek() and ftell() functions will process as though all blocks are the same length. Record format must be specified as there is no default.

Table 18. fopen() and freopen() Parameters Supported by File Type (continued)

File Type	Parameter	Value
SAM ESDS Disk Files (VSAM-managed SAM)	mode	r, r+, w, w+, rb, r+b, wb, w+b, a, a+, ab, and a+b. If the number of blocks that will fit in a CI is greater than 255, the ftell(), fseek(), fgetpos(), and fsetpos() functions are disabled. **Note:** The check performed by the library determines if there can be more than 255 of the maximum size blocks per CI. Files which are not recfm=FBS might contain blocks which are shorter than the specified BLKSIZE and might therefore contain more than 255 blocks per CI. If they do, it will cause an error during execution. a, ab, a+ and a+b will be accepted if DISP=OLD is specified on the DLBL JCL statement. DISP=OLD indicates the file is to be created if it does not exist, or extended if it does exist. If a, ab, a+ or a+b is specified and DISP=OLD is not specified on the DLBL JCL statement, the DLBL will be changed to add DISP=OLD to cause the writes to append to the file. This can only be added if the additional VSAM label record is present. That is, the DLBL JCL statement specifies disposition, space allocation, or buffer space for the SAM ESDS file. The open will fail if the file is opened for append and the additional VSAM label record is not present. If w, wb, w+ or w+b is specified, and DISP=OLD is specified on the DLBL JCL statement, it will be treated as for append.
	blksize	As required. (Must be a multiple of LRECL for files in fixed blocked record format.) If the file has been defined explicitly using IDCAMS, the block size is available from the VSAM catalog. For default values, see Table 11 on page 20.
	lrecl	As required. If the file has been defined explicitly using IDCAMS, the logical record length might be available from the VSAM catalog. For default values, see Table 11 on page 20.
	recfm	Fixed blocked and unblocked, variable blocked and unblocked, and undefined. ASA print-control characters or machine control codes can be specified. Record format must be specified if the file does not exist (that is, if the file has not been predefined) as there is no default. If the file has been predefined, RECFM is obtained from the VSAM catalog. **Note:** VSAM-managed SAM does not support spanned records.

Table 18. `fopen()` and `freopen()` Parameters Supported by File Type (continued)

File Type	Parameter	Value
Tape	`mode`	`r`, `w`, `rb`, `wb`, `a`, and `ab`. `a` and `ab` will only be accepted for labeled tapes. If `a` or `ab` is specified, and `DISP=OLD` or `DISP=MOD` is specified on the `TLBL`, the file is opened for output and writes will append to the file. `DISP=OLD` indicates the file is to be extended and `DISP=MOD` indicates it is to be created if it does not exist, or extended if it does exist. If `a` or `ab` is specified and `DISP=OLD` or `DISP=MOD` has not been specified on the `TLBL`, the `TLBL` will be updated to add `DISP=MOD`. If `w` or `wb` is specified and `DISP=OLD` or `DISP=MOD` has been specified on the `TLBL`, the file is opened for output and writes will append to the file.
	`blksize`	As required. (Must be a multiple of LRECL for files in fixed blocked record format.) For default values, see Table 11 on page 20.
	`lrecl`	As required. For default values, see Table 11 on page 20.
	`recfm`	Fixed blocked and unblocked, variable blocked and unblocked, spanned blocked and unblocked, and undefined. ASA print-control characters or machine control codes can be specified. Record format must be specified is the file does not exist as there is no default.
Card Reader	`mode`	`r` and `rb`.
	`blksize`	The value specified is not used.
	`lrecl`	As required. Default is 80. Maximum is 512.
	`recfm`	Fixed, variable, and undefined. If blocked, ASA print-control characters, or machine control codes are specified, they are ignored. Default is `F`.
Card Punch	`mode`	`w`, `wb`, `a`, and `ab`. If `a` or `ab` is specified, it will be treated as if `w` or `wb` was specified.
	`blksize`	The value specified is not used.
	`lrecl`	As required. Maximum is 512. Default is 80 if neither ASA print-control characters nor machine control codes are specified—otherwise 81.
	`recfm`	Fixed, variable, and undefined. If blocked is specified, it is ignored. ASA print-control characters are ignored unless specified with `type=record`. Machine control codes can be specified. Default is `F`.
Printer	`mode`	`w`, `wb`, `a`, and `ab`. If `a` or `ab` is specified, it will be treated as if `w` or `wb` was specified.
	`blksize`	The value specified is ignored.
	`lrecl`	As required. For fixed or undefined record format, the default is 133 if using ASA or 132 if not. For variable record format, the default is 129 if using ASA or 128 if not. Maximum is 512.
	`recfm`	Fixed, variable, and undefined. If blocked is specified, it is ignored. ASA print-control characters or machine control codes can be specified. Default is `F`.

Buffering

LE/VSE C Run-Time uses buffers to map C I/O to system-level I/O.

When LE/VSE C Run-Time performs I/O operations, it uses one of the following buffering modes:

Line buffering
: Characters are transmitted to the system when a newline character is encountered. Line buffering is meaningless for binary and record I/O files.

Full buffering
: Characters are transmitted to the system when a buffer is filled.

C provides a third buffering mode, unbuffered I/O, which is not supported for SAM files.

You can use the `setvbuf()` and `setbuf()` library functions to set the buffering mode before you perform any I/O operation to the file. `setvbuf()` fails if you specify unbuffered I/O. It also fails if you try to specify line buffering for an `FBS` data set opened in text mode, where the device does not support repositioning. This failure happens because LE/VSE C Run-Time cannot deliver records at line boundaries without violating `FBS` format. Do not try to change the buffering mode after you have performed any I/O operation to the file.

For all files except `stderr`, full buffering is the default, but you can use `setvbuf()` to specify line buffering. For binary files, record I/O files, and unblocked text files, a block is written out as soon as it is full, regardless of whether you have specified line buffering or full buffering. Line buffering is different from full buffering only for blocked text files.

DTF (Define The File) Attributes

LE/VSE C Run-Time determines the device type by checking for the presence of `DLBL` or `TLBL` JCL statements and by determining the device assigned to the programmer or system logical unit. LE/VSE C Run-Time merges the file attributes specified in a call to `fopen()` with the file attributes retrieved from the VSAM catalog (if available) and the `DLBL` or `TLBL` JCL statement. The DTF is then built according to the type of device assigned and the resultant file attributes.

For a SAM disk file, the `blksize` specified on the `DLBL` JCL statement overrides the `blksize` specified on the call to `fopen()`. The `blksize` specified on the `DLBL` JCL statement is the `blksize` for an input file and the `blksize`+8 for an output file.

For an explicitly defined SAM ESDS file, the `blksize` from the VSAM catalog overrides the `blksize` specified on the call to `fopen()`. The `recfm` and `lrecl` (if specified on the call to `fopen()`) must match the values on the VSAM catalog.

For an implicitly defined SAM ESDS file, the `recfm`, `lrecl`, and `blksize` specified on the call to `fopen()`, override the values from the VSAM catalog.

Reading from Files

You can use the following library functions to read in information from files:
- `fread()`
- `fgets()`
- `gets()`

- fgetc()
- getc()
- getchar()
- scanf()
- fscanf()

fread() is the only interface allowed for reading record I/O files. A read operation directly after a write operation without an intervening call to fflush(), fsetpos(), fseek(), or rewind() fails. LE/VSE C Run-Time treats the following as read operations:
- Calls to read functions that request 0 bytes
- Read requests that fail because of a system error
- Calls to the ungetc() function

LE/VSE C Run-Time does not consider a read to be at EOF until you try to read past the last byte visible in the file. For example, in a file containing 3 bytes, the feof() function returns FALSE after three calls to fgetc(). Calling fgetc() one more time causes feof() to return TRUE.

You can set up a SIGIOERR handler to catch read or write system errors. See Chapter 14, “Debugging I/O Programs,” on page 131 for more information.

Reading from Binary Files

LE/VSE C Run-Time reads binary records in the order that they were written to the file. Any null padding is visible and treated as data. Record boundaries are meaningless.

Reading from Text Files

For non-ASA variable text files, the default for LE/VSE C Run-Time is to ignore any empty physical records in the file. If a physical record contains a single blank, LE/VSE C Run-Time reads in a logical record containing only a newline. However, if the environment variable _EDC_ZERO_RECLEN was set to Y, LE/VSE C Run-Time reads an empty physical record as a logical record containing a newline, and a physical record containing a single blank as a logical record containing a blank *and* a newline. Thus, when _EDC_ZERO_RECLEN is set to Y, LE/VSE C Run-Time differentiates between empty records and records containing single blanks, and does not ignore either of them. For more information about how LE/VSE C Run-Time treats empty records in variable format, see “Mapping C Types to Variable Format” on page 10.

For ASA variable text files, if a file was created without a control character as its first byte, the first byte defaults to the ' ' character (blank). When the file is read back, the first character is read as a newline.

On input, ASA characters are translated to the corresponding sequence of control characters. For more information about using ASA files, refer to Chapter 5, “ASA Text Files,” on page 25.

For undefined format text files, reading a file causes a newline character to be inserted at the end of each record. On input, a record containing a single blank character is considered an empty record and is translated to a newline character. Trailing blanks are preserved for each record.

For files opened in fixed text format, rightmost blanks are stripped off a record at input, and a newline character is placed in the logical record. This means that a record consisting of a single newline character is represented by a fixed-length record made entirely of blanks.

Reading from Record I/O Files

For files opened in record format, `fread()` is the only interface that supports reading. Each time you call `fread()` for a record I/O file, `fread()` reads one record. If you call `fread()` with a request for less than a complete record, the requested bytes are copied to your buffer, and the file position is set to the start of the next record. If the request is for more bytes than are in the record, one record is read and the position is set to the start of the next record. LE/VSE C Run-Time does not strip any blank characters or interpret any data.

`fread()` returns the number of items read successfully, so if you pass a `size` argument equal to 1 and a `count` argument equal to the maximum expected length of the record, `fread()` returns the length, in bytes, of the record read. If you pass a `size` argument equal to the maximum expected length of the record, and a `count` argument equal to 1, `fread()` returns either 0 or 1, indicating whether a record of length `size` was read. If a record is read successfully but is less than `size` bytes long, `fread()` returns 0.

A failed read operation may lead to undefined behavior until you reposition successfully.

Writing to Files

You can use the following library functions to write to a file:
- `fwrite()`
- `printf()`
- `fprintf()`
- `vprintf()`
- `vfprintf()`
- `puts()`
- `fputs()`
- `fputc()`
- `putc()`
- `putchar()`

`fwrite()` is the only interface allowed for writing to record I/O files. See *LE/VSE C Run-Time Library Reference* for more information on these library functions.

A write operation directly after a read operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails unless the read operation has reached `EOF`. The file pointer does not reach `EOF` until after you have tried to read *past* the last byte of the file.

LE/VSE C Run-Time counts a call to a write function writing `0` bytes or a write request that fails because of a system error as a write operation.

If you are updating a file and a system failure occurs, LE/VSE C Run-Time tries to set the file position to the end of the last record updated successfully. For a fully-buffered file, this is at the end of the last record in a block. For a line-buffered file, this may be any record in the current block. If you are writing new data at the time of a system failure, LE/VSE C Run-Time puts the file position at the end of the last block of the file. In files opened for blocked output, you may lose data written by other writes to that block before the system failure. The contents of a file after a system write failure are indeterminate.

If one user opens a file for writing, and another later opens the same file for reading, the user who is reading the file can check for records that may have been written past the end of the file by the other user. If the file is a spanned variable text file, the reader can read part of a spanned record and reach the end of the file before reading in the last segment of the spanned record.

Writing to Binary Files

Data flows over record boundaries in binary files. Writes or updates past the end of a record go to the next record. When you are writing to files and not making any intervening calls to `fflush()`, blocks are written to the system as they are filled. If a fixed record is incomplete when you close the file, LE/VSE C Run-Time completes it with nulls. You cannot change the length of existing records in a file by updating them.

If you are using variable binary files, note the following:
- On input and on update, records that have no length are ignored; you will not be notified. On output, zero-length records are not written. However, in spanned files, if the first segment of a record has been written to the system, and the user flushes or closes the file, a zero-length last segment may be written to the file.
- If you are writing new data in a `recfm=VB` file, LE/VSE C Run-Time may add a short record at the end of a block, to fill the block out to the full block size.
- If your file is spanned, records are written up to length LRECL, spanning multiple blocks if necessary. You can create a spanned file by specifying a RECFM containing `V` and `S` on the `fopen()` call.

Writing to Text Files

LE/VSE C Run-Time treats the control characters as follows when you are writing to a non-ASA text file:

`\a`	Alarm. Placed directly into the file; LE/VSE C Run-Time does not interpret it.
`\b`	Backspace. Placed directly into the file; LE/VSE C Run-Time does not interpret it.
`\f`	Form feed. Placed directly into the file; LE/VSE C Run-Time does not interpret it.
`\n`	Newline. Defines a record boundary; LE/VSE C Run-Time does not place it in the file.
`\r`	Carriage return. Defines a record boundary; LE/VSE C Run-Time does not place it in the file. Treated like a newline character.
`\t`	Horizontal tab character. Placed directly into the file; LE/VSE C Run-Time does not interpret it.
`\v`	Vertical tab character. Placed directly into the file; LE/VSE C Run-Time does not interpret it.
`\x0E`	DBCS shift-out character. Indicates the beginning of a DBCS string, if `MB_CUR_MAX > 1`. Placed into the file.
`\x0F`	DBCS shift-in character. Indicates the end of a DBCS string, if `MB_CUR_MAX >`

1. Placed into the file. See Chapter 6, "LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)," on page 29 for more information about `MB_CUR_MAX`.

The way LE/VSE C Run-Time treats text files depends on whether they are in fixed, variable, or undefined format, and whether they use ASA.

As with ASA files in other environments, the first character of each record is reserved for the ASA print-control character that represents a newline, a carriage return, or a form feed.

See Chapter 5, "ASA Text Files," on page 25 for more information.

Writing to Fixed-Format Text Files

Records in fixed-format files are all the same length. You complete each record with a newline or carriage return character. For fixed text files, the newline character is not written to the file. Blank padding is inserted to the LRECL of each record of the block, and the block, when full, is written. For a more complete description of the way fixed-format files are handled, see "Fixed-Format Records" on page 6.

A logical record can be shortened to be an empty record (containing just a newline) or extended to a record containing LRECL bytes of data plus a newline. Because the physical record represents the newline position by using padding blanks, the newline position can be changed on an update as long as it is within the physical record.

Note: Using `ftell()` or `fgetpos()` values for positions that do not exist after you have shortened records results in undefined behavior.

When you are updating a file, writing new data into an existing record replaces the old data and, if the new data is longer or shorter than the old data, changes the size of the logical record by changing the number of blank characters in the physical record. When you extend a record, thereby writing over the old newline, a newline character is implied after the last character of the update. Calling `fflush()` flushes the data out to the file and inserts blank padding between the last data character and the end of the record. Once you have called `fflush()`, you can call any of the read functions, which begin reading at the newline. Once the newline is read, reading continues at the beginning of the next record.

Writing to Variable-Format Text Files

In a file with variable-length records, each record may be a different length. The variable length formats permit both variable-length records and variable-length blocks. The first 4 bytes of each block are reserved for the Block Descriptor Word (BDW); the first 4 bytes of each record are reserved for the Record Descriptor Word (RDW).

For ASA and non-ASA, the `'\n'` (newline) character implies a record boundary. On output, the newline is not written to the physical file; instead, it is assumed to follow the data of the record.

If you have not set `_EDC_ZERO_RECLEN`, LE/VSE C Run-Time writes out a record containing a single blank character to represent a single newline. On input, a record containing a single blank character is considered an empty record and is translated to a newline character. Note that a single blank followed by a newline is written out as a single blank, and is treated as just a newline on input. When `_EDC_ZERO_RECLEN` is set, writing a record containing only a newline results in a zero-length variable record.

For more information about environment variables, refer to Chapter 21, "Using Environment Variables," on page 219. For more information about how LE/VSE C Run-Time treats empty records in variable format, see "Mapping C Types to Variable Format" on page 10.

Attempting to shorten a record on update by specifying less data before the newline causes the record to be padded with blanks to the original record size. For spanned records, updating a record to a shorter length results in the same blank padding to the original record length, over multiple blocks, if applicable.

Attempts to lengthen a record on update generally result in truncation. The exception to this rule is extending an empty record to a 1-byte record when the environment variable `_EDC_ZERO_RECLEN` is not set. Because the physical representation for an empty record is a record containing one blank character, it is possible to extend the logical record to a single non-blank character followed by a newline character. For standard streams, truncation in text files does not occur; data is wrapped automatically to the next record as if you had added a newline.

When you are writing data to a non-blocked file without intervening flush or reposition requests, each record is written to the system when a newline or carriage return character is written or when the file is closed.

When you are writing data to a blocked file without intervening flush or reposition requests, if the file is opened in full buffering mode, the block is written to the system on completion of the record that fills the block. If the blocked file is line buffered, each record is written to the system when it is completed. If you are using full buffering for a `VB` format file, a write may not fill a block completely. The data does not go to the system unless a block is full; you can complete the block with another write. If the subsequent write contains more data than is needed to fill the block, it flushes the current block to the system and starts writing your data to a new block.

When you are writing data to a spanned file without intervening flush or reposition requests, if the record spans multiple blocks, each block is written to the system once it is full and the user writes an additional byte of data.

For ASA variable text files, if a file was created without a control character as its first byte or record (after the RDW and BDW), the first byte defaults to the `' '` character. When the file is read back, the first character is read as a newline.

Writing to Undefined-Format Text Files

In an undefined-format file, there is only one record per block. Each record may be a different length, up to a maximum length of BLKSIZE. Each record is completed with a newline or carriage return character. The newline character is not written to the physical file; it is assumed to follow the data of the record. However, if a record contains only a newline character, LE/VSE C Run-Time writes a record containing a single blank to the file to represent an empty record. On input, the blank is read in as a newline.

Once a record has been written, you cannot change its length. If you try to shorten a logical record by updating it with a shorter record, LE/VSE C Run-Time completes the record with blank padding. If you try to lengthen a record by updating it with more data than it can hold, LE/VSE C Run-Time truncates the new data. The only instance in which this does not happen is when you extend an empty record so that it contains a single byte. Any data beyond the single byte is truncated.

Truncation vs. Splitting

If you try to write more data to a record than LE/VSE C Run-Time allows, and the file you are writing to is not one of the standard streams (the defaults, or those redirected by `freopen()` or through the JCL `EXEC` statement `PARM` field), output is cut off at the record boundary and the remaining bytes are discarded. LE/VSE C Run-Time does not count the discarded characters as characters that have been written out successfully.

In all truncation cases, the `SIGIOERR` signal is raised if the action for `SIGIOERR` is not `SIG_IGN`. The user error flag is set so that `ferror()` will return `TRUE`. For more information about `SIGIOERR`, `ferror()`, and other I/O-related debugging tools, see Chapter 14, “Debugging I/O Programs,” on page 131. LE/VSE C Run-Time continues to discard new output until you complete the current record by writing a newline or carriage return character, close the file, or change the file position.

If you are writing to one of the standard streams, attempting to write more data than a record can hold results in the data being split across multiple records.

Writing to Record I/O Files

`fwrite()` is the only interface allowed for writing to a file opened for record I/O. To open a file for record I/O, the `fopen()` mode string must contain `type=record`. Only one record is written at a time. If you attempt to write more new data than a full record can hold or you try to update a record with more data than it currently has, LE/VSE C Run-Time truncates your output at the record boundary. When LE/VSE C Run-Time performs a truncation, it sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`.

When you update a record, you can update less than the full record. The remaining data that you do not update is left untouched in the file.

When you are writing new records to a fixed-record I/O file, if you try to write a short record, LE/VSE C Run-Time pads the record with nulls out to LRECL.

At the completion of an `fwrite()`, the file position is at the start of the next record. For new data, the block is flushed out to the system as soon as it is full.

Flushing Buffers

You can use the library function `fflush()` to flush streams to the system. For more information about `fflush()`, see *LE/VSE C Run-Time Library Reference*.

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of streams. If you call one LE/VSE C Run-Time program from another LE/VSE C Run-Time program by using the ANSI `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller.

`fflush()` is ignored for files on all non-disk devices.

Updating Existing Records

Calling fflush() while you are updating flushes the updates out to the system. If you call fflush() when you are in the middle of updating a record, LE/VSE C Run-Time writes the partially updated record out to the system. A subsequent write continues to update the current record.

Reading Updated Records

If you have a file open for read at the same time that the file is open for update in the same application (see "Simultaneous Reader/Writer" on page 64), you will be able to see the new data if you call fflush() to refresh the contents of the input buffer, as in the following example:

EDCXGOS3

```
 /* EDCXGOS3
    This example demonstrates how updated records are read.
  */

#include <stdio.h>

int main()
{
   FILE *fp1, *fp2;
   int rc1, rc2, rc3, rc4;

   remove("'a.b'");          /* Ensure file is deleted */
   /* Create an empty file and open it for update... */
   if ((fp1 = fopen("'a.b'","w+,recfm=u,blksize=4000")) == NULL)
      perror("Error opening file for write");
   if ((freopen("'a.b'", "r+,recfm=u,blksize=4000", fp1)) == NULL)
      perror("Error opening file for read/write");
 fprintf(fp1,"first record");

   /* Open a simultaneous reader... */
   if ((fp2 = fopen("'a.b'","r,recfm=u,blksize=4000")) == NULL)
      perror("Error opening file for read");

   /* Following gets EOF since fp1 has not completed first
      line of output so nothing will be flushed to file yet */
   rc1 = fgetc(fp2);
   if (rc1 == EOF) puts("At EOF");
      else printf("Read char: %c instead of EOF...\n", rc1);

   fputc('\n', fp1);         /* This will complete second line */
   fflush(fp1);              /* Ensures data is flushed to file */

   rc2 = fgetc(fp2);         /* Gets 'f', 1st char of first record */
   if (rc2 == EOF) puts("At EOF but shouldn't be...");
      else printf("Read char: %c\n", rc2);
```

Figure 8. Example of Reading Updated Records (Part 1 of 2)

```
   rewind(fp1);

   fprintf(fp1,"some updates\n");

   rc3 = fgetc(fp2);         /* Gets 'i', does not know about update */
   if (rc3 == EOF) puts("At EOF but shouldn't be...");
      else printf("Read char: %c\n", rc3);

   fflush(fp1);              /* Ensures update makes it to file */

   fflush(fp2);              /* This updates reader's buffer */

   rc4 = fgetc(fp2);         /* Gets 'm', 3rd char of updated record */
   if (rc4 == EOF) puts("At EOF but shouldn't be...");
      else printf("Read char: %c\n", rc4);

   exit(0);
}
```

Figure 8. Example of Reading Updated Records (Part 2 of 2)

Simultaneous Reader/Writer

If, within a single application, a file is first opened for update and then, without an intermediate close, is opened for read one or more times, then the situation is known as *simultaneous read/write*. However, the following restrictions must be observed:

- The file must exist prior to being opened for update and no `w`-type open modes are allowed on the `fopen()` or `freopen()` call. (The file might be empty—refer to "EDCXGOS3" on page 63 for an example of how to create an empty file.)
- The file must be a SAM file (it cannot be a SAM ESDS file).
- The file must be opened for update before the first (or only) open for read.

The "writer" (that is, the stream opened for update) is known as the *simultaneous writer*. There can be only one such writer.

Each "reader" (that is, each stream opened for read) is known as a *simultaneous reader*. There can be any number of readers.

feof(): The behavior of `feof()` in this situation is illustrated in the following scenario:

1. A *simultaneous reader* reads to end of file (that is, until `feof()` returns `TRUE`).
2. The *simultaneous writer* appends to the file.
3. `feof()` remains `TRUE` until one of the following occurs:
   - The next read (for example, using `fread()`).

     -or-
   - A call to `fflush()` by the *simultaneous writer*.

Writing New Records

Binary Streams

LE/VSE C Run-Time treats line buffering and full buffering the same way for binary files.

If the file has a variable length or undefined record format, `fflush()` writes the current record out. This may result in short records. In blocked files, this means that the block is written to disk, and subsequent writes are to a new block. For fixed files, no incomplete records are flushed.

For single-volume disk files in `FBS` format, `fflush()` flushes complete records in an incomplete block out to the file. For all other types of `FBS` files, `fflush()` does not flush an incomplete block out to the file.

For files in `FB` format, `fflush()` always flushes out all complete records in the current block. For disk files, new completed records are added to the end of the flushed block if it is short. For other files, any new record will start a new block.

Text Streams

- Line-Buffered Streams

  `fflush()` has no effect on line-buffered text files, because LE/VSE C Run-Time writes all records to the system as they are completed. All incomplete new records remain in the buffer.
- Fully Buffered Streams

  Calling `fflush()` flushes all completed records in the buffer, that is, all records ending with a newline or carriage return (or form feed character, if you are using ASA), to the system. LE/VSE C Run-Time holds any incomplete record in the buffer until you complete the record or close the file.

For ASA text files, if a flush occurs while an ASA character that indicates more than one newline is being updated, the remaining newlines will be discarded and a read will continue at the first data character. For example, if `'\n\n\n'` is updated to be `'\n\n'` and a flush occurs, then a `'0'` will be written out in the `ASA` character position.

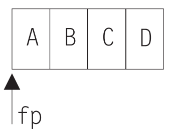
Record I/O

LE/VSE C Run-Time treats line buffering and full buffering the same way for record I/O. For files in `FB` format, calling `fflush()` writes all records in the buffer to the system. For single-volume disk files in `FBS` format, `fflush()` will flush complete records in an incomplete block out to the file. For all other types of `FBS` files, `fflush()` will not flush an incomplete block out to the file. For all other formats, calling `fflush()` has no effect, because `fwrite()` has already written the records to disk.

`ungetc()` Considerations

`ungetc()` pushes characters back onto the input stream for binary and text files. `ungetc()` handles only single-byte characters. You can use it to push back as many as four characters onto the `ungetc()` buffer. For every character pushed back with `ungetc()`, `fflush()` backs up the file position by one character and clears all the pushed-back characters from the stream. Backing up the file position may end up going across a record boundary. Remember that for text files, LE/VSE C Run-Time counts the newlines added to the records as single-byte characters when it calculates the file position.

For example, given the stream
you can run the following code fragment:



```
fgetc(fp);       /* Returns A and puts the file position at    */
                 /*    the beginning of the character B        */
ungetc('Z',fp);  /* Logically inserts Z ahead of B             */
fflush(fp);      /* Moves the file position back by one to A,  */
                 /*    removes Z from the logical stream       */
```

If you want `fflush()` to ignore `ungetc()` characters, you can set the _EDC_COMPAT environment variable. See Chapter 21, "Using Environment Variables," on page 219 for more information.

Repositioning within Files

You can use the following library functions to help you position within a SAM file:
- `fseek()`
- `ftell()`
- `fgetpos()`
- `fsetpos()`
- `rewind()`

See *LE/VSE C Run-Time Library Reference* for more information on these library functions.

Opening a file with `fopen()` and specifying the `noseek` parameter disables all of these library functions except `rewind()`. A call to `rewind()` causes the file to be reopened, unless the file is a tape file opened for write-only. In this case, `rewind()` sets `errno` and raises `SIGIOERR` (if `SIGIOERR` is not set to `SIG_IGN`, which is its default).

Note: If a SAM ESDS file is opened for `r`, `rb`, `w` or `wb`, with `noseek` specified and `DISP=(,DELETE)` is specified on the `DLBL` statement, calling `rewind()` will cause the file to be deleted.

Calling any of these functions flushes all complete and updated records out to the system. If a repositioning operation fails, LE/VSE C Run-Time attempts to restore the original file position and treats the operation as a call to `fflush()`, except that it does not account for the presence of `ungetc()` or `ungetwc()` characters, which are lost. After a successful repositioning operation, `feof()` always returns `0`, even if the position is just after the last byte of data in the file.

The `fsetpos()` and `fgetpos()` library functions are generally more efficient than `ftell()` and `fseek()`. The `fgetpos()` function can encode the current position into a structure that provides enough room to hold the system position as well as position data specific to C. The `ftell()` function must encode the position into a single word of storage, which it returns. This compaction forces `fseek()` to calculate certain position information specific to C at the time of repositioning. For variable-format binary files, you can choose to have `ftell()` return relative byte offsets. In previous releases, `ftell()` returned only encoded offsets, which contained the relative block number. Since you cannot calculate the block number from a relative byte offset in a variable-format file, `fseek()` may have to read through the file to get to the new position. `fsetpos()` has system position information available within the `fpos_t` structure and can generally reposition directly to the desired location.

You can use the `fseek()`, `ftell()`, `fgetpos()` and `fsetpos()` functions with single-volume disk files only. The files must have no more than 255 blocks per track (CKD devices) or 255 blocks per CI (VSAM-managed SAM and FBA devices).

You can use the `rewind()` function with disk or tape files only. When `noseek` is not specified on `fopen()` for a multivolume disk file, calling `rewind()` will cause the file to be closed and reopened.

`ungetc()` Considerations

For binary and text files, the library functions `fgetpos()` and `ftell()` take into account the number of characters you have pushed back onto the input stream with `ungetc()`, and adjust the file position accordingly. `ungetc()` backs up the file position by a single byte each time you call it. For text files, LE/VSE C Run-Time counts the newlines added to the records as single-byte characters when it calculates the file position.

If you make so many calls to `ungetc()` that the logical file position is before the beginning of the file, the next call to `ftell()` or `fgetpos()` fails.

When you are using `fseek()` with an *`origin`* of `SEEK_CUR`, the starting point for the reposition also accounts for the presence of `ungetc()` characters and compensates as `ftell()` and `fgetpos()` do.

If you want `fgetpos()` and `fseek()` to ignore `ungetc()` characters, you can set the _EDC_COMPAT environment variable. See Chapter 21, “Using Environment Variables,” on page 219 for details. `ftell()` is not affected by the setting of _EDC_COMPAT.

How Long `fgetpos()` and `ftell()` Values Last

As long as you do not re-create a file or shorten logical records, you can rely on the values returned by `ftell()` and `fgetpos()`, even across program boundaries and calls to `fclose()`. (Calling `fopen()` or `freopen()` with any of the `w` modes re-creates a file.) Using `ftell()` and `fgetpos()` values that point to information deleted or re-created results in undefined behavior. For more information about shortening records, see “Writing to Variable-Format Text Files” on page 60.

Using `fseek()` and `ftell()` in Binary Files

With binary files, `ftell()` returns two types of positions:
- Relative byte offsets
- Encoded offsets

Relative Byte Offsets

You get byte offsets by default when you are seeking or positioning in fixed-format binary files. You can also use byte offsets on a variable or undefined format file opened in binary mode with the `byteseek` parameter specified on the `fopen()` or `freopen()` function call. You can specify `byteseek` to be the default for `fopen()` calls by setting the environment variable `_EDC_BYTE_SEEK` to `Y`. See Chapter 21, “Using Environment Variables,” on page 219 for information on how to set environment variables.

You do not need to acquire an offset from `ftell()` to seek to a relative position; you may specify a relative offset to `fseek()` with an *origin* of `SEEK_SET`. However, you cannot specify a negative offset to `fseek()` when you have specified `SEEK_SET`, because a negative offset would indicate a position before the beginning of the file. Also, you cannot specify a negative offset with *origin*s of `SEEK_CUR` or `SEEK_END` such that the resulting file position would be before the beginning of the file. If you specify such an offset, `fseek()` fails.

If your file is not opened read-only, you can specify a position that is beyond the current `EOF`. In such cases, a new end-of-file position is created; null characters are automatically added between the old `EOF` and the new `EOF`.

`fseek()` support of byte offsets in variable-format files generally requires reading all records from the *origin* to the new position. The impact on performance is greatest if you open an existing file for append in `byteseek` mode and then call `ftell()`. In this case, `ftell()` has to read from the beginning of the file to the current position to calculate the required byte offset. Support for byteseeking is intended to ease portability from other platforms. If you need better performance, consider using `ftell()`-encoded offsets, discussed in the next section.

Encoded Offsets

If you do not specify the `byteseek` parameter and you set the `_EDC_BYTE_SEEK` variable to `N`, any variable- or undefined-format binary file gets encoded offsets from `ftell()`. This keeps this release of LE/VSE C Run-Time compatible with code generated by old releases of C/370.

Encoded offsets are values representing the block number and the relative byte within that block, all within one `long int`. Because LE/VSE C Run-Time does not document its encoding scheme, you cannot rely on any encoded offset not returned by `ftell()`, except `0`, which is the beginning of the file. This includes encoded offsets that you adjust yourself (for example, with addition or subtraction). When you call `fseek()` with the *origin* `SEEK_SET`, you must use either `0` or an encoded offset returned from `ftell()`. For *origin*s of `SEEK_CUR` and `SEEK_END`, however, you specify relative byte offsets. If you want to seek to a certain relative byte offset, you can use `SEEK_SET` with an offset of `0` to rewind the file to the beginning, and then you can use `SEEK_CUR` to specify the desired relative byte offset.

In earlier releases of C, `ftell()` could determine position only for files with no more than 131,071 blocks. In the new design, this number increases depending on the block size. From a maximum block size of 32,760, every time this number decreases by half, the number of blocks that can be represented doubles.

If your file is not opened read-only, you can use `SEEK_CUR` or `SEEK_END` to specify a position that is beyond the current `EOF`. In such cases, a new end-of-file position is created; null characters are automatically added between the old `EOF` and the new `EOF`. For `SEEK_SET`, because you are restricted to using offsets returned by `ftell()`, any offset that indicates a position outside the current file is invalid and causes `fseek()` to fail.

Using `fseek()` and `ftell()` in Text Files (ASA and Non-ASA)

In text files, `ftell()` produces only encoded offsets. It returns a `long int`, in which the block number and the byte offset within the block are encoded. You cannot rely on any encoded offset not returned by `ftell()` except `0`. This includes encoded offsets that you adjust yourself (for example, with addition or subtraction).

When you call `fseek()` with the *origin* `SEEK_SET`, you must use an encoded offset returned from `ftell()`. For *origin*s of `SEEK_CUR` and `SEEK_END`, however, you specify relative byte offsets. If you want to seek to a certain relative byte offset, you can use `SEEK_SET` with an offset of `0` to rewind the file to the beginning, and then you can use `SEEK_CUR` to specify the desired relative byte offset. LE/VSE C Run-Time counts newline characters and skips to the next record each time it reads one.

Unlike binary files you cannot specify offsets for `SEEK_CUR` and `SEEK_END` that set the file position past the end of the file. Any offset that indicates a position outside the current file is invalid and causes `fseek()` to fail.

In earlier releases, `ftell()` could determine position only for files with no more than 131071 blocks. In the new design, this number increases depending on the block size. From a maximum block size of 32760, every time this number decreases by half, the number of blocks that can be represented doubles. A maximum block size of 65528, instead of 32760, can be used for disk files when the appropriate hardware-device support is available.

Repositioning flushes all updates before changing position. An invalid call to `fseek()` is now always treated as a flush. It flushes all updated records or all complete new records in the block, and leaves the file position unchanged. If the flush fails, any characters in the `ungetc()` buffer are lost. If a block contains an incomplete new record, the block is saved and will be completed by another write or by closing the file.

Using `fseek()` and `ftell()` in Record Files

For files opened with `type=record`, `ftell()` returns relative record numbers. The behavior of `fseek()` and `ftell()` is similar to that when you use relative byte offsets for binary files, except that the unit is a record rather than a byte. For example,

```
fseek(fp,-2,SEEK_CUR);
```

seeks backward two records from the current position.

```
fseek(fp,6,SEEK_SET);
```

seeks to relative record 6. You do not need to get an offset from `ftell()`.

You cannot seek past the end or before the beginning of a file.

The first record of a file is relative record 0.

Porting Old C Code that Uses `fseek()` or `ftell()`

The LE/VSE C Run-Time encoding scheme used by `ftell()` in non-`byteseek` mode is different from that used in older versions of the C/370 compiler.

- If your code obtains `ftell()` values and passes them to `fseek()`, the change to the encoding scheme should not affect your application. On the other hand, your application may not work if you have saved encoded `ftell()` values in a file and your application reads in these encoded values to pass to `fseek()`. For non-record I/O files, you can set the environment variable _EDC_COMPAT with the `ftell()` encoding set to tell LE/VSE C Run-Time that you have old `ftell()` values. Files opened for record I/O do not support old `ftell()` values saved across the program boundary.
- In previous versions, the `fseek()` support for the `ftell()` encoding scheme inadvertently supported seeking from `SEEK_SET` with a byte offset up to 32K.

This will no longer be supported. Users of this support will have to change to `byteseek` mode. You can do this without changing your source code; just use the `_EDC_BYTE_SEEK` environment variable.

Closing Files

Use the `fclose()` library function to close a file. LE/VSE C Run-Time automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend. See *LE/VSE C Run-Time Library Reference* for more information on this library function.

For files opened in fixed binary mode, incomplete records will be padded with null characters when you close the file.

For files opened in variable binary mode, incomplete records are flushed to the system. In a spanned file, closing a file can cause a zero-length segment to be written. This segment will still be part of the non-zero-length record. For files opened in undefined binary mode, any incomplete output is flushed on close.

Closing files opened in text mode causes any incomplete new record to be completed with a newline character. All records not yet flushed to the file are written out when the file is closed.

For files opened for record I/O, closing causes all records not yet flushed to the file to be written out.

Renaming and Removing Files

You can remove or rename a SAM file using the `remove()` or `rename()` library functions, respectively. `rename()` and `remove()` both accept file IDs. `rename()` does not accept `DLBL`/`TLBL`-name or logical unit specification, but `remove()` does.

When using `remove()` with the `DLBL`/`TLBL`-name specification, the `DLBL` or `TLBL` cannot be removed.

When using `remove()` with the file ID specification:
- The Partition Temporary Labels *only* will be searched for a `DLBL` or `TLBL` matching the file ID.
- SAM ESDS files will be deleted from the default catalog specified by the `IJSYSUC DLBL` if present, or the master catalog otherwise.

When using `rename()` with the file ID specification:
- The Partition Temporary Labels *only* will be searched for a `DLBL` matching the file ID.

  Note: The file ID on the `DLBL` statement is not updated.
- SAM ESDS files will be renamed in the default catalog specified by the `IJSYSUC DLBL` if present, or the master catalog otherwise.

`fldata()` Behavior

- Any of the `__recfm` bits may be set on for SAM files; see the example below.

- The *filename* field is fully qualified and includes quotation marks if you have opened the file by its file ID. If you have opened it by DLBL/TLBL-name and/or logical unit, the name returned will be the same as the name provided on the call to the fopen() or freopen() function, including the dd: prefix.
- All SAM files set on __dsorgPS.
- The __dsorgMem, __dsorgVSAM, __dsorgPO and __dsorgPDSmem bits are never set on for SAM files. The fields __vsamtype, __vsamkeylen, and __vsamRKP are used only for VSAM; they are not set for SAM files.
- Valid devices are: __DISK, __TAPE, __PRINTER, and __OTHER.
- __blksize is the physical block size read from or written to the device, including all control information.
- __maxreclen is the maximum number of data bytes in each logical record, excluding all control information but including ASA print-control characters or machine control codes if specified.

```
struct __fileData {
     unsigned int   __recfmF      : 1,  /* if mapping ==> Fixed               */
                    __recfmV      : 1,  /* if mapping ==> Variable            */
                    __recfmU      : 1,  /* if mapping ==> Undefined           */
                    __recfmS      : 1,  /* if mapping ==> Spanned or Standard */
                    __recfmBlk    : 1,  /* if mapping ==> Blocked             */
                    __recfmASA    : 1,  /* only if Text mode and ASA          */
                    __recfmM      : 1,  /* only if machine print-cntrl codes  */
                    __dsorgPO     : 1,  /* see above                          */
                    __dsorgPDSmem : 1,  /* see above                          */
                    __dsorgPDSdir : 1,  /* N/A; never on                      */
                    __dsorgPS     : 1,  /* see above                          */
                    __dsorgConcat : 1,  /* N/A; never on                      */
                    __dsorgMem    : 1,  /* N/A; never on                      */
                    __dsorgHiper  : 1,  /* N/A; never on                      */
                    __dsorgTemp   : 1,  /* only if file created by tmpfile()  */
                    __dsorgVSAM   : 1,  /* never on                           */
                    __reserve1    : 1,  /*                                    */
                    __openmode    : 2,  /* normal setting                     */
                    __modeflag    : 4,  /* normal setting                     */
                    __reserve2    : 9,  /*                                    */
                                                                              */
     char           __device;           /* see above                          */
     unsigned long  __blksize,          /* block size (may include BDW, RDWs) */
                    __maxreclen;        /* data length of records (includes   */
                                        /* ASA character if if this is an     */
                                        /* ASA file)                          */
     unsigned short __vsamtype;         /* not used                           */
     unsigned long  __vsamkeylen;       /* not used                           */
     unsigned long  __vsamRKP;          /* not used                           */
     char *         __dsname;           /* filled in for SAM __DISK files     */
                                        /* with file-ID.                      */
                                        /* The name is fully qualified but    */
                                        /* contains no quotation marks.       */
                                        /* The name is always uppercased.     */
                                        /* For non-disk devices, this field   */
                                        /* is set to NULL.                    */

     unsigned int   __reserve4;         /*                                    */
};
```

Chapter 9. Performing VSE/Librarian I/O Operations

This chapter describes using VSE/Librarian I/O under VSE batch.

VSE/Librarian I/O supports text, binary, and record I/O in fixed (`F`) record format and binary I/O in undefined (`U`) record format.

VSE/Librarian I/O is not supported under CICS.

See Chapter 6, "LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)," on page 29 for information about using wide-character I/O with LE/VSE C Run-Time.

Opening Files

To open a VSE/Librarian sublibrary member, use the standard C `fopen()` or `freopen()` library functions. These are described in general terms in *LE/VSE C Run-Time Library Reference*. Details about them specific to all LE/VSE C Run-Time I/O are discussed in Chapter 3, "Opening Files," on page 15. This section describes considerations for using `fopen()` and `freopen()` with VSE/Librarian sublibrary members.

Using `fopen()` or `freopen()`

Files are opened with a call to `fopen()` or `freopen()` in the format: `fopen("`*`filename`*`", "mode")`.

Filenames for VSE/Librarian Sublibrary Members

The syntax for the *filename* argument on your `fopen()` or `freopen()` call is shown in the following diagram:

```
►►──DD:──┬────────────┬──(──member.type──)──────────────────────────►◄
         └─lib.sublib─┘
```

lib.sublib
: Is a valid Librarian sublibrary name.

member.type
: Specifies the sublibrary member name and type.

If the sublibrary name is omitted, the default libraries specified by the `LIBDEF` JCL statement will be searched. The sublibraries searched will depend on the member type as follows:

PHASE
: Sublibraries in the PHASE search chain (`LIBDEF PHASE,SEARCH`) are searched for input. Phases can only be read.

PROC
: Sublibraries in the PROCEDURE search chain (`LIBDEF PROC,SEARCH`) are searched for input.

OBJ
: Sublibraries in the OBJECT search chain (`LIBDEF OBJ,SEARCH`) are searched for input.

DUMP
: Sublibraries in the DUMP search chain (`LIBDEF DUMP,CATALOG`) specifies the sublibraries to write a DUMP to.

Other Sublibraries in the SOURCE search chain (`LIBDEF SOURCE,SEARCH`) are searched for input.

When a VSE/Librarian member is created and no sublibrary is specified, the member is created in the first sublibrary in the search chain.

File Modes Supported for VSE/Librarian I/O

The only file modes supported for VSE/Librarian I/O are `r`, `rb`, `w`, `wb`, `a` and `ab`.

`fopen()` and `freopen()` Parameters

The following table lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are allowed and applicable for VSE/Librarian I/O, and lists the option values that are valid for the applicable ones. Detailed descriptions of these options follow the table.

Table 19. Parameters for the `fopen()` and `freopen()` Functions for VSE/Librarian I/O

Parameter	Allowed?	Applicable?	Notes
`recfm=`	Yes	Yes	Must be fixed or undefined. See the parameter list below for details.
`lrecl=`	Yes	Yes	Must be `80` if record format is fixed. Ignored otherwise.
`blksize=`	Yes	Yes	Any positive integer up to 65528 is valid. Must be a multiple of LRECL if fixed record format. See the parameter list below for details.
`space=`	Yes	No	Not used for VSE/Librarian I/O.
`type=`	Yes	Yes	May be omitted. If you do specify it, `type=record` is the only valid value. See the parameter list below for details.
`acc=`	Yes	No	Not used for VSE/Librarian I/O.
`password=`	Yes	No	Not used for VSE/Librarian I/O.
`asis`	Yes	No	Ignored.
`byteseek`	Yes	Yes	Used for binary files to specify that the seeking functions should use relative byte offsets instead of encoded offsets. See the parameter list below for details.
`noseek`	Yes	Yes	Used to disable seeking functions for improved performance. See the parameter list below for details.
`OS`	Yes	No	Ignored.
`rewind=`	Yes	Yes	Not used for VSE/Librarian I/O.
`dsn=`	Yes	No	Ignored.

`recfm=`
Must be either fixed unblocked (ASA print-control characters or machine control codes can be specified) or undefined. `recfm=U` is only allowed when the file is opened for binary processing. `type=record` is only allowed when `recfm=F` is specified. Specifying `recfm=*` is not supported under LE/VSE C Run-Time.

VSE/Librarian members with a type of `DUMP` must be `recfm=U` while members with a type of either `OBJ` or `PROC` must be `recfm=F`.

`lrecl=`
Must be `80` if record format is fixed. Ignored otherwise.

blksize=
: For fixed record format files, this parameter is used to specify the buffer size for VSE/Librarian I/O to allow multiple records to be read and/or written at a time. The block size must be a multiple of LRECL.

  For undefined record format files, the value specified determines the size of the I/O buffer used by the VSE/Librarian.

  If omitted, BLKSIZE defaults to `4000`.

space=
: This parameter is not valid for VSE/Librarian I/O. If you specify it, LE/VSE C Run-Time ignores it.

type=
: You can omit this parameter. If you specify it, the only valid value for VSE/Librarian I/O is `type=record`, which opens a file for record I/O. `type=record` is only allowed when `recfm=F` is specified and the file is opened for binary processing.

acc=
: This parameter is not valid for VSE/Librarian I/O. If you specify it, LE/VSE C Run-Time ignores it.

password=
: This parameter is not valid for VSE/Librarian I/O. If you specify it, LE/VSE C Run-Time ignores it.

asis
: If you specify this parameter, LE/VSE C Run-Time ignores it.

byteseek
: When you specify this parameter and open a file in binary mode, all repositioning functions (such as `fseek()` and `ftell()`) use relative byte offsets from the beginning of the file instead of encoded offsets. To have the `byteseek` parameter set as the default for all your calls to `fopen()` or `freopen()`, you can set the environment variable `_EDC_BYTE_SEEK` to `Y`. See Chapter 21, "Using Environment Variables," on page 219 for more information.

noseek
: Specifying this parameter on the `fopen()` call disables the repositioning functions `ftell()`, `fseek()`, `fgetpos()`, and `fsetpos()` for as long as the file is open. When you have specified `noseek` and have opened a disk file for read only, the only repositioning function allowed on the file is `rewind()`, if the device supports rewinding. Otherwise, a call to `rewind()` sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`. Calls to `ftell()`, `fseek()`, `fsetpos()`, or `fgetpos()` return `EOF`, set `errno`, and set the stream error flag on.

  The use of the `noseek` parameter may improve performance when you are reading and writing files.

  Note: If you specify the `noseek` parameter when you open a file for writing, you must specify `noseek` on any subsequent `fopen()` call that simultaneously opens the file for reading; otherwise, you will get undefined behavior.

OS
: If you specify this parameter, LE/VSE C Run-Time ignores it.

rewind=
: This parameter is not valid for VSE/Librarian I/O. If you specify it, LE/VSE C Run-Time ignores it.

dsn=
This parameter is not valid for VSE/Librarian I/O. If you specify it, LE/VSE C Run-Time ignores it.

Buffering

Same as for SAM I/O. (See “Buffering” on page 56.)

Unbuffered I/O is not supported for VSE/Librarian files.

Reading from Files

Same as for SAM I/O. (See “Reading from Files” on page 56.)

Writing to Files

Same as for SAM I/O. (See “Writing to Files” on page 58.)

Flushing Buffers

The `fflush()` function will write out any completed records.

Repositioning within Files

Same as for SAM I/O. (See “Repositioning within Files” on page 66.)

Closing Files

Same as for SAM I/O. (See “Closing Files” on page 70.)

Renaming and Removing Files

You can use `remove()` and `rename()` on individual members of VSE/Librarian sublibraries only. Neither function can be used on entire sublibraries.

When using `remove()`, if the library/sublibrary is specified, the member is deleted from that sublibrary. If not, the member is deleted from the first sublibrary in the chain.

When using `rename()`:
- If the library/sublibrary is specified as part of the “old” name, then the library/sublibrary specified as part of the “new” name must be the same, or be omitted. The member will be renamed in the specified sublibrary.
- If the library/sublibrary is *not* specified as part of the “old” name, then the library/sublibrary should not be specified as part of the “new” name either. The member will be renamed in the first sublibrary in the chain.

`fldata()` Behavior

- Refer to the example below for information about which of the `__recfm` bits that may be set on for VSE/Librarian files.
- The *filename* field is identical to the *filename* specified on the call to `fopen()`.
- The `__dsorgPO` and `__dsorgPDSmem` fields are always set on when you are reading a member of a VSE/Librarian sublibrary.

```
struct __fileData {
     unsigned int   __recfmF      : 1,  /* if mapping ==> Fixed              */
                    __recfmV      : 1,  /* N/A; never on                     */
                    __recfmU      : 1,  /* if mapping ==> Undefined          */
                    __recfmS      : 1,  /* N/A; never on                     */
                    __recfmBlk    : 1,  /* N/A; never on                     */
                    __recfmASA    : 1,  /* only if Text mode and ASA         */
                    __recfmM      : 1,  /* only if machine print-cntrl codes */
                    __dsorgPO     : 1,  /* see above                         */
                    __dsorgPDSmem : 1,  /* see above                         */
                    __dsorgPDSdir : 1,  /* N/A; never on                     */
                    __dsorgPS     : 1,  /* see above                         */
                    __dsorgConcat : 1,  /* N/A; never on                     */
                    __dsorgMem    : 1,  /* N/A; never on                     */
                    __dsorgHiper  : 1,  /* N/A; never on                     */
                    __dsorgTemp   : 1,  /* only if file created by tmpfile() */
                    __dsorgVSAM   : 1,  /* never on                          */
                    __reserve1    : 1,  /*                                   */
                    __openmode    : 2,  /* normal setting                    */
                    __modeflag    : 4,  /* normal setting                    */
                    __reserve2    : 9,  /*                                   */
                                                                             */
     char           __device;           /* __DISK                            */
     unsigned long  __blksize,          /* block size                        */
                    __maxreclen;        /* data length of records (includes  */
                                        /* ASA character if if this is an    */
                                        /* ASA file)                         */
     unsigned short __vsamtype;         /* not used                          */
     unsigned long  __vsamkeylen;       /* not used                          */
     unsigned long  __vsamRKP;          /* not used                          */
     char *         __dsname;           /* this is the full name of the VSE  */
                                        /* Librarian file in the format      */
                                        /* dd:lib.sublib(member.type)        */

     unsigned int   __reserve4;         /*                                   */
};
```

Chapter 10. Performing VSAM I/O Operations

This chapter outlines the use of Virtual Storage Access Method (VSAM) data sets in LE/VSE C Run-Time. Three I/O processing modes for VSAM data sets are available in LE/VSE C Run-Time:
- Record
- Text Stream
- Binary Stream

Because VSAM is a record-based access method, record mode is the logical processing mode and is specified by coding the `type=record` keyword parameter on the `fopen()` function call. LE/VSE C Run-Time also provides limited support for VSAM text streams and binary streams. Because of the record-based nature of VSAM, this chapter is organized differently from the other chapters in this section. The focus of this chapter is on record I/O. Only those aspects of text and binary I/O that are specific to VSAM are discussed, at the end of the chapter.

VSAM I/O is not supported under CICS, except through the CICS command level interface.

For more information about the facilities of VSAM, see "Where to Find More Information" on page xxi.

See Chapter 6, "LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)," on page 29 for information about using wide-character I/O with LE/VSE C Run-Time.

VSAM Types (Data Set Organization)

There are three types of VSAM data sets supported by LE/VSE C Run-Time, all of which are held on disk devices.
- Key-Sequenced Data Set (KSDS) is used when a record is accessed through a key field within the record (for example, an employee directory file where the employee number can be used to access the record). KSDS also supports sequential access. Each record in a KSDS must have a unique key value.
- Entry-Sequenced Data Set (ESDS) is used for data that is primarily accessed in the order it was created (or the reverse order). It supports direct access by Relative Byte Address (RBA), and sequential access.
- Relative Record Data Set (RRDS) is used for data in which each item has a particular number, and the relevant record is accessed by that number (for example, a telephone system with a record associated with each number). It supports direct access by Relative Record Number (RRN), and sequential access.

In addition to the primary VSAM access described above, for KSDS and ESDS, there is also direct access by one or more additional key fields within each record. These additional keys can be unique or non-unique; they are called an alternate index (AIX).

Note: VSAM Linear Data Sets are not supported in LE/VSE C Run-Time I/O.

Access Method Services

Access Method Services are generally known by the name IDCAMS on VSE. For more information, see *Access Method Services* manual.

Before a VSAM data set is used for the first time, its structure is defined to the system by the Access Method Services `DEFINE CLUSTER` command. This command defines the type of VSAM data set, its structure, and the space it requires.

Before a VSAM alternate index is used for the first time, its structure is defined to the system by the Access Method Services `DEFINE ALTERNATEINDEX` command. To enable access to the base cluster records through the alternate index, use the `DEFINE PATH` command. Finally, to build the alternate index, use the `BLDINDEX` command.

Once you have built the alternate index, you call `fopen()` and specify the `PATH` in order to access the base cluster through the alternate index. Do not use `fopen()` to access the alternate index itself.

Note: You cannot use the `BLDINDEX` command on an empty base cluster.

Choosing VSAM Data Set Types

When you plan your program, you must first decide the type of data set to use. Figure 9 on page 81 shows you the possibilities available with the types of VSAM data sets.

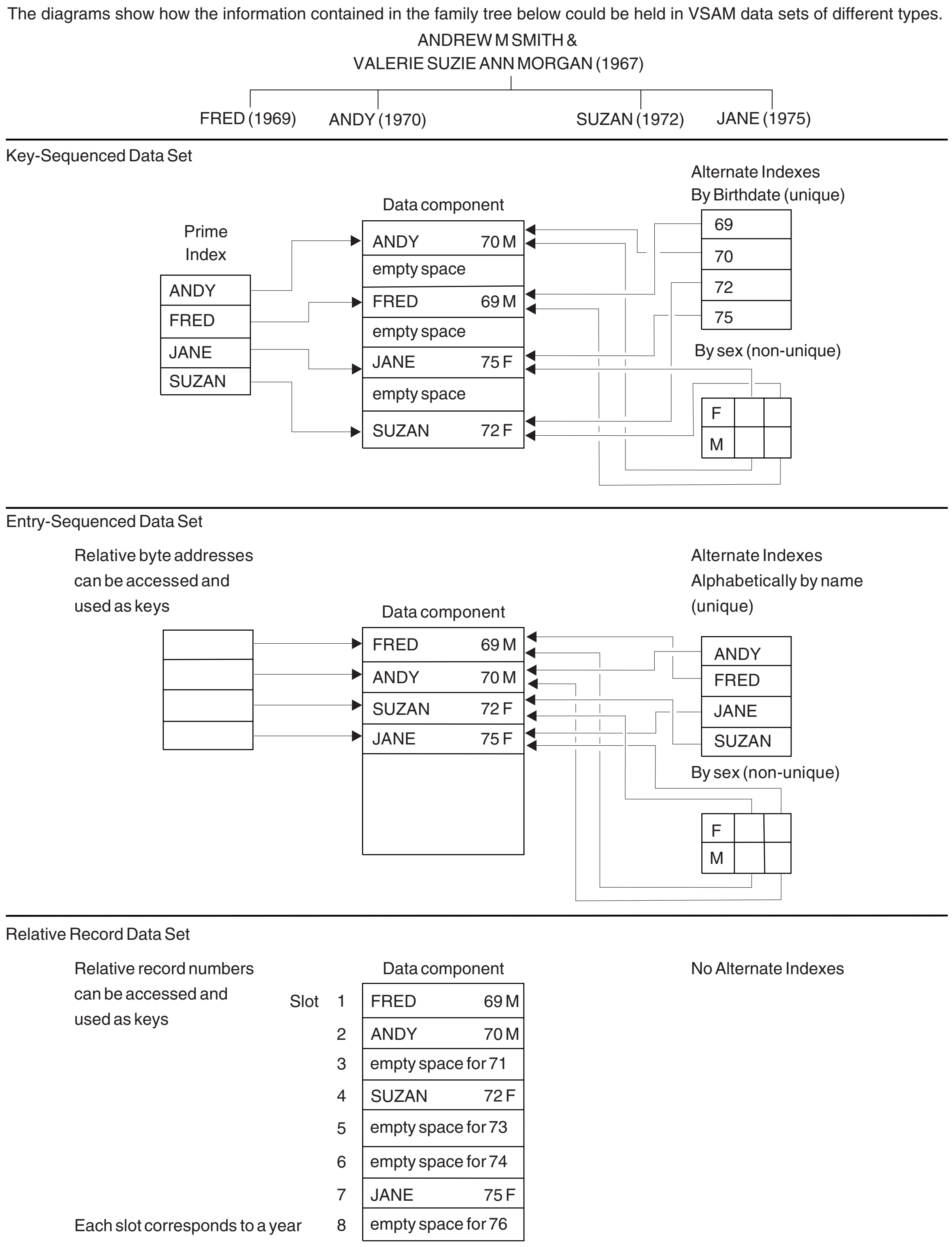


Figure 9. Types and Advantages of VSAM Data Sets

When choosing the VSAM data set type, you should base your choice on the most common sequence in which you require data. You should follow a procedure similar to the one suggested below to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and its primary access.
   - sequentially—favors ESDS.
   - by key—favors KSDS.
   - by number—favors RRDS.
2. Determine whether you require access through an alternate index path. These are only supported on KSDS and ESDS. If you do, determine whether the alternate index is to have unique or non-unique keys. You should keep in mind that making an assumption that all future records will have unique keys may not be practical, and an attempt to insert a record with a non-unique key in an index that has been created for unique keys causes an error.
3. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported.

Keys, RBAs and RRNs

All VSAM data sets have keys associated with their records. For KSDS, KSDS AIX, and ESDS AIX, the key is a defined field within the logical record. For ESDS, the key is the *relative byte address* (RBA) of the record. For RRDS, the key is a *relative record number* (RRN).

Keys for Indexed VSAM Data Sets: For KSDS, KSDS AIX, and ESDS AIX, keys are part of the logical records recorded on the data set. For KSDS, the length and location of the keys are defined by the `DEFINE CLUSTER` command of Access Method Services. For KSDS AIX and ESDS AIX, the keys are defined by the `DEFINE ALTERNATEINDEX` command.

Relative Byte Addresses: Relative byte addresses enable you to access ESDS files directly. The RBAs are `unsigned long int` fields, and their values are computed by VSAM.

Notes:
1. KSDS can also use RBAs. However, because the RBA of a KSDS record can change if an insert, delete or update operation is performed elsewhere in the file, it is not recommended.
2. You can call `flocate()` with RBA values in an RRDS cluster, but `flocate()` with RBA values does not work across control intervals. Therefore, using RBAs with RRDS clusters is not recommended. The RRDS access method does not support RBAs. LE/VSE C Run-Time supports the use of RBAs in an RRDS cluster by translating the RBA value to an RRN. It does this by dividing the RBA value by the LRECL.
3. Alternate indexes do not allow positioning by RBA.

The RBA value is stored in the C structure `__amrc`, which is defined in the C stdio.h header file. You can access the field `__amrc->__RBA` as shown below:

EDCXGVS1:

```
 /* EDCXGVS1
    This example shows how to access the __amrc->__RBA field.
    It assumes that an ESDS has already been defined, and has been
    assigned the DLBL-name ESDSCLU.
  */

#include <stdio.h>
#include <stdlib.h>

main() {
   FILE *ESDSfile;
   unsigned long myRBA;
   char recbuff[100]="This is record one.";
   int w_retcd;
   int l_retcd;
   int r_retcd;

   printf("calling fopen(\"dd:esdsclu\",\"rb+,type=record\");\n");
   ESDSfile = fopen("dd:esdsclu", "rb+,type=record");
   printf("fopen() returned 0X%.8x\n",ESDSfile);
   if (ESDSfile==NULL) exit;

   w_retcd = fwrite(recbuff, 1, sizeof(recbuff), ESDSfile);
   printf("fwrite() returned %d\n",w_retcd);
   if (w_retcd != sizeof(recbuff)) exit;
   myRBA = __amrc->__RBA;

   l_retcd = flocate(ESDSfile, &myRBA, sizeof(myRBA), __RBA_EQ);
   printf("flocate() returned %d\n",l_retcd);
   if (l_retcd !=0) exit;

   r_retcd = fread(recbuff, 1, sizeof(recbuff), ESDSfile);
   printf("fread() returned %d\n",r_retcd);
   if (l_retcd !=0) exit;

   return(0);
}
```

Figure 10. VSAM Example

For more information about the `__amrc` structure, refer to Chapter 14, “Debugging I/O Programs,” on page 131.

Relative Record Numbers: Records in an RRDS are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record position. Only RRDS files support accessing a record by its relative record number.

Summary of VSAM I/O Operations

Table 20 summarizes VSAM data set characteristics and the allowable I/O operations on them.

Table 20. Summary of VSAM Data Set Characteristics and Allowable I/O Operations

	KSDS	**ESDS**	**RRDS**
Record Length	Variable. Length can be changed by update.	Variable. Length cannot be changed by update.	Fixed.
Alternate Index	Allows access using unique or non-unique keys.	Allows access using unique or non-unique keys.	Not supported by VSAM.
Record Read (Sequential)	The order is determined by the VSAM key. Reads proceed in key sequence for the key of reference.	By entry sequence.	By relative record number.
Record Write (Direct)	Position determined by the value in the field designated as the key.	Record written at the end of the file.	By relative record number.
Positioning for Record Read	By key or by RBA value. Positioning by RBA value is not recommended because changes to the file change the RBA.	By RBA value. Alternate index allows use by key.	By relative record number.
Delete (Record)	If not already in correct position, reposition the file; read the record using `fread()`; delete the record using `fdelrec()`. `fread()` must immediately precede `fdelrec()`.	Not supported by VSAM.	If not already in correct position, position the file; read the record using `fread()`; delete the record using `fdelrec()`. `fread()` must immediately precede `fdelrec()`.
Update (Record)	If not already in correct position, reposition the file; read the record using `fread()`; update the record using `fupdate()`. `fread()` must immediately precede `fupdate()`.	If not already in correct position, reposition the file; read the record using `fread()`; update the record using `fupdate()`. `fread()` must immediately precede `fupdate()`.	If not already in correct position, reposition the file; read the record using `fread()`; update the record using `fupdate()`. `fread()` must immediately precede `fupdate()`.
Empty the File	Define the file as reusable using `DEFINE CLUSTER` definition, and then open the data set in write (`"wb,type=record"` or `"wb+,type=record"`) mode. Not supported for alternate indexes.	Define the file as reusable using `DEFINE CLUSTER` definition, and then open the data set in write (`"wb,type=record":` or `"wb+,type=record"`) mode. Not supported for alternate indexes.	Define the file as reusable using `DEFINE CLUSTER` definition, and then open the data set in write (`"wb,type=record"` or `"wb+,type=record"`) mode.
Stream Read	Supported by LE/VSE C Run-Time.	Supported by LE/VSE C Run-Time.	Supported by LE/VSE C Run-Time.
Stream Write/Update	Not supported by LE/VSE C Run-Time.	Supported by LE/VSE C Run-Time.	Supported by LE/VSE C Run-Time.
Stream Repositioning	Supported by LE/VSE C Run-Time.	Supported by LE/VSE C Run-Time.	Supported by LE/VSE C Run-Time.

Opening VSAM Data Sets

To open a VSAM data set, use the standard C library functions `fopen()` and `freopen()` just as you would for opening non-VSAM data sets. The `fopen()` and `freopen()` functions are described in *LE/VSE C Run-Time Library Reference*.

This section describes considerations for using `fopen()` and `freopen()` with VSAM files. Remember that a VSAM file must exist and be defined as a VSAM cluster before you call `fopen()`.

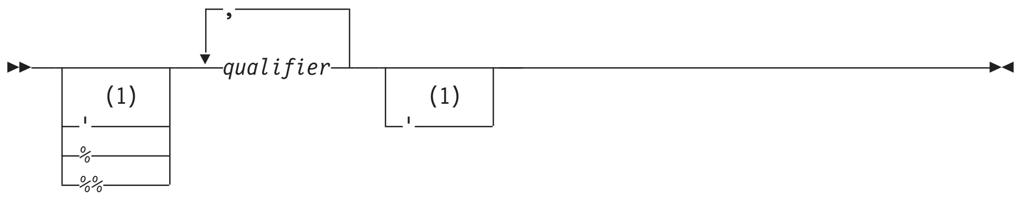
For information regarding VSAM-managed SAM (SAM ESDS files), see Chapter 8, "Performing SAM I/O Operations," on page 47.

Using `fopen()` or `freopen()`

Files are opened with a call to `fopen()` or `freopen()` in the format: `fopen("`*`filename`*`", "mode")`.

Filenames for VSAM Data Sets

Using a Data Set Name: The syntax for the *filename* argument on your `fopen()` or `freopen()` call when using a data set name is shown in the following diagram:



Notes:

1 The single quotation marks must be matched; if you use one, you must use the other.

A sample construct is:

```
'qualifier1.qualifier2'
```

' When you enclose a data set name in single quotation marks, the data set name is *fully qualified*. The file opened is the one specified by the data set name inside the quotation marks. If the data set name is not fully qualified, LE/VSE C Run-Time appends the job name to the front of the data set name. For example, the statement `fopen("a.b","w");` opens a file *jobname*.A.B, where *jobname* is the name of the job submitted. If the data set name is fully qualified, LE/VSE C Run-Time does not append a job name.

% A single %-sign in front of the data set name indicates that you want VSE/VSAM to append a partition identifier to the data set name specified.

%%
Two %-signs in front of the data set name indicates that you want VSE/VSAM to append a unique processor identification and a partition identifier to the data set name specified.

qualifier
Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national ($, #, @), the hyphen, or the character X'C0'. The first character should be either alphabetic or national.

You can join qualifiers with periods. The maximum length of a data set name is 44 characters, including periods.

Using a DLBL/TLBL-name: The syntax for the *filename* argument on your `fopen()` or `freopen()` call when using a `DLBL/TLBL`-name is shown in the following diagram:

```
►►─DD:─┬─DLBL──────────────┬──────────────────────────────────────────────►◄
       ├─TLBL──────────────┤
       │             (1)   │
       └─LU───┬─DLBL───────┤
              │      (2)  │
              └─TLBL──────┘
```

Notes:

1 If both *LU* and *DLBL* or *TLBL* are specified, a dash (-) must be used as separator (no blanks are allowed).

2 If both *LU* and *DLBL* or *TLBL* are specified, a dash (-) must be used as separator (no blanks are allowed).

LU
Specification of logical unit (*LU*) is ignored.

DLBL **or** *TLBL*
A 1- to 8-character name of which at most the first 7 characters are used. These characters may be alphanumeric or national ($, #, @). The first character must be either alphabetic or national.

The following example shows how to access a cluster or path by `DLBL`-name by writing the required `DLBL` statement and calling `fopen()`:

If your data set is `VSAM.CLUSTER1` and your C program refers to this data set by the `DLBL`-name `CFILE`, you can write the `DLBL` statement:

```
// DLBL CFILE,'VSAM.CLUSTER1',0,VSAM
```

and code the following in your C source program:

```
#include <stdio.h>

FILE *infile;
main()
{
   infile=fopen("DD:CFILE", "ab+, type=record");
   .
   .
}
```

Note: LE/VSE C Run-Time does not check the value of `shareoptions` at open time, nor does it provide support for read-integrity and write-integrity, as required to share files under `shareoptions` 3 and 4.

To ensure data integrity on concurrent VSAM reads and writes by using common buffers, the `dsn=` keyword must be specified on all calls to `fopen()` for a given data set (either the base cluster or any of its alternate indexes). Using the `dsn=` keyword facilitates VSAM DSN (data set name) sharing regardless of the `shareoptions` specification.

For more information on `shareoptions`, see the information on `DEFINE CLUSTER` in the books listed in "Where to Find More Information" on page xxi.

Specifying `fopen()` and `freopen()` Keywords

The *mode* argument is a character string specifying the type of access requested for the file.

The *mode* argument contains one positional parameter (access mode) followed by keyword parameters. A description of these parameters, along with an explanation of how they apply to VSAM data sets is given in the following sections.

Specifying Access Mode: The access mode is specified by the positional parameter of the `fopen()` function call. The possible record I/O and binary modes you can specify are:

`rb` Open for reading. If the file is empty, `fopen()` fails.

`wb` Open for writing. If the cluster is defined as reusable, the existing contents of the cluster are destroyed. If the cluster is defined as not reusable (clusters with paths are, by definition, not reusable), `fopen()` fails. However, if the cluster has been defined but not loaded, this mode can be used to do the initial load of both reusable and non reusable clusters.

`ab` Open for writing.

`rb+` or `r+b`
Open for reading, writing, and/or updating.

`wb+` or `w+b`
Open for reading, writing, and/or updating. If the cluster is defined as reusable, the existing contents of the cluster are destroyed. If the cluster is defined as not reusable (clusters with paths are, by definition, not reusable), the `fopen()` fails. However, if the cluster has been defined but not loaded, this mode can be used to do the initial load of both reusable and non reusable clusters.

`ab+` or `a+b`
Open for reading, writing, and/or updating.

For text files, you can specify the following modes: `r`, `w`, `a`, `r+`, `w+`, and `a+`.

Note: For keyed VSAM data sets (KSDS, KSDS AIX and ESDS AIX) in text and binary I/O, the only valid modes are `r` and `rb`, respectively. (See also "Text and Binary I/O in VSAM" on page 98.)

`fopen()` and `freopen()` Parameters

The following table lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for VSAM I/O, and lists the values that are valid for the applicable ones.

Table 21. Parameters for the `fopen()` and `freopen()` Functions for VSAM Data Sets

Parameter	Allowed?	Applicable?	Notes
`recfm=`	Yes	No	Ignored.
`lrecl=`	Yes	No	Ignored.
`blksize=`	Yes	No	Ignored.
`space=`	Yes	No	Ignored.
`type=`	Yes	Yes	May be omitted. If you do specify it, `type=record` is the only valid value. See the parameter list below for details.
`acc=`	Yes	Yes	Specifies the access direction for VSAM data sets. Valid values are `BWD` and `FWD`. See the parameter list below for details.
`password=`	Yes	Yes	Specifies the password for a VSAM data set. See the parameter list below for details.
`asis`	Yes	No	Ignored.
`byteseek`	Yes	Yes	Used for binary stream files to specify that the seeking functions should use relative byte offsets instead of encoded offsets. This is the default setting.
`noseek`	Yes	No	Ignored.
`OS`	Yes	No	Ignored.
`rewind=`	Yes	Yes	Not used for VSAM I/O.
`dsn=`	Yes	Yes	Specifies the number of strings to be allocated for data set name sharing. See the parameter list below for details.

`recfm=`
Any values passed into `fopen()` are ignored.

`lrecl=` and `blksize=`
These parameters are set to the maximum record size of the cluster as initialized in the cluster definition. Any values passed into `fopen()` are ignored.

`space=`
This parameter is not supported under VSAM.

`type=`
If you use the `type=` parameter, the only valid value for VSAM data sets is `type=record`. This opens a file for record I/O.

`acc=`
For VSAM files opened with the parameter `type=record`, you can specify the direction by using the `acc=`*`access_type`* parameter on the `fopen()` function call. For text and binary files, the access direction is always *forward*. Attempts to open a VSAM data set with `acc=BWD` for either binary or text stream I/O will fail.

The *access_type* can be one of the following:

`FWD` The `acc=FWD` parameter specifies that the file be processed in a forward direction. When the file is opened, it will be positioned at the beginning of the first physical record, and any subsequent read operations sets the file position indicator to the beginning of the next record.

The default value for the access parameter is `acc=FWD`.

BWD The `acc=BWD` parameter specifies that the file be processed in a backward direction. When the file is opened, it is positioned at the beginning of the last physical record and any subsequent read operation sets the file position indicator to the beginning of the preceding record.

You can change the direction of sequential processing (from forward to backward or from backward to forward) by using the `flocate()` library function. For more information about `flocate()`, see "Repositioning within Record I/O Files" on page 94.

Note: When opening paths, records with duplicate alternate index keys are processed in order of arrival time (oldest to newest) regardless of the current processing direction.

password=

VSAM facilities provide password protection for your data sets. You access a data set that has password protection by specifying the password on the `password` keyword parameter of the `fopen()` function call; the password resides in the VSAM catalog entry for the named file. There can be more than one password in the VSAM catalog entry; data sets can have different passwords for different levels of authorization such as reading, writing, updating, inserting, or deleting. For a complete description of password protection on VSAM files, see the list of publications listed in "Where to Find More Information" on page xxi.

The `password` parameter has the form:

```
password=nx
```

where `x` is a 1- to 8-character password, and **n** is the exact number of characters in the password. The password can contain special characters such as blanks and commas.

If a required password is not supplied, or if an incorrect password is given, `fopen()` fails.

asis

If you specify this parameter, LE/VSE C Run-Time ignores it.

byteseek

When you specify this parameter and open a file in binary stream mode, `fseek()` and `ftell()` use relative byte offsets from the beginning of the file. This is the default setting.

noseek

This parameter is ignored for VSAM data sets.

OS

This parameter is ignored for VSAM data sets.

rewind=

This parameter is not valid for VSAM I/O. If you specify it, LE/VSE C Run-Time ignores it.

dsn=

This parameter allows you to specify the number of strings to be allocated to the data set for shared access.

The `dsn` paremeter has the form:

```
dsn=n
```

where *n* specifies a decimal number. The number of strings allocated to a data set is determined by the first `fopen()` call for that data set which specifies `dsn=`*n*. VSAM uses the value *n* to set the ACB BSTRNO value. Subsequent calls to `fopen()` for the same data set requires a `dsn=`*n* keyword parameter in order to participate in DSN (data set name) sharing, but the value *n* is ignored.

All files opened with a `dsn=` parameter use ACB MACRF=OUT. VSE/VSAM does not allow mixed MACRF=IN|OUT ACBs for DSN sharing. This does not affect C file processing options, such as the access mode.

For additional information about VSAM DSN sharing, refer to "Where to Find More Information" on page xxi.

Buffering

Full buffering is the default. You can specify line buffering, but LE/VSE C Run-Time treats line buffering as full buffering for VSAM data sets. Unbuffered I/O is not supported under VSAM; if you specify it, your `setvbuf()` call fails.

To find out how to optimize VSAM performance by controlling the number of VSAM buffers used for your data set, read the section "Optimizing the Performance of VSE/VSAM" in *IBM VSE/Virtual Storage Access Method (VSAM) User's Guide*.

Record I/O in VSAM

This section describes how to use record I/O in VSAM.

RRDS Record Structure

For RRDS files opened in record mode, LE/VSE C Run-Time defines the following key structure in the C header file stdio.h:

```
typedef struct {
                long unsigned int __fill,
                                  __recnum;  /* the RRN, starting at 1 */
}__rrds_key_type;
```

In your source program, you can define an RRDS record structure as either:

```
struct {
        __rrds_key_type rrds_key;      /* __fill value always 0 */
        char           data[MY_REC_SIZE];
}  rrds_rec_0;
```

or:

```
struct {
        __rrds_key_type rrds_key;      /* __fill value always 1 */
        char           *data;
}  rrds_rec_1;
```

The LE/VSE C Run-Time library recognizes which type of record structures you have used by the value of `rrds_key.__fill`. Zero indicates that the data is contiguous with `rrds_key` and 1 indicates that a pointer to the data follows `rrds_key`.

Reading Record I/O files

To read from a VSAM data set opened with `type=record`, use the standard C `fread()` library function. If you set the `size` argument to 1 and the `count` argument to the maximum record size, `fread()` returns the number of bytes read successfully. For more information on `fread()`, see *LE/VSE C Run-Time Library Reference*.

`fread()` reads one record from the system from the current file position. Thus, if you want to read a certain record, you can call `flocate()` to position the file pointer to point to it; the subsequent call to `fread()` reads in that record.

If you use an `fread()` call to request more bytes than the record about to be read contains, `fread()` reads the entire record and returns the number of bytes read. If you use `fread()` to request fewer bytes than the record about to read contains, `fread()` reads the number of bytes that you specified and returns your request.

LE/VSE C Run-Time VSAM Record I/O does not allow a read operation to immediately follow a write operation without an intervening reposition. LE/VSE C Run-Time treats the following as read operations:
- Calls to read functions that request `0` bytes
- Read requests that fail because of a system error
- Calls to the `ungetc()` function

Calling `fread()` several times in succession, with no other operations on this file in between, reads several records in sequence (sequential processing), which can be forward or backward, depending on the access direction, as described below.

- **KSDS, KSDS AIX and ESDS AIX**

  The records are retrieved according to the sequence of the key of reference, or in reverse key sequence.

  Note: Records with duplicate alternate index keys are processed in order of arrival time (oldest to newest) regardless of the current processing direction.
- **ESDS**

  The records are retrieved according to the sequence they were written to the file (entry sequence), or in reverse entry sequence.
- **RRDS**

  The records are retrieved according to relative record number sequence or reverse relative record number sequence.

  When records are being read, RRNs without an associated record are ignored. For example, if a file has relative records of 1, 2, and 5, the nonexistent records 3 and 4 are ignored.

  By default, in record mode, `fread()` must be called with a pointer to an RRDS record structure. The field `__rrds_key_type.__fill` must be set to either 0 or 1 indicating the type of the structure, and the `count` argument must include the length of the `__rrds_key_type`. `fread()` returns the RRN number in the `__recnum` field, and includes the length of the `__rrds_key_type` in the return value. You can override these operations by setting the _EDC_RRDS_HIDE_KEY environment variable to `Y`. Once this variable is set, `fread()` is called with a data buffer and not an RRDS data structure. The return value of `fread()` is now only the length of the data read. In this case, `fread()` cannot return the RRN. For information on setting environment variables, see Chapter 21, “Using Environment Variables,” on page 219.

Writing to Record I/O Files

To write new records to a VSAM data set opened with `type=record`, use the standard C `fwrite()` library function. If you set `size` to 1 and `count` to the desired record size, `fwrite()` returns the number of bytes written successfully. For more information on `fwrite()` and the `type=record` parameter, see *LE/VSE C Run-Time Library Reference*.

In general, C I/O does not allow a write operation to follow a read operation without an intervening reposition or `fflush()`. LE/VSE C Run-Time counts a call to a write function writing `0` bytes or a write request that fails because of a system error as a write operation. However, LE/VSE C Run-Time VSAM record I/O allows a write to directly follow a read.

The process of writing to a data set for the first time is known as *initial loading*. Using the `fwrite()` function, you can write to a new VSAM file in *initial load* mode just as you would to a file not in *initial load* mode. Writing to a KSDS PATH or an ESDS PATH in *initial load* mode is not supported.

If your `fwrite()` call does not try to write more bytes than the maximum record size, `fwrite()` writes a record of the length you asked for and returns your request. If your `fwrite()` call asks for more than the maximum record size, `fwrite()` writes the maximum record size, sets `errno`, and returns the maximum record size. In either case, the next call to `fwrite()` writes to the following record.

Note: If an `fwrite()` fails, you must reposition the file before you try to read or write again.

- **KSDS, KSDS AIX**

  Records are written to the cluster according to the value stored in the field designated as the prime key.

  You can load a KSDS in any key order but it is most efficient to perform the `fwrite()` operations in key sequence.
- **ESDS, ESDS AIX**

  Records are written to the end of the file.
- **RRDS**

  Records are written according to the value stored in the relative record number field.

  `fwrite()` is called with the RRDS record structure.

  By default, in record mode, `fwrite()` and `fupdate()` must be called with a pointer to an RRDS record structure. The `__rrds_key_type` fields `__fill` and `__recnum` must be set. `__fill` is set to `0` or 1 to indicate the type of the structure. The `__recnum` field specifies the RRN to write, and is required for `fwrite()` but not `fupdate()`. The `count` argument must include the length of the `__rrds_key_type`. `fwrite()` and `fupdate()` include the length of the `__rrds_key_type` in the return value.

Updating Record I/O Files

The `fupdate()` function, a LE/VSE C Run-Time extension to the SAA C library, is used to update records in a VSAM file. For more information on this function, see *LE/VSE C Run-Time Library Reference*.

- **KSDS, ESDS, and RRDS**

  To update a record in a VSAM file, you must perform the following operations:

1. Open the VSAM file in update mode (`rb+/r+b`, `wb+/w+b`, or `ab+/a+b` specified as the required positional parameter of the `fopen()` function call and `type=record`).
2. If the file is not already positioned at the record you want to update, reposition to that record.
3. Read in the record using `fread()`.

   Once the record you want to update has been read in, you must ensure that no reading, writing, or repositioning operations are performed before `fupdate()`.
4. Make the necessary changes to the copy of the record in your buffer area.
5. Update the record from your local buffer area using the `fupdate()` function.

   If an `fupdate()` fails, you must reposition using `flocate()` before trying to read or write.

Notes:

1. If a file is opened in update mode, a read operation can result in the locking of control intervals, depending on `shareoptions` specification of the VSAM file. If after reading a record, you decide not to update it, you may need to unlock a control interval by performing a file positioning operation to the same record, such as an `flocate()` using the same key.
2. If `fupdate()` wrote out a record the file position is the start of the next record. If the `fupdate()` call did not write out a record, the file position remains the same.

- **KSDS and KSDS PATH**

  You can change the length of the record being updated. If your request does not exceed the maximum record size of the file, `fupdate()` writes a record of the length requested and returns the request. If your request exceeds the maximum record size of the file, `fupdate()` writes a record that is the maximum record size, sets `errno`, and returns the maximum record size.

  You cannot change the prime key field of the record, and in KSDS AIX, you cannot change the key of reference of the record.
- **ESDS**

  You cannot change the length of the record being updated. If the size of the record being updated is less than the current record size, `fupdate()` updates the amount you specify and does not alter the data remaining in the record. If your request exceeds the length of the record that was read, `fupdate()` writes a record that is the length of the record that was read, sets `errno`, and returns the length of the record that was read.
- **ESDS PATH**

  You cannot change the length of the record being updated or the key of reference of the record. If the size of the record being updated is less than the current record size, `fupdate()` updates the amount you specify and does not alter the data remaining in the record. If your request exceeds the length of the record that was read, `fupdate()` writes a record that is the length of the record that was read, sets `errno`, and returns the length of the record that was read.
- **RRDS**

  RRDS files have fixed record length. If you update the record with less than the record size, only those characters specified are updated, and the remaining data is not altered. If your request exceeds the record size of the file, `fupdate()` writes a record that is the record size, sets `errno`, and returns the length of the record that was read.

Deleting Records

To delete records, use the library function `fdelrec()`, a LE/VSE C Run-Time extension to the SAA C library. For more information on this function, see *LE/VSE C Run-Time Library Reference*.

- **KSDS, KSDS PATH, and RRDS**

  To delete records, you must perform the following operations:

  1. Open the VSAM file in update mode (`rb+`/`r+b`, `ab+`/`a+b`, or `wb+`/`w+b` specified as the required positional parameter of the `fopen()` function call and `type=record`).
  2. If the file is not already positioned at the record you want to delete, reposition to that record.
  3. Read the record using the `fread()` function.

     Once the record you want to delete has been read in, you must ensure that no reading, writing, or repositioning operations are performed before `fdelrec()`.
  4. Delete the record using the `fdelrec()` function.

  Note: If the data set was opened with an access mode of `rb+` or `r+b`, a read operation can result in the locking of control intervals, depending on `shareoptions` specification of the VSAM file. If after reading a record, you decide not to delete it, you may need to unlock a control interval by performing a file-positioning operation to the same record, such as an `flocate()` using the same key.

- **ESDS and ESDS PATH**

  VSAM does not support deletion of records in ESDS files.

Repositioning within Record I/O Files

You can use the following functions to locate a record within a VSAM data set:
- `flocate()`
- `ftell()` and `fseek()`
- `fgetpos()` and `fsetpos()`
- `rewind()`

For complete details on these library functions, see *LE/VSE C Run-Time Library Reference*.

flocate()

The `flocate()` C library function can be used to locate a specific record within a VSAM data set given the key, relative byte address, or the relative record number. The `flocate()` function also sets the access direction.

The following `flocate()` parameters set the access direction to *forward*:
- `__KEY_FIRST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ`
- `__KEY_GE`
- `__RBA_EQ`

The following `flocate()` parameters all set the access direction to *backward* and are only valid for record I/O:
- `__KEY_LAST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ_BWD`
- `__RBA_EQ_BWD`

Note: The __RBA_EQ and __RBA_EQ_BWD parameters are not valid for paths and are not recommended for KSDS and RRDS data sets.

You can use the rewind() library function instead of calling flocate() with __KEY_FIRST.

- **KSDS, KSDS AIX, and ESDS AIX**

  The key parameter of flocate() for the options __KEY_EQ, __KEY_GE, and __KEY_EQ_BWD is a pointer to the key of reference of the data set. The key_len parameter is the key length as defined for the data set for a full key search, or less than the defined key length for a generic key search (a partial key match).

  For KSDSs, __RBA_EQ and __RBA_EQ_BWD are supported, but are not recommended.

  Alternate indexes do not allow positioning by RBA.
- **ESDS**

  The key parameter of flocate() is a pointer to an unsigned long integer containing the specified RBA value. The key_len parameter is 4, because RBAs are unsigned long integers.
- **RRDS**

  For __KEY_EQ, __KEY_GE, and __KEY_EQ_BWD, the key parameter of flocate() is a pointer to an unsigned long integer containing the specified relative record number. For __RBA_EQ and __RBA_EQ_BWD, the key parameter of flocate() is a pointer to an unsigned long integer containing the specified RBA. However, seeking to RBA values is not recommended, because it is not supported across control intervals. The key_len parameter is 4, because RRNs and RBAs are unsigned long integers.

fgetpos() and fsetpos()

fgetpos() is used to store the current file position and access direction. fsetpos() is used to relocate to a file position stored by fgetpos() and restore the saved access direction.

- **KSDS**

  fgetpos() stores the RBA value. This RBA value may be invalidated by subsequent insertions, deletions, or updates.
- **KSDS AIX and ESDS AIX**

  fgetpos() and fsetpos() are not supported for PATHs.
- **ESDS and RRDS**

  There are no special considerations.

ftell() and fseek()

ftell() is used to store the current file position. fseek() is used to relocate to one of the following:
- A file position stored by ftell()
- A calculated record number (SEEK_SET)
- A position relative to the current position (SEEK_CUR)
- A position relative to the end of the file (SEEK_END)

ftell() and fseek() offsets in record mode I/O are relative record offsets. For example, the following call moves the file position to the start of the previous record:

```
fseek(fp, -1L, SEEK_CUR);
```

You cannot use `fseek()` to reposition to a file position before the beginning of the file or to a position beyond the end of the file.

Note: In general, the performance of this method is inferior to `flocate()`.

The access direction is unchanged by the repositioning.

- **KSDS and RRDS**

  There are no special considerations.
- **KSDS AIX and ESDS AIX**

  `ftell()` and `fseek()` are not supported.
- **ESDS**

  `ftell()` is not supported.
- **RRDS**

  `fseek()` seeks to a relative position in the file, and not to an RRN value. For example, in a file consisting of RRNs 1, 3, 5 and 7, `fseek(fp, 3L, SEEK_SET);` followed by an `fread()` would read in RRN 7, which is at offset 3 in the file.

rewind()

The `rewind()` function repositions the file position to the beginning of the file, and clears the error setting for the file.

`rewind()` does not reset the file access direction. For example, a call to `flocate()` with `__KEY_LAST` sets the file pointer to the end of the file and sets the access direction to backwards. A subsequent call to `rewind()` sets the file pointer to the beginning of the file, but the access direction remains backwards.

Flushing Buffers

You can use the C library function `fflush()` to flush buffers. However, `fflush()` writes nothing to the system, because all records have already been written there by `fwrite()`.

`fflush()` after a read operation does not refresh the contents of the buffer.

For more information on `fflush()`, see *LE/VSE C Run-Time Library Reference*.

Summary of VSAM Record I/O Operations

Table 22. Summary of VSAM Record I/O Operations

	KSDS	ESDS	RRDS	PATH
`fopen()`, `freopen()`	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+
`fwrite()`	rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	rb+, ab, ab+
`fread()`	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+
`ftell()`	rb, rb+, ab, ab+, wb, wb+[1]		rb, rb+, ab, ab+, wb, wb+	
`fseek()`	rb, rb+, ab, ab+, wb, wb+[1]	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fgetpos()`	rb, rb+, ab, ab+, wb, wb+[2]	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	

Table 22. Summary of VSAM Record I/O Operations (continued)

	KSDS	ESDS	RRDS	PATH
`fsetpos()`	rb, rb+, ab, ab+, wb, wb+[2]	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`flocate()`	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+
`rewind()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fflush()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fdelrec()`	rb+, ab+, wb+		rb+, ab+, wb+	rb+, ab+ (not ESDS)
`fupdate()`	rb+, ab+, wb+	rb+, ab+, wb+	rb+, ab+, wb+	rb+, ab+
`ferror()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`feof()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`clearerr()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fclose()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fldata()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+

Notes:

1 The saved position is based on the relative position of the record within the data set. Subsequent insertions or deletions may invalidate the saved position.

2 The saved position is based on the RBA of the record. Subsequent insertions, deletions or updates may invalidate the saved position.

Text and Binary I/O in VSAM

Because VSAM is primarily record-based, this section only discusses those aspects of text and binary I/O that are specific to VSAM. For general information on text and binary I/O, refer to the respective sections in Chapter 8, "Performing SAM I/O Operations," on page 47.

Reading from Text and Binary I/O Files

- **RRDS**

  All the read functions support reading from text and binary RRDS files. `fread()` is called with a character buffer instead of an RRDS record structure.

Writing to and Updating Text and Binary I/O Files

- **KSDS, KSDS AIX, and ESDS AIX**

  LE/VSE C Run-Time VSAM support for streams does not provide for writing and updating these types of data sets opened for text or binary stream I/O.
- **ESDS**

  Writes are supported for ESDSs opened as binary or text streams. Updating data in an ESDS stream cannot change the length of the record in the external file. Therefore, in a binary stream:
  - updates for less than the existing record length leave existing data beyond the updated length unchanged;
  - updates for longer than the existing record length flow over the record boundary and update the start of the next record.

  In text streams:
  - updates that specify records shorter than the original record pad the updated record to the existing record length with blanks;
  - updates for longer than the existing record length result in truncation, unless the original record contained only a newline character, in which case it may be updated to contain one byte of data plus a newline character.
- **RRDS**

  `fwrite()` is called with a character buffer instead of an RRDS record structure.

  Records are treated as contiguous. Once the current record is filled, the next record in the file is written to. For example, if the file consisted of only record 1, record 5, and record 28, a write would complete record 1 and then go directly to record 5.

  Writing past the last record in the file is allowed, up to the maximum size of the RRDS data set. For example, if the last record in the file is record 28, the next record to be written is record 29.

  Insertion of records is not supported. For example, in a file of records 1, 5, and 28, you cannot insert record 3 into the file.

Deleting Records in Text and Binary I/O Files

`fdelrec()` is not supported for text and binary I/O in VSAM.

Repositioning within Text and Binary I/O Files

You can use the following functions to locate a record within a VSAM data set:
- `flocate()`
- `ftell()` and `fseek()`
- `fgetpos()` and `fsetpos()`

- `rewind()`

For complete details on these library functions, see *LE/VSE C Run-Time Library Reference*.

`flocate()`

The `flocate()` C library function can be used to reposition to the beginning of a specific record within a VSAM data set given the key, relative byte address, or the relative record number. For more information on this function, see *LE/VSE C Run-Time Library Reference*.

The following `flocate()` parameters set the direction access to *forward*:
- `__KEY_FIRST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ`
- `__KEY_GE`
- `__RBA_EQ`

The following `flocate()` parameters all set the access direction to *backward* and are not valid for text and binary I/O, because backwards access is not supported:
- `__KEY_LAST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ_BWD`
- `__RBA_EQ_BWD`

You can use the `rewind()` library function instead of calling `flocate()` with `__KEY_FIRST`.

- **KSDS, KSDS AIX, and ESDS AIX**

  The key parameter of `flocate()` for the options `__KEY_EQ` and `__KEY_GE` is a pointer to the key of reference of the data set. The `key_len` parameter is the key length as defined for the data set for a full key search, or less than the defined key length for a generic key search (a partial key match).

  Alternate indexes do not allow positioning by RBA.

  Note: The `__RBA_EQ` parameter is not valid for paths and is not recommended.
- **ESDS**

  The key parameter of `flocate()` is a pointer to an unsigned long integer containing the specified RBA value. The `key_len` parameter is 4, because RBAs are unsigned long integers.
- **RRDS**

  For `__KEY_EQ` and `__KEY_GE`, the key parameter of `flocate()` is a pointer to an unsigned long integer containing the specified relative record number. For `__RBA_EQ`, the key parameter of `flocate()` is a pointer to an unsigned long integer containing the specified RBA. However, seeking to RBA values is not recommended, because it is not supported across control intervals. The `key_len` parameter is 4, because RRNs and RBAs are unsigned long integers.

`fgetpos()` and `fsetpos()`

`fgetpos()` saves the access direction, an RBA value, and the file position, and `fsetpos()` restores the saved access direction.

`fgetpos()` accounts for the presence of characters in the `ungetc()` buffer unless you have set the _EDC_COMPAT variable. See Chapter 21, “Using Environment Variables,” on page 219 for information about _EDC_COMPAT. If `ungetc()` characters back the file position up to before the start of the file, calls to `fgetpos()` fail.

- **KSDS**

  `fgetpos()` stores the RBA value. This RBA value may be invalidated by subsequent insertions, deletions or updates.
- **KSDS PATH and ESDS PATH**

  `fgetpos()` and `fsetpos()` are not supported for `PATH`s.
- **ESDS and RRDS**

  There are no special considerations.

`ftell()` and `fseek()`

Using `fseek()` to seek beyond the current end of file in a writable ESDS or RRDS binary file results in the file being extended with nulls to the new position. An incomplete last record is completed with nulls, records of length `lrecl` are added as required, and the current record is filled with the remaining number of nulls and left in the current buffer. This is supported for relative byte offset from `SEEK_SET`, `SEEK_CUR` and `SEEK_END`.

Table 23 provides a summary of the `fseek()` and `ftell()` parameters in binary and text.

Table 23. Summary of `fseek()` and `ftell()` Parameters in Text and Binary

Type	Mode	`ftell()` return values	`fseek()` `SEEK_SET`	`SEEK_CUR`	`SEEK_END`
KSDS	Binary	Relative byte offset	Relative byte offset	Relative byte offset	Relative byte offset
	Text	Not supported	Zero only	Relative byte offset	Relative byte offset
ESDS	Binary	Relative byte offset	Relative byte offset	Relative byte offset	Relative byte offset
	Text	Not supported	Zero only	Relative byte offset	Relative byte offset
RRDS	Binary	Encoded byte offset	Encoded byte offset	Relative byte offset	Relative byte offset
	Text	Encoded byte offset	Encoded byte offset	Relative byte offset	Relative byte offset
PATH	Binary	Not supported	Not supported	Not supported	Not supported
	Text	Not supported	Not supported	Not supported	Not supported

Flushing Buffers

You can use the C library function `fflush()` to flush data.

For text files, calling `fflush()` to flush an update to a record causes the new data to be written to the file.

If you call `fflush()` while you are updating, the updates are flushed out to VSAM.

For more information on `fflush()`, see *LE/VSE C Run-Time Library Reference*.

Summary of VSAM Text I/O Operations

Table 24. Summary of VSAM Text I/O Operations

	KSDS	ESDS	RRDS	PATH
`fopen()`, `freopen()`	r	r, r+, a, a+, w, w+ (empty cluster or reuse specified for w & w+)	r, r+, a, a+, w, w+ (empty cluster or reuse specified for w & w+)	r
`fwrite()`		r+, a, a+, w, w+	r+, a, a+, w, w+	

Table 24. Summary of VSAM Text I/O Operations (continued)

	KSDS	ESDS	RRDS	PATH
`fprintf()`		r+, a, a+, w, w+	r+, a, a+, w, w+	
`fputs()`		r+, a, a+, w, w+	r+, a, a+, w, w+	
`fputc()`		r+, a, a+, w, w+	r+, a, a+, w, w+	
`putc()`		r+, a, a+, w, w+	r+, a, a+, w, w+	
`vfprintf()`		r+, a, a+, w, w+	r+, a, a+, w, w+	
`vprintf()`		r+, a, a+, w, w+	r+, a, a+, w, w+	
`fread()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`fscanf()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`fgets()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`fgetc()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`getc()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`ungetc()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`ftell()`			r, r+, a, a+, w, w+	
`fseek()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
`fgetpos()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
`fsetpos()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
`flocate()`	r	r, r+, a+, w+	r, r+, a+, w+	r
`rewind()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
`fflush()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
`ferror()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
`fdelrec()`				
`fupdate()`				
`feof()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
`clearerr()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
`fclose()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
`fldata()`	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r

Summary of VSAM Binary I/O Operations

Table 25. Summary of VSAM Binary I/O Operations

	KSDS	ESDS	RRDS	PATH
`fopen()`, `freopen()`	rb	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb
`fwrite()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`fprintf()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`fputs()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`fputc()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`putc()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`vfprintf()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	

Table 25. Summary of VSAM Binary I/O Operations (continued)

	KSDS	ESDS	RRDS	PATH
`vprintf()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`fread()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`fscanf()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`fgets()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`fgetc()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`getc()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`ungetc()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`ftell()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fseek()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fgetpos()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fsetpos()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`flocate()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`rewind()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fflush()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`ferror()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fdelrec()`				
`fupdate()`				
`feof()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`clearerr()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fclose()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fldata()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb

Closing VSAM Data Sets

To close a VSAM data set, use the standard C `fclose()` library function as you would for closing non-VSAM files. See *LE/VSE C Run-Time Library Reference* for more details on the `fclose()` library function.

For ESDS binary files, if `fclose()` is called and there is a new record in the buffer that is less than the maximum record size, this record is written to the file at its current size. A new RRDS binary record that is incomplete when the file is closed is filled with null characters to the record size.

A new ESDS or RRDS text record that is incomplete when the file is closed is completed with a newline.

VSAM Return Codes

When failing return codes are received from LE/VSE C Run-Time VSAM I/O functions, you can access the `__amrc` structure to help you diagnose errors. The `__amrc_type` structure is defined in the header file stdio.h (when the compiler option `LANGLVL(EXTENDED)` is used).

Note: The `__amrc struct` is global and can be reset by another I/O operation (such as `printf()`).

The following fields of the structure are important to VSAM users:

`__amrc.__code.__feedback.__rc`
Stores the VSAM R15.

`__amrc.__code.__feedback.__fdbk`
Stores the VSAM error code or reason code.

`__amrc.__RBA`
Stores the RBA after some operations.

`__amrc.__last_op`
Stores a code for the last operation. The codes are defined in the header file stdio.h.

For definitions of these return codes and feedback codes, refer to the publications listed in "Where to Find More Information" on page xxi.

You can set up a `SIGIOERR` handler to catch read or write system errors. See Chapter 14, "Debugging I/O Programs," on page 131 for more information.

VSAM Examples

This section provides several examples of using I/O under VSAM.

KSDS Example

The example below shows two functions from an employee record entry system with a mainline driver to process selected options (display, display next, update, delete, create).

The update routine is an example of KSDS clusters, and the display routine is an example of both KSDS clusters and alternate indexes.

For these examples, the clusters and alternate indexes should be defined as follows:
- The KSDS cluster has a record size of 150 with a key length of 4 with offset 0
- The unique KSDS AIX has a key length of 20 with an offset of 10
- The non-unique KSDS AIX has a key length of 40 with an offset of 30

The update routine is passed the following:
- `data_ptr`, which points to the information that is to be updated
- `orig_data_ptr`, which points to the information that was originally displayed using the display option
- A file pointer to the KSDS cluster

The display routine is passed the following:
- `data_ptr`, which points to the information that was entered on the screen for the search query
- `orig_data_ptr`, which is returned with the information for the record to be displayed if it exists
- File pointers for the primary cluster, unique alternate index and non-unique alternate index

By definition, the primary key is unique and therefore the employee number was chosen for this key. The `user_id` is also a unique key; therefore, it was chosen as the unique alternate index key. The name field may not be unique; therefore, it was chosen as the non-unique alternate index key.

KSDS Example

EDCXGVS2:

```
 /* EDCXGVS2
    This example demonstrates the use of a KSDS file.
    Part 1 of 2-other file is EDCXGVS3.
  */

#include <stdio.h>
#include <string.h>

 /* global definitions                                                */

struct data_struct {
              char   emp_number[4];
              char   user_id[8];
              char   name[20];
              char   pers_info[37];
};

static void print_amrc(void);
int update_emp_rec(struct data_struct *, struct data_struct *, FILE *);
int display_emp_rec(struct data_struct *, struct data_struct *,
    FILE *, FILE *, FILE *);

#define  REC_SIZE               69
#define  CLUS_KEY_SIZE           4
#define  AIX_UNIQUE_KEY_SIZE     8
#define  AIX_NONUNIQUE_KEY_SIZE  20
 /* main() function definition                                        */

int main() {
    FILE*              clus_fp;
    FILE*              aix_ufp;
    FILE*              aix_nufp;
    int                i;
    struct data_struct  buf1, buf2;

    char data[3][REC_SIZE+1] = {
"   1LARRY   LARRY              HI, I'M LARRY,                     ",
"   2DARRYL1 DARRYL             AND THIS IS MY BROTHER DARRYL,     ",
"   3DARRYL2 DARRYL                                                "
    };
```

Figure 11. KSDS Example (Program) (Part 1 of 7)

```
/* open file three ways                                          */
clus_fp = fopen("dd:cluster", "rb+,type=record,dsn=3");
if (clus_fp == NULL) {
   print_amrc();
   printf("Error: fopen(\"dd:cluster\"...) failed\n");
   return 5;
}
/* assume base cluster was loaded with at least one dummy record  */
/* so aix could be defined                                       */
aix_ufp = fopen("dd:aixuniq", "rb,type=record,dsn=3");
if (aix_ufp == NULL) {
   print_amrc();
   printf("Error: fopen(\"dd:aixuniq\"...) failed\n");
   return 10;
}
/* assume base cluster was loaded with at least one dummy record  */
/* so aix could be defined                                       */
aix_nufp = fopen("dd:aixnunq", "rb,type=record,dsn=3");
if (aix_nufp == NULL) {
   print_amrc();
   printf("Error: fopen(\"dd:aixnunq\"...) failed\n");
   return 15;
}
/* load sample records                                           */
for (i = 0; i < 3; ++i) {
   if (fwrite(data[i],1,REC_SIZE,clus_fp) != REC_SIZE) {
      print_amrc();
      printf("Error: fwrite(data[%d]...) failed\n", i);
      return 66+i;
   }
}

/* display sample record by primary key                          */
memcpy(buf1.emp_number, "   1", 4);
if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
   return 69;
```

Figure 11. KSDS Example (Program) (Part 2 of 7)

```
    /* display sample record by non-unique aix key                    */
    memset(buf1.emp_number, '\0', 4);
    buf1.user_id[0] = '\0';
    memcpy(buf1.name, "DARRYL                    ", 20);
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 70;

    /* display sample record by unique aix key                        */
    memcpy(buf1.user_id, "DARRYL2 ", 8);
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 71;

    /* update record just read with new personal info                 */
    memcpy(&buf1, &buf2, REC_SIZE);
    memcpy(buf1.pers_info, "AND THIS IS MY OTHER BROTHER DARRYL. ", 37);
    if (update_emp_rec(&buf1, &buf2, clus_fp) != 0) return 72;

    /* display sample record by unique aix key                        */
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 73;

    return 0;
}

static void print_amrc() {
    __amrc_type currErr = *__amrc; /* copy contents of __amrc         */
                                   /* structure so that values        */
                                   /* do not get jumbled by printf    */
    printf("R15 value   = %d\n", currErr.__code.__feedback.__rc);
    printf("Reason code = %d\n", currErr.__code.__feedback.__fdbk);
    printf("RBA         = %d\n", currErr.__RBA);
    printf("Last op     = %d\n", currErr.__last_op);
    return;
}
```

Figure 11. KSDS Example (Program) (Part 3 of 7)

```
 /* update_emp_rec() function definition                           */

int update_emp_rec (struct data_struct *data_ptr,
                    struct data_struct *orig_data_ptr,
                    FILE   *fp)
{
    int          rc;
    char         buffer[REC_SIZE+1];

  /*    Check to see if update will change primary key (emp_number)   */
    if (memcmp(data_ptr->emp_number,orig_data_ptr->emp_number,4) != 0) {
       /* Check to see if changed primary key exists                  */
       rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
       if (rc == 0) {
          print_amrc();
          printf("Error: new employee number already exists\n");
          return 10;
       }

       clearerr(fp);

       /* Write out new record                                        */
       rc = fwrite(data_ptr,1,REC_SIZE,fp);
       if (rc != REC_SIZE || ferror(fp)) {
          print_amrc();
          printf("Error: write with new employee number failed\n");
          return 20;
       }

       /* Locate to old employee record so it can be deleted          */
       rc = flocate(fp,&(orig_data_ptr->emp_number),CLUS_KEY_SIZE,
                    __KEY_EQ);
       if (rc != 0) {
          print_amrc();
          printf("Error: flocate to original employee number failed\n");
          return 30;
       }

       rc = fread(buffer,1,REC_SIZE,fp);
       if (rc != REC_SIZE || ferror(fp)) {
          print_amrc();
          printf("Error: reading old employee record failed\n");
          return 40;
       }

       rc = fdelrec(fp);
       if (rc != 0) {
          print_amrc();
          printf("Error: deleting old employee record failed\n");
          return 50;
       }

   } /* end of checking for change in primary key               */
```

Figure 11. KSDS Example (Program) (Part 4 of 7)

```
    else { /* Locate to current employee record                  */
       rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
       if (rc == 0) {
          /* record exists, so update it                            */
          rc = fread(buffer,1,REC_SIZE,fp);
          if (rc != REC_SIZE || ferror(fp)) {
             print_amrc();
             printf("Error: reading old employee record failed\n");
             return 60;
          }

          rc = fupdate(data_ptr,REC_SIZE,fp);
          if (rc == 0) {
             print_amrc();
             printf("Error: updating new employee record failed\n");
             return 70;
          }
       }
       else { /* record does not exist so write out new record   */
          clearerr(fp);
          printf("Warning: record previously displayed no longer\n");
          printf("       : exists, new record being created\n");
          rc = fwrite(data_ptr,1,REC_SIZE,fp);
          if (rc != REC_SIZE || ferror(fp)) {
             print_amrc();
             printf("Error: write with new employee number failed\n");
             return 80;
          }
       }
    }
    return 0;
}
```

Figure 11. KSDS Example (Program) (Part 5 of 7)

```
 /* display_emp_rec() function definition                              */

int display_emp_rec (struct data_struct *data_ptr,
                     struct data_struct *orig_data_ptr,
                     FILE *clus_fp, FILE *aix_unique_fp,
                     FILE *aix_non_unique_fp)
{
    int     rc = 0;
    char    buffer[REC_SIZE+1];

    /* Primary Key Search                                              */
    if (memcmp(data_ptr->emp_number, "\0\0\0\0", 4) != 0) {
       rc = flocate(clus_fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,
                    __KEY_EQ);
       if (rc != 0) {
          printf("Error: flocate with primary key failed\n");
          return 10;
       }

       /* Read record for display                                      */
       rc = fread(orig_data_ptr,1,REC_SIZE,clus_fp);
       if (rc != REC_SIZE || ferror(clus_fp)) {
          printf("Error: reading employee record failed\n");
          return 15;
       }
    }
    /* Unique Alternate Index Search                                   */
    else if (data_ptr->user_id[0] != '\0') {
       rc = flocate(aix_unique_fp,data_ptr->user_id,AIX_UNIQUE_KEY_SIZE,
                   __KEY_EQ);
       if (rc != 0) {
          printf("Error: flocate with user id failed\n");
          return 20;
       }

       /* Read record for display                                      */
       rc = fread(orig_data_ptr,1,REC_SIZE,aix_unique_fp);
       if (rc != REC_SIZE || ferror(aix_unique_fp)) {
          printf("Error: reading employee record failed\n");
          return 25;
       }
    }
```

Figure 11. KSDS Example (Program) (Part 6 of 7)

```
    /* Non-unique Alternate Index Search                              */
    else if (data_ptr->name[0] != '\0') {
       rc = flocate(aix_non_unique_fp,data_ptr->name,
                    AIX_NONUNIQUE_KEY_SIZE,__KEY_GE);
       if (rc != 0) {
          printf("Error: flocate with name failed\n");
          return 30;
       }

       /* Read record for display                                     */
       rc = fread(orig_data_ptr,1,REC_SIZE,aix_non_unique_fp);
       if (rc != REC_SIZE || ferror(aix_non_unique_fp)) {
          printf("Error: reading employee record failed\n");
          return 35;
       }
    }
    else {
       printf("Error: invalid search argument; valid search arguments\n"
              "     : are either employee number, user id, or name\n");
       return 40;
    }
    /* display record data                                            */
    printf("Employee Number: %.4s\n", orig_data_ptr->emp_number);
    printf("Employee Userid: %.8s\n", orig_data_ptr->user_id);
    printf("Employee Name:   %.20s\n", orig_data_ptr->name);
    printf("Employee Info:   %.37s\n", orig_data_ptr->pers_info);
    return 0;
}
```

Figure 11. KSDS Example (Program) (Part 7 of 7)

The following JCL can be used to test the previous example:

EDCXGVS3:

```
// JOB EDCXGVS3
/* This example illustrates the use of a KSDS file
/* Part 2 of 2-other file is EDCXGVS2
/* ------------------------------------------------------------------
/* Delete cluster, AIX, and PATH
/* ------------------------------------------------------------------
// EXEC IDCAMS,SIZE=AUTO
    DELETE -
        userid.KSDS.CLUSTER -
        CLUSTER -
        PURGE -
        ERASE
```

Figure 12. KSDS Example (JCL) (Part 1 of 3)

```
/* ----------------------------------------------------------------
/* Define KSDS
/* ----------------------------------------------------------------
// EXEC IDCAMS,SIZE=AUTO
    DEFINE CLUSTER -
     (NAME(userid.KSDS.CLUSTER) -
     TRK(4 4)  -
     RECSZ(69 100) -
     INDEXED -
     NOREUSE -
     KEYS(4 0) -
     VOLUMES(volume) -
     OWNER(userid) -
     ) -
      DATA -
        (NAME(userid.KSDS.DA)) -
      INDEX -
        (NAME(userid.KSDS.IX))

/* ---------------------------------------------------------------
/* Repro data into KSDS
/* ---------------------------------------------------------------
// DLBL FILEOUT,'userid.KSDS.CLUSTER',,VSAM
// EXEC IDCAMS,SIZE=AUTO
    REPRO -
        INFILE(SYSIPT) -
       OUTFILE(FILEOUT)
0000ZZZZZZZZDUMMY_RECORD

/*/* ---------------------------------------------------------------
/* Define unique AIX, define and build PATH
/* ---------------------------------------------------------------
// EXEC IDCAMS,SIZE=AUTO
    DEFINE AIX -
        (NAME(userid.KSDS.UAIX) -
        RECORDS(25)  -
        KEYS(8,4)    -
        VOL(volume)  -
        UNIQUEKEY -
        RELATE(userid.KSDS.CLUSTER)) -
      DATA -
        (NAME(userid.KSDS.UAIXDA)) -
      INDEX -
        (NAME(userid.KSDS.UAIXIX))
    DEFINE PATH -
        (NAME(userid.KSDS.UPATH) -
        PATHENTRY(userid.KSDS.UAIX))
    BLDINDEX -
        INDATASET(userid.KSDS.CLUSTER) -
        OUTDATASET(userid.KSDS.UAIX)
/*
```

Figure 12. KSDS Example (JCL) (Part 2 of 3)

```
/* -------------------------------------------------------------------
/* Define non-unique AIX, define and build PATH
/* -------------------------------------------------------------------
// EXEC IDCAMS,SIZE=AUTO
    DEFINE AIX -
        (NAME(userid.KSDS.NUAIX) -
        RECORDS(25)  -
        KEYS(20, 12)    -
        VOL(volume)  -
        NONUNIQUEKEY -
        RELATE(userid.KSDS.CLUSTER)) -
      DATA -
        (NAME(userid.KSDS.NUAIXDA)) -
      INDEX -
        (NAME(userid.KSDS.NUAIXIX))
    DEFINE PATH -
        (NAME(userid.KSDS.NUPATH) -
        PATHENTRY(userid.KSDS.NUAIX))
    BLDINDEX -
        INDATASET(userid.KSDS.CLUSTER) -
        OUTDATASET(userid.KSDS.NUAIX)
/*

/* ------------------------------------------------------------------
/* Run the testcase
/* ------------------------------------------------------------------
// LIBDEF *,SEARCH=(PRD2.SCEEBASE,...)
// DLBL CLUSTER,'userid.KSDS.CLUSTER',,VSAM
// DLBL AIXUNIQ,'userid.KSDS.UPATH',,VSAM
// DLBL AIXNUNQ,'userid.KSDS.NUPATH',,VSAM
// EXEC EDCXGVS2,SIZE=AUTO
/*

/* -------------------------------------------------------------------
/* Print out the cluster
/* -------------------------------------------------------------------
// EXEC IDCAMS,SIZE=AUTO
   PRINT -
       INFILE(CLUSTER) CHAR
/*
/&
```

Figure 12. KSDS Example (JCL) (Part 3 of 3)

RRDS Example

The following program illustrates the use of an RRDS file. It performs the following operations:

1. Opens an RRDS file in record mode (the cluster must be defined)
2. Writes three records (RRN 2, RRN 10, and RRN 32)
3. Sets the file position to the first record
4. Reads the first record in the file
5. Deletes it
6. Locates the last record in the file and sets the access direction to backwards
7. Reads the record
8. Updates the record
9. Sets the _EDC_RRDS_HIDE_KEY environment variable
10. Reads the next record in sequence (RRN 10) into a character string

EDCXGVS4

```
 /* EDCXGVS4
    This example illustrates the use of an RRDS file
  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct rrds_struct {
   __rrds_key_type   rrds_key;
   char             *rrds_buf;
};

typedef struct rrds_struct RRDS_STRUCT;

main() {

FILE            *fileptr;
RRDS_STRUCT      RRDSstruct;
RRDS_STRUCT     *rrds_rec = &RRDSstruct;
char             buffer1[80] =
                    "THIS IS THE FIRST RECORD IN THE FILE.  I"
                    "T WILL BE WRITTEN AT RRN POSITION 2.   ";
char             buffer2[80] =
                    "THIS IS THE SECOND RECORD IN THE FILE. I"
                    "T WILL BE WRITTEN AT RRN POSITION 10.  ";
char             buffer3[80] =
                    "THIS IS THE THIRD RECORD IN THE FILE.  I"
                    "T WILL BE WRITTEN AT RRN POSITION 32.  ";
char             outputbuf[80];
unsigned long    flocate_key = 0;
```

Figure 13. RRDS Example (Part 1 of 3)

```
 /*------------------------------------------------------------------*/
 /*| select RRDS record structure 2 by setting __fill to 1          */
 /*|                                                                */
 /*| 1. open an RRDS file record mode  (the cluster must be defined) */
 /*| 2. write three records (RRN 2, RRN 10, RRN 32)                 */
 /*------------------------------------------------------------------*/
    rrds_rec->rrds_key.__fill = 1;

    fileptr = fopen("DD:RRDSFIL", "wb+,type=record");
    if (fileptr == NULL) {
       perror("fopen");
       exit(99);
    }
    rrds_rec->rrds_key.__recnum = 2;
    rrds_rec->rrds_buf = buffer1;
    fwrite(rrds_rec,1,88, fileptr);

    rrds_rec->rrds_key.__recnum = 10;
    rrds_rec->rrds_buf = buffer2;
    fwrite(rrds_rec,1,88, fileptr);

    rrds_rec->rrds_key.__recnum = 32;
    rrds_rec->rrds_buf = buffer3;
    fwrite(rrds_rec,1,88, fileptr);

/*------------------------------------------------------------------*/
 /*| 3. set file position to the first record                       */
 /*| 4. read the first record in the file                           */
 /*| 5. delete it                                                   */
 /*------------------------------------------------------------------*/
    flocate(fileptr, &flocate_key, sizeof(unsigned long), __KEY_FIRST);

    memset(outputbuf,0x00,80);
    rrds_rec->rrds_buf = outputbuf;

    fread(rrds_rec,1, 88, fileptr);
    printf("The first record in the file (this will be deleted):\n");
    printf("RRN %d: %s\n\n",rrds_rec->rrds_key.__recnum,outputbuf);

    fdelrec(fileptr);
```

Figure 13. RRDS Example (Part 2 of 3)

```
 /*-----------------------------------------------------------------*/
 /*| 6. locate last record in file and set access direction backwards*/
 /*| 7. read the record                                              */
 /*| 8. update the record                                            */
 /*-----------------------------------------------------------------*/
    flocate(fileptr, &flocate_key, sizeof(unsigned long), __KEY_LAST);

    memset(outputbuf,0x00,80);
    rrds_rec->rrds_buf = outputbuf;

    fread(rrds_rec,1, 88, fileptr);
    printf("The last record in the file (this one will be updated):\n");
    printf("RRN %d: %s\n\n",rrds_rec->rrds_key.__recnum,outputbuf);

    memset(outputbuf,0x00,80);
    memcpy(outputbuf,"THIS IS THE UPDATED STRING... ",30);
    fupdate(rrds_rec,88,fileptr);

/*-----------------------------------------------------------------*/
 /*| 9. set _EDC_RRDS_HIDE_KEY environment variable                */
 /*|10. read the next record in sequence (ie. RRN 10) into a       */
 /*|    + character string                                         */
 /*-----------------------------------------------------------------*/

    setenv("_EDC_RRDS_HIDE_KEY","Y",1);
    memset(outputbuf,0x00,80);
    fread(outputbuf, 1, 80, fileptr);
    printf("The middle record in the file (read into char string):\n");
    printf("%80s\n\n",outputbuf);

    fclose(fileptr);
}
```

Figure 13. RRDS Example (Part 3 of 3)

`fldata()` Behavior

Following is a sample of `fldata()` settings for an RRDS file opened as `wb+` in record mode.

If the file is opened by `DLBL/TLBL`-name and/or logical unit, the *filename* field returned by `fldata()` is the same as the *filename* parameter specified on the call to `fopen()`, including the `DD:` prefix. Otherwise, if opened by file ID, the *filename* field is the fully qualified data set name including quotation marks.

```
     __recfmF             = 1   /* fixed record format (all RRDS)     */
     __recfmV             = 0   /* variable record format             */
     __recfmU             = 0   /* undefined record format            */
     __recfmS             = 0   /* N/A                                */
     __recfmBlk           = 0   /* FALSE                              */
     __recfmASA           = 0   /* FALSE                              */
     __recfmM             = 0   /* N/A                                */
     __dsorgPO            = 0   /* N/A                                */
     __dsorgPDSmem        = 0   /* N/A                                */
     __dsorgPDSdir        = 0   /* N/A                                */
     __dsorgPS            = 0   /* N/A                                */
     __dsorgConcat        = 0   /* N/A                                */
     __dsorgMem           = 0   /* N/A                                */
     __dsorgHiper         = 0   /* N/A                                */
     __dsorgTemp          = 0   /* N/A                                */
     __dsorgVSAM          = 1   /* always TRUE for VSAM files         */
     __reserve1           = 0   /* N/A                                */
     __openmode           = 2   /* __BINARY, __RECORD, __TEXT         */
```

```
__modeflag          = 10  /* __READ, __WRITE, ...              */
__reserve2          = 0   /* N/A                               */

__device            = 0   /* __DISK                            */
__blksize           = max record length + size of RRN field (8)
__maxreclen         = max record length + size of RRN field (8)
__vsamtype          = 3   /* 0=__NOTVSAM, 1=__ESDS, 2=__KSDS    */
                          /* 3=__RRDS, 4=__ESDS_PATH,          */
                          /* 5=__KSDS_PATH                     */
__vsamkeylen        = 0   /* N/A for RRDS                      */
__vsamRKP;          = 0   /* N/A for RRDS                      */
__dsname            = jobname.RRDS.CLUSTER
__reserve4          = 0   /* N/A                               */
```

All values shown whose name starts with two underscore characters (__) have `#define`s for them in stdio.h. For a complete explanation of the `fldata()` function, see *LE/VSE C Run-Time Library Reference*.

Chapter 11. Performing Memory File I/O Operations

This chapter describes how to perform memory file I/O operations.

LE/VSE C Run-Time supports files known as *memory files*. Memory files are temporary work files that are stored in main memory rather than in external storage.

Memory files can be written to, read from, and repositioned within like any other type of file. Memory files exist for the life of your root program, unless you explicitly delete them by using the `remove()` or `clrmemf()` functions. The root program is the first `main()` to be invoked. Any `main()` program called by a `system()` call is known as a *child program*. When the root program terminates, LE/VSE C Run-Time removes memory files automatically. Memory files may give you better performance than other types of files.

Note: There may not be a one-to-one correspondence between the bytes in a memory file and the bytes in some other external representation of the file, such as a disk file. Applications that mix open modes on a file (for example, writing a file as text file and reading it back as binary) may not port readily from external I/O to memory file I/O.

See Chapter 6, "LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)," on page 29 for information about using wide-character I/O with LE/VSE C Run-Time.

Opening Files

To open a memory file, use the standard C `fopen()` or `freopen()` library functions. These are described in general terms in *LE/VSE C Run-Time Library Reference*. Details about them specific to all LE/VSE C Run-Time I/O are discussed in Chapter 3, "Opening Files," on page 15. This section describes considerations for using `fopen()` and `freopen()` with memory files.

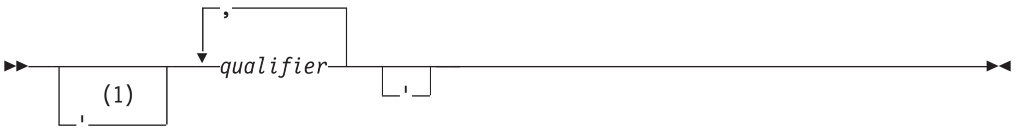
Memory files are always treated as binary streams of bytes, regardless of the parameters you specify on the function call that opens them.

Using `fopen()` or `freopen()`

Files are opened with a call to `fopen()` or `freopen()` in the format: `fopen("filename", "mode")`.

Filenames for Memory Files

Using a File ID: The syntax for the *filename* argument on your `fopen()` or `freopen()` call when using a file ID is shown in the following diagram:



Notes:

1 The single quotation marks must be matched; if you use one, you must use the other.

A sample construct is:

```
'qualifier1.qualifier2'
```

' When you enclose a file ID in single quotation marks, the file ID is *fully qualified*. The file opened is the one specified by the file ID inside the quotation marks. If the file ID is not fully qualified, LE/VSE C Run-Time appends the job name to the front of the file ID. For example, the statement `fopen("a.b","w");` opens a file *jobname*.A.B, where *jobname* is the name of the job submitted. If the file ID is fully qualified, LE/VSE C Run-Time does not append a job name.

qualifier
Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national ($, #, @), the hyphen, or the character X'C0'. The first character should be either alphabetic or national.

You can join qualifiers with periods. The maximum length of a file ID is 44 characters, including periods.

Note: *filename* may optionally be specified as "*" to have LE/VSE C Run-Time generate a temporary name for the memory file. See "Performance Tips" on page 123 for additional information.

Using a DLBL/TLBL-name and/or Logical Unit: You can specify names that comply with the rules for using a `DLBL/TLBL`-name and/or logical unit with `fopen()` as described for SAM I/O on page 48 and for VSAM I/O on page 86. However, LE/VSE C Run-Time treats the entire file description, including the `dd:`, as the *filename*.

Using a VSE/Librarian Sublibrary Member: You can specify names that comply with the rules for using a VSE/Librarian sublibrary member with `fopen()` as described for VSE/Librarian I/O on page 73. However, LE/VSE C Run-Time treats the entire file description, including the `dd:`, as the *filename*.

`fopen()` and `freopen()` Parameters

The following table lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for memory file I/O, and lists the values that are valid for the applicable ones.

Table 26. Parameters for the `fopen()` and `freopen()` Functions for Memory File I/O

Parameter	Allowed?	Applicable?	Notes
`recfm=`	Yes	No	If you specify a RECFM, it must have correct syntax. Otherwise ignored for memory files.
`lrecl=`	Yes	No	If you specify an LRECL, it must have correct syntax. Otherwise ignored for memory files.
`blksize=`	Yes	No	If you specify a BLKSIZE, it must have correct syntax. Otherwise ignored for memory files.
`acc=`	Yes	No	Ignored for memory files.
`password=`	Yes	No	Ignored for memory files.
`space=`	Yes	No	Ignored for memory files.
`type=`	Yes	Yes	Must be `memory`. See the parameter list below for details.
`asis`	Yes	No	Ignored.
`byteseek`	Yes	No	Ignored for memory files, as they use byteseeking by default.
`noseek`	Yes	No	Ignored for memory files.
`OS`	No	No	May cause errors for memory files.
`rewind=`	Yes	No	Ignored for memory files.
`dsn=`	Yes	No	Ignored for memory files.

`recfm=`
LE/VSE C Run-Time parses your specification for this value. If it does not have the correct syntax, your function call fails. If it does, LE/VSE C Run-Time ignores its value and continues.

`lrecl=` and `blksize=`
LE/VSE C Run-Time parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, LE/VSE C Run-Time ignores their values and continues.

`acc=`
This parameter is ignored for memory files.

`password=`
This parameter is ignored for memory files.

`space=`
This parameter is ignored for memory files.

`type=`
To create a memory file, you must specify `type=memory`. You cannot specify `type=record`; if you do, `fopen()` or `freopen()` fails.

For parameter compatibility with C on other platforms, specification of `type=memory(hiperspace)` is treated as if `type=memory` was specified.

`asis`
If you specify this parameter, LE/VSE C Run-Time ignores it.

`byteseek`
This parameter is ignored for memory files.

`noseek`
This parameter is ignored for memory files.

OS
: This parameter is not allowed for memory files. If you specify it, it may cause errors.

rewind=
: This parameter is ignored for memory files.

dsn=
: This parameter is ignored for memory files.

Once a memory file has been created, it can be accessed by the phase that created it as well as by any function or phase that is subsequently invoked (including phases that are called using the `system()` library function), and by any phases in the current chain of `system()` calls. Once the file has been created, you can open it with the same name, without specifying the `type=memory` parameter. You cannot specify `type=record` for a memory file.

This is how LE/VSE C Run-Time searches for memory files:
1. `fopen("my.file","w....,type=memory");` LE/VSE C Run-Time checks the open files to see whether a file with that name is already open. If not, it creates a memory file.
2. `fopen("my.file","w......");` LE/VSE C Run-Time checks the open files to see whether a file with that name is already open. If not, it then checks to see whether a memory file exists with that name. If so, it opens the memory file; if not, it creates a disk file.
3. `fopen("my.file","a.....,type=memory");` LE/VSE C Run-Time checks the open files to see whether a file with that name is already open. If not, it searches the existing memory files to see whether a memory file exists with that name. If so, LE/VSE C Run-Time opens it; if not, it creates a new memory file.
4. `fopen("my.file","a....");` LE/VSE C Run-Time checks the open files to see whether a file with that name is already open. If not, LE/VSE C Run-Time searches existing files (both disk and memory) according to file mode, and opens the first file that has that name. If there is no such file, LE/VSE C Run-Time creates a disk file.
5. `fopen("my.file","r....,type=memory");` LE/VSE C Run-Time searches the memory files to see whether a file with that name exists. If one does, LE/VSE C Run-Time opens it. Otherwise, the `fopen()` call fails.
6. `fopen("my.file","r....");` LE/VSE C Run-Time searches first through memory files. If it does not find the specified one, it then tries to open a disk file.

If you specify a memory *filename* that has an asterisk (*) as the first (or only) character, a name is created for that file (you can acquire this name by using `fldata()`.). For example, you can specify `fopen("*","type=memory");`. Opening a memory file this way is faster than using the `tmpnam()` function.

All valid *filename*s are accepted for memory files. However, if invalid disk *filename*s are used for memory files, difficulties could occur if you try to port memory-file applications to disk-file applications.

Memory files are always opened in fixed binary mode regardless of the open mode. There is no blank padding, and control characters such as the new line are written directly into the file (even if the `fopen()` specifies text mode).

Buffering

Memory files are not buffered. Any parameters passed to `setvbuf()` are ignored. Each character that you write is written directly to the memory file.

Reading from Files

You can use the following library functions to read information from memory files:
- `fread()`
- `fgets()`
- `gets()`
- `fgetc()`
- `getc()`
- `getchar()`
- `scanf()`
- `fscanf()`

See *LE/VSE C Run-Time Library Reference* for more information on these library functions.

The `gets()`, `getchar()`, and `scanf()` functions read from `stdin`, which can be redirected to a memory file.

You can open an existing file for read one or more times, even if it is already open for write. You cannot open a file for write if it is already open (for either read or write). If you want to update or truncate a file or append to a file that is already open for reading, you must first close all the other streams that refer to that file.

For memory files, a read operation directly after a write operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails. LE/VSE C Run-Time treats the following as read operations:
- Calls to read functions that request `0` bytes
- Read requests that fail because of a system error
- Calls to the `ungetc()` function

Writing to Files

You can use the following library functions to write to a file:
- `fwrite()`
- `printf()`
- `fprintf()`
- `vprintf()`
- `vfprintf()`
- `puts()`
- `fputs()`
- `fputc()`
- `putc()`
- `putchar()`

See *LE/VSE C Run-Time Library Reference* for more information on these library functions.

The `printf()`, `puts()`, `putchar()`, and `vprintf()` functions write to `stdout`, which can be redirected to a memory file.

LE/VSE C Run-Time counts a call to a write function writing `0` bytes, or a write request that fails because of a system error, as a write operation. For memory files, the only possible system error that can occur is an error in acquiring storage.

Flushing Records

`fflush()` does not move data from an internal buffer to a memory file, because the data is written to the memory file as it is generated. However, `fflush()` does make the data visible to readers who have a memory file open for reading while a user has it open for writing.

The `fclose()` function also invokes `fflush()` when it detects an incomplete buffer for a file that is open for writing or appending.

`ungetc()` Considerations

`ungetc()` pushes characters back onto the input stream for memory files. `ungetc()` handles only single-byte characters. You can use it to push back as many as four characters onto the `ungetc()` buffer. For every character pushed back with `ungetc()`, `fflush()` backs up the file position by one character and clears all the pushed-back characters from the stream. Backing up the file position may end up going across a record boundary.

If you want `fflush()` to ignore `ungetc()` characters, you can set the _EDC_COMPAT environment variable. See Chapter 21, “Using Environment Variables,” on page 219 for more information.

Repositioning within Files

You can use the following library functions to help you position within a memory file:
- `fgetpos()`
- `fsetpos()`
- `fseek()`
- `ftell()`
- `rewind()`

See *LE/VSE C Run-Time Library Reference* for more information on these library functions.

Using `fseek()` to seek past the end of a memory file extends the file using null characters. This may cause LE/VSE C Run-Time to attempt to allocate more storage than is available as it tries to extend the memory file.

When you use the `fseek()` function with memory files, it supports byte offsets from `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.

All file positions from `ftell()` are relative byte offsets from the beginning of the file. `fseek()` supports these values as offsets from `SEEK_SET`.

`fgetpos()`, `fseek()` with an offset of `SEEK_CUR`, and `ftell()` handle `ungetc()` characters unless you have set the _EDC_COMPAT environment variable, in which case `fgetpos()` and `fseek()` do not. See Chapter 21, “Using Environment Variables,” on page 219 for more information about _EDC_COMPAT. If in handling these characters, if the current position goes beyond the start of the file, `fgetpos()` returns the `EOF` value, and `ftell()` returns `-1`.

fgetpos() values generated by code from previous releases of C are not supported by fsetpos().

Closing Files

Use the fclose() library function to close a memory file. See *LE/VSE C Run-Time Library Reference* for more information on this library function. LE/VSE C Run-Time automatically closes memory files at the termination of the C root main environment.

Performance Tips

Memory files perform more efficiently if large amounts of data (10K or more) are written in one request (that is, if you pass 10K or more of data to the fwrite() function). You should use fopen("*","type=memory") both to generate a name for a memory file and to open the file instead of calling fopen() with a name returned by tmpnam(). You can acquire the file's generated name by using fldata().

Removing Memory Files

The memory file remains accessible until the file is removed by the remove() or clrmemf() library functions or until the root program has terminated. You cannot remove an open memory file, except when you use clrmemf(). See *LE/VSE C Run-Time Library Reference* for more information on these library functions.

fldata() Behavior

When you call the fldata() function for an open memory file, it fills in a data structure as shown in the following.

If the *filename* specified on the call to fopen() begins with DD: (that is, the file is opened by DLBL/TLBL-name and/or logical unit, or VSE/Librarian sublibrary member), the *filename* field returned by fldata() is the same as the *filename* parameter specified on the call to fopen(), including the DD: prefix.

If opened by file ID, the *filename* field returned is the fully qualified data set name including quotation marks.

If opened using an "*", the *filename* field returned is the name generated by the tmpnam() function.

```
struct __fileData {
     unsigned int   __recfmF      : 1,  /* TRUE                            */
                    __recfmV      : 1,  /* FALSE                           */
                    __recfmU      : 1,  /* FALSE                           */
                    __recfmS      : 1,  /* FALSE                           */
                    __recfmBlk    : 1,  /* FALSE                           */
                    __recfmASA    : 1,  /* FALSE                           */
                    __recfmM      : 1,  /* FALSE                           */
                    __dsorgPO     : 1,  /* N/A                             */
                    __dsorgPDSmem : 1,  /* N/A                             */
                    __dsorgPDSdir : 1,  /* N/A                             */
                    __dsorgPS     : 1,  /* always TRUE                     */
                    __dsorgConcat : 1,  /* N/A                             */
                    __dsorgMem    : 1,  /* TRUE                            */
                    __dsorgHiper  : 1,  /* N/A                             */
                    __dsorgTemp   : 1,  /* N/A                             */
                    __dsorgVSAM   : 1,  /* N/A                             */
                    __reserve1    : 1,  /* N/A                             */
                    __openmode    : 2,  /*                                 */
```

```
                __modeflag    : 4,  /*                              */
                __reserve2    : 9;

     char           __device;          /* __MEMORY                     */
     unsigned long  __blksize,         /*                              */
                    __maxreclen;       /*                              */
     unsigned short __vsamtype;        /* N/A                          */
     unsigned long  __vsamkeylen;      /* N/A                          */
     unsigned long  __vsamRKP;         /* N/A                          */
     char *         __dsname;          /* name used on fopen() call    */
     unsigned int   __reserve4;
};
```

For a complete explanation of the `fldata()` function, see *LE/VSE C Run-Time Library Reference*.

Example Program

The following example shows the use of a memory file. The program EDCXGMF3 creates a memory file, calls program EDCXGMF4 and redirects the output of the called program to the memory file. When control returns to the first program, the program reads and prints the string in the memory file.

For more information on the `system()` library function, see *LE/VSE C Run-Time Library Reference*.

EDCXGMF3

```
 /* EDCXGMF3
    This example demonstrates the use of a memory file.
    Part 1 of 2-other file is EDCXGMF4.
  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    char buffer[20];
    char *rc;

 /* Open the memory file to create it */
    if ((fp = fopen("PROG.DAT","wb+,type=memory")) != NULL)
     {
       /* Close the memory file so that it can be used as stdout */
       fclose(fp);

       /* Call EDCXGMF4 and redirect its output to memory file   */
       /* EDCXGMF4 must be an executable PHASE                   */
       system("EDCXGMF4 >PROG.DAT");
```

Figure 14. Memory File Example 1 (Part 1 of 2)

```
 /* Now print the string contained in the file */

       fp = fopen("PROG.DAT","rb");
       rc = fgets(buffer,sizeof(buffer),fp);
       if (rc == NULL)
       {
          perror(" Error reading from file ");
          exit(99);
       }
       printf("%s", buffer);
     }

     return(0);
}
```

Figure 14. Memory File Example 1 (Part 2 of 2)

EDCXGMF4

```
 /* EDCXGMF4
    This example demonstrates the use of a memory file.
    Part 2 of 2-other file is EDCXGMF3.
  */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
       char item1[] = "Hello World\n";
       int rc;

 /* Write the data to the stdout which, at this point, has been
    redirected to the memory file */
       rc = fputs(item1,stdout);
       if (rc == 0) {
          perror("Error putting to file ");
          exit(99);
       }

       return(0);

}
```

Figure 15. Memory File Example 2

Chapter 12. Performing CICS I/O Operations

LE/VSE C Run-Time under CICS supports only three kinds of I/O:

CICS I/O
LE/VSE C Run-Time applications can access the CICS I/O commands through the CICS command level interface. For additional information regarding this interface, see *CICS Transaction Server for VSE/ESA Application Programming Guide* and *CICS Transaction Server for VSE/ESA Application Programming Reference*.

Files
Memory files are the only type of file that LE/VSE C Run-Time supports under CICS.

Note: Under CICS, even if the *filename* specified on the call to `fopen()` or `freopen()` is an unqualified file ID (quotation marks omitted), no prefixing by *jobname* will be performed.

VSAM files can be accessed through the CICS command level interface.

CICS data queues
Under CICS, LE/VSE C Run-Time implements the standard output (`stdout`) and standard error (`stderr`) streams as CICS transient data queues. These data queues must be defined in the CICS Destination Control table (DCT) by the CICS system administrator before the CICS cold start. Output from all users' transactions that use `stdout` (or `stderr`) is written to the queue in the order of occurrence. To help differentiate the output, the user's terminal name, the CICS transaction identifier, and the current date and time is automatically added by the run-time library to the beginning of each line written to the queue.

The queues are as follows:

Stream	Queue
`stdout`	CESO
`stderr`	CESE
`stdin`	Not supported

To access any other queues, you must use the command level interface.

For complete information about using LE/VSE C Run-Time and LE/VSE C Run-Time I/O under CICS, see "Using Input and Output" on page 275.

For information on using wide characters in the CICS environment, see Chapter 6, "LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS)," on page 29.

Chapter 13. Performing LE/VSE Message File Operations

This chapter describes input and output with the LE/VSE message file. This file is write-only; it is non-readable and non-seekable.

The default open mode for the LE/VSE Message File is text. Binary and record I/O modes are not supported.

See Chapter 6, “LE/VSE C Run-Time Support for the Double-Byte Character Set (DBCS),” on page 29 for information about using wide-character I/O with LE/VSE C Run-Time.

The standard stream `stderr` defaults to using the LE/VSE message file.

Opening Files

The default is for `stderr` to go to the message file automatically. The message file is available only as `stderr`; you cannot use the `fopen()` or `freopen()` library function to open it or to change the access mode.

The record format is always treated as undefined (`U`) and the logical record length is always treated as 255 (the maximum length defined by LE/VSE Message File system write interface).

Reading from Files

The LE/VSE Message file is non-readable.

Writing to Files

- Data written to the LE/VSE Message File is always appended to the end of the file.
- When the data written is longer than the maximum LRECL of the LE/VSE Message File (255 bytes), it is written by LE/VSE C Run-Time to the LE/VSE Message File 255 bytes at a time, with the last write possibly less than 255 bytes. No truncation will occur.
- When the output data is shorter than the actual LRECL of the LE/VSE Message File, it is padded with blank characters by the LE/VSE system write interface.
- When the output data is longer than the actual LRECL of the LE/VSE Message File, it is split into multiple records by the LE/VSE system write interface. The LE/VSE system write interface splits the output data at the last blank before the LRECL-th byte, and begins writing the next record with the first nonblank character. Note that if there are no blanks in the first LRECL bytes (DBCS for instance), the LE/VSE system write interface splits the output data at the LRECL-th byte. It also closes off any DBCS string on the first record with a X'0F' character, and begins the DBCS string on the next record with a X'0E' character.
- The hex characters X'0E' and X'0F' have special meaning to the LE/VSE system write interface. The LE/VSE system write interface removes adjacent pairs of these characters (normalization).
- You can set up a `SIGIOERR` handler to catch system write errors. See Chapter 14, “Debugging I/O Programs,” on page 131 for more information.

Flushing Buffers

The `fflush()` function has no effect on the LE/VSE Message File.

Repositioning within Files

The `ftell()`, `fgetpos()`, `fseek()`, and `fsetpos()` functions are not allowed, because the LE/VSE Message File is a non-seekable file. The `rewind()` function only resets error flags.

Closing Files

Do not use the `fclose()` library function to close the LE/VSE message file. LE/VSE C Run-Time automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend.

Chapter 14. Debugging I/O Programs

This chapter will help you locate and diagnose problems in programs that use input and output. It discusses several diagnostic methods specific to I/O. Diagnostic methods for I/O errors include:

- Using return codes from I/O functions
- Using `errno` values and the associated `perror()` message
- Using the `__amrc` structure
- Using the `__amrc2` structure

The information provided with the return code of I/O functions and with the `perror()` message associated with `errno` values may help you locate the source of errors and the reason for program failure. Since return codes and `errno` values do not exist for every possible system I/O failure, return codes and `errno` values are not useful for diagnosing all I/O errors. This chapter discusses the use of the `__amrc` structure and the `__amrc2` structure.

Using the `__amrc` Structure

`__amrc` is a structure defined in stdio.h (when the compile-time option `LANGLVL(EXTENDED)` is in effect) to help you determine errors resulting from an I/O operation. This structure is changed during system I/O and some C specific error situations.

When looking at `__amrc`, be sure to copy the structure into a temporary structure of type `__amrctype` since any I/O function calls will change the value of `__amrc`.

Figure 16 on page 132 shows the `__amrc` structure as it appears in stdio.h.

```
typedef struct __amrctype {

   union {      1
      long int __error;     2


      struct {
        unsigned short __syscode,
                       __rc;
      } __abend;     3
      struct {
         unsigned char __fdbk_fill,
                       __rc,
                       __ftncd,
                       __fdbk;
      } __feedback;     4
      struct {
         unsigned short __svc99_info,
                        __svc99_error;
      } __alloc;      5
   } __code;
   unsigned long __RBA;      6

   unsigned int      __last_op;      7
   struct {
    unsigned long  __len_fill;
    unsigned long  __len;
    char           __str[120];
    unsigned long  __parmr0;
    unsigned long  __parmr1;
    unsigned long  __fill2[2];
    char           __str2[64];
   } __msg;      8
} __amrc_type;
```

Figure 16. __amrc Structure

1 __code
The error or warning value from an I/O operation is in either `__error`, `__abend`, `__feedback`, or `__alloc`. You must look at `__last_op` to determine how to interpret the `__code` union.

2 __error
This field contains the return code from the system macro or utility. Refer to Table 27 on page 134 for further information.

3 __abend
This structure contains the abend code when `errno` is set to indicate a recoverable I/O abend. `__syscode` is the system cancel code and `__rc` is set to zero. The macros `__abendcode()` and `__rsncode()` may be set to the abend code and reason code of a command when invoked with `system()`.

4 __feedback
This structure is used for VSAM only. The `__rc` field stores the VSAM register 15 and the `__fdbk` field stores the VSAM error code or reason code. See also the `__RBA` field below.

5 __alloc
This structure contains no valid information under VSE.

6 __RBA
This is the relative byte address (RBA) value returned by VSAM after an

ESDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. It may be used in subsequent calls to `flocate()`.

7 **`__last_op`**
This field contains a value that indicates the last I/O operation being performed by LE/VSE C Run-Time at the time the error occurred. These values are shown in Table 27 on page 134.

8 **`__msg`**
This may contain an error message from read or write operations, but is not always filled.

This field is used by the `SIGIOERR` handler.

Figure 17 demonstrates how to print the `__amrc` structure after an error has occurred to get information that may help you to diagnose an I/O error.

EDCXGDI1

```
 /* EDCXGDI1
    This example demonstrates how to print the __amrc structure
  */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   FILE *fp;
   __amrc_type save_amrc;
   char buffer[80];
   int i = 0;

   /* open a binary file */

   fp = fopen("testfull.file","wb, recfm=F, lrecl=80");
   if (fp == NULL) exit(99);

   memset(buffer, 'A', 80);

   /* write to file until it runs out of extents */

   while (fwrite(buffer, 1, 80, fp) == 80)
      ++i;

   save_amrc = *__amrc;  /* need copy of __amrc structure */

   printf("number of successful fwrites of 80 bytes = %d\n", i);

   printf("last fwrite errno=%d lastop=%d syscode=%X rc=%d\n",
           errno,
           save_amrc.__last_op,
           save_amrc.__code.__abend.__syscode,
           save_amrc.__code.__abend.__rc);

   return 0;
}
```

Figure 17. Example of Printing the `__amrc` Structure

The program writes to a file until it is full. When the file is full, the program will fail. Following the I/O failure the program makes a copy of the `__amrc` structure, and prints the number of successful writes to the file, the `errno`, the `__last_op` code, the abend system code and the return code.

Using the `__amrc2` Structure

The `__amrc2` structure is an extension of `__amrc`. There are only 2 fields defined for `__amrc2`. Like the `__amrc` structure, `__amrc2` is changed during system I/O and some C specific error situations.

Note: See "Using the SIGIOERR Signal" on page 137 for information on restrictions that exist when comparing file pointers if you are using the `__amrc2` structure.

Figure 18 shows the `__amrc2` structure as it appears in stdio.h.

```
struct {
    long int       __error2;     1                    */
    FILE          *__fileptr;    2                    */
    long int       __reserved[6];
}
```

Figure 18. `__amrc2` Structure

1 **`__error2`**
This field is not used under VSE.

2 **`__fileptr`**
This field is used by the signal `SIGIOERR` to pass back a `FILE` pointer that can then be passed to `fldata()` to get the name of the file causing the error.

Using `__last_op` Codes

The `__last_op` field is the most important of the `__amrc` fields. It defines the last I/O operation LE/VSE C Run-Time was performing at the time of the I/O error. You should note that the structure is neither cleared nor set by non-I/O operations so querying this field outside of a `SIGIOERR` handler should only be done immediately after I/O operations. Table 27 lists `__last_op` codes you may receive and where to look for further information.

Table 27. `__last_op` Codes and Diagnosis Information

Code	Further Information
`__IO_INIT`	Will never be seen by `SIGIOERR` exit value given at initialization.
`__BSAM_OPEN`	Sets __error with return code from VSE OPEN macro.
`__BSAM_CLOSE`	Sets __error with return code from VSE CLOSE macro.
`__BSAM_READ`	No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).
`__BSAM_NOTE`	NOTE returned `0` unexpectedly, no return code.
`__BSAM_POINT`	This will not appear as an error lastop.
`__BSAM_WRITE`	No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).

Table 27. `__last_op` Codes and Diagnosis Information (continued)

Code	Further Information
`__BSAM_CLOSE_T`	Not supported under VSE, but retained for compatibility.
`__BSAM_BLDL`	Not supported under VSE, but retained for compatibility.
`__BSAM_STOW`	Not supported under VSE, but retained for compatibility.
`__TGET_READ`	Not supported under VSE, but retained for compatibility.
`__TPUT_WRITE`	Not supported under VSE, but retained for compatibility.
`__IO_DEVTYPE`	Sets __error with return code from EXTRACT macro.
`__IO_TRKCALC`	Sets __error with return code from GETVCE macro.
`__IO_OBTAIN`	Not supported under VSE, but retained for compatibility.
`__IO_LOCATE`	Not supported under VSE, but retained for compatibility.
`__IO_CATALOG`	Not supported under VSE, but retained for compatibility.
`__IO_UNCATALOG`	Not supported under VSE, but retained for compatibility.
`__IO_RENAME`	Not supported under VSE, but retained for compatibility.
`__C_TRUNCATE`	Set when LE/VSE C Run-Time truncates output data. Usually this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an `fwrite()` writing more data than the record can hold. Truncation is always rightmost data. There is no return code.
`__C_FCBCHECK`	Set when LE/VSE C Run-Time FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.
`__C_DBCS_TRUNCATE`	This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A newline is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if MB_CUR_MAX is 1.
`__C_DBCS_SO_TRUNCATE`	This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.
`__C_DBCS_SI_TRUNCATE`	This occurs only when there was not enough room to start a DBCS string and data was written anyway, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.
`__C_DBCS_UNEVEN`	This occurs when an SI is written before the last double byte character is completed, thereby forcing LE/VSE C Run-Time to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.
`__C_CANNOT_EXTEND`	This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a VSE/Librarian sublibrary being opened for update.
`__VSAM_OPEN_FAIL`	Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.
`__VSAM_OPEN_ESDS`	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
`__VSAM_OPEN_RRDS`	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is RRDS.
`__VSAM_OPEN_KSDS`	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is KSDS.

Table 27. __last_op Codes and Diagnosis Information (continued)

Code	Further Information
__VSAM_OPEN_ESDS_PATH	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS PATH or AIX.
__VSAM_OPEN_KSDS_PATH	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is KSDS PATH or AIX.
__VSAM_MODCB	Set when a low level VSAM MODCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_TESTCB	Set when a low level VSAM TESTCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_SHOWCB	Set when a low level VSAM SHOWCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_GENCB	Set when a low level VSAM GENCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_GET	Set when the last op was a low level VSAM GET; if the GET fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_PUT	Set when the last op was a low level VSAM PUT; if the PUT fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_POINT	Set when the last op was a low level VSAM POINT; if the POINT fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_ERASE	Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_ENDREQ	Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_CLOSE	Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets __rc and __fdbk in the __amrc struct.
__QSAM_GET	__error is not set (if abend (errno == 92), __abend is set, otherwise if read error (errno == 66), look at __msg.
__QSAM_PUT	__error is not set (if abend (errno == 92), __abend is set, otherwise if write error (errno == 65), look at __msg.
__QSAM_TRUNC	This is an intermediate operation. You will only see this if an I/O abend occurred.
__QSAM_CLOSE	Sets __error to result of VSE CLOSE macro.
__QSAM_OPEN	Sets __error to result of VSE OPEN macro.
__CICS_WRITEQ_TD	Sets __error with error code from `EXEC CICS WRITEQ TD`.

Using the SIGIOERR Signal

`SIGIOERR` is a signal used by the library to pass control to an error handler when an I/O error occurs. The default for this signal is `SIG_IGN`. Setting up a `SIGIOERR` handler is like setting up any other error handler. The example in Figure 19 on page 138 adds a `SIGIOERR` handler to the example shown in Figure 17 on page 133. Note the way `fldata()` and the `__amrc2` field `__fileptr` are used to get the name of the file that caused the error.

EDCXGDI2

```
 /* EDCXGDI2
    This example demonstrates how to use SIGIOERR
  */

#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

void iohdlr(int);

int main(void) {
   FILE *fp;
   char buffer[80];
   int i = 0;

   signal(SIGIOERR, iohdlr);

   /* open a binary file */

   fp = fopen("testfull.file","wb, recfm=F, lrecl=80");
   if (fp == NULL) exit(99);

   memset(buffer, 'A', 80);

   /* write to file until it runs out of extents */

   while (fwrite(buffer, 1, 80, fp) == 80)
      ++i;

   printf("number of successful fwrites of 80 bytes = %d\n", i);

   return 0;
}

 void iohdlr (int signum) {
   __amrc_type save_amrc;
   __amrc2_type save_amrc2;
   char filename[FILENAME_MAX];
   fldata_t info;

   save_amrc = *__amrc;    /* need copy of __amrc structure  */
   save_amrc2 = *__amrc2;  /* need copy of __amrc2 structure */

   /* get name of file causing error from fldata */

   if (fldata(save_amrc2.__fileptr, filename, &info) == 0)
      printf("error on file %s\n",filename);

   perror("io handler");  /* give errno message */
   printf("lastop=%d syscode=%X rc=%d\n",
           save_amrc.__last_op,
           save_amrc.__code.__abend.__syscode,
           save_amrc.__code.__abend.__rc);

   signal(SIGIOERR, iohdlr);
}
```

Figure 19. Example of Using SIGIOERR

When control is given to a `SIGIOERR` handler, the `__amrc2` structure field `__fileptr` will be filled in with a file pointer. The only operation permitted on the file pointer is `fldata()`. This operation can be used to extract information about the file that caused the error. Other than `freopen()` and `fclose()`, all I/O operations will fail since the file pointer is marked invalid. Do not issue `freopen()` or `fclose()` in a `SIGIOERR` handler that returns control. This will result in unpredictable behavior, likely an abend.

If you choose not to return from the handler, the file is still locked from all operations except `fldata()`, `freopen()`, or `fclose()`. The file is considered open and can prevent other incorrect access, such as a SAM file opened more than once for a write. Like all other files, the file is closed automatically at program termination if not already explicitly closed.

When you exit a `SIGIOERR` handler and do not return, the state of the file at closing is indeterminate. The state of the file is indeterminate because certain control block fields are not set correctly at the point of error and they do not get corrected unless you return from the handler.

For example, if your handler were invoked due to a truncation error and you performed a `longjmp()` out of your `SIGIOERR` handler, the file in error would remain open, yet inaccessible to all I/O functions other than `fldata()`, `fclose()`, and `freopen()`. If you were to close the file or it was closed at termination of the program, it is still likely that the record that was truncated will not appear in the final file.

You should be aware that for a standard stream passed across a `system()` call, the state of the file will be indeterminate even after you return to the parent program. For this reason, you should not jump out of a `SIGIOERR` handler. For further information on `system()` calls and standard streams, see Chapter 7, "Standard Streams and Redirection," on page 37.

I/O with files other than the file causing the error is perfectly valid within a `SIGIOERR` handler. For example, it is valid to call `printf()` in your `SIGIOERR` handler if the file causing the error is not `stdout` Comparing the incoming file pointer to the standard streams is not a reliable mechanism of detecting whether any of the standard streams are in error. This is because the file pointer in some cases is only a pointer to a file structure that points to the same`__file` as the stream supplied by you. The FILE pointers will not be equal if compared, but a comparison of the `__file` fields of the corresponding FILE pointers will be. See the stdio.h header file for details of type FILE.

If `stdout` or `stderr` are the originating files of a `SIGIOERR`, you should open a special log file in your handler to issue messages about the error.

Part 2. Interlanguage Calls with LE/VSE C Run-Time

This part describes the LE/VSE C Run-Time-specific considerations about interlanguage calls in the LE/VSE environment. For complete information about interlanguage calls (ILC) with LE/VSE C Run-Time and LE/VSE, refer to *LE/VSE Writing Interlanguage Communication Applications*.

Chapter 15. Combining C and Assembler

This chapter describes how to communicate between LE/VSE C Run-Time and assembler programs.

There are different prologs and epilogs shipped with LE/VSE. The prolog and epilog shown in this chapter set up your assembler to imitate LE/VSE C Run-Time. For more information on LE/VSE with assembler, see *LE/VSE Writing Interlanguage Communication Applications*.

Establishing the LE/VSE C Run-Time Environment

Before a LE/VSE C Run-Time function can be called from assembler, a suitable environment must be established. Do one of the following:

- Call the assembler program from a C `main()`, even though you simply want to make a call from assembler to LE/VSE C Run-Time. This establishes the C environment. Then you can call LE/VSE C Run-Time from assembler by following the OS linkage conventions. An example is shown on page 146.
- Use preinitialization to set up the C environment. See "Retaining the C Environment Using Preinitialization" on page 149 for more information on this task.

Specifying Linkage for C and Assembler

There are two ways to specify the linkage between LE/VSE C Run-Time and assembler:

- LE/VSE C Run-Time provides a `#pragma linkage` directive that enables it to generate and accept parameter lists, using a linkage convention known as OS linkage. Although functionally different, both *calling* an assembler routine and *being called by* an assembler routine are handled by the same `#pragma`. Its format is:

  `#pragma linkage(`*identifier*`, OS)`

  where *identifier* is the name of the assembler function to be called from C or the C function to be called from assembler. The `#pragma linkage` directive must occur before the call to the entry point.
- In the absence of a `#pragma linkage`, a LE/VSE C Run-Time internal linkage is used. Assembler code written using this method will be more difficult to migrate to improved linkages. While C code can be migrated by recompiling, assembler code will have to be rewritten.

You can call LE/VSE C Run-Time library functions when using the OS `#pragma linkage`, but it must be done indirectly, through intervening C code, as shown in Figure 21 on page 146.

Note: Do not use the macros `va_arg()` and `va_start()` in functions that participate in interlanguage calls.

Parameter List for OS Linkage

A parameter list for OS linkage is a list of pointers. The most significant bit of the last parameter in the parameter list is turned on by the compiler when the list is created.

If a parameter is an address type parameter, the address itself is directly stored into the parameter list. Otherwise, a copy is created for a value parameter and the address of this copy is stored into the parameter list.

The type of a parameter is specified by the prototype of a function. In absence of a prototype, the creation of a parameter list is determined by the types of the actual parameters passed to the function. Figure 20 shows an example of the parameter list for OS linkage.

In the list, the first parameter and the third parameter are value parameters and the second parameter is an address parameter.

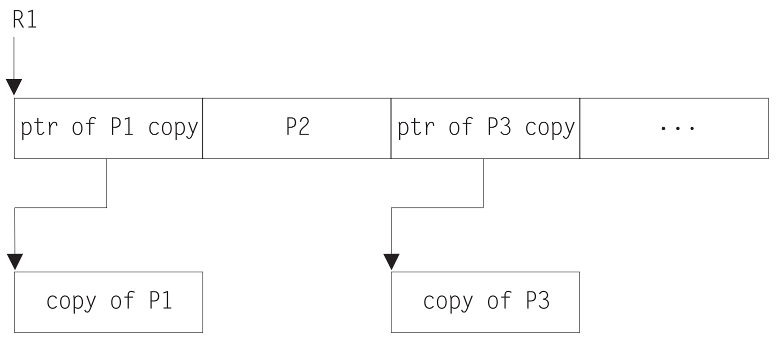


Figure 20. Example of Parameter Lists for OS Linkages

Using Standard Macros

To communicate properly, assembler routines must preserve the use of certain registers and particular storage areas, in a fashion consistent with code from the compiler. LE/VSE C Run-Time provides three macros for use with assembler routines. These macros are in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE). They must be assembled using High Level Assembler. The macros are:

EDCPRLG	Generates the prolog for assembler code
EDCEPIL	Generates the epilog for assembler code
EDCDSAD	Accesses automatic memory

The advantage of writing assembler code using these macros is that the assembler routine will then participate fully in the C environment, enabling the assembler routine to call C functions. To establish a C environment, the main program module must be a C function. The macros also manage automatic storage, and make the assembler code easier to debug.

Assembler Prolog

Use the EDCPRLG macro to generate assembler prolog code at the start of assembler routines.

```
►►──name──EDCPRLG──┬──────────────────────┬──────────────────────────────►◄
                   ├─USRDSAL──=──ulen─────┤
                   ├─BASEREG──=──register─┤
                   └─DSALEN──=──dlen──────┘
```

name Is inserted in the prolog. It is used in the processing of certain exception conditions and is useful in debugging and in reading memory dumps. If *name* is absent, the name of the current CSECT is used.

USRDSAL=*ulen*
: Is used only when automatic storage (in bytes) is needed. To address this storage, see the EDCDSAD macro description. The *ulen* value is the requested length of the user space in the DSA.

BASEREG=*register*
: Designates the required base register. The macro generates code needed for setting the value of the register and for establishing addressability. The default is Register 3. If *register* equals `NONE`, no code for establishing the base and addressability is generated.

DSALEN=*dlen*
: Is the total requested length of the DSA. The default is `120`. If fewer than `120` bytes are requested, `120` bytes are allocated. If both *dlen* and *ulen* are specified, then the greater of *dlen* or *ulen*`+120` is allocated. If `DSALEN=NONE` is specified, no code is generated for DSA storage allocation, and R13 will still point to the caller's DSA. Therefore, you should not use the EDCEPIL macro to terminate the assembler routine. Instead, you have to restore the registers yourself from the current DSA. To do this, you can use an assembler instruction such as

```
LM 13,12,12(13)
```

You should not use EDCDSAD to access automatic memory if you have specified `DSALEN=NONE`, since DSECT is addressable using R13.

Assembler Epilog

Use the EDCEPIL macro to generate assembler epilog code at the end of assembler routines. Do not use this macro in conjunction with an EDCPRLG macro that specifies `DSALEN=NONE`.

```
►►──┬──────┬──EDCEPIL────────────────────────────────────────────────►◄
    └─name─┘
```

name Is the optional name operand, which then becomes the label on the exit from this code. The name does not have to match the prolog.

Accessing Automatic Memory

Use the EDCDSAD macro to access automatic memory. Automatic memory is reserved using the `USRDSAL`, or the `DSALEN` operand of the EDCPRLG macro. The length of the allocated area is derived from the *ulen* and/or *dlen* values specified on the EDCPRLG macro. EDCDSAD generates a DSECT, which reserves space for the stack frame needed for the C environment.

```
►►──┬──────┬──EDCDSAD────────────────────────────────────────────────►◄
    └─name─┘
```

name Is the optional name operand, which then becomes the name of the generated DSECT.

The DSECT is addressable using Register 13 which is initialized by the prolog code. If you have specified DSALEN=NONE with EDCPRLG you should not use EDCDSAD.

Example

The following example shows how to use the #pragma linkage OS and call C library functions from the assembler routine.

EDCXGCA4

```
 /* EDCXGCA4
    This example demonstrates C/Assembler ILC.
    Part 1 of 3-other files are EDCXGCA2, EDCXGCA5.
  */

#pragma linkage(callprtf, OS)

int main(void) {
   callprtf();

   return(0);
}
```

Figure 21. Calling C Library Functions from Assembler Using OS Linkage

EDCXGCA2

```
* EDCXGCA2
* This example demonstrates C/Assembler ILC.
* Part 2 of 3-other files are EDCXGCA4, EDCXGCA5.
CALLPRTF CSECT
         EDCPRLG
         LA    1,ADDR_BLK              parameter address block in r1
         L     15,=V(@PRINTF4)         address of routine
         BALR  14,15                   call it
         EDCEPIL
ADDR_BLK DC   A(FMTSTR)                parameter address block with..
         DC   A(X'80000000'+INTVAL)    ..high bit on the last address
FMTSTR   DC   C'Sample formatting string'
         DC   C' which includes an int -- %d --'
         DC   AL1(NEWLINE,NEWLINE)
         DC   C'and two newline characters'
         DC   AL1(NULL)
*
INTVAL   DC   F'222'           The integer value displayed
*
NULL     EQU  X'00'            C NULL character
NEWLINE  EQU  X'15'            C \n character
         END
```

Figure 22. Calling C Library Functions from Assembler Using OS Linkage

EDCXGCA5

```
 /* EDCXGCA5
    This example demonstrates C/Assembler ILC.
    Part 3 of 3-other files are EDCXGCA2, EDCXGCA4.
  */

 /***********************************************************
 * This routine is an interface between assembler code      *
 * and the LE/VSE C Run-Time library function printf().  OS *
 * linkage will not tolerate variable length parameter      *
 * lists, so this routine is specific to a formatting string *
 * and a single 4-byte substitution parameter.  It is       *
 * specified as an int here.                                *
 ***********************************************************/

#pragma linkage(_printf4,OS) /*function will be called from assembler*/

#include <stdio.h>

int _printf4(char *str,int i) {

   return printf(str,i);  /* call LE/VSE C Run-Time library function */

}
```

Figure 23. Calling C Library Functions from Assembler Using OS Linkage

Register Content at Entry to an ASM Routine Using OS linkage

When control is passed to an assembler routine that uses OS linkage, the contents of the registers are as follows:

Register	Contents
R0	0
R1	Points to the parameter list. The parameter list consists of a vector of addresses, each of which points to an actual parameter. The address of the last parameter has its high order bit set on, to indicate the end of the list.
R2 to R11	Undefined.
R12	Points to an internal control block. It can be used by the called routine but must be restored to its entry value if it calls a routine that expects a LE/VSE C Run-Time environment.
R13	Points to the caller's DSA. Part of the DSA is used by EDCPRLG and EDCEPIL to save and restore registers. If EDCPRLG is in the called routine, it changes R13 from pointing to the caller's DSA to the called routine's DSA.
R14	The return address.
R15	The address of the entry point being called.

Register Content at Exit from an ASM Routine to LE/VSE C Run-Time

Registers have the following content when control returns to the point of call:

Register	Contents
R0	Undefined.
R1	Undefined.
R2 to R13	Must be restored to entry values. This is done by EDCEPIL and EDCPRLG.
R14	Return address.
R15	Return value for integer types (long int, short int, char) and pointer types. Otherwise set to `0`.
FP0	Returns value for float or double parameters.
FP0	Returns value if long double is passed.
FP2	Returns value if long double is passed.

All other floating point registers are undefined.

Retaining the C Environment Using Preinitialization

If an assembler routine is to call the same C program repeatedly, the creation and termination of the C environment for each call will be unnecessarily inefficient. The solution is to create the C environment only once by preinitializing the C program. You must use the callable service CEEPIPI to preinitialize the environment for your applications. For more information about this service, see *LE/VSE Programming Guide*.

Part 3. Coding: Advanced Topics

This part contains information that you may find useful once you become more familiar with using the LE/VSE C Run-Time.

Chapter 16. Reentrancy in LE/VSE C Run-Time

This chapter describes the concept of reentrancy and how it can help to make your programs more efficient, and tells you how to prepare reentrant programs and control writable static in reentrant code.

Reentrant programs are structured to allow more than one user to share a single copy of a phase or to use a phase repeatedly without reloading it. C achieves reentrancy by splitting your program into two parts. The first part, which consists of executable code and constant data, does not change during program execution. The second part may be altered in the course of the program. This part includes the DSA (also known as the stack) and a piece of storage known as the writable static area, which contains all static variables that can be altered. Both of these parts are areas of memory that are maintained until the program terminates.

If the program is installed in the SVA, only a single copy of the first (constant or reentrant part) exists within a single partition, no matter how many users are running the program simultaneously. This reentrant part may be shared across partitions or across sessions. In this case, the phase is loaded only once. Separate concurrent invocations of the program share or reenter the same copy of the phase, which is write-protected. If the program is not installed in the SVA, each invocation receives a private copy of the code part, but this part may not be write-protected.

The modifiable writable static part of the program consists of:
- All program variables with the `static` storage class
- All program variables receiving the `extern` storage class
- All writable strings
- All function descriptors for all referenced functions
- All variable descriptors to reference imported variables

Each user running the program receives a private copy of the second (variable or nonreentrant) part. This part, the code area, is modifiable by each user.

The code part of the program consists of:
- Executable instructions
- Read-only constants
- Objects with the variable `NORENT #pragma`

Reentrant programs can be categorized as having natural or constructed reentrancy. Programs that contain no references to the writable static objects listed above have natural reentrancy. Programs that refer to writable static objects must be processed with the LE/VSE prelinker to make them reentrant; such programs have constructed reentrancy.

You do not need to use the `RENT` compiler option if your program is naturally reentrant.

Limitations of Reentrancy

Reentrancy is only an advantage if there will be concurrent users of a program. These advantages become apparent only when the program is large.

Even if a program is large and will have more than one user at the same time, there are also these limitations to consider:

- Reentrancy requires an extra preparation step if your program contains any writable static. After the program is compiled, you must use the LE/VSE prelinker, as described in *LE/VSE Programming Guide*.
- The shared portion of the program may be installed in the Shared Virtual Area (SVA).

Using the LE/VSE Prelinker for Reentrancy

If your program contains writable static, 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 do not need to use the prelinker to ensure reentrancy. If you compile your code and wish to link it using the VSE system linkage editor, you must first call the LE/VSE prelinker.

The LE/VSE prelinker is not a post-compiler. That is, you do not prelink the object modules individually into separate prelinked object modules as if running the prelinker were an extension of the compile step. Instead, you prelink all the object modules together in the same job into one output prelinked object module. This is because the prelinker cannot process each object deck one at a time—it needs to calculate how the single writable static area for the program will be structured and thus needs all of the object decks input in a single step.

The LE/VSE prelinker:

- Maps input L-names from the object modules to output S-names (8 characters maximum)
- Collects compile-time initialization information on static objects
- Collects objects that exist in writable static into one area by assigning an offset within the writable static area to each object
- Removes all relocation and name information of objects in the writable static area

The output of the LE/VSE prelinker is a single prelinked object module. You can link this object module only on the same platform as where you prelinked it.

Because the prelinker maps names and removes the relocation information, you cannot use the resulting object module as input for another prelink, nor can you use the linkage editor to replace a control section (CSECT) that either defines or references writable static objects.

The LE/VSE prelinker can handle object modules from languages other than C, but only C or assembler code using the macros EDCDXD and EDCLA may refer to writable static objects. To generate a reentrant phase, you must follow these steps:

1. If your program contains no writable static, compile your program as you would normally (that is, no special compiler options are required), and then go directly to step 4.

If you are unsure about whether your program contains writable static, compile it with the RENT option. Invoking the LE/VSE 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.

2. If your program contains writable static, you must compile your C source files using the RENT compiler option.
3. Use the LE/VSE 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 LE/VSE prelinker.
4. Link the program with the SVA option on the PHASE card and install the program in the SVA area of your system.

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

Controlling External Static

Certain program variables with the extern storage class may be constant and never written to. If this is the case, every user does not need to have a separate copy of these variables. In addition, there may be a need to share constant program variables between C and another language.

You can force a variable to be the part of the program that includes executable code and constant data by using the #pragma variable(varname, NORENT) directive. The following program fragment illustrates how this can be done:

```
#pragma options(RENT)

#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
    /* ... */
}
```

Figure 24. Controlling External Static

In this example, the source file is to be compiled with the RENT option. The variable rates is included with the executable code because #pragma variable(rates, NORENT) has been specified. The variable totals is included with the writable static. Each user has their own copy of the array totals, and the array rates is shared among all users of the program.

The #pragma variable(varname, NORENT) does not apply to, and has no effect on, program variables with the static storage class. Program variables with the static storage class are always included with the writable static. An informational message will appear if you do try to write to a nonreentrant variable when the CHECKOUT compiler option is specified.

When #pragma variable(varname, NORENT) is specified for a variable, you must ensure that this variable is never written to. Program exceptions or unpredictable program behavior may result. In addition, #pragma variable(varname, NORENT) must be included in every source file where the variable is referenced or defined.

Controlling Writable Strings

In a large number of programs, character strings may be constant and never written to. If this is the case, every user does not need a separate copy of these strings.

You can force all strings in a given source file to be the part of the program that includes executable code and constant data by using `#pragma strings(readonly)`. The following program fragment illustrates how to make the strings constant:

EDCXGRE1

```
 /* EDCXGRE1
    This example demonstrates how to make strings constant
  */

#pragma strings(readonly)
#include <stdio.h>

int main(void)
{
   printf("hello world\n");

   return(0);
}
```

Figure 25. Making Strings Constant

In this example, the string `"hello world\n"` will be included with the executable code because `#pragma strings(readonly)` is specified. This can yield a performance and storage benefit. Ensure that read-only strings are never written to. Program exceptions or unpredictable program behavior may result if an attempt is made to write to a read-only string.

Using Writable Static in Assembler Code

Programming in C can eliminate most, if not all, of the need to code in assembler, but there may be instances when you must. In addition, you may also have to modify writable static from within an assembler program.

One way to modify writable static is to pass the address of the writable static variable as a parameter to the assembler program. This may be difficult in some cases. Two assembler macros are provided to make this easier:
- EDCDXD
- EDCLA

These are both in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE). The restriction on the names of writable static objects accessible in assembler code is that they are S-names. This means that they may be at most 8 characters long and may contain only characters allowed in external names by the assembler code.

The macro EDCDXD is used to declare a writable static variable. EDCLA is used to load the address of the writable static variable into a register. Using the EDCLA macro in assembler code necessitates coding EDCDXD as well.

The following assembler program illustrates how they are used:

EDCXGRE3

```
* EDCXGRE3                                                        *
* static area, from assembler code.                               *
* Part 1 of 2-other file is EDCXGRE4.                             *
*                                                                 *
* parameters:  none                                               *
* return:      none                                               *
* action:      store contents of register 13 ( callers dynamic    *
*              storage area) in variable DSA which exists in      *
*              the writable static area                           *
*                                                                 *
* Macros:    EDCPRLG, EDCEPIL, EDCDXD, EDCLA in the LE/VSE        *
*            installation sublibrary (default is PRD2.SCEEBASE)   *
*******************************************************************
DSA      EDCDXD 0F          ; declaration of DSA   in writable static
TBL@A    EDCDXD 20F         ; definition  of TBL@A in writable static
GETDSA   CSECT
GETDSA   AMODE ANY
GETDSA   RMODE ANY
         EDCPRLG            ;  prolog (save registers etc.)
         EDCLA 1,DSA        ;  load register 1 with address of DSA
         ST 13,0(,1)        ;  store contents of reg 13 in DSA
         EDCEPIL            ;  epilog (restore registers etc.)
         END
```

Figure 26. Referencing Objects in the Writable Static Area-Part 1

In this example, the external variable DSA is declared with the EDCDXD macro. The size value of 0F (zero fullwords) is used to indicate that DSA is to be treated as an extern declaration in C. Because DSA is an extern declaration and not a definition, DSA must be defined in another C or assembler program. The EDCLA macro is used to load general purpose register 1 with the address of DSA, which exists in the writable static area.

The external variable TBL@A is declared with the EDCDXD macro. It is also defined because its size is 20F (20 fullwords or 80 bytes) and corresponds to an external data definition in C. When the program starts, TBL@A is initialized to zero. Because TBL@A is an external data definition, there should not be another definition of it in a C or assembler program.

When these macros are used, pseudo-registers cannot be used within the same assembler program.

There are no assembler macros for static initialization of a variable with a nonzero value. You can do this by defining and initializing the variable in C and making an extern declaration for it in the assembler program. In the example assembler program, DSA has been declared this way.

The following C program illustrates how to call the above assembler program.

EDCXGRE4

```
 /* EDCXGRE4
    This example shows how to reference objects in the writable
    static area, from assembler code.
    Part 2 of 2-other file is EDCXGRE3.
  */

#include <stdio.h>
#pragma map(tbl_a,"TBL@A")       /* map to match assembler name */

void GETDSA(void);               /* assembler routine modifies DSA */

#define SZ 20                    /* maximum call depth */
extern void * tbl_a[SZ];         /* defined in assembler program */
void * DSA;                      /* define it here, source name */
                                 /* same as assembler name */

 /* call yourself deeper and deeper */
 /* save DSA pointers as you go */
void deeper( int i)
{
  if (i >= SZ)     /* if deep enough just return */
    return;
  GETDSA();        /* assign value to DSA */
  tbl_a[i] = DSA;  /* save value in table */
  deeper(i+1);     /* go deeper in call chain */
}

int main(void) {
  int i;
  deeper(0);
  for(i=0; i<SZ; i++)
    printf("depth %3d, DSA was at %p\n", i, tbl_a[i]);
  return 0;
}
```

Figure 27. Referencing Objects in the Writable Static Area-Part 2

Chapter 17. Using the Decimal Data Type

This chapter describes the fixed-point decimal data types supported by LE/VSE C Run-Time. This chapter refers to fixed-point decimal data types as “decimal types”. The decimal data type is an extension of the ANSI C language definition. You can use decimal data types to represent large numerical quantities accurately, especially in business and commercial applications for financial calculations.

The decimal data types allow expressions of up to `DEC_DIG` significant digits including integral and fractional parts. The value of `DEC_DIG` is specified in the header file decimal.h.

You can pass decimal arguments in function calls and `define` macros. You can also declare decimal variables, typedefs, arrays, structures, and unions having decimal members. The following operators apply on decimal variables:
- Arithmetic
- Relational
- Assignment
- Comma
- Conditional
- Equality
- Logical
- Primary
- Unary

When using the decimal data types, you must include the decimal.h header file in your source code.

Note: To generate more efficient code for decimal operations, use the `OPTIMIZE` compiler option.

Declaring Data Types

Use the type specifier `decimal(`*n*`,`*p*`)` to declare decimal variables and to initialize them with fixed-point decimal constants. The decimal macro is defined in decimal.h.

Decimal types are classified as arithmetic types. `decimal(`*n*`,`*p*`)` designates a decimal number with *n* digits, and *p* decimal places. *n* is the total number of digits for the integral and decimal parts combined, and *p* is the number of digits for the decimal part only. For example decimal(5,2) represents a number, such as 123.45, where *n*=5 and *p*=2. The value for *p* is optional. If it is left out, the default value is `0`.

n and *p* have a range of allowed values according to the following rules:

```
p ≤ n 1 ≤ n ≤ DEC_DIG 0 ≤ p ≤
DEC_PRECISION
```

Note: `DEC_DIG` (the maximum number of digits *n*) and `DEC_PRECISION` (the maximum precision *p*) are defined in decimal.h. Currently, a maximum of 31 digits is used for both limits.

Declaring Fixed-Point Decimal Constants

The syntax for fixed-point decimal constants is:

```
fixed-point-decimal-constant:
  fractional-constant fixed-point-decimal-suffix

fractional-constant (use any one of the following formats):
  digit-sequence . digit-sequence
  . digit-sequence
  digit-sequence .
  digit-sequence

digit-sequence (use any one of the following formats):
  digit
  digit-sequence digit

fixed-point-decimal-suffix (use any one of the following formats):
  D
  d
```

A fixed-point decimal constant has a numeric part and a suffix that specifies its type. The components of the numeric part may include a digit sequence representing the integral part, followed by a decimal point (.), followed by a digit sequence representing the fractional part. Either the integral part, the fractional part, or both are present.

Each fixed-point decimal constant has the attributes *number of digits* (**digits**) and *number of decimal places* (**precision**). No leading or trailing zeros are stripped off when the **digits** and the **precision** are determined.

Table 28 gives examples of fixed-point decimal constants and their corresponding attributes:

Table 28. Fixed-Point Decimal Constants and Their Attributes

Fixed-Point decimal Constant	**(digits, precision)**
1234567890123456D	(16, 0)
12345678.12345678D	(16, 8)
12345678.d	(8, 0)
.1234567890d	(10, 10)
12345.99d	(7, 2)
000123.990d	(9, 3)
0.00D	(3, 2)

Declaring Decimal Variables

The following example shows how you can declare a variable as a decimal data type:

```
decimal(10,2)
 x; decimal(5,0)   y; decimal(5)     z; decimal(18,10) *ptr;
decimal(8,2)   arr[100];
```

In the previous example:
- *x* can have values between -99999999.99D to +99999999.99D.
- *y* and *z* can have values between -99999D to +99999D.
- *ptr* is a pointer to type decimal(18,10).
- *arr* is an array of 100 elements, where each element is of type decimal(8,2).

The syntax for the decimal type specifier is as follows:

```
►►—decimal—(—constant-expression——————————————————————)——————————————►◄
                                  └—, constant-expression—┘
```

The constant-expression is evaluated as a positive integral constant expression. A second constant-expression is optional. If it is left out, the default value is 0. decimal(*n*,*0*) and decimal(*n*) are type compatible.

Defining Decimal Related Constants

Use the following numerical limits to define the decimal value in assignments and expressions. These predefined values are contained in decimal.h.

- Smallest number that can be represented in a decimal type

  DEC_MIN
  -9999999999999999999999999999999D
- Largest positive number that can be represented in a decimal type

  DEC_MAX
  +9999999999999999999999999999999D
- Smallest number greater than zero that can be represented in a decimal type

  DEC_EPSILON
  .0000000000000000000000000000001D
- Maximum number of significant digits that decimal types can hold

  DEC_DIG
  31
- Maximum number of decimal places that decimal types can hold

  DEC_PRECISION
  31

Using Operators

You can use arithmetic, relational, assignment, comma, conditional, equality, logical, primary, and unary cast operators on a decimal data type. Conversions follow these arithmetic conversion rules:

- First, if either operand has type long double, the other operand is converted to type long double.
- Otherwise, if either operand has type double, the other operand is converted to type double.
- Otherwise, if either operand has type float, the other operand is converted to type float.
- Otherwise, if either operand has type decimal, the other operand is converted to type decimal.
- Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:
  – If either operand has type unsigned long int, the other operand is converted to unsigned long int.
  – Otherwise, if one operand has type long int and the other has type unsigned int, if a long int can represent all values of an unsigned int, the operand of

type unsigned int is converted to long int; if a long int cannot represent all the values of an unsigned int, both operands are converted to unsigned long int.

– Otherwise, if either operand has type long int, the other operand is converted to long int.
– Otherwise, if either operand has type unsigned int, the other operand is converted to unsigned int.
– Otherwise, both operands have type int.

Arithmetic Operators

This example shows how to use arithmetic operators and then describes certain arithmetic, assignment, unary, and cast operators in more detail.

It summarizes how to add, subtract, multiply and divide decimal variables.

EDCXGDC1

```
 /* EDCXGDC1
    This example demonstrates arithmetic operations on decimal
    variables
  */

#include <decimal.h>            /* decimal header file */
#include <stdio.h>

int main(void)
{

decimal(10,2) op_1 = 12d;
decimal(5,5) op_2 = -.12345d;
decimal(24,12) op_3 = 12.34d;
decimal(20,5) op_4 = 11.01d;
decimal(14,5) res_add;
decimal(25,2) res_sub;
decimal(15,7) res_mul;
decimal(31,14) res_div;

res_add = op_1 + op_2;
res_sub = op_3 - op_1;
res_mul = op_2 * op_1;
res_div = op_3 / op_4;

printf("res_add =%D(*,*)\n",digitsof(res_add),
       precisionof(res_add),res_add);
printf("res_sub =%D(*,*)\n",digitsof(res_sub),
       precisionof(res_sub),res_sub);
printf("res_mul =%D(*,*)\n",digitsof(res_mul),
       precisionof(res_mul),res_mul);
printf("res_div =%D(*,*)\n",digitsof(res_div),
       precisionof(res_div), res_div);

return(0);
}
```

Figure 28. Arithmetic Operators Example

Additive Operators

Additive operators follow the arithmetic conversions defined in “Using Operators” on page 161.

Note: For performance reasons, generating negative zero is possible.

Refer to “Intermediate Results” on page 165 for details on how to get the convert type during alignment of the decimal point.

Multiplicative Operators

Multiplicative operators follow the arithmetic conversions defined “Using Operators” on page 161.

Note: For performance reasons generating negative zero is possible.

Refer to "Intermediate Results" on page 165 for details on how to get the convert type during alignment of the decimal point.

Relational Operators

Relational operators follow the arithmetic conversions defined in "Using Operators" on page 161.

The following example shows you how to use a relational expression less than (<) for decimals. In this example, decimal types are also compared with other arithmetic types (integer, float, double, long double), and the implicit conversion of the decimal types is performed using the arithmetic conversion rules in "Converting Decimal Data Types" on page 167. Leading zeros in the example are shown to indicate the size of the number of digits in the decimal data type. You do not need to enter leading zeros in your decimal type variable initialization.

EDCXGDC2

```
 /* EDCXGDC2
    This example shows how to use a relational expression with the
    decimal data type
   */
#include <decimal.h>

decimal(10,3) pdval = 0000023.423d;    /* Decimal declaration*/
int ival = 1233;                       /* Integer declaration*/
float fval = 1234.34;                  /* Float declaration*/
double dval = 251.5832;                /* Double declaration*/
long double lval = 37486.234;          /* Long double declaration*/
int main(void)
{
  decimal(15,6) value = 000485860.085999d;
  /*Perform relational operation between other data types and decimal*/
   if (pdval < ival) printf("pdval is the smallest !\n");
   if (pdval < fval) printf("pdval is the smallest !\n");
   if (pdval < dval) printf("pdval is the smallest !\n");
   if (pdval < lval) printf("pdval is the smallest !\n");
   if (pdval < value) printf("pdval is the smallest !\n");

   return(0);
}
```

Figure 29. Relational Operators Example

Refer to "Intermediate Results" on page 165 for details on how to get the convert type during alignment of the decimal point.

Equality Operators

Equality operators follow the arithmetic conversions defined in "Using Operators" on page 161. Where the operands have types and values suitable for the relational operators, the semantics for relational operators applies.

Note: Positive zero and negative zero compare equal. In the following example, the expression always evaluates to TRUE:

```
(-0.00d == +0.00000d)
```

Refer to "Intermediate Results" on page 165 below for details on how to get the convert type during alignment of the decimal point.

Conditional Operators

Conditional operators follow the arithmetic conversions defined in "Using Operators" on page 161. If both the second and third operands have an arithmetic type, the usual arithmetic conversions are performed to bring them to a common type. If both operands are decimal types, the operands are converted to the convert type and the result has that type.

Refer to "Intermediate Results" on page 165 below for details on how to get the convert type during alignment of the decimal point.

Intermediate Results

Use one of the following tables to calculate the size of the result. The tables summarize the intermediate expression results with the four basic arithmetic operators and conditional operators when applied to the decimal types. Both tables assume the following:

- *x* has type decimal(n_1, p_1)
- *y* has type decimal(n_2, p_2)
- `decimal(n,p)` is the result type

Use Table 29 to calculate the size of the result when no overflow is assumed. You can use this table most of the time. If overflow occurs, use Table 30 to determine the resulting type.

Table 29. Intermediate Results (Without Overflow in n or p)

Expression	**(n,p)**
x * y	$n = n_1 + n_2$ $p = p_1 + p_2$
x / y	n = DEC_DIG p = DEC_DIG - (($n_1 - p_1 + p_2$)
x + y	$p = \max(p_1, p_2)$ $n = \max(n_1 - p_1, n_2 - p_2) + p + 1$
x - y	same rule as addition
z ? x : y	$p = \max(p_1, p_2)$ $n = \max(n_1 - p_1, n_2 - p_2) + p$

You can use Table 30 to calculate the size of the result whether there is an overflow or not.

Table 30. Intermediate Results (In the General Form)

Expression	**(*n*, *p*)**
*x * y*	$n = \min(n_1 + n_2,$ DEC_DIG) $p = \min(p_1 + p_2,$ DEC_DIG - $\min((n_1 - p_1) + (n_2 - p_2),$ DEC_DIG))
x / y	n = DEC_DIG $p = \max($DEC_DIG - $((n_1 - p_1) + p_2), 0)$
x + y	$i_r = \min(\max(n_1 - p_1, n_2 - p_2) + 1,$ DEC_DIG) $p = \min(\max(p_1, p_2),$ DEC_DIG - $i_r)$ $n = i_r + p$
x - y	same rule as addition

Table 30. Intermediate Results (In the General Form) (continued)

Expression	(*n*, *p*)
z ? x : y	$i_r = \max(n_1 - p_1, n_2 - p_2)$ $p = \min(\max(p_1, p_2), DEC\_DIG - i_r)$ $n = i_r + p$

If overflow occurs in *n* or *p* the decimal places are truncated and a message is issued. As much of the integral part is reserved as possible. If the integral part is truncated as an expression in the static/extern initialization, an error message is issued. If the integral part is truncated inside the block scope, only a warning is issued. On each operation, the complete result is calculated before truncation occurs.

Assignment Operators

Assignment operators follow the arithmetic conversion defined in "Using Operators" on page 161.

When values are assigned, `SIGFPE` may be raised if the operands contain values that are not valid.

Unary Operators

Use the following unary operators to determine the digits in a decimal data type:

`sizeof` Determines the total number of bytes occupied by the decimal type

`digitsof`
Determines the number of digits (*n*)

`precisionof`
Determines the number of decimal digits (*p*)

sizeof Operator

When you use the `sizeof` operator with `decimal(`*n*`,`*p*`)`, the result is an integer constant. The `sizeof` operator returns the total number of bytes occupied by the decimal type.

Each decimal digit occupies a halfbyte. In addition, a halfbyte is used for the sign. The number of bytes used by `decimal(`*n*`,`*p*`)` is the smallest whole number greater than or equal to (*n* + 1)/2 (that is, `sizeof(decimal(`*n*`,`*p*`))` = ceil((*n* + 1)/2). The `sizeof` result is calculated using this method because the C/VSE compiler uses packed decimals to implement decimal data types.

The following example shows you how to determine the total number of bytes occupied by the decimal type:

```
int y; decimal (5, 2) x;                    /* This would be calculated
to be 3 bytes*/ y = sizeof(x);        /* (5+1)/2 = 3.
      */
```

digitsof Operator

When you use the `digitsof` operator with a decimal type, the result is an integer constant. The `digitsof` operator returns the number of significant digits (*n*) in a decimal type.

This example gives you the number of digits (*n*) in a decimal type:

```
decimal (5, 2) x; int n; n = digitsof(x);
 /* the result is n=5 */
```

Note: `digitsof` can be applied only to a decimal type.

precisionof Operator

When you use the `precisionof` operator with a decimal type, the result is an integer constant. The `precisionof` operator tells you the number of decimal digits (*p*) of the decimal type.

This example gives you the number of decimal digits (*p*) of the decimal type:

```
decimal (5, 2) x; int p; p = precisionof(x); /*
the result is p=2 */
```

Note: `precisionof` can be applied only to a decimal type.

Cast Operator

You can explicitly convert the following types:
- Decimal types to decimal types
- Decimal types to and from floating types
- Decimal types to and from integer types

Note: When you are explicitly casting to a decimal type, the truncation on the leading nonzero digits does not cause an exception at run time. For more information about how compiler messages and run-time exceptions are suppressed, refer to "Converting Decimal Data Types" on page 167.

Summary of Operators Used With Decimal Types

Table 31 summarizes all of the operators that can be used with decimal types.

Table 31. Operators Used With Decimal Types

Operator Name	Associativity	Operators
Primary	left to right	`()`

Unary	right to left	`++ -- +`
		`- ! &`
		(typename) `sizeof`
		`digitsof precisionof`

Multiplicative	left to right	`* /`
Additive	left to right	`+ -`
Relational	left to right	`< > <= >=`
Equality	left to right	`== !=`
Conditional	right to left	`? :`

Assignment	right to left	`= += -=`
		`*= /=`

Comma	left to right	`,`

Converting Decimal Data Types

The C/VSE compiler implicitly converts the following types:
- Decimal types to decimal types
- Decimal types to and from floating types
- Decimal types to and from integer types

Converting Decimal Types to Decimal Types

If the value of the decimal type to be converted is within the range of values that can be represented exactly, the value of the decimal type is not changed.

If the value of the decimal type to be converted is outside the range of values that can be represented, the value of the decimal type is truncated. Truncation may occur on the integral part or the fractional part or both.

When truncation occurs on the fraction part, neither a compile time message nor a run-time exception is generated.

When truncation occurs on the integral part, a compile time message or a run-time exception or both are generated as follows:

- In the initialization of static or external variables
  - Compiler error if nonzero digits are truncated in the integral part
- In the initialization of automatic variables, an assignment or function call with prototype
  - Checkout warning at compile time
  - Run-time exception `SIGFPE` is raised if non-zero digits are truncated in the integral part at run time

Note: An explicit cast can be used to suppress compile time messages and run-time exceptions. A run-time exception is generated if and only if the leading nonzero digits are truncated and the operation is not an explicit cast operation.

Examples

In the following examples, *message* represents a compile message and *exception* represents a run-time exception (that is, `SIGFPE` is raised).

Fractional Part Cannot Be Represented

Conversion of one decimal object to another decimal object with smaller precision involves truncation on the right of the decimal point.

```
#include <decimal.h>

void func(void);
void dec_func(decimal( 7, 1 ));
decimal( 7, 4 ) x = 123.4567D;
decimal( 7, 1 ) y;
decimal( 7, 1 ) z = 123.4567D;  /* z = 000123.4D <-- No message,  */
                                /*                 No exception */
void func(void) {
  decimal( 7, 1 ) a = 123.4567D;  /* a = 000123.4D <-- No message,  *  /
                                  /*                 No exception */
  y = x;         /* y = 000123.4D <-- No message, No exception */
  y = 123.4567D;  /* y = 000123.4D <-- No message, No exception */
  dec_func(x);                  /* <-- No message, No exception */
}
```

Figure 30. Fractional Part Cannot Be Represented

Integral Part Cannot Be Represented

Conversion of one decimal object to another decimal object with fewer digits involves truncation on the left of the decimal point.

```
void func(void);
void dec_func(decimal( 5, 2 ));
decimal( 8, 2 ) w = 000456.78D;
decimal( 8, 2 ) x = 123456.78D;
decimal( 5, 2 ) y;
decimal( 5, 2 ) z = 123456.78D;   /* <-- Compile error            */
decimal( 5, 2 ) z1 = (decimal( 5, 2 )) 123456.78D;
                                  /* z1 = 456.78D <-- No message,  */
                                  /*                No exception */

void func(void) {
  decimal( 5, 2 ) a = 123456.78D;   /*  <-- Checkout warning        */
                                    /*                and exception */
  decimal( 5, 2 ) a1 = (decimal( 5, 2 )) 123456.78D;
                                    /* a1 = 456.78D <-- No message,  */
                                    /*                 No exception */
  y = w;          /* y = 456.78D <-- Checkout warning, No exception */
  y = x;                       /* <-- Checkout warning and exception */
  y = 123456.78D;              /* <-- Checkout warning and exception */
  dec_func(x);                 /* <-- Checkout warning and exception */

  y = (decimal( 5, 2 )) w;
                         /* y = 456.78D <-- No message, No exception */
  y = (decimal( 5, 2 )) x;
                         /* y = 456.78D <-- No message, No exception */
  y = (decimal( 5, 2 )) 123456.78D;
                         /* y = 456.78D <-- No message, No exception */
  dec_func((decimal( 5, 2 )) x);
                                     /* <-- No message, No exception */
}
```

Figure 31. Integral Part Cannot Be Represented

Converting Decimal Types to and from Integer Types

Conversion from Integer Types

When a value of integer type is implicitly converted to decimal type, the integer type is converted to type decimal(10,0).

When a value of integer type is explicitly converted to decimal type, the conversion proceeds as though these two steps are followed:

1. The integer type is converted to type decimal(10,0). A run-time exception can never occur in this step.
2. Type decimal(10,0) is then converted to decimal(n,p). All rules for decimal type to decimal type conversion apply in this step.

An unsigned integer type is converted to a positive decimal value.

If the value of the integral part cannot be represented by the decimal type, the result of the conversion is undefined and SIGFPE is raised.

Conversion to Integer Types

When a value of decimal type is converted to integer type, the fractional part is discarded. If the value of the integral part cannot be represented by the integer type, the result of the conversion is undefined. An exception does not occur and execution continues.

When a negative decimal type is converted to an unsigned integer type, the conversion proceeds as though these steps are followed:

1. The decimal type is converted to a signed integer type with the same size as the unsigned integer type.
2. The signed integer type is converted to the unsigned integer type.

Examples of Conversion from Integer Type

```
#include <decimal.h>

decimal(10,2) pd01 = 1234;    /* pd01 = 00001234.00d */
decimal(5,0) pd02 = 987654;   /* compile error */
int main(void) {
  decimal(5,0) pd03 = 987654;   /* run-time exception */
  decimal(13,4) pd04;

  /* The number 321 is converted to decimal(10,0) before the */
  /* addition is performed.                                  */
  pd04 = 1234.56d + 321;           /* pd04 = 000001555.5600d */

}
```

Figure 32. Conversion from Integral Type

Examples of Conversion to Integer Type

```
int i = 1234.5678d;           /* i = 1234 */
int j = -789d;                /* j = -789 */
int k = 9876543210d;          /* k is undefined */
```

Figure 33. Conversion to Integer Type

Converting Decimal Types to and from Floating Types

Conversion from Floating Types

When a value of floating type is converted to decimal type and the value being converted cannot be represented by the decimal type, the result is truncated. If the value of the floating type to be converted is within the range of values that can be represented, but cannot be represented exactly, the result is also truncated. The result retains as much value as possible. When the leading non-zero digits are truncated and the operation is not an explicit cast operation, a decimal overflow exception occurs at run time and SIGFPE is raised.

When a conversion from a floating type is made with static or external variable initialization, a compile error message is issued.

The result of the conversion may not be exact because the internal representation of System/370 floating-point instructions is hexadecimal based. The mapping between the two representations is not one-to-one, even when the value of a float type is within the range of the decimal type.

Conversion to Floating Types

The result of the conversion might not be exact because of:
- The limitations of significant digits in different floating types
- The degree to which a value can be stored exactly in a floating type
- The loss of precision during conversion
- The internal hexadecimal representation of the System/370 floating point instructions.

Examples of Conversion from Floating Type

```
#include <decimal.h>

decimal(10,2) pd11 = 1234.0;  /* pd11 = 00001234.00d */
decimal(5,0) pd12 = 987654.0; /* compile error */
int main(void) {
  decimal(5,0) pd13 = 987654.0;  /* run-time exception */
  decimal(13,4) pd14 = 12.34567890;  /* fractional part is truncated */
}
```

Figure 34. Conversion from Floating Type

Examples of Conversion to Floating Type

```
/* The content of each floating type variables depends on    */
/* their limitation of significant digits that are specified */
/* in <float.h>.                                             */
float       a = 12345678901234567890.1234567890d;
double      b = 12345678901234567890.1234567890d;
long double c = 12345678901234567890.1234567890d;
```

Figure 35. Conversion to Floating Type

Calling Functions

There are no default argument promotions on arguments that have type decimal when the called function does not include a prototype. If the expression for the called function has a type that includes a prototype, the behavior is as documented in ANSI with the exception of prototype with an ellipsis (...). If the prototype ends with an ellipsis (...), default argument promotions are not performed on arguments with decimal types.

A function may change the values of its parameters, but these changes cannot affect the values of the arguments. However, it is possible to pass a pointer to a decimal object, and the function may change the value of the decimal object pointed to.

Using Library Functions

You can use variable arguments and input/output operations with decimals.

Using Variable Arguments with Decimal Data Types

You can use the `va_arg` macro with a decimal data type `decimal(`*n*`,`*p*`)`.

```
var_type va_arg( va_list arg_ptr, var_type );
```

Each invocation of `va_arg` modifies *arg_ptr* so that the values of successive arguments are returned in turn.

Formatting Input and Output Operations

Use the following functions to print the value of a decimal type:
- `fprintf()`
- `printf()`
- `sprintf()`
- `vfprintf()`
- `vprintf()`
- `vsprintf()`

Use the following functions to read the value of a decimal type:
- `fscanf()`
- `scanf()`
- `sscanf()`

Using `fprintf()`

The formatting behavior of the `printf()`, `sprintf()`, `vfprintf()`, `vprintf()`, and `vsprintf()` functions is the same as that of the `fprintf()` function.

The following is added to the optional precision's description.
- An optional **precision** that gives the minimum digits to appear for the **d, i, o, u, x,** and **X** conversions
- The number of digits to appear after the decimal-point character for **e, E, f,** and **D(n,p)** conversions
- The maximum number of significant digits for the **g** and **G** conversions
- The maximum number of characters to be written from a string in an **s** conversion

The precision takes the form of a period (.) followed either by an asterisk (*) or by an optional decimal integer. If only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

The following are added to the list of flag characters and their meanings:

The result is to be converted to an alternative form as follows:

- For **o** conversion, the precision is increased to force the first digit of the result to be a zero.
- For **x** or (**X**) conversion, a nonzero result will have 0x (or 0X) prefixed to it.
- For **e, E, f, g, G, and D(n,p)** conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally a decimal-point character appears in the result of these conversions only if a digit follows it.)
- For **g** and **G** conversions, trailing zeros will *not* be removed from the result.
- For other conversions, the behavior is undefined.

0 The conversions are as follows:

- For **d, i, o, u, x, X, e, E, f, f, G** and **D(n,p)** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **-** flags both appear, the **0** flag will be ignored.
- For **d, i, o, u, x** and **X** conversions, if a precision is specified, the **0** flag will be ignored.
- For other conversions, the behavior is undefined.

The following are added to the list of conversion specifiers:

D(n,p) The decimal argument is converted in the style ***[-] ddd.ddd***, where the number of digits after the decimal-point character is equal to the precision of specification. If the precision is missing, it is taken as the *p* value. If the precision is zero and # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is truncated to the appropriate number of digits according to the precision.

The (*n*,*p*) descriptor is used to describe the characteristic of the decimal argument. Both *n* and *p* have to be in the form of decimal integers. If *p* is missing, a default value of zero is assumed. Blank spaces are allowed in the conversion specifier, for example

```
%D(   10   ,     2    )
```

The number of digits *n* and the number of precisions *p* can be an asterisk (*), in which case an argument from the argument list supplies the value. The *n* and *p* argument must precede the value being formatted, and follow the width and precision, if any, in the argument list, for example

```
"%*.*D(*,*)",width,precision,n,p,pdec
```

If the specifier is in a form not stated above, the behavior is undefined.

Using `fscanf()`

The formatting behavior of the `scanf()` and `sscanf()` functions is the same as that of the `fscanf()` function.

The following is added to the list of conversion specifiers:

D(n,p) Matches a decimal number. The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal point character, but no decimal suffix.

The subject sequence is defined as the longest initial sub-sequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

Validating Values

It is possible to have invalid representation of decimal value stored in memory, such as input from file or overlay memory. If the invalid decimal value is used in an operation or assignment, the result may not be as expected. A built-in function can be used to report whether the decimal representation is valid or not. The function call can be in the following form:

```
status = decchk ( x );
```

The built-in function `decchk()` accepts a decimal types expression as argument and returns a status value of type `int`.

The status can be interpreted as follows:

0 Valid decimal representation value (including nonpreferred but valid sign, A-F)

1 Leftmost halfbyte is not zero in a decimal type number that has an even number of digits (for example, 123 is stored in decimal(2,0))

2 Incorrect digits (not 0-9)

4 Incorrect sign (not A-F)

Macro define name for function return status (in decimal.h):

```
    #define  DEC_VALUE_OK      0
    #define  DEC_BAD_NIBBLE    1
    #define  DEC_BAD_DIGIT     2
    #define  DEC_BAD_SIGN      4
```

The function return status is masked to return multiple status.

Fix Sign

A built-in function can be used to fix nonpreferred sign variables. The function call can be in the following form:

```
x = decfix ( x );
```

The built-in function `decfix()` accepts a decimal types expression as argument and returns a decimal value that has the same size (that is, same decimal types) and same value as the argument with the correct preferred sign. The function does not change the content of the argument.

Decimal Absolute

The function call can be in the following form:

```
y = decabs ( x );
```

The built-in function `decabs()` accepts a decimal types expression as argument and returns the absolute value of the decimal argument (i.e., same decimal types as the argument). The function does not change the content of the argument.

See *LE/VSE C Run-Time Library Reference* for more information on the `decabs()`, `decchk()`, and `decfix()` library functions.

Programming Examples

Example One

EDCXGDC3

```
 /* EDCXGDC3
    This example demonstrates the use of the decimal data type always
    include decimal.h when decimal data type is used
  */

#include <decimal.h>

 /* Declares a decimal(10,2) variable */
decimal(10,2) pd01;

 /* Declares a decimal(15,4) variable and initializes it with the */
 /* value 1234.56d                                                */
decimal(15,4) pd02 = 1234.56d;

 /* Structure that has decimal related members */
struct pdec
  {                             /* members' data types        */
  int m;                        /* - integer                  */
  decimal(23,10) pd03;          /* - decimal(23,10)           */
  decimal(10,2) pd04[3];        /* - array of decimal(10,2)   */
  decimal(10,2) *pd05;          /* - pointer to decimal(10,2) */
  } pd06,
   *pd07 = &pd06;              /* pd07 points to pd06        */

 /* Array of decimal(31,30) */
decimal(31,30) pd08[2];

 /* Prototype for function that accepts decimal(10,2) and int as */
 /* arguments and has return type decimal(25,5)                  */
decimal(25,5) product(decimal(10,2), int);

decimal(5,2) PdCnt;             /* decimal loop counter */
int i;
```

Figure 36. Decimal Data Type Example 1 (Part 1 of 3)

```
int main(void)
{
  pd01 = -789.45d;              /* simple assignment */
  pd06.m = digitsof(pd06.pd03) + precisionof(pd02);  /* 23 + 4 */
  pd06.pd03 = sizeof(pd01);
  pd06.pd04[0] = pd02 + pd01;   /* decimal addition */
  *(pd06.pd04 + 1) = (decimal(10,2)) product(pd07->pd04[0], pd07->m);
  pd07->pd04[2] = product(pd07->pd04[0], pd07->pd04[1]);
  pd07->pd05 = &pd01;           /* taking the address of a */
                                /*   decimal variable      */
  /* These two statements are different */
  pd08[0] = 1 / 3d;
  pd08[1] = 1d / 3d;

  printf("pd01 = %D(10,2)\n", pd01);
  printf("pd02 = %*.*D(*,*)\n",
          20, 5, digitsof(pd02), precisionof(pd02), pd02);
  printf("pd06.m = %d, pd07->m = %d\n", pd06.m, pd07->m);
  printf("pd06.pd03 = %D(23,10), pd07->pd03 = %D(23,10)\n",
          pd06.pd03, pd07->pd03);

 /* You will get an infinite loop if floating type is  */
  /* used instead of the decimal data types.           */
  for (PdCnt = 0.0d; PdCnt != 3.6d; PdCnt += 1.2d)
  {
    i = PdCnt / 1.2d;
    printf("pd06.pd04[%d] = %D(10,2), \
            pd07->pd04[%d] = %D(10,2)\n",
            i, pd06.pd04[i], i, pd07->pd04[i]);
  }

  printf("*(pd06.pd05) = %D(10,2), *(pd07->pd05) = %D(10,2)\n",
          *(pd06.pd05), *(pd07->pd05));

  printf("pd08[0] = %D(31,30)\n", pd08[0]);
  printf("pd08[1] = %D(31,30)\n", pd08[1]);

  return(0);
}
```

Figure 36. Decimal Data Type Example 1 (Part 2 of 3)

```
 /* Function definition for product() */
decimal(25,5) product(decimal(10,2) v1, int v2)
{

  /* The following happens in the return statement */
  /* - v2 is converted to decimal(10,0)                        */
  /* - after the multiplication, the expression has resulting    */
  /*   type decimal(20,2) (i.e. (10,2) * (10,0) ==> (20,2))      */
  /* - the result is then converted implicitly to decimal(25,5)  */
  /*   before it is returned                                     */
  return( v1 * v2 );

}
```

Figure 36. Decimal Data Type Example 1 (Part 3 of 3)

Output from Programming Example One

```
pd01 = -789.45
pd02 =          1234.56000
pd06.m = 27, pd07->m = 27
pd06.pd03 = 6.0000000000, pd07->pd03 = 6.0000000000
pd06.pd04[0] = 445.11,            pd07->pd04[0] = 445.11
pd06.pd04[1] = 12017.97,           pd07->pd04[1] = 12017.97
pd06.pd04[2] = 5348886.87,          pd07->pd04[2] = 5348886.87
*(pd06.pd05) = -789.45, *(pd07->pd05) = -789.45
pd08[0] = 0.3333333333333333333333000000000
pd08[1] = 0.3333333333333333333333333333333
```

Example Two

EDCXGDC4

```
 /* EDCXGDC4
    This example demonstrates the use of the decimal data type
  */

#include <decimal.h>

decimal(31,4) pd01 = 1234.5678d;
decimal(29,4) pd02 = 1234.5678d;

int main(void)
{
  /* The results are different in the next two statements */
  pd01 = pd01 + 1d;
  pd02 = pd02 + 1d;

  printf("pd01 = %D(31,4)\n", pd01);
  printf("pd02 = %D(29,4)\n", pd02);

  /* Warning: The decimal variable with size 31 should not be      */
  /*          used in arithmetic operation.                        */
  /*          In the above example: (31,4) + (1,0) ==> (31,3)      */
  /*                                (29,4) + (1,0) ==> (30,4)      */

  return(0);
}
```

Figure 37. Decimal Data Type Example 2

Note: See “Intermediate Results” on page 165 to understand the output from this example and why decimal variables with size 31 should be used with caution in arithmetic operations.

Output from Programming Example Two

```
pd01 = 1235.5670
pd02 = 1235.5678
```

Decimal Exception Handling

LE/VSE C Run-Time decimal instructions produce the following exceptions that are unique to decimal operations:

- Data exception (interrupt code hex ’7’)

  This may be caused by invalid sign or digit codes in a packed decimal number operated on by packed decimal instructions, for example, ADD DECIMAL or COMPARE DECIMAL.

  When an operation is performed on decimal operands and the assignment is not through an explicit cast operation, the following situations cause run-time exceptions at execution time and `SIGFPE` is raised.
- Decimal-overflow exception (interrupt code hex ’A’)

  This exception may be caused when nonzero digits are lost because the destination field in a decimal operation is too short to contain the result.

  Note: The unhandled decimal overflow message

  ```
  CEE3210S The system detected a Decimal-overflow exception.
  ```

  is the same for both decimal overflow and fixed overflow conditions. Fixed overflow conditions should not occur because the fixed overflow condition is normally disabled (masked) and is ignored by the run-time.
- Decimal-divide exception (interrupt code hex ’B’)

  This exception may be caused when, in decimal division, the divisor is zero, or the quotient exceeds the specified data-field size. The decimal divide is indicated only if the sign codes of both the divisor and dividend are valid and only if the digit or digits used in establishing the exception are valid.

  Note: The unhandled divide message

  ```
  CEE3211S The system detected a Decimal-divide exception.
  ```

  does not distinguish between a decimal divide condition and a fixed divide by zero condition. Both are mapped into the same error message.
- A decimal data exception may be produced by the `printf()` family when processing an invalid decimal operand. This may result in abnormal termination of your program with the run-time message:

  Under VSE

  ```
  CEE3207S The system detected a Data exception
  ```

  Under CICS

  ```
  EDCK007 ABEND=8097 Data Exception
  ```

Other exceptions indicated by the decimal instruction set are not unique.

Restrictions

- **`printf()` and `scanf()`**

  You must ensure that valid packed decimal data is present when attempting to use it with run-time library decimal routines. No additional validation is performed on decimal data to ensure format correctness. Use the `decchk()` routine to validate decimal data operands in such circumstances.
- **Additional Considerations**

- When the operands of a decimal operation contain invalid digits, the result is undefined, and a run-time exception can occur. To validate a decimal number, call the `decchk()` built-in function in your code.
- Code should be written in a manner that does not depend on the ability of the run-time to recover from a decimal overflow exception.
- In a multiprocessor configuration, decimal operations cannot be used safely to update a shared storage location when the possibility exists that another processor may also be updating that location. This possibility arises because the bytes of a decimal operand are not necessarily accessed concurrently.
- If a decimal exception occurs in user code or library routines, the expected results of the instruction causing the exception or the library routine where the exception occurred are undefined. The results produced by the library routine's execution are also undefined.
- If a `SIGFPE` handler is coded to handle decimal exceptions, it should re-enable itself prior to resuming normal execution or recovery from the error. This reestablishes the exception environment consistent with good programming practice.

Decimal Exceptions and Interlanguage Calls

Support for C enabled with decimal exception has not changed for PL/I and COBOL.

Assembler Interlanguage Calls

Calls to an assembler language procedure or function assume that the called routine will save and restore the value of the Program Mask if the routine alters it. Ensure that the Program Mask is preserved across an assembler language interface. If it is not preserved, the recognition of subsequent decimal overflow exceptions in C code will be unpredictable.

Chapter 18. Handling Error Conditions and Signals

This chapter discusses how to handle error conditions and signals with LE/VSE C Run-Time. It describes how to establish, enable and raise a signal, and provides a list of signals supported by LE/VSE C Run-Time.

The LE/VSE environment uses a stack-based model to handle error conditions. This environment establishes a last-in, first-out (LIFO) queue of 0 or more user condition handlers for each stack frame. The LE/VSE condition handler calls the user condition handler at each stack frame to handle error conditions when they are detected. For more information about the callable services in LE/VSE, refer to "Handling Signals using LE/VSE Callable Services" on page 181.

Handling Signals Using `signal()` and `raise()`

The LE/VSE C Run-Time environment provides two functions that alter the signal handling capabilities available in the run-time environment. These are the `signal()` function and the `raise()` function. The `signal()` function registers a condition handler and the `raise()` function raises the condition.

You can use the `signal()` function to perform one of the following actions:

- Ignore the condition. For example, use the `SIG_IGN` condition to specify `signal(SIGFPE,SIG_IGN)`.
- Reset the Global Error Table for default handling. For example, use the `SIG_DFL` condition to specify `signal(SIGSEGV,SIG_DFL)`. See "C Condition Handling Semantics" in *LE/VSE Programming Guide* for information regarding the C Global Error Table.
- Register a function to handle the specific condition. For example, pass a pointer to a function for the specific condition with `signal(SIGILL,cfunc1)`. The function registered for `signal()` must be declared with C linkage.

Handling Signals using LE/VSE Callable Services

You can set up user signal handlers with the LE/VSE condition handling services. Some of the LE/VSE callable services available for condition handling are:

CEEHDLR	Register a user-written condition handler.
CEEHDLU	Remove a registered user-written condition handler.
CEESGL	Raise a LE/VSE condition.

In addition, with LE/VSE, when an exception occurs after an interlanguage call, the exception may be handled where it occurs or percolated to its caller (written in any LE/VSE-conforming language) or promoted. For more information on how to handle exceptions under the LE/VSE condition handling model, refer to *LE/VSE Programming Guide*.

Specific considerations for C under LE/VSE:

1. The `TRAP` run-time option determines how the LE/VSE condition manager is to act upon error conditions and program interrupts. If the `TRAP(OFF)` run-time option is in effect, conditions detected by the operating system, often due to machine interrupts, will not be handled by the LE/VSE environment and thus cannot be handled by a LE/VSE C Run-Time program.

Note: `TRAP(OFF)` only blocks the handling of hardware (program checks) and operating system (abend) conditions. It does not block software conditions such those that are associated with a raise or `CEESGL`. Any conditions that are blocked because of `TRAP(OFF)` are not presented to any handlers (whether registered by a signal or by `CEEHDLR`). In particular, even for `TRAP(OFF)`, conditions that are initiated by a signal or by `CEESGL` are presented to handlers registered by either `signal()` or `CEEHDLR`.

The use of the `TRAP(OFF)` option is not recommended; refer to *LE/VSE Programming Reference* for more information.

2. You can use the `ERRCOUNT` run-time option to specify how many errors are to be tolerated during the execution of your program before an abend occurs. The counter is incremented by one for every severity 2, 3, or 4 condition that occurs. Both hardware-generated and software-generated signals increment the counter.

Note: The LE/VSE C Run-Time registered condition handlers (those registered by `signal()` and `raise()`), are activated after the LE/VSE registered condition handlers for the current stack frame are activated. This means that if there are condition handlers for both LE/VSE C Run-Time and LE/VSE, the LE/VSE handlers are activated first.

LE/VSE C Run-Time Signal Handling Features

The terms used to describe implementation features and concepts are:

Establishing a Signal

A signal, `sig_num`, becomes established when `signal(sig_num, sig_handler)` is executed. (Two values of `sig_handler` are reserved: `SIG_IGN` and `SIG_DFL`. They are pointers to library-supplied functions that establish the action taken.) `sig_handler` is a pointer to a function to be called when the signal is raised. This function is also known as a *signal handler*. The function must be written in C with the default linkage in effect. That is, `sig_handler` cannot have `OS`, `PLI`, or `COBOL` linkage. The signal ceases to be established when:

- The signal is explicitly reset to the system default by using `signal(sig_num, SIG_DFL)`.
- The signal is explicitly reset by using `signal(sig_num, SIG_IGN)`.
- The signal is implicitly reset to the system default when the signal is raised. When `sig_handler` is called, signal handling is reset to the default as if an implicit `signal(sig_num, SIG_DFL)` had been executed. Depending on the purpose of the signal handler, you may want to reestablish the signal from within the signal handler.
- A phase is deleted using the `release()` function and a signal handler for the signal resides in the phase. In this case, default handling will be reset for all the affected signals.

Enabling a Signal

A signal is enabled when the occurrence of the condition will result in either the execution of an established signal handler or the default system response. The signal is disabled when the occurrence is to be ignored. This can be done by making the call `signal(sig_num, SIG_IGN)`.

Interrupting a Program

Program interrupts or errors detected by the hardware and identified to the program by operating system mechanisms are known as hardware signals. For example, the hardware can detect a divide by zero and this result can be raised to the program.

Raising a Signal

Signals that are explicitly raised by the user (by using the `raise()` function) are known as software signals.

Identifying Hardware and Software Signals

The following is a list of the LE/VSE C Run-Time supported signals:

SIGABND	System abend.
SIGABRT	Abnormal termination (software only).
SIGFPE	Erroneous arithmetic operation (hardware and software).
SIGILL	Invalid object module (hardware and software).
SIGINT	Interactive attention interrupt by `raise()` (software only).
SIGIOERR	Serious software error such as a system read or write. You can assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. This minimizes the time required to locate the source of a serious error.
SIGSEGV	Invalid access to memory (hardware and software).
SIGTERM	Termination request sent to program (software only).
SIGUSR1	Reserved for user (software only).
SIGUSR2	Reserved for user (software only).

The applicable hardware signals or exceptions that are supported are listed in Table 32. It also lists those hardware exceptions that are not supported (for example, fixed-point overflow) and are masked.

The applicable software signals or exceptions that are supported are listed in Table 33 on page 184.

Table 32. Hardware Exceptions - Default Run-Time Messages and System Actions

C Signal	Hardware Exception	Default Run-Time Message with LE/VSE	Default System Action with LE/VSE Library
SIGILL	Operation exception	CEE3201	Abnormal termination VSE rc=3000
	Privileged operation exception	CEE3202	
	Execute exception	CEE3203	
SIGSEGV	Protection exception	CEE3204	Abnormal termination VSE rc=3000
	Addressing exception	CEE3205	
	Specification exception	CEE3206	

Table 32. Hardware Exceptions - Default Run-Time Messages and System Actions (continued)

C Signal	Hardware Exception	Default Run-Time Message with LE/VSE	Default System Action with LE/VSE Library
SIGFPE	Data exception	CEE3207	Abnormal termination VSE rc=3000
	Fixed-point divide	CEE3209	
	Decimal overflow	CEE3210	
	Decimal divide	CEE3211	
	Exponent overflow	CEE3212	
	Floating point divide	CEE3215	

Note: The default run-time program mask is enabled for decimal overflow exceptions.

Table 33 shows software signals or exceptions, their origin, default run-time messages and default system actions.

Table 33. Software Exceptions - Default Run-Time Messages and System Actions

C Signal	Software Exception	Default Run-Time Message with LE/VSE	Default System Action with LE/VSE Library
SIGILL	raise(SIGILL)	EDC6001	Abnormal Termination VSE rc=3000
SIGSEGV	raise(SIGSEGV)	EDC6002	Abnormal Termination VSE rc=3000
SIGFPE	raise(SIGFPE)	EDC6002	Abnormal Termination VSE rc=3000
SIGABND	raise(SIGABND)	EDC6003	Abnormal Termination VSE rc=3000
SIGTERM	raise(SIGTERM)	EDC6004	Abnormal Termination VSE rc=3000
SIGINT	raise(SIGINT)	EDC6005	Abnormal Termination VSE rc=3000
SIGABRT	raise(SIGABRT)	EDC6006	Abnormal Termination VSE rc=2000
SIGUSR1	raise(SIGUSR1)	EDC6007	Abnormal Termination VSE rc=3000
SIGUSR2	raise(SIGUSR2)	EDC6008	Abnormal Termination VSE rc=3000
SIGIOERR	raise(SIGIOERR)	EDC6009	Signal is ignored.

SIGABND Considerations

When the `SIGABND` signal is registered with an address of a LE/VSE C Run-Time specific handler using the `signal()` function, control cannot resume at the instruction following the abend or the invocation of `raise()` with `SIGABND`. If the C signal handler is returned, the abend is percolated and the default behavior occurs. The `longjmp()` or `exit()` function can be invoked from the handler to control the behavior.

If SIG_IGN is the specified action for SIGABND and an abend occurs (or SIGABND was raised), the abend will not be ignored because a resume cannot occur. The abend will percolate and the default action will occur.

Two macros are available in signal.h header file that provide information about an abend. The __abendcode() macro returns the abend that occurred and __rsncode() returns the corresponding reason code for the abend. These values are available in a C signal handler that has been registered with the SIGABND signal. If you are looking for the abend and reason codes, using these macros, they should only be checked when in a signal handler. The values returned by the __abendcode() and __rsncode() macros are undefined if the macros are used outside a registered signal handler.

SIGIOERR Considerations

When the SIGIOERR signal is raised, codes for the last operation will be set in the __amrc structure to aid you in error diagnosis. See "Using the __amrc Structure" on page 131 for more information.

Default Handling of Signals

The run-time environment will perform default handling of a given signal unless the signal is established (signal(sig_num, sig_handler)) or the signal is disabled (signal(sig_num, SIG_IGN)). A user can also set or reset default handling by coding:

```
signal(sig_num, SIG_DFL);
```

The default handling depends upon the signal that was raised. Refer to the two preceding tables for information on the default handling of a given signal.

Note: When using the atexit() library function, the atexit list will not be run if the application is abnormally terminated.

MAP 0010: Summary of LE/VSE Error Handling

001

Signal is raised. Is `SIG_IGN` set for the signal? Or is the signal blocked?
Yes No

- **002**
  Continue at Step 006.

003

Is the signal for a SIGABND?
Yes No

- **004**
  Resume at the next instruction.

005
Condition is percolated for default behavior.

006

Was the signal previously blocked?
Yes No

- **007**

  Is a LE/VSE user handler registered?
  Yes No

  - **008**

    Is a C handler established for the signal by `signal()`?
    Yes No

    - **009**
      Continue at Step 015 on page 187.

    010
    Run C handler and resume at the next instruction.

  011
  Run LE/VSE user handler. The handler can resume, percolate or promote the signal. See *LE/VSE Programming Guide* for more details.

$\boxed{\textbf{012}}$

Is a C handler established for the signal?

Yes No

$\boxed{\textbf{013}}$

Perform default processing.

$\boxed{\textbf{014}}$

Run C handler and transfer control to the next instruction following interrupt.

$\boxed{\textbf{015}}$

At stack frame 0?

Yes No

$\boxed{\textbf{016}}$

Default handling for the signal and percolate to next stack frame.

$\boxed{\textbf{017}}$

Perform default processing.

Example of C Signal Handling Under LE/VSE C Run-Time

In the following example, the call to `signal()` in `main()` establishes the function `handler` to process the interrupt signal when it occurs. An error value returned from this call to `signal()` causes the program to end with a printed error message.

EDCXGEC1

```
 /* EDCXGEC1
    This example demonstrates signal handling
  */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void handler(int);
```

Figure 38. Example Illustrating Signal Handling (Part 1 of 2)

```
int main(void) {
   if (signal(SIGINT,handler) == SIG_ERR) {
      perror("Could not set SIGINT");
      abort();
   }
 /* add code here if desired */
   raise(SIGINT);
 /* add code here if desired */
   return(0);
}

void handler(int sig_num) {
   signal(SIGINT, handler);
   printf("Signal handler entered\n");
   exit(0);
}
```

Figure 38. Example Illustrating Signal Handling (Part 2 of 2)

Chapter 19. Optimizing Code

This chapter briefly describes the optimization methods used by the C/VSE compiler and discusses some programming practices that can further improve the execution performance of your code.

For optimization, the compiler changes the unoptimized code sequences, derived from the source code, into equivalent code sequences that execute faster and usually require less memory space. It is possible for an expression that would normally cause an exception to be removed by optimization, thus preventing the exception.

Note: The C/VSE compiler provides two optimization levels. To generate unoptimized code, specify the `NOOPTIMIZE` option. To generate optimized code, specify `OPTIMIZE`. Some releases of C/370 supported three levels, `OPTIMIZE(0)`, `OPTIMIZE(1)`, and `OPTIMIZE(2)`. If you run a program which has been compiled with one of these, LE/VSE C Run-Time maps them to their corresponding values. `OPTIMIZE(0)` maps to `NOOPTIMIZE`; `OPTIMIZE(1)` and `OPTIMIZE(2)` map to `OPTIMIZE`.

Because the optimization is achieved by transforming the code using knowledge obtained from a larger program context, the direct correspondence between source and object code is often lost. Therefore, debugging information for programs compiled with the optimization option is limited to setting break points at function entry points only. Optimized code is also more sensitive to subtle coding errors. For these reasons, optimization should not be used while a program is under development. Only the final version of a program should be compiled with optimization.

Using Optimization Facilities

The compiler performs the following optimizations:

Inlining

Inlining replaces certain function calls with the actual code of the function and is performed before all other optimizations. Inlining not only eliminates the linkage overhead but also exposes the entire function to the caller and thus allows the compiler to better optimize your code.

Inlining is performed when the compiler option `INLINE` is specified. Any `OPTIMIZE` level including `NOOPT` can be used.

Consider the following program:

EDCXGOP1

```
 /* EDCXGOP1
    This example demonstrates optimization
  */

#include <stdio.h>

int which_group(int);
#pragma inline(which_group)

int main (void) {

   int j;

   j = which_group (7);

   return(0);
}

int which_group (int a) {
   if (a < 0) {
      printf("first group\n");
      return(99);
   }
   else if (a == 0) {
      printf("second group\n");
      return(88);
   }
   else {
      printf("third group\n");
      return(77);
   }
}
```

Figure 39. Optimization Example 1

In this example, if you specify the `#pragma inline` directive for the function `which_group()`, and compile with the `OPTIMIZE` and `INLINE` options, after optimizations, the compiler determines that the above code is equivalent to:

EDCXGOP2

```
 /* EDCXGOP2
    This example demonstrates optimization
  */

#include <stdio.h>

int main(void) {

   int j;

   printf("third group\n");   /* a lot less code generation */
   j = 77;

   return(0);
}
```

Figure 40. Optimization Example 2

Value Numbering
: Value numbering involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

Straightening
: Straightening is rearranging the program code to minimize branching logic and to combine physically separate blocks of code.

Common Expression Elimination
: Common expressions recalculate the same value in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done even for intermediate expressions within expressions. For example, if your program contains the following statements:

```
a = c + d;
      .
      .
      .
f = c + d + e;
```

the common expression `c + d` is saved from its first evaluation and is used in the subsequent statement to determine the value of `f`.

Code Motion
: If variables used in a computation within a loop are not altered within the loop, it may be possible to perform the calculation outside of the loop and use the results within the loop.

Strength Reduction
: Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Constant Propagation
: Constants used in an expression are combined and new ones generated. Some mode conversions are done, and compile time evaluation of some intrinsic functions takes place.

Instruction Scheduling
: Instructions are reordered to minimize execution time.

Dead Store Elimination
: The compiler eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary, and is therefore removed.

Dead Code Elimination
: The compiler may eliminate code for calculations that are not required. Other optimization techniques may cause code to become dead.

Under `OPT`, these optimization techniques are performed locally and can be achieved with minimal increases in storage and compilation time requirements over `NOOPT`.

Programming Recommendations

The following section contains tips on how to write code to best use the optimization techniques used by the compiler.

Specifying Inline Functions

To get maximum performance improvements, specify the `INLINE` compile-time option with `OPTIMIZE`. Use the `REPORT` suboption when tuning your code.

If you inline too large a function, your run-time performance may degrade.

Two types of calls are not inlined:
- The number of parameters on the call does not match the function definition. An example of this is a `var arg` function call.
- The call is directly recursive; the routine calls itself.

Selective Mode

If you know exactly which functions are frequently invoked from within a compile unit, you can simply add the appropriate `#pragma inline` directives in your source and compile with `INLINE (NOAUTO,REPORT,,)`.

If your code contains complex macros, the macros can now be made into static routines at no execution-time cost. All static routines that are interfaces to a data object can be placed into a header file.

Note: You can use Debug Tool for VSE/ESA to get this information or use available execution time analyzers.

For best run-time performance, the inliner should be used in selective mode (`INLINE(NOAUTO,,,)`) to fine-tune your final application rather than rely on the inliner in automatic mode (`INLINE(AUTO,,,)`).

Automatic Mode

To provide assistance in choosing which routines to inline, you can compile with `INLINE(AUTO,REPORT,,)`. Specifying larger values for *threshold* and *limit* will inline more functions and thus increase the size of functions containing inlined functions. The *threshold* and *limit* parameters are defined as follows:

threshold
Maximum relative size of a function to inline. The default value 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 the compiler. Specifying a threshold of `0` is equivalent to specifying `NOAUTO`.

limit Maximum relative size a function can grow before auto-inlining stops. The default is 1000 ACUs for the specific function. Specifying a limit of `0` is equivalent to specifying `NOAUTO`.

Note: When functions become too large, run-time performance can degrade.

Inlining a function that is rarely invoked can degrade performance. Use the `#pragma noinline` directive to instruct the automatic inliner not to inline these types of functions.

Once you are satisfied with the selection of inlined routines, you should add the appropriate `#pragma inline` directives to the source. That is, once the selected routines are forced with these directives, you can then compile the program in selective mode. This way, you do not need to be affected by changes made to the heuristics used in the auto inliner.

Automatic mode is provided to assist you in starting to optimize your code. It is not recommended for final compilation of production level code.

Using Variables

Use local variables, preferably automatic variables, as much as possible. The compiler can accurately analyze their use, while it has to make several worst case assumptions about global variables. These assumptions tend to hinder optimizations. For example, if you code a function that uses external variables heavily, and also calls several external functions, the compiler has to assume that every call to an external function could change the value of every external variable. If you know that none of the function calls will affect the global variables that you are using, and you have to read them frequently with function calls interspersed, copying the global variables to local variables and then using these local variables will help the compiler to perform optimizations that otherwise would not be done.

If you want to share variables between functions within the same compilation unit, use static variables instead of external variables.

Choose static variables rather than external variables wherever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about the variables.

To access an external variable, the compiler has to make an extra memory access to obtain the variable's address. The compiler removes extraneous address loads, but this means that the compiler has to use a register to keep the address. Using many external variables simultaneously takes up many registers, thereby causing spilling of registers to storage. Group external data into structures (all elements of an external structure use the same base address) or arrays wherever it makes sense to do so.

The compiler treats register variables the same way it treats automatic variables that do not have their address taken. If you specify the `OPTIMIZE` compiler option, using the register attribute can greatly affect a program's performance.

Using Pointers

Because it is difficult, and in some cases impossible, to keep track of where pointers point to, use of pointers inhibits most memory optimizations such as dead store elimination and store motion.

Passing Function Arguments

Optimization is effective when function arguments are used. It is usually better to pass a value as an argument to a function than to let the function take the value from a global variable.

Coding Expressions

If components of an expression are duplicate expressions, code them either at the left end of the expression or within parentheses. For example:

```
a = b*(x*y*z);           /* Duplicates recognized */
c = x*y*z*d;
e = f + (x + y);
g = x + y + h;

a = b*x*y*z;             /* No duplicates recognized */
c = x*y*z*d;
e = f + x + y;
g = x + y + h;
```

The compiler can recognize `x*y*z` and `x + y` as duplicate expressions because they are either coded in parentheses or coded at the left end of the expression.

When components of an expression in a loop are constant, code the expressions either at the left end of the expression or within parentheses. If `c`, `d`, and `e` are constant and `v`, `w`, and `x` are variable, the following examples show the difference in evaluation:

```
v*w*x*(c*d*e);      /* Constant expressions recognized */
c + d + e + v + w + x;

v*w*x*c*d*e;        /* Constant expressions not recognized */
v + w + x + c + d + e;
```

Coding Conversions

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic. For example:

EDCXGOP3

```
 /* EDCXGOP3
    This example shows how numeric conversions are done
  */

int main(void)
{
   int i;
   float array[10];
   float x = 1.0;
   for (i = 0; i < 10; i++)
      {
       array[i] = array[i]*x;  /* No conversions needed */
       x = x + 1.0;
      }

   for (i = 1; i <= 9; i++)
      array[i] = array[i]*i;  /* Multiple conversions needed */

   return(0);
}
```

Figure 41. Numeric Conversions Example

When you must use mixed-mode arithmetic, code the fixed-point and floating-point arithmetic in separate computations as much as possible.

Using Arithmetic Constructions

Wherever possible, use multiplication rather than division. For example,

```
x*(1.0/3.0);  /*  1.0/3.0 is evaluated at compile time  */
```

produces faster code than:

```
x/3.0;
```

Assigning the divisor's reciprocal to a temporary variable and then multiplying by that variable is beneficial, especially if you divide many values by the same number in your code.

Input/Output Considerations

Consider the use of the file when choosing file attributes:
- Specify largest possible BLKSIZE (blocked files).
- `fseek()` on sequential files is most efficient when using `recfm=F` or `recfm=FBS`.

The proper choice of file attributes is important for efficient I/O.

When accessing files:
- Use the `putc()` or `getc()` macros instead of `fputc()` or `fgetc()`, respectively, if you must read or write a character.

  The `fputc()` function, as defined by ANSI, will put a single character to the text stream. Special action is taken when a control character is written. On the other hand, the `putc()` macro buffers characters in storage and invokes `fputc()` only when a control character is encountered. This reduces call overhead when writing characters one at a time.
- Use `fread()` instead of `fgets()` and `fwrite()` in place of `fputs()` wherever possible.
- Avoid using `fscanf()` or `fprintf()` if you can use other I/O routines instead.
- When using `fflush()` beware of `NULL` file pointers; `fflush(NULL)` will flush all streams.
- Use `fgetpos()` and `fsetpos()` instead of `ftell()` and `fseek()` when you are saving a position you will return to later. `fgetpos()` saves more information about the position than `ftell()`.
- Use memory files rather than files created with `tmpfile()`.

Using Built-In Library Functions and Macros

Include the appropriate library header files to trigger the use of built-in functions (that is, compiler-generated expansion for the function).

Including the proper library header files also prevents parameter type mismatch and ensures optimal performance. For a list of the built-in functions, see Appendix H, "Using Built-In Functions," on page 467. If you want an explicit call, you can enclose the function in parentheses, for example, `(memcpy)(buf1, buf2, len)` to force the function call.

For best performance, you should always include the ctype.h header file to use the following macros rather than their equivalent functions:

isalpha()	**islower()**	**isupper()**
isalnum()	**isprint()**	**isxdigit()**
iscntrl()	**ispunct()**	**toupper()**
isdigit()	**isspace()**	**tolower()**
isgraph()		

Use `memcmp()` to compare arrays, as in the following example:

```
if (!memcmp (a, b, sizeof(a)))
  /*    arrays are equal    */
```

is more efficient than a comparison in a loop such as:

```
int    a[1000], b[1000];

for (i = 0; i < 1000; ++i)
  if (a[i] != b[i])
    break;

if (i == 1000)
  /*    arrays are equal     */
```

Arrays are often compared using a loop (one element at a time). When two arrays are being compared for equality, the loop can be replaced with a memcmp(). In some cases, this means that the execution of hundreds (or thousands) of machine instructions are replaced by the execution of a few.

The C language does not allow structure comparison, because structures may contain padding bytes with undefined values. When it is known that no padding bytes exist, memcmp() should be used to compare structures. The AGGREGATE compiler option can be used to obtain a structure and union map.

As well, use memset() to clear structs, unions, arrays or character buffers as follows:

```
char c[10];

for (i = 0; i < 10; i++)           /*  do not use  */
  c[i] = ' ';

memset (c, ' ', sizeof (c));       /*  better      */
```

The memset() library function should be used to initialize a character buffer and when an array needs to be initialized to a repetitive byte pattern (such as zeros).

When using strlen() do not hide size information. Less code is needed for strlen() when the upper bound is known at compile time.

```
char    small_str_array[100];
char   *small_str_ptr;
   ⋮
x = strlen(small_str_ptr);   /*  unknown upper bound  */

x = strlen(small_str_array); /*  better  */
```

For best performance, if you are concatenating strings, use strcat() instead of sprintf(). If you are performing character to integer conversions, use atoi() rather than sscanf().

Try to replace str*xxx*() functions with their corresponding mem*xxx*() functions, because mem*xxx*() functions are more efficient. Some ways to minimize the execution cost of a str*xxx*() function are to use fixed length character buffers or to save the length of incoming string (including null terminator) for subsequent calls to memcpy() and memcmp().

```
total_len = strlen (s) + 1;
   ⋮
for (i = 0; i < 10; i++)
  if (memcmp (s, t[i], total_len) == 0)
```

```
⋮

memcpy (a, s, total_len);
```

For efficient string processing, save the length of a null-terminated string and use `mem`*xxx*`()` function calls; subsequent operations to compare or copy the string can use this length.

Note: You cannot replace all `strcmp()` calls with a `memcmp()` call with a `strlen()` value of one of the strings. `memcmp()` will not stop comparing strings when it encounters a null in one of the strings, possibly resulting in an attempt to access protected storage which follows the shorter string. This, in turn, could result in an exception.

Using Loops and Control Constructs

For the `for`-loop index variable:
- Some data types are preferred in terms of efficiency of reference: `int` and `double`.
- If you do not need to use `float` type variables, use `int` or `double`. If you do not need `double`, use `int`.
- If you use `enum` variable, expand the variable to be a fullword.
- Do not use the address operator (`&`) on the index.
- Index should not be a member of `union`.
- Use the `auto` or `register` storage class over the `extern` or `static` storage class.

When using `if` statements, order the `if` conditions efficiently; put the most decisive tests first and the most expensive tests last. By performing the most common tests before performing the less common tests, you increase the efficiency of your code; fewer loops are required to meet the test conditions.

```
if (command.is_classg &&;
    command.len == 6  &&;
   !strcmp (command.str, "LOGON"))  /* call to strcmp() most expensive */
  logon ();
```

Declaring a Data Type

Use the `int` data type instead of `char` when performing arithmetic:

```
char_var += '0';

int_var  += '0';         /*  better  */
```

A `char` type variable is efficient when you are:
- Assigning a literal to a `char` variable
- Comparing the variable with a `char` literal

```
char_var = 27;

if (char_var == 'D')
```

These data types are more expensive to reference:
- Unsigned short
- Signed char
- Float
- Long double

For example, use a `double` rather than a `float` when possible.

```
float_var++;

double_var++;     /*  better  */
```

For storage efficiency, the compiler will pack `enumeration` variables in 1, 2 or 4 bytes depending on the largest value of constant. If performance is critical, expand size to fullword by adding an enumeration constant with large value.

```
enum byte { land, sea, air, space };

enum word { low, medium, high, expand_to_fullword = INT_MAX };
```

For example, fullword `enumeration` variables are preferred when used as function parameters.

For efficient use of `extern` variables,
- Place scalars ahead of arrays in `extern struct`.
- Copy heavily referenced scalar to `auto` or `register` variables (especially when used in a loop).

When using bit fields:
- Even though the compiler supports a bit field spanning more than 4 bytes, the cost of referencing it is higher.
- An `unsigned` bit field is preferred over a `signed` bit field.
- A single bit member is referenced more efficiently than multiple bits.
- A bit field used to store integer values should have length 8, 16, or 24 bits and be on a byte boundary.

```
struct {    unsigned   xval  :8,
                       xbool :1,
                       xmany :6,
                       xset  :1;
} b;

if (b.xval == 3)
   ⋮
if (b.xmany + 5 == x)     /*  inefficient because it does not */
                          /*  fall on a byte boundary         */
   ⋮
if (b.xbool)
   ⋮
```

Using Library Extensions

Consider `fetch()` instead of `system()` for calling other LE/VSE C Run-Time modules.

A `system()` call does full environment initialization and termination, but a `fetch()`ed routine shares the environment of the calling routine. As well, you have control of when the module is deleted with `release()`, and you can easily pass parameters to a `fetch()`ed module.

Use memory files as efficient temporary files, by using the `type=memory` attribute in `fopen()` before creating the temporary file. Some applications use temporary files to pass data between program modules.

Optimizing Dynamic Memory

Use the `STACK`, `HEAP`, and `RPTSTG(ON)` run-time options to optimize your run-time space requirements. See *LE/VSE Programming Guide* for more information on run-time storage.

Part 4. LE/VSE C Run-Time Environments

This part describes the different LE/VSE C Run-Time environments.

Chapter 20. Using Run-Time User Exits

This chapter shows how to use run-time user exits with the LE/VSE run-time library. This is general-use programming interface information and associated guidance information for using the library.

This chapter is provided here for your convenience. For further information on using run-time user exits in the LE/VSE environment, refer to *LE/VSE Programming Guide*.

Using Run-Time User Exits in LE/VSE

LE/VSE provides user exits that you can use for functions at your installation. You can use the assembler user exit (CEEBXITA) or the HLL user exit (CEEBINT). This section provides information about using these run-time user exits.

Understanding the Basics

User exits are invoked under LE/VSE to perform enclave initialization functions and both normal and abnormal termination functions. User exits offer you a chance to perform certain functions at a point where you would not otherwise have a chance to do so. In an assembler initialization user exit, for example, you can specify a list of run-time options that establish characteristics of the environment. This is done prior to the actual execution of any of your application code. Another example is using an assembler termination user exit to request a dump after your application has terminated with an abend.

In most cases, you do not need to modify any user exit to run your application. Instead, you can accept the IBM-supplied default versions of the exits, or the defaults as defined by your installation. To do so, run your application in the normal manner and the default versions of the exits are invoked. You may also want to read the sections “User Exits Supported under LE/VSE” and “Order of Processing of User Exits” on page 202, which provide an overview of the user exits and describe when they are invoked.

If you plan to modify either of the user exits to perform some specific function, you must link the modified exit to your application before running, as described in “Using Installation-Wide or Application-Specific User Exits” on page 203. In addition, the sections “Using the Assembler User Exit” on page 204 and “High Level Language User Exit Interface” on page 214 describe the respective user exit interfaces to which you must adhere to change an assembler or HLL user exit.

User Exits Supported under LE/VSE

LE/VSE provides two user exit routines, one written in assembler and the other in an LE/VSE-conforming HLL. You can find sample jobs containing these user exits in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE).

The user exits supported by LE/VSE are shown in Table 34.

Table 34. User Exits Supported under LE/VSE

Name	Type of User Exit	When Invoked
CEEBXITA	Assembler user exit	Enclave initialization Enclave termination Process termination
CEEBINT	HLL user exit. CEEBINT can be written in LE/VSE C Run-Time, PL/I, LE/VSE-conforming assembler (see restrictions in "Order of Processing of User Exits").	Enclave initialization

Order of Processing of User Exits

The location and order in which user exits are driven for your application are summarized in Figure 42.

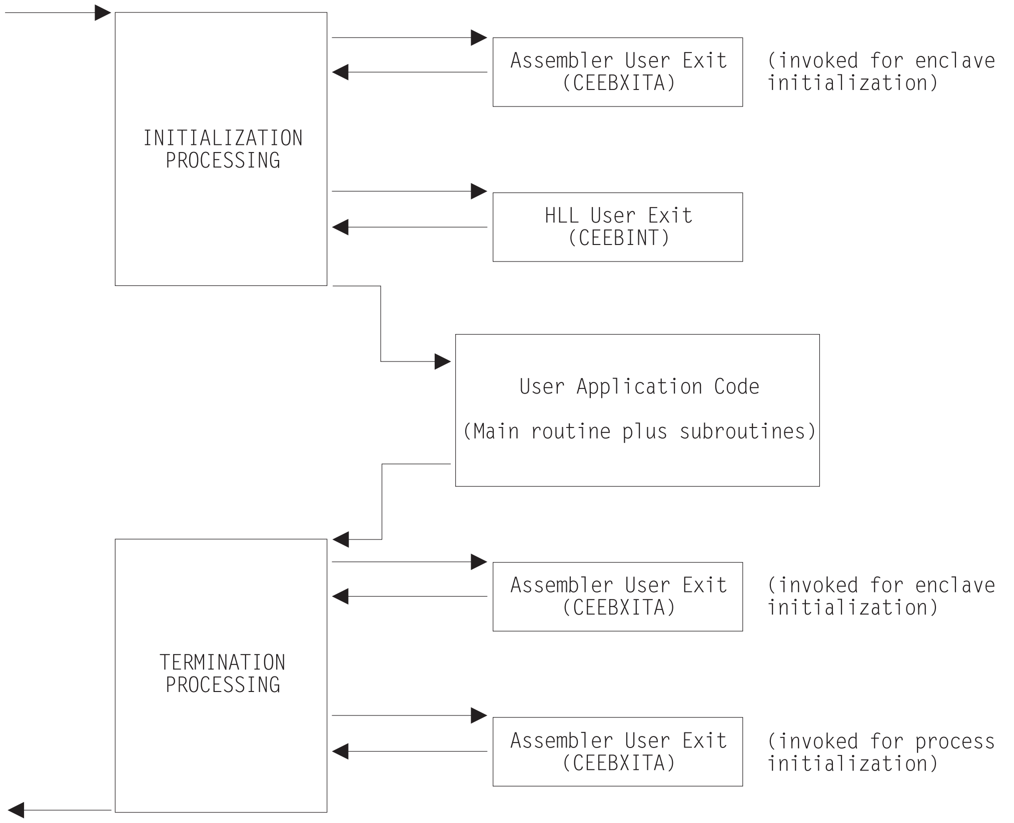

INITIALIZATION PROCESSING
Assembler User Exit (CEEBXITA)
(invoked for enclave initialization)
HLL User Exit (CEEBINT)
User Application Code
(Main routine plus subroutines)
TERMINATION PROCESSING
Assembler User Exit (CEEBXITA)
(invoked for enclave initialization)
Assembler User Exit (CEEBXITA)
(invoked for process initialization)

Figure 42. Location of User Exits

In Figure 42, run-time user exits are invoked in the following sequence:

1. Assembler user exit is invoked for enclave initialization.

   The assembler user exit (CEEBXITA) is invoked very early during the initialization process before the enclave initialization is complete. Early invocation of the assembler exit allows the enclave initialization code to benefit from any changes that might be contained in the exit. If run-time options are provided in the assembler exit, the enclave initialization code is aware of the new options.

2. Environment is established.
3. HLL user exit is invoked.

   The HLL initialization exit (CEEBINT) is invoked just before the invocation of the application code. In LE/VSE, this exit can be written in C PL/I, or LE/VSE-conforming assembler. However, you can only write CEEBINT in C if the following conditions are met:

   - CEEBINT must be declared with OS linkage, that is, you must include the `#pragma linkage(CEEBINT,OS)` preprocessor directive. Your application code must be compiled with the `RENT` compile-time option.
   - You must prelink together your application code and CEEBINT object modules.
   - CEEBINT must be used as an application-specific user exit, rather than as an installation-wide user exit (refer to “Using Installation-Wide or Application-Specific User Exits” for more information).

   The HLL initialization exit can *not* be written in COBOL, although COBOL applications can use this HLL user exit. At the time when CEEBINT is invoked, the run-time environment is fully operational and all LE/VSE-conforming HLLs are supported.
4. Main routine is invoked.
5. Main routine returns control to caller.
6. Environment is terminated.
7. Assembler user exit is invoked for termination of the enclave.

   CEEBXITA is invoked for enclave termination processing after all application code in the enclave has completed, but prior to any enclave termination activity.
8. Assembler user exit is invoked for termination of the process.

   CEEBXITA is invoked again when the LE/VSE process terminates.

Although both the assembler and HLL exits are invoked for initialization, they do not perform exactly the same functions. See “CEEBXITA Behavior during Enclave Initialization” on page 204 and “High Level Language User Exit Interface” on page 214 for a detailed description of each exit.

LE/VSE provides the CEEBXITA assembler user exit for termination but does not provide a corresponding HLL termination user exit.

Using Installation-Wide or Application-Specific User Exits

IBM offers default versions of CEEBXITA and CEEBINT. You can use the IBM-supplied default version of either exit, or you can customize CEEBXITA or CEEBINT for use on an installation-wide basis. When CEEBXITA or CEEBINT is linked with the LE/VSE initialization/termination library routines during installation, it functions as an installation-wide user exit.

Finally, you can customize CEEBXITA or CEEBINT yourself for use on your application. When CEEBXITA or CEEBINT is linked in your phase, it functions as an application-specific user exit. The application-specific exit is used only when you run that application. The installation-wide assembler user exit is not executed.

In order to obtain an application-specific user exit, you must explicitly include it at link-edit time in the application phase using an INCLUDE link-edit control statement. Any time that the application-specific exit is modified, it must be relinked with the application.

The assembler user exit interface is described in “CEEBXITA Assembler User Exit Interface” on page 206. The HLL user exit interface is described in “High Level Language User Exit Interface” on page 214.

Using the Assembler User Exit

The assembler user exit CEEBXITA tailors the characteristics of the enclave before it is established. CEEBXITA must be written in assembler language because an HLL environment may not yet be established when the exit is invoked. CEEBXITA is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave in the process or a nested enclave. CEEBXITA can differentiate easily between first and nested enclaves. For more information about nested enclaves, see *LE/VSE Programming Guide*.

CEEBXITA behaves differently depending on when it is invoked, as described in the following sections.

Using Sample Assembler User Exits

Sample assembler user exit programs are distributed with LE/VSE. You can use them and modify the code for the requirements of your own application. Choose a sample program appropriate for your application. The following assembler exit user programs are delivered with LE/VSE.

Table 35. Sample Assembler User Exits for LE/VSE

Example User Exit	Operating System	Language (if Language Specific)
CEEBXITA	VSE (default)	
CEECXITA	CICS (default)	
CEEBX05A	VSE	COBOL

Notes:

1. CEEBXITA and CEECXITA are the defaults on your system for VSE and CICS, if LE/VSE is installed at your site without modification.
2. The source code for CEEBXITA and CEEBX05A can be found on VSE in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE).
3. CEEBX05A is an example user exit program for COBOL applications on VSE.

CEEBXITA Behavior during Enclave Initialization

The CEEBXITA assembler user exit is invoked before enclave initialization is performed. You can use it to help guide the establishment of the environment in which your application runs. The user exit can interrogate program parameters supplied in the JCL and change them if desired. In addition, you can specify run-time options in the user exit using the `CEEAUE_OPTION` field of the assembler interface (see “CEEBXITA Assembler User Exit Interface” on page 206 for information about how to do this).

CEEBXITA performs no special tasks, but simply returns control to LE/VSE initialization.

CEEBXITA Behavior during Enclave Termination

The CEEBXITA assembler exit is invoked after the user code for the enclave has completed, but before the occurrence of any enclave termination activity. For example, CEEBXITA is invoked before the storage report is produced (if one was requested), data sets are closed, and HLLs are invoked for enclave termination. In other words, the assembler user exit for termination is invoked when the environment is still active.

The VSE assembler user exits allow you to request an abend. Under VSE (as well as CICS under VSE), you can also request a dump to assist in problem diagnosis. Note that termination activities have not yet begun when the user exit is invoked. Thus, the majority of storage has not been modified when the dump is produced.

It is possible to request an abend and dump in the enclave termination user exit for all enclave-terminating events.

Example code that shows how to request an abend and dump when there is an unhandled condition of severity 2 or greater can be found in the member `CEEBX05A.A` in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE).

CEEBXITA Behavior during Process Termination

The CEEBXITA assembler exit is invoked after:
- All enclaves have terminated.
- The enclave resources have been relinquished.
- Any LE/VSE-managed files have been closed.
- Debug Tool for VSE/ESA has terminated.

This allows you to free files at this time, and it presents another opportunity to request an abend.

During termination, CEEBXITA can interrogate the LE/VSE reason and return codes and, if necessary, request an abend with or without a dump. This can be done at either enclave or process termination.

The IBM-supplied CEEBXITA performs no special tasks, but simply returns control to LE/VSE termination.

Specifying Abend Codes to Be Percolated by LE/VSE

The assembler user exit, when invoked for initialization in the batch environment, can return a list of VSE cancel codes, program-interruption codes, and user abend codes (contained in the `CEEAUE_CODES` field of the assembler user exit interface—see “CEEBXITA Assembler User Exit Interface” on page 206) that are to be exempted from LE/VSE condition handling.

When an abend or program interrupt occurs in your application, and `TRAP(ON)` is in effect, and the VSE cancel code, program-interruption code, or user abend code is in the `CEEAUE_CODES` list, LE/VSE produces an abnormal termination message and issues an abend to terminate the enclave. Normal LE/VSE condition handling is never invoked to handle these conditions. The feature is useful when you do not want LE/VSE condition handling to intervene for certain abends, and when you want to produce a system dump.

When `TRAP(OFF)` is specified and there is a program interrupt, the user exit for termination is not driven. The use of `TRAP(OFF)` is not recommended; refer to *LE/VSE Programming Reference* for more information.

Actions Taken for Errors that Occur within the Assembler User Exit

If any errors occur during the enclave initialization user exit, the standard system action occurs because LE/VSE condition handling has not yet been established.

Any errors occurring during the enclave termination user exit lead to abnormal termination (through an abend) of the LE/VSE environment.

If there is a program check during the enclave termination user exit and TRAP(ON) is in effect, the application ends abnormally with ABEND code 4044 and reason code 44. If there is a program check during the enclave termination exit and TRAP(OFF) has been specified, the application ends abnormally without additional error checking support. LE/VSE provides no condition handling; error handling is performed by the operating system. The use of TRAP(OFF) is not recommended; refer to *LE/VSE Programming Reference* for more information.

LE/VSE takes the same actions as described above for program checks during the process termination user exit.

CEEBXITA Assembler User Exit Interface

You can modify CEEBXITA to perform any function desired, although the exit must have the following attributes after you modify it:

- The user-supplied exit must be named CEEBXITA.
- The exit must be reentrant.
- The exit must be capable of executing in AMODE(ANY) and RMODE(ANY).
- The exit must be relinked with the application after modification (if you want an application-specific user exit), or relinked with LE/VSE initialization/termination routines after modification (if you want an installation-wide user exit).

If a user exit is modified, you are responsible for conforming to the interface shown in Figure 43 on page 207. This user exit must be written in assembler.

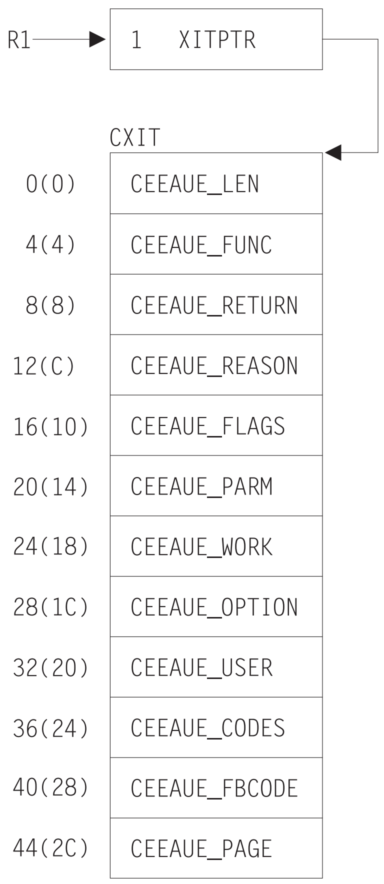


Figure 43. Interface for Assembler User Exits

When the user exit is called, register 1 (R1) points to a word that contains the address of the CXIT control block. The high order bit is on.

The CXIT control block contains the following fullwords:

CEEAUE_LEN (input parameter)
A fullword integer that specifies the total length of this control block. For LE/VSE, the length is 48 bytes.

CEEAUE_FUNC (input parameter)
A fullword integer that specifies the function code. In LE/VSE, the following function codes are supported:
1 – Initialization of the first enclave within a process
2 – Termination of the first enclave within a process
3 – Nested enclave initialization
4 – Nested enclave termination
5 – Process termination

The user exit should ignore function codes other than those numbered from 1 through 5.

CEEAUE_RETURN (input/output parameter)
A fullword integer that specifies the return or abend code. CEEAUE_RETURN has different meanings, depending on whether it is an input parameter or an output parameter:

- As an input parameter, CEEAUE_RETURN is the enclave return code.

- As an output parameter, `CEEAUE_RETURN` has different meanings, depending on the flag `CEEAUE_ABND` (see below):
  - If the flag `CEEAUE_ABND` is off, `CEEAUE_RETURN` is interpreted as the LE/VSE return code placed in register 15.
  - If the flag `CEEAUE_ABND` is on, `CEEAUE_RETURN` is interpreted as an abend code used when an abend is issued. (In batch, run-time message CEE3322C is produced and an operating system request is issued to terminate the enclave; in CICS, an `EXEC CICS ABEND` is issued.)

See *LE/VSE Programming Guide* for more information about how LE/VSE computes return and reason codes.

CEEAUE_REASON (input/output parameter)
A fullword integer that specifies the reason code for `CEEAUE_RETURN`. `CEEAUE_REASON` has different meanings, depending on whether it is an input parameter or an output parameter:

- As an input parameter, `CEEAUE_REASON` is the LE/VSE return code modifier.
- As an output parameter, `CEEAUE_REASON` has different meanings, depending on the flag `CEEAUE_ABND` (see below):
  - If the flag `CEEAUE_ABND` is off, `CEEAUE_REASON` is interpreted as the LE/VSE return code modifier placed in register 0.
  - If the flag `CEEAUE_ABND` is on, `CEEAUE_REASON` is interpreted as an abend reason code used when an abend is issued. (`CEEAUE_REASON` is used in the batch abnormal-termination run-time message CEE3322C, but is ignored in the CICS environments when an `EXEC CICS ABEND` is issued.)

See *LE/VSE Programming Guide* for more information about how LE/VSE computes return and reason codes.

CEEAUE_FLAGS (input/output parameter)
Contains four flag bytes. CEEBXITA uses only the first byte but reserves the remaining bytes. All unspecified bits and bytes must be zero. The layout of these flags is shown in Figure 44.

```
Byte 0
        x... .... - CEEAUE_ABTERM
        0... .... - Normal termination
        1... .... - Abnormal termination
        .x.. .... - CEEAUE_ABND
        .0.. .... - Terminate with CEEAUE_RETURN
        .1.. .... - Abend with CEEAUE_RETURN and CEEAUE_REASON given
        ..x. .... - CEEAUE_DUMP
        ..0. .... - If CEEAUE_ABND=0, abend with no dump
        ..1. .... - If CEEAUE_ABND=1, abend with a dump
        ...0 0000 - Reserved bits (must be zero)
Byte 1
        00 - Reserved for future use
Byte 2
        00 - Reserved for future use
Byte 3
        00 - Reserved for future use
```

Figure 44. CEEAUE_FLAGS Format

Byte 0 (`CEEAUE_FLAG1`) has the following meaning:

CEEAUE_ABTERM (input parameter)

OFF Indicates that the enclave terminates normally (severity 0 or 1 condition).

ON Indicates that the enclave terminates with an LE/VSE return code modifier of 2 or greater. This could, for example, indicate that a condition of severity 2 or greater was raised that was unhandled.

CEEAUE_ABND (input/output parameter)

OFF Indicates that the enclave should terminate without an abend being issued. Thus, `CEEAUE_RETURN` and `CEEAUE_REASON` are placed into register 15 and register 0 respectively and returned to the enclave creator.

ON Indicates that the enclave terminates with an abend. Thus, `CEEAUE_RETURN` and `CEEAUE_REASON` are used by LE/VSE in the invocation of the abend. When running in the batch environment, run-time message CEE3322C is produced and an operating system request is issued to terminate the enclave. When running under CICS, an `EXEC CICS ABEND` command is issued using the abend code contained in `CEEAUE_RETURN`. `CEEAUE_REASON` is ignored under CICS.

The `TRAP` run-time option does not affect the setting of `CEEAUE_ABND`.

CEEAUE_DUMP (output parameter)

OFF Indicates that if you request an abend, an abend is issued without requesting a system dump.

ON Indicates that if you request an abend, an abend is issued requesting a system dump.

CEEAUE_PARM (input/output parameter)
A fullword pointer to the parameter address list of the application program.

As an input parameter, this fullword contains the register 1 value passed to the main routine. The exit can modify this value, and the value is then passed to the main routine. If run-time options are present in the invocation command string, they are stripped off before the exit is called.

If the parameter inbound to the main routine is a character string, `CEEAUE_PARM` contains the address of a fullword address that points to a halfword prefixed string. If this string is altered by the user exit, the string must not be extended in place.

CEEAUE_WORK (input parameter)
Contains a fullword pointer to a 256-byte work area that the exit can use. On entry, it contains binary zeros and is doubleword-aligned.

This area does not persist across exits.

CEEAUE_OPTION (output parameter)
On return, this field contains a fullword pointer to the address of a halfword length prefixed character string that contains run-time options. These options are processed for enclave initialization only. When invoked for enclave termination, this field is ignored.

These run-time options override all other sources of run-time options except those that are specified as nonoverrideable in the installation default run-time options.

Under CICS, the STACK run-time option cannot be modified using the assembler user exit.

CEEAUE_USER (input/output parameter)
Contains a fullword whose value is maintained without alteration and passed to every user exit. On entry to the enclave initialization user exit, it is zero. Thereafter, the value of the user word is not altered by LE/VSE or any member libraries. The user exit can change the value of this field and LE/VSE maintains this value. This allows a user exit to initialize the fullword and pass it to subsequent user exits.

CEEAUE_CODES (output parameter)
During the initialization exit, CEEAUE_CODES contains the fullword address of a table of VSE cancel codes, program-interruption codes, and user-abend codes that the LE/VSE condition handler exempts from normal condition handling. Therefore, the application is not given the opportunity to field the abend. The table consists of:

- A fullword count of the number of cancel codes, program-interruption codes, and abend codes that are to be exempted from LE/VSE condition handling, and passed to the operating system.
- A fullword for each of the particular cancel codes, program-interruption codes, or abend codes that are to be exempted from LE/VSE condition handling, and passed to the operating system.
  - User abend codes are specified as F'uuu'. For example, if you want user abend 777 to be exempted from LE/VSE condition handling, code F'777'.
  - VSE cancel codes are specified as X'000000cc'. Avoid specifying the value X'00000020', which indicates a program check has occurred. If you specify the value X'00000020', LE/VSE ignores it, and normal LE/VSE condition handling semantics take effect. If you want to exempt specific program checks from LE/VSE condition handling, specify the program-interruption codes.
  - Program-interruption codes are specified as X'800000ii'. For example, if you want an operation exception to be exempted from LE/VSE condition handling, code X'80000001'.

This function is not enabled under CICS.

CEEAUE_FBCODE (input parameter)
Contains the fullword address of the condition token with which the enclave terminated. If the enclave terminates normally (that is, not because of a condition), the condition token is zero.

CEEAUE_PAGE (input/output parameter)
Usage of this field is related to PL/I BASED variables that are allocated storage outside of AREAs. You can indicate whether storage should be allocated on a 4K-page boundary. You can specify the minimum number of bytes of storage that you want allocated. Your allocation request must be an exact multiple of 4K. The IBM-supplied default setting for CEEAUE_PAGE is 32768 (32K).

If CEEAUE_PAGE is set to zero, PL/I BASED variables can be placed on other than 4K-page boundaries.

CEEAUE_PAGE is honored only during enclave initialization (that is, when CEEAUE_FUNC is 1 or 3).

Parameter Values in the Assembler User Exit

The parameters described in the following sections contain different values depending on how the user exit is used. Possible values are shown for the parameters based on how the assembler user exit is invoked.

First Enclave within Process Initialization—Entry

CEEAUE_LEN	48
CEEAUE_FUNC	1 (first enclave within process initialization function code).
CEEAUE_RETURN	0
CEEAUE_REASON	0
CEEAUE_FLAGS	0
CEEAUE_PARM	The register 1 value from the operating system.
CEEAUE_WORK	Address of a 256-byte work area of binary zeros.
CEEAUE_USER	0
CEEAUE_FBCODE	0
CEEAUE_PAGE	Minimum number of storage bytes to be allocated for PL/I BASED variables (default is 32768).

First Enclave within Process Initialization—Return

CEEAUE_RETURN	0, or if CEEAUE_ABND = 1, the abend code.
CEEAUE_REASON	0, or if CEEAUE_ABND = 1, the reason code for CEEAUE_RETURN.
CEEAUE_FLAGS	CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.
	CEEAUE_DUMP = 1 if the abend should request a dump.
CEEAUE_PARM	Register 1, used as the new parameter list.
CEEAUE_OPTION	Pointer to the address of a halfword prefixed character string containing run-time options, or 0.
CEEAUE_USER	Value of CEEAUE_USER for all subsequent exits.
CEEAUE_CODES	Pointer to the abend code table, or 0.
CEEAUE_PAGE	User specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default is 32768).

First Enclave within Process Termination—Entry

CEEAUE_LEN	48
CEEAUE_FUNC	2 (first enclave within process termination function code).
CEEAUE_RETURN	Return code issued by the application that is terminating.
CEEAUE_REASON	Reason code that accompanies CEEAUE_RETURN.

CEEAUE_FLAGS	CEEAUE_ABTERM = 1 if the application is terminating with a LE/VSE return code modifier of 2 or greater, or 0 otherwise.
	
	CEEAUE_ABND = 0
	
	CEEAUE_DUMP = 0

CEEAUE_WORK	Address of a 256-byte work area of binary zeros.
CEEAUE_USER	Return value from the previous exit.
CEEAUE_FBCODE	Feedback code causing termination.

First Enclave within Process Termination—Return

CEEAUE_RETURN	If CEEAUE_ABND = 0, the return code placed in register 15 when the enclave terminates. If CEEAUE_ABND = 1, the abend code.
CEEAUE_REASON	If CEEAUE_ABND = 0, the enclave reason code. If CEEAUE_ABND = 1, the abend reason code.
CEEAUE_FLAGS	CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing. CEEAUE_DUMP = 1 if the abend should request a dump.
CEEAUE_USER	The value of CEEAUE_USER for all subsequent exits.

Nested Enclave Initialization—Entry

CEEAUE_LEN	48
CEEAUE_FUNC	3 (nested enclave initialization function).
CEEAUE_RETURN	0
CEEAUE_REASON	0
CEEAUE_FLAGS	0
CEEAUE_PARM	The register 1 value discovered in a nested enclave creation.
CEEAUE_WORK	Address of a 256-byte work area of binary zeros.
CEEAUE_USER	The return value from previous exit.
CEEAUE_FBCODE	0
CEEAUE_PAGE	Minimum number of storage bytes to be allocated for PL/I BASED variables (default is 32768).

Nested Enclave Initialization—Return

CEEAUE_RETURN	0, or if CEEAUE_ABND = 1, the abend code.
CEEAUE_REASON	0, or if CEEAUE_ABND = 1, the reason code for CEEAUE_RETURN.
CEEAUE_FLAGS	CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

	CEEAUE_DUMP = 1 if the abend should request a dump.
CEEAUE_PARM	Register 1 used as the new parameter list.
CEEAUE_OPTION	Pointer to a fullword address that points to a halfword prefixed string containing run-time options, or 0.
CEEAUE_USER	The value of CEEAUE_USER for all subsequent exits.
CEEAUE_CODES	Pointer to the abend code table, or 0.
CEEAUE_PAGE	User specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default is 32768).

Nested Enclave Termination—Entry

CEEAUE_LEN	48
CEEAUE_FUNC	4 (termination function).
CEEAUE_RETURN	Return code issued by the enclave that is terminating.
CEEAUE_REASON	Reason code that accompanies CEEAUE_RETURN.
CEEAUE_FLAGS	CEEAUE_ABTERM = 1 if the application is terminating with an LE/VSE return code modifier of 2 or greater, or 0 otherwise.
	CEEAUE_ABND = 0
	CEEAUE_DUMP = 0
CEEAUE_WORK	Address of a 256-byte work area of binary zeros.
CEEAUE_USER	Return value from previous exit.
CEEAUE_FBCODE	Feedback code causing termination.

Nested Enclave Termination—Return

CEEAUE_RETURN	If CEEAUE_ABND = 0, the return code from the enclave.
	If CEEAUE_ABND = 1, the abend code.
CEEAUE_REASON	If CEEAUE_ABND = 0, the enclave reason code.
	If CEEAUE_ABND = 1, the enclave reason code.
CEEAUE_FLAGS	CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.
	CEEAUE_DUMP = 1 if the abend should request a dump.
CEEAUE_USER	Value of CEEAUE_USER for all subsequent exits.

Process Termination—Entry

CEEAUE_LEN	48
CEEAUE_FUNC	5 (process termination function).
CEEAUE_RETURN	Return code presented to the invoking system in

	register 15 that reflects the value returned from the first enclave within process termination.
CEEAUE_REASON	Reason code accompanying CEEAUE_RETURN that is presented to the invoking system in register 0 and reflects the value returned from the first enclave within process termination.
CEEAUE_FLAGS	CEEAUE_ABTERM = 1 if the last enclave is terminating abnormally (that is, an LE/VSE return code modifier is 2 or greater). This reflects the value returned from the first enclave within process termination (function code 2).
	CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing first enclave within process termination (function code 2).
	CEEAUE_DUMP = 0
CEEAUE_WORK	Address of a 256-byte work area of binary zeros.
CEEAUE_USER	The return value from previous exit.
CEEAUE_FBCODE	The feedback code causing termination.

Process Termination—Return

CEEAUE_RETURN	If CEEAUE_ABND = 0, the return code from the process.
	If CEEAUE_ABND = 1, the abend code.
CEEAUE_REASON	If CEEAUE_ABND = 0, the reason code for CEEAUE_RETURN from the process.
	If CEEAUE_ABND = 1, reason code for the CEEAUE_RETURN abend reason code.
CEEAUE_FLAGS	CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.
	CEEAUE_DUMP = 1 if the abend should request a dump.
CEEAUE_USER	The value of CEEAUE_USER for all subsequent exits.

High Level Language User Exit Interface

LE/VSE provides CEEBINT, an HLL user exit, for enclave initialization. You can code CEEBINT in LE/VSE C Run-Time, PL/I, or LE/VSE C Run-Time (subject to the restrictions in “Order of Processing of User Exits” on page 202), or LE/VSE-conforming assembler. The HLL user exit cannot be written in COBOL. COBOL programmers can use an HLL exit written in LE/VSE C Run-Time, PL/I, LE/VSE-conforming assembler, LE/VSE C Run-Time (again, subject to the restrictions in “Order of Processing of User Exits” on page 202), or default to the IBM-supplied default HLL user exit (which is written in LE/VSE C Run-Time).

The HLL enclave initialization exit is invoked after the enclave has been established, after the Debug Tool for VSE/ESA initial command string has been processed, and prior to the invocation of compiled code. When invoked, it is passed a parameter list that conforms to the LE/VSE definition. The parameters are all fullwords and are defined as follows:

Number of arguments in parameter list (input)
A fullword binary integer.
- On entry: Contains 7.
- On exit: Not applicable.

Return code (output)
A fullword binary integer.
- On entry: 0.
- On exit: Able to be set by the exit, but not interrogated by LE/VSE.

Reason code (output)
A fullword binary integer.
- On entry: 0
- On exit: Able to be set by the exit, but not interrogated by LE/VSE.

Function code (input)
A fullword binary integer.
- On entry: 1, indicating the exit is being driven for initialization.
- On exit: Not applicable.

Address of the main program entry point (input)
A fullword binary address.
- On entry: The address of the routine that gains control first.
- On exit: Not applicable.

User word (input/output)
A fullword binary integer.
- On entry: Value of the user word (CEEAUE_USER) as set by the assembler user exit. See page 210 for a description of the CEEAUE_USER field.
- On exit: The value set by the user exit, maintained by LE/VSE and passed to subsequent user exits.

Exit List Address (output)
A fullword binary integer reserved for future use.

This allows the establishment of one or more user exits when the enclave user exit sets this field to a list of user exits. Currently, only one user exit is supported in LE/VSE.

A_Exits
The address of the exit list control block, Exit_list.
- On entry: 0.
- On exit: 0, unless you establish a hook exit, in which case you would set this pointer and fill in relevant control blocks. The control blocks for Exit_list and Hook_exit are shown in Figure 45 on page 216.

As supplied, CEEBINT has only one exit defined that you can establish: the hook exit described by the Hook_exit control block. This exit gains control when hooks generated by the PL/I compile-time TEST option are executed. You can establish this exit by setting appropriate pointers (A_Exits to Exit_list to Hook_exit). Figure 45 on page 216 illustrates the Exit_list and Hook_exit control blocks.

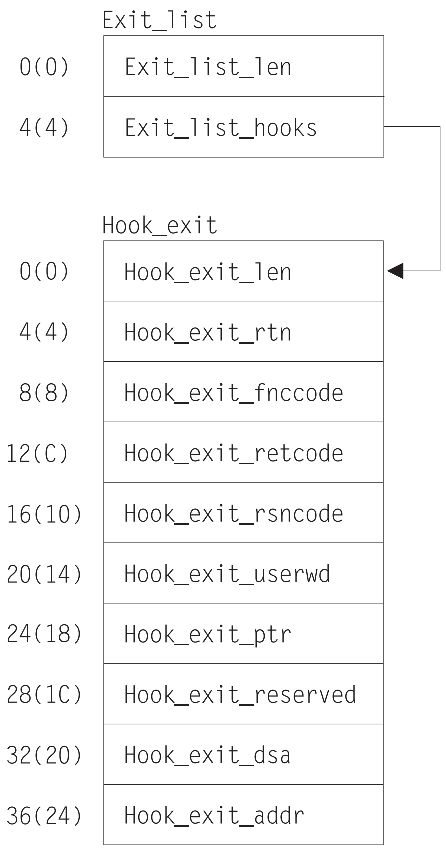


Figure 45. Exit_list and Hook_exit Control Blocks

The control block `Exit_list` exit contains the following fields:

Exit_list_len
The length of the control block. It must be 1.

Exit_list_hooks
The address of the `Hook_exit` control block.

The control block for the hook exit must contain the following fields:

Hook_exit_len
The length of the control block.

Hook_exit_rtn
The address of a routine you want invoked for the exit. When the routine is invoked, it is passed the address of this control block. Because this routine is invoked only if the address you specify is nonzero, you can turn the exit on and off.

Hook_exit_fnccode
The function code with which the exit is invoked. This is always 1.

Hook_exit_retcode
The return code set by the exit. You must ensure it conforms to the following specifications:

0	Requests that Debug Tool for VSE/ESA be invoked next
4	Requests that the program resume immediately

16 Requests that the program be terminated

Hook_exit_rsncode
The reason code set by the exit. This is always zero.

Hook_exit_userwd
The user word passed to the user exits.

Hook_exit_ptr
An exit-specific user word.

Hook_exit_reserved
Reserved.

Hook_exit_dsa
The contents of register 13 when the hook was executed.

Hook_exit_addr
The address of the hook instruction executed.

Usage Requirements

1. The user exit must not be a main-designated routine. For example, it cannot be a C `main()` function.
2. The HLL exit routines must be linked with compiled code. If you do not provide an initialization user exit, an IBM-supplied default, which simply returns control to your application, is linked with the compiled code.
3. The exit cannot be written in COBOL.
4. The exit should be coded so that it returns for all unknown function codes.
5. LE/VSE C Run-Time constructs such as the `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` functions terminate the enclave.
6. A PL/I `EXIT` or `STOP` statement terminates the enclave.
7. Use the callable service IBMBHKS to turn hooks on and off. For more information about IBMBHKS, see *IBM PL/I for VSE/ESA Programming Guide*.

Chapter 21. Using Environment Variables

This chapter describes environment variables that affect the LE/VSE C Run-Time environment. You can use environment variables to define the characteristics of a specific environment. They may be set, retrieved, and used during the execution of a LE/VSE C Run-Time program.

The following environment variables affect the LE/VSE C Run-Time environment if they are on when an application program runs. The variables that begin with _EDC_ and _CEE_ are described in detail in "Environment Variables Specific to the LE/VSE C Run-Time Library" on page 221. See "Locale Source Files" on page 329 for more information on the locale-related environment variables.

_CEE_ENVFILE
Used to read environment variables from a specified file.

_EDC_BYTE_SEEK
Specifies that `fseek()` and `ftell()` should use relative byte offsets.

_EDC_COMPAT
Specifies that LE/VSE C Run-Time should use specific functional behavior from previous releases of C/370.

_EDC_RRDS_HIDE_KEY
Relevant for VSAM RRDS files opened in record mode. Enables calls to `fread()` that specify a pointer to a character string and do not append the Relative Record Number to the beginning of the string.

_EDC_STOR_INCREMENT
Sets the size of increments to the internal library storage subpool.

_EDC_STOR_INITIAL
Sets the initial size of the internal library storage subpool.

_EDC_ZERO_RECLEN
Enables processing of zero-length records in a SAM file opened in variable format.

LANG
Determines the locale to use for the locale categories when neither the LC_ALL environment variable nor the individual locale environment variables specify locale information. This environment variable does not interact with the language setting for messages.

LC_ALL
Determines the locale to be used to override any values for locale categories specified by the settings of the LANG environment variable or any individual locale environment variables.

LC_COLLATE
Determines the locale to be used to define the behavior of ranges, equivalence classes, and multicharacter collating elements.

LC_CTYPE
Determines the locale for the interpretation of byte sequences of text data as characters (for example, single-byte versus multibyte characters in arguments and input files).

LC_MESSAGES
Determines the locale which defines the language in which messages are written.

LC_MONETARY
Determines the locale for monetary-related numeric formatting information.

LC_NUMERIC
Determines the locale for numeric formatting (for example, thousands separator and radix character) information.

LC_TIME
Determines the locale for date and time formatting information.

LC_TOD
Determines the locale for time of day and Daylight Savings Time formatting information.

Working with Environment Variables

The following library functions affect environment variables:
- `setenv()`
- `clearenv()`
- `getenv()`

The `setenv()` function adds, changes, and deletes environment variables in the Environment Variable Table. The `getenv()` function retrieves the values from the table. If it does not find an environment variable, `getenv()` returns `NULL`. The `clearenv()` function clears the environment variable table, and resets to default behavior the actions affected by LE/VSE C Run-Time-specific environment variables.

For a complete description of these functions, refer to *LE/VSE C Run-Time Library Reference*.

Environment variables may be set any time in an application program or user exit. You can use the exit routine CEEBINT to set environment variables through calls to `setenv()`. For more information on the LE/VSE user exit CEEBINT, refer to “Using Run-Time User Exits in LE/VSE” on page 201. You can also set environment variables by using the `ENVAR` run-time option. The syntax for this option is

```
ENVAR("1st_var=1st_value", "2nd_var=2nd_value").
```

For more information on this run-time option, refer to *LE/VSE Programming Reference*.

Specifying the _CEE_ENVFILE environment variable with a *filename* on the `ENVAR` option enables you to read more environment variables from that file. See “Environment Variables Specific to the LE/VSE C Run-Time Library” on page 221 for more information about _CEE_ENVFILE.

Environment variables set with the `setenv()` function exist only for the life of the program, and are not saved before program termination. Child programs are initialized with the environment variables of the parent. However, environment variables set by a child program are not propagated back to the parent upon termination of the child program.

Naming Conventions

Avoid the following when creating names for environment variables:

=	This is invalid and will generate an error message.
EDC	This is reserved for LE/VSE C Run-Time-specific environment variables.
CEE	This is reserved for LE/VSE C Run-Time-specific environment variables used with LE/VSE. See "Environment Variables Specific to the LE/VSE C Run-Time Library" for more information.
DBCS Characters	Multibyte and DBCS characters should not be used in environment variable names. Their use can result in unpredictable behavior. Multibyte and DBCS characters are allowed in environment variable values; however, the values are not validated, and redundant shifts are not removed.
White Space	Blank spaces are valid characters and should be used carefully in environment variable names and values. For example, `setenv(" my name","  David ",1)` sets the environment variable `<space>my<space>name` to `<space><space>David`. A call to `getenv("my name");` returns `NULL` indicating that the variable was not found. You must specifically query `getenv(" my name")` to retrieve the value of `"  David"`.

The environment variable names are case sensitive.

The empty string is a valid environment variable name.

Environment Variables Specific to the LE/VSE C Run-Time Library

The following LE/VSE C Run-Time-specific environment variables are supported to provide various functions. LE/VSE C Run-Time variables have the prefix _CEE_ or _EDC_. You should not use these prefixes to name your own variables.

- _EDC_BYTE_SEEK
- _EDC_COMPAT
- _EDC_RRDS_HIDE_KEY
- _EDC_STOR_INCREMENT
- _EDC_STOR_INITIAL
- _EDC_ZERO_RECLEN
- _CEE_ENVFILE

There are no default settings for the environment variables that begin with _EDC_. There are, however, default *actions* that occur if these environment variables are undefined or are set to invalid values. See the descriptions of each variable below.

The LE/VSE C Run-Time specific environment variables may be set with the `setenv()` function.

_EDC_BYTE_SEEK

The environment variable _EDC_BYTE_SEEK indicates to LE/VSE C Run-Time that, for all binary files, `ftell()` should return relative byte offsets, and `fseek()` should use relative byte offsets as input. The default behavior is for only binary files with a fixed record format to support relative byte offsets.

_EDC_BYTE_SEEK is set with the command:

```
setenv("_EDC_BYTE_SEEK","Y",1);
```

_EDC_COMPAT

The environment variable _EDC_COMPAT indicates to LE/VSE C Run-Time that it should use old functional behavior for various items in code ported from old releases of C/370. These functional items are specified by the value of the environment variable.

_EDC_COMPAT is set with the command

```
setenv("_EDC_COMPAT","x",1);
```

where *x* is an integer. LE/VSE C Run-Time converts the string *"x"* into its decimal integer equivalent, and treats this value as a bit mask to determine which functions to use in compatibility mode. The following table interprets the least significant bit as bit zero.

Bit	Function affected
0	`ungetc()`
1	`ftell()`
2	Reserved
3 through 31	Unused

For this release, calls to `fseek()` with an offset of `SEEK_CUR`, `fgetpos()`, and `fflush()` take into account characters pushed back with the `ungetc()` library function. You must set the _EDC_COMPAT environment variable for `ungetc()` if you want these functions to ignore `ungetc()` characters as they did in old C/370 code.

For `ftell()`, LE/VSE C Run-Time uses an encoding scheme that varies according to the attributes of the underlying file. You must set the _EDC_COMPAT environment variable for `ftell()` if you want to use encoded `ftell()` values generated in old C/370 code.

Here are some examples of how you can set _EDC_COMPAT:

```
setenv("_EDC_COMPAT","1",1);
```

invokes old `ungetc()` behavior.

```
setenv("_EDC_COMPAT","2",1);
```

invokes old `ftell()` behavior.

```
setenv("_EDC_COMPAT","3",1);
```

invokes both old `ungetc()` behavior and old `ftell()` behavior.

_EDC_RRDS_HIDE_KEY

The LE/VSE C Run-Time environment variable _EDC_RRDS_HIDE_KEY applies to VSAM RRDS files opened in record mode. When this environment variable is set, you can call `fread()` with a pointer to a character string, and the Relative Record Number is not appended to the beginning of the record.

The _EDC_RRDS_HIDE_KEY environment variable is set with the command

```
setenv("_EDC_RRDS_HIDE_KEY","Y",1);
```

By default, when you open a VSAM record in record mode, the `fread()` function is called with the RRDS record structure, and the record is preceded by the Relative Record Number.

_EDC_STOR_INCREMENT

This environment variable is used to set the size of increments to the internal library storage subpool. By default, when the storage subpool is filled, its size is incremented by 8K. When _EDC_STOR_INCREMENT is set, its value string is translated to its decimal integer equivalent. This integer is then the new setting of the subpool storage increment size.

The _EDC_STOR_INCREMENT value must be greater than zero, and must be a multiple of 4K. If the value is less than zero, the default setting of 8K is used. If the value is not a multiple of 4K, then it is rounded up to the next 4K interval. If _EDC_STOR_INCREMENT is set to an invalid value that must be modified internally to be divisible by 4K, this modification is not reflected in the character string that appears in the environment variable table.

Consider the case where `setenv()` is called as follows:

```
   setenv("_EDC_STOR_INCREMENT","9000",1);
```

Internally, the storage subpool increment value is set to 12288 (that is, 12K). However, the subsequent call

```
   getenv("_EDC_STOR_INCREMENT");
```

returns `"9000"`, as set by the call to `setenv()`.

_EDC_STOR_INITIAL

This environment variable is used to set the initial size of the internal library storage subpool. The default subpool storage size is 12K. When _EDC_STORE_INITIAL is set, its value string is translated to its decimal integer equivalent. This integer is then the new setting of the subpool storage increment size.

The _EDC_STORE_INITIAL value must be greater than zero, and must be a multiple of 4K. If the value is less than zero, the default setting of 12K is used. If the value is not a multiple of 4K, then it is rounded up to the next 4K interval. If _EDC_STORE_INITIAL is set to an invalid value that must be modified internally to be divisible by 4K, this modification is not reflected in the character string that appears in the environment variable table.

Consider the case where `setenv()` is called from CEEBINT as follows:

```
   setenv("_EDC_STORE_INITIAL","16000",1);
```

with the CEEBINT user exit linked to the application.

Internally, the storage subpool is initialized to 16384 (that is, 16K). However, the subsequent call

```
   getenv("_EDC_STORE_INITIAL");
```

returns "16000" as set by the `setenv()` call.

_EDC_ZERO_RECLEN

This environment variable allows processing of zero-length records in a SAM Variable file opened in either record or text mode.

_EDC_ZERO_RECLEN is set with the command

```
setenv("_EDC_ZERO_RECLEN","Y",1);
```

For details on the behavior of this environment variable, refer to Chapter 8, "Performing SAM I/O Operations," on page 47.

_CEE_ENVFILE

This environment variable enables a list of environment variables to be set from a specified file. This environment variable only takes effect when it is set through the run-time option `ENVAR` on initialization of a parent program.

When _CEE_ENVFILE is defined under these conditions, its value is taken as the name of the file to be used.

Depending on the format of the name, LE/VSE C Run-Time will open the file using different attributes:

- If the name begins with `DD:` (a `DLBL`/`TLBL`-name and/or logical unit, or a VSE/Librarian sublibrary member), the file is opened as fixed length record format with an LRECL of 80 bytes.
- If the name does *not* begin with `DD:`, the file is opened as variable length record format with an LRECL of 80 bytes and a BLKSIZE of 4000 bytes.

Note: If using any type of file other than a member of a VSE/Librarian sublibrary, care must be taken to ensure that the attributes of the file are compatible with the above.

Example

To read a member called MYVARS.Z from a VSE/Librarian sublibrary called MY.LIB, you would call your program with the `ENVAR` run-time option as follows:

```
ENVVAR("_CEE_ENVFILE=DD:MY.LIB(MYVARS.Z)")
```

Because the name begins with `DD:`, the specified file is opened as a fixed length record file. Each record consists of *`NAME`*`=`*`VALUE`*. For example, a file with the following two records:

```
_EDC_RRDS_HIDE_KEY=Y
World_Champions=Toronto_Blue_Jays
```

would set the environment variable _EDC_RRDS_HIDE_KEY to the value `Y`, and the environment variable World_Champions to the value `Toronto_Blue_Jays`.

Note: Using _CEE_ENVFILE to set environment variables through a file is not supported under CICS.

Example

The following example sets the environment variable _EDC_BYTE_SEEK. A child program is then initiated by a system call. This example illustrates that environment variables are propagated forward, but not backward.

EDCXGEV1

```
 /* EDCXGEV1
    This example shows how environment variables are propagated.
    Part 1 of 2-other file is EDCXGEV2.
  */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

   char *x;

   /*   set the environment variable _EDC_BYTE_SEEK    */
   setenv("_EDC_BYTE_SEEK","Y",1);

   /*   set x to the current value of _EDC_BYTE_SEEK   */
   x = getenv("_EDC_BYTE_SEEK");

   printf("edcxgev1 _EDC_BYTE_SEEK = %s\n",
      (x != NULL) ? x : "undefined");

   /*   call the child program   */
   system("edcxgev2");

   /*   set x to the current value of _EDC_BYTE_SEEK   */
   x = getenv("_EDC_BYTE_SEEK");

   printf("edcxgev1 _EDC_BYTE_SEEK = %s\n",
      (x != NULL) ? x : "undefined");

   return(0);
}
```

Figure 46. Environment Variables Example-Part 1

EDCXGEV2

```
 /* EDCXGEV2
    This example shows how environment variables are propagated.
    Part 2 of 2-other file is EDCXGEV1.
  */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

   char *x;

   /*   set x to the current value of _EDC_BYTE_SEEK   */
   x = getenv("_EDC_BYTE_SEEK");

   printf("edcxgev2 _EDC_BYTE_SEEK = %s\n",
      (x != NULL) ? x : "undefined");

   /*   clear the Environment Variables Table   */
   clearenv();

   /*   set x to the current value of  _EDC_BYTE_SEEK   */
   x = getenv("_EDC_BYTE_SEEK");
   printf("edcxgev2 _EDC_BYTE_SEEK = %s\n",
      (x != NULL) ? x : "undefined");

   return(0);
}
```

Figure 47. Environment Variables Example-Part 2

The preceding program produces the following output:

```
edcxgev1 _EDC_BYTE_SEEK = Y
edcxgev2 _EDC_BYTE_SEEK = Y
edcxgev2 _EDC_BYTE_SEEK = undefined
edcxgev1 _EDC_BYTE_SEEK = Y
```

Chapter 22. Using the System Programming C Facilities

Note on Documentation

Chapter 22, "Using the System Programming C Facilities" and Chapter 23, "Library Functions for the System Programming C Facilities" explain how to use the system programming C (SPC) facilities with LE/VSE. Note that this support is also available with the **VSE C Run-Time Support** of z/VSE (which is a subset of LE/VSE).

When C applications are compiled, many routines are needed to support the LE/VSE C Run-Time environment that are not included in your executable phase. These routines, which are in the C Run-Time Library, are dynamically loaded at run time. This reduces the size of the loaded phase to its practical minimum and provides for the sharing of C Run-Time library code by allowing its placement in the Shared Virtual Area.

The C Run-Time Library provides the environment and services that make LE/VSE C Run-Time ANSI-compatible. It sets up the environment, performing such services as error handling, low-level storage management and run-time option parsing, and contains the ANSI C library functions provided by LE/VSE. The library also provides an environment suitable for using Debug Tool/VSE, LE/VSE's full-screen debugging tool; for the `ctest.h`, `cdump()`, `csnap()`, and `ctrace()` library functions; and for the performance of interlanguage calls among the many languages that LE/VSE supports.

There are, however, some situations in which the library is either not desired or not available. In some supported environments there may be specific cases, such as system exit routines, where there is no vehicle for locating or loading the dynamic library or the overhead of doing so may make its use impractical.

LE/VSE makes the use of the C Run-Time Library optional. Note that without the library, most of the services it provides are not available. For example, there is no support for Debug Tool/VSE; the `ctest.h`, `cdump()`, `csnap()`, and `ctrace()` library functions; interlanguage call facilities; or the specification of options at run-time; support for the `RENT` [1] compiler option and for most functions normally provided by the C Run-Time Library, including the C file system and the mathematical functions.

System programming facilities enable you to run applications without the C Run-Time Library, and to:

- Use a subset of the C language to develop specialized applications that do not require the C Run-Time Library on the machines where the application will run.

  For more information on this type of application, see "Creating Freestanding Applications" on page 229.
- Use C as an assembler language alternative, such as for writing exit routines.

  For more information on this type of application, see "Creating System Exit Routines" on page 234.

1. Except in freestanding applications described in "Initializing a Freestanding Application" on page 230

- Develop applications featuring a persistent C environment, where a C environment is created once and used repeatedly for C function execution.

  For more information on this type of application, see "Creating and Using Persistent C Environments" on page 238.
- Develop co-routines using a two-stack model, as used in client-server style applications. In this style, the user application calls upon the applications server to perform services independently of the user and then returns to the user.

  For more information on this type of application, see "Developing Services in the Application Service Routine Environment" on page 242.

Note: Using the decimal data type and its related functions (`decabs()`, `decchk()`, and `decfix()`) without the C Run-Time Library is not supported.

Using Functions in the System Programming C Environment

The following functions are available in the SPC environment:

- The following *built-in* functions provided by the C/VSE Compiler;:

Mathematical	`abs()`, `fabs()`
Memory manipulation	`memchr()`, `memcmp()`, `memcpy()`, `memset()`, `cds()`, `cs()`
String operations	`strcat()`, `strchr()`, `strcmp()`, `strcpy()`, `strlen()`, `strrchr()`

  The built-in versions of these functions are available only if the appropriate header file (`string.h`, `math.h`, or `stdlib.h`) is included in the source file. The use of these functions is described in the *LE/VSE C Run-Time Library Reference*.
- The memory management functions, including complete support for:
  - The `malloc()` function
  - The `calloc()` function
  - The `realloc()` function
  - The `free()` function
  - The HEAP run-time option
- The `exit()` function
- The `sprintf()` function.

Additional memory management functions are available in the system programming C environment, as follows:

`__4kmalc()`	to allocate page-aligned storage
`__24malc()`	to allocate storage below the 16MB (where MB is 1048576 bytes) line in ESA systems even when `HEAP(ANYWHERE)` is specified.

Storage allocated by these functions is not part of the heap, so freeing it is your responsibility (using `free()`); it is not freed when the environment is terminated.

In this environment, low-level memory management functions and contents supervision (loading and deleting executable code) are supported by low-level routines that you can replace to support non-standard environments. This is described in "Tailoring the System Programming C Environment" on page 255.

System Programming C Facility Considerations and Restrictions

When using any system programming C environment, consider the following:

- The `fetch()` function is not supported when you are running in a system programming C environment. You can use the `EDCXLOAD` routine, as described in "EDCXLOAD" on page 259, to simulate some of the functionality of the `fetch()` function.
- The DLI parameter list established by the `#pragma runopts(PLIST(DLI))` directive is not supported in any of the system programming environments. However, this does not preclude the use of DLI within these environments, because the registers upon entry are available using the `__xregs()` function and `ctdli` is bound statically. For more information on `__xregs()`, refer to "__xregs — Get Registers on Entry" on page 265.
- Interlanguage calls to COBOL and PL/I are not supported.
- SPC is not supported under CICS.
- The only run-time options supported under the system programming C environment ares `STACK` and `HEAP`.
- Redirection of standard streams is not supported.
- The default value for `STACK` is the minimum size required to start the C program. (This default is different from the non-systems programming C environments.) If a size is specified, that actual value is used, provided it is large enough. If the value specified is smaller than the requirements for the program, the required value is used.
- Exception handling is not supported.
- The POSIX locale features and coded character set conversion routines are not available.

Creating Freestanding Applications

Freestanding applications are C modules that run without the C Run-Time library.

The initialization routine provided by SPC for building freestanding applications is EDCXSTRT. The applications can use no C Run-Time library functions.

Certain restrictions apply to freestanding applications initialized by `EDCXSTRT`. These restrictions are as follows:

- They cannot perform interlanguage calls, except with assembler language routines that preserve register 12 and use the IBM-supplied macros for entry and exit.
- The parameters received by the `main()` function (normally `argc` and `argv`) are undefined. `__xregs()` (described in "__xregs — Get Registers on Entry" on page 265) can be used to examine the parameters passed by the calling environment.
- They cannot do arithmetic using `long double` variables on pre-`XA` machines (that is, on machines that do not support the `DXR` instruction).

Creating Modules without CEESTART

In many of the environments described in this chapter, the initialization normally performed by LE/VSE is replaced by special-purpose routines that are tailored to the specific requirements of the type of application. This requires replacing the initialization routine (`CEESTART`) normally used by LE/VSE.

When you do not use the System Programming C Facilities, the compiler generates a CEESTART CSECT (control section) whenever a main() or *fetchable* function is encountered in the source file. With the NOSTART compiler option, you can suppress the generation of CEESTART for source files that contain a main() function where this is required. In a system programming C environment, you must compile using the NOSTART option. The object modules created will then be suitable for inclusion in applications that use the alternative initialization routines described in this chapter.

Including an Alternative Initialization Routine

When NOSTART is used to suppress the generation of CEESTART, an alternative initialization routine must be explicitly included in the executable phase by the user at Link Edit. Use the Linkage Editor INCLUDE and ENTRY control statements. For example, you can use the following linkage editor statements to specify EDCXSTRT as an alternative initialization routine:

```
//OPTION LINK
  PHASE phase_name
  INCLUDE EDCXSTRT
  INCLUDE EDC0XSPC
  INCLUDE main_function
  ENTRY EDCXSTRT
```

Figure 48. Specifying Alternative Initialization at Link Edit

The alternative initialization routines are in the LE/VSE library.

Another example of specifying alternative initialization is shown in Figure 50 on page 231.

Initializing a Freestanding Application

EDCXSTRT must be explicitly included in the executable phase and specified as the executable phase entry point.

Under this environment, only the following library routines are supported:

- Built-in compiler functions. For a list of these functions, refer to the table on page on page 228.
- Memory management routines, including malloc(), calloc(), realloc(), and free().
- The exit() and sprintf() functions.
- The __4kmalc() and __24malc() functions.

The value returned to the host system will be the return value from main().

The RENT compiler option is supported in this environment.

Building Freestanding Applications

The routine to support this function (EDCXSTRT) is a CEESTART replacement (described in "Creating Modules without CEESTART" on page 229) in your module. Therefore, it must be explicitly included ahead of your module at link edit.

A simple freestanding routine is shown in Figure 49 on page 231.

EDCJL084

```
int main(void)
{
  int   x = 1;

  x = x + 10;
  return(x);
}
```

Figure 49. Sample Freestanding Routine

This routine is compiled with the NOSTART option and link edited using control statements shown in Figure 50.

```
  ACTION NOMAP
  PHASE CSPC1,*
  INCLUDE EDCXSTRT
  INCLUDE EDC0XSPC
  INCLUDE
.
.
.
(Object deck)
.
.
.
  ENTRY EDCXSTRT
```

Figure 50. Link Edit Control Statements Used to Build a Freestanding Routine

Note: EDC0XSPC is an include book which includes the modules required for most SPC programs.

Figure 51 on page 232 shows how to compile and link a freestanding program.

EDCJN018

```
* $$ JOB JNM=CSPC1,LDEST=(*,uid),PDEST=(*,uid),CLASS=Z
// JOB CSPC1
// LIBDEF *,SEARCH=(MY.LIB,PRD2.DBASE,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=MY.LIB
// OPTION CATAL,NODUMP
  ACTION NOMAP
  PHASE CSPC1,*
  INCLUDE EDCXSTRT
  INCLUDE EDC0XSPC
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='NOSTART'
int main(void)
{
  int   x = 1;

  x = x + 10;
  return(x);
}
/*
// LIBDEF *,SEARCH=PRD2.SCEEBASE
// EXEC  PGM=LNKEDT
/&
* $$ EOJ
```

Figure 51. Compile and Link

Special Considerations for Reentrant Phases

A simple freestanding routine is shown in Figure 52. To develop a reentrant phase, this routine must be compiled with both the `RENT` (because the phase contains writable static at **1**) and `NOSTART` (because this is a system programming environment) compiler options. This routine uses the `exit()` function (**2**), which is normally part of the C Run-Time library. Like `sprintf()`, it is available to freestanding routines without requiring the LE/VSE run-time library.

EDCJL086

```
main()
{
  static  int i[5]={0,1,2,3,4};   1
  exit(320+i[1]);   2
}
```

Figure 52. Sample Reentrant Freestanding Routine

The JCL required to build and execute this routine is shown in Figure 53 on page 233.

EDCJN019

```
* $$ JOB JNM=CSPCRENT,LDEST=(*,uid),PDEST=(*,uid),CLASS=Z
// JOB CSPCRENT
// LIBDEF *,SEARCH=(MY.LIB,PRD2.DBASE,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=MY.LIB
// OPTION CATAL,NODUMP
  ACTION NOMAP
  PHASE CSPCRENT,*
  INCLUDE EDCXSTRT    1
  INCLUDE EDCXEXIT    2
  INCLUDE EDCRCINT    3
  INCLUDE EDCKSMSK
  INCLUDE EDCXFREE
  INCLUDE EDCXGET
  INCLUDE EDCX4KGT
  INCLUDE EDCXABND
  INCLUDE EDCXBTCA
  INCLUDE EDCXCEE
  INCLUDE EDCXHEAP
  INCLUDE EDCXHFRE
  INCLUDE EDCXHGET
  INCLUDE EDCXOBJP
  INCLUDE EDCXTOVF
  INCLUDE CEESG003
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='/NOSTART RENT'
main()
{
  static  int i[5]={0,1,2,3,4};
  exit(320+i[1]);
}
/*
// EXEC  PGM=EDCPRLK   4

// EXEC  PGM=LNKEDT    5
// LIBDEF *,SEARCH=(MY.LIB)   6
// EXEC  PGM=CSPCRENT
// EXEC  PGM=LISTLOG
/&
* $$ EOJ
```

Figure 53. Building and Running a Reentrant Freestanding Routine

The following notes refer to Figure 53.

1 The alternative initialization routine (EDCXSTRT in this example) must be included explicitly in the module. If this is not the first CSECT in the module, it must be explicitly named as the module entry point.

2 EDCXEXIT must be explicitly included if the `exit()` function is used in the application.

3 The routine EDCRCINT must be explicitly included in the module if the RENT compiler option is used. No error will be detected at load time if this routine is not explicitly included. At execution time, abend 2106, reason code 7205, will result if EDCRCINT is required but not included.

4 The LE/VSE prelinker must be used for modules compiled with the RENT compiler option.

5 The output from the prelinker is made available to the linkage editor.

6 Because the resultant phase is freestanding, the run-time libraries are not required. The LIBDEF statement removes these libraries.

Table 36 lists the parts used for freestanding applications and their function and location.

Table 36. Parts Used for Freestanding Applications

Part Name	Function	Inclusion in Executable Phase		Location
			Notes	
EDCXSTRT	This module is the mainline.	1	This CSECT must be the module entry point.	Member of PRD2.SCEEBASE
EDCXSPRT	System programming version of sprintf().	2		Member of PRD2.SCEEBASE
EDCXEXIT	System programming version of exit().	2		Member of PRD2.SCEEBASE
EDCXMEM	System programming version of malloc(), calloc(), realloc(), free(), __4kmalc(), and __24malc().	2		Member of PRD2.SCEEBASE
EDCRCINT	This must be included if the compiler option RENT is to be used.	2		Member of PRD2.SCEEBASE

Notes:

1 This module must be explicitly included in the Executable Phase using the VSE INCLUDE link edit control statement.

2 This module must be explicitly included if you want to use the system programming version of the function.

Creating System Exit Routines

LE/VSE C Run-Time allows the creation of routines that have no environmental requirements on entry *except*:
- Register 13 must point to a 72-byte save area
- Register 14 must contain the return address
- Register 15 must contain the entry address.

There is no requirement on the name of the entry point (that is, it does not have to be main()), so several different entry points, with names specified by the calling environment, can be combined in the same executable phase.

C routines that do not require the LE/VSE environment should specify:

```
#pragma environment( function-name)
```

This pragma causes the compiler to generate a different prolog for the specified function. The prolog contains the instructions at the beginning of the routine that perform the housekeeping necessary for the function to run, including allocation of the function's automatic storage. This prolog will set up a C environment sufficient for both the function in which it is specified and any function that may be called. Called functions should not specify this pragma, unless they are called elsewhere without a C environment present.

The RENT compiler option is not supported in this environment; if you require reentrant system exit routines, the routine must be naturally reentrant. See *LE/VSE C Run-Time Library Reference* for more information about reentrancy.

System exit routines can be linked with their callers or dynamically loaded and invoked.

Building System Exit Routines

The PRD2.SCEEBASE object library must be available at link-edit time. You should include EDCXENV first or explicitly name the entry point with an ENTRY statement.

An Example of a System Exit

The following C program shown in Figure 54 is an example of a system exit routine called by assembler program CSPCEX1 shown in Figure 55 on page 236. Figure 56 on page 236 shows an example of the JCL required to compile, link, and execute the sample exit.

EDCJN020

```
#pragma environment(cspcex2)   1

#include <spc.h>
#include <string.h>

int cspcex2()
{
  void  **parm;
  short   parmlen;
  char    arg[20];

  parm = (void *)__xregs(1);

  memset(arg, '\0', sizeof(arg));
  memcpy(arg, (char *)*parm, 4);

  if (strcmp(arg, "fred") == 0)
    exit(0);
  else
    exit(2);
}
```

Figure 54. System Exit Example—C

The following note refers to Figure 54.

1 The #pragma environment directive sets up an entry point CSPCEX2.

EDCJN021

```
CSPCEX1  CSECT
         USING *,15
         STM   14,12,12(13)
*
         ST    13,SAVEAREA+4         Backchain save area
         LA    13,SAVEAREA           Activate mine
         DROP  15
*
         BALR  3,0
         USING *,3
*
         WTO   'In CSPCEX1'
         CALL  CSPCEX2,STRING
         LTR   15,15
         BZ    OK
         WTO   'Returned to CSPCEX1 with RC ¬= 0'
         B     EXIT
OK       WTO   'Returned to CSPCEX1 with RC = 0'
*
EXIT     L     13,4(,13)             Address of caller's save area
         LM    14,12,12(13)
         SR    15,15
         BR    14
*
         DS    0F
SAVEAREA DS    CL72
STRING   DC    C'fred'
         END
```

Figure 55. System Exit Example—Assembler

EDCJN022

```
* $$ JOB JNM=CSPCEX1,LDEST=(*,uid),PDEST=(*,uid),CLASS=5
// JOB CSPCEX1
// LIBDEF *,SEARCH=(MY.LIB,PRD2.DBASE,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=MY.LIB
// OPTION DECK,NODUMP
// DLBL IJSYSPH,'PUNCH.OUTPUT',0,SD
// EXTENT SYSPCH,SYSWK2,1,0,10000,100
ASSGN SYSPCH,DISK,VOL=SYSWK2,SHR
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='/NAME(CSPCEX2), NOSTART'
.
.
.
(C source code from
Figure 54 on page 235)
.
.
.
/*
CLOSE SYSPCH,FED
```

Figure 56. System Exit Example—JCL (Part 1 of 2)

```
// DLBL IJSYSIN,'PUNCH.OUTPUT',0,SD
// EXTENT SYSIPT
ASSGN SYSIPT,DISK,VOL=SYSWK2,SHR
// EXEC LIBR,PARM='ACCESS SUBLIB=MY.LIB'
CLOSE SYSIPT,SYSRDR
/*
// DLBL IJSYSPH,'PUNCH.OUTPUT',0,SD
// EXTENT SYSPCH,SYSWK2,1,0,10000,100
ASSGN SYSPCH,DISK,VOL=SYSWK2,SHR
// EXEC ASMA90,SIZE=512K,PARM='EXIT(LIBEXIT(EDECKXIT))'
         PUNCH 'CATALOG CSPCEX1.OBJ REPLACE=YES'
.
.
.
(Assembler source code from
Figure 55 on page 236)
.
.
.
/*

CLOSE SYSPCH,FED
// DLBL IJSYSIN,'PUNCH.OUTPUT',0,SD
// EXTENT SYSIPT
ASSGN SYSIPT,DISK,VOL=SYSWK2,SHR
// EXEC LIBR,PARM='ACCESS SUBLIB=MY.LIB'
CLOSE SYSIPT,SYSRDR
/*
// OPTION CATAL,NODECK,NODUMP
// LIBDEF *,SEARCH=(MY.LIB,PRD2.SCEEBASE)
/*

// OPTION CATAL,NODECK,NODUMP
  ACTION NOMAP
  PHASE CSPCEX1,*
  INCLUDE CSPCEX1
  INCLUDE EDCXENV
  INCLUDE EDCXREGS
  INCLUDE EDCXEXIT
  INCLUDE EDC0XSPC
// EXEC  PGM=LNKEDT
// LIBDEF *,SEARCH=(MY.LIB)
// EXEC  PGM=CSPCEX1
// EXEC  PGM=LISTLOG
/&
* $$ EOJ
```

Figure 56. System Exit Example—JCL (Part 2 of 2)

Table 37 lists the parts used by exit routines, and their function and location.

Table 37. Parts Used by Exit Routines

Part Name	Function	Inclusion in Executable Phase		Location
			Notes	
`EDCXENV`	Extended prolog code for exits that do not require the library.	1		Member of `PRD2.SCEEBASE`
`EDCXSPRT`	System programming version of `sprintf()`	2		Member of `PRD2.SCEEBASE`
`EDCXEXIT`	System programming version of `exit()`	2		Member of `PRD2.SCEEBASE`

Table 37. Parts Used by Exit Routines (continued)

Part Name	Function	Inclusion in Executable Phase		Location
			Notes	
EDCXMEM	System programming version of malloc(), calloc(), realloc(), free(), __4kmalc(), and __24malc().	2		Member of PRD2.SCEEBASE

Notes:

1 This module must be explicitly included in the Executable Phase using the VSE INCLUDE link-edit control statement.

2 This module must be explicitly included if you want to use the system programming version of the function.

Creating and Using Persistent C Environments

Three routines are available to create and use a persistent C environment. These routines are used by an assembler language application that needs a C environment available to support the C functions (not including main()) that it calls.

An initialization routine, EDCXHOTC, is called to create a C environment. This call returns a *handle* that can be used (through EDCXHOTU) to call C subroutines. The environment persists until it is explicitly terminated by calling EDCXHOTT.

The routines are:

EDCXHOTC	Sets up a persistent C environment (no run-time library)
EDCXHOTU	Runs a function in a persistent C environment
EDCXHOTT	Terminates a persistent C environment

The functions that act as entry points for these routines are __xhotc(), __xhotu(), and __xhott(), respectively. For more information on these functions, refer to Chapter 23, "Library Functions for the System Programming C Facilities," on page 263.

The RENT compiler option is not supported in the persistent environment described in this chapter.

Exception handling is not supported in persistent C environments.

Building Applications That Use Persistent C Environments

There are no special restrictions for building applications that use persistent C environments.

An Example of Persistent C Environments

The assembler routine shown in Figure 58 on page 239 illustrates the use of this feature to call a C function shown in Figure 57 on page 239.

EDCJL089

```
#pragma linkage(crtn,OS)   1
#include <string.h>
#include <stdio.h>
#define INSIZE 300           /* the maximum length we'll tolerate */

void crtn(int p1,char *p2,char *outstring) {
     char hold[2+INSIZE];
     char *endptr;
     int  i;

     endptr=memchr(p2,'@',INSIZE);
     if (NULL==endptr)
          i=INSIZE;      /* no ender? use max */
     else
          i=endptr-p2;        /* length of stuff before it */

     memcpy(hold,p2,i);       /* copy formatting string */
     hold[i++]='\n';          /* add a new-line.. */
     hold[i]='\0';            /* ..and a null terminator */

     sprintf(outstring,hold,p1);        /* print it out */

     return;                  /* and return */
}
```

Figure 57. Example Function Used in a Persistent C Environment

This C function accepts two parameters: an integer and a `printf()`-style formatting string. The formatting string has a maximum length of 300 bytes; it is terminated by an @ if shorter. This routine *must* use `OS` linkage (Figure 57 **1**). The routine scans the formatting string for the terminator, copies it to a local work area, adds a trailing newline and `NULL` character, and prints the integer according to the formatting string.

The structure of the assembler caller is shown in Figure 58.

EDCJL090

```
ENVA     CSECT
ENVA     AMODE  ANY
ENVA     RMODE  ANY
         STM    R14,R12,12(R13)    1
         LR     R3,R15
         USING  ENVA,R3
         LA     R0,DSALEN
         GETVIS LENGTH=(R0)
         LTR    R15,R15
         BZ     GOTSTOR
         WTO    'ENVA - GETVIS failed'
         B      EXIT
```

Figure 58. Using a Persistent C Environment (Part 1 of 3)

```
GOTSTOR   ST     R13,4(,R1)
          LR     R13,R1
          USING  DSA,R13
          WTO    'In ENVA'
          LA     R4,HANDLE   2
          LA     R5,STKSIZE
          LA     R6,STKLOC
          STM    R4,R6,PARMLIST
          OI     PARMLIST+8,X'80'
          WTO    'Calling EDCXHOTC'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTC)
          BALR   R14,R15
          LA     R9,10    3
LOOP      DS     0H
          ST     R9,LOOPCTR    4
          LA     R4,HANDLE
          LA     R5,USEFN
          LA     R6,LOOPCTR
          LA     R7,FMTSTR1
          LA     R8,OUTSTRING
          STM    R4,R8,PARMLIST
          OI     PARMLIST+16,X'80'
          MVC    OUTSTRING,CLEAR
          WTO    'Calling EDCXHOTU'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTU)
          BALR   R14,R15
          MVC    WTOLIST+4(L'OUTSTRING),OUTSTRING
          WTO    MF=(E,WTOLIST)
          LA     R7,FMTSTR2    5
          LA     R8,OUTSTRING
          STM    R4,R8,PARMLIST
          OI     PARMLIST+16,X'80'
          MVC    OUTSTRING,CLEAR
          WTO    'Calling EDCXHOTU'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTU)
          BALR   R14,R15
          MVC    WTOLIST+4(L'OUTSTRING),OUTSTRING
          WTO    MF=(E,WTOLIST)
          BCT    R9,LOOP
          ST     R4,PARMLIST   6
          OI     PARMLIST,X'80'
          WTO    'Calling EDCXHOTT'
          LA     R1,PARMLIST
          L      R15,=V(EDCXHOTT)
          BALR   R14,R15
          LR     R1,R13    7
          L      R13,4(0,R13)
          LA     R0,DSALEN
          FREEVIS  ADDRESS=(R1),LENGTH=(R0)
```

Figure 58. Using a Persistent C Environment (Part 2 of 3)

```
EXIT     WTO   'Leaving ENVA'
         LM    R14,R12,12(R13)
         SR    R15,R15
         BR    R14
USEFN    DC    V(CRTN)
STKSIZE  DC    A(4096)
STKLOC   DC    A(1)
FMTSTR1  DC    C'1st value of loopctr is %i@'
FMTSTR2  DC    C'value on 2nd call is %i@'
CLEAR    DC    C' '
OUTSTRING DS   CL70
         LTORG
WTOLIST  WTO  ('                                                  X
                         '),MF=L
DSA      DSECT ,           The dynamic storage area
SAVEAREA DS    18A         The save area
PARMLIST DS    5A
HANDLE   DC    A(0)
LOOPCTR  DC    A(1)
DSALEN   EQU   *-DSA
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   ENVA
```

Figure 58. Using a Persistent C Environment (Part 3 of 3)

The following notes refer to Figure 58 on page 239.

1 This routine is entered with standard linkage conventions. It saves the registers in the save area pointed to by register 13, acquires a dynamic storage area for its own use, and chains the save areas together.

2 A C environment is created by calling `EDCXHOTC`. The parameter list for this call is the address of the handle (for the persistent C environment created), the address of a word containing the initial stack size, and the address of a word containing the initial stack location (0 for below the 16MB line and 1 for above). This parameter list uses the normal `OS` linkage format.

3 The routine loops 10 times calling the C function `crtn` twice each time through the loop.

4 The parameter list for the first call is the address of the handle, the address of a word pointing to the function, and the parameters to be received by the function. `EDCXHOTU` is called. This causes the specified C function, `crtn()` to be given control with register 1 pointing to the remaining parameters, `LOOPCTR` and `FMTSTR1`.

5 The C function is called again, this time with `FMTSTR2` as the second parameter.

6 When the loop ends, EDCXHOTT is called to terminate the environment created at **2** .

7 The routine terminates by freeing its dynamic storage area and returning to its caller.

Table 38 lists the parts used by persistent environments, and their function and location.

Table 38. Parts Used by Persistent Environments

Part Name	Function	Inclusion in Executable Phase		Location
			Notes	
EDCXHOTC	This module is called to set up a C environment with no Library.			Member of PRD2.SCEEBASE
EDCXHOTT	This module is called to terminate a C environment set up by EDCXHOTC.			Member of PRD2.SCEEBASE
EDCXHOTU	This module is called to use a C environment set up by EDCXHOTC.			Member of PRD2.SCEEBASE
EDCXSPRT	System programming version of `sprintf()`.	1		Member of PRD2.SCEEBASE
EDCXEXIT	System programming version of `exit()`.	1		Member of PRD2.SCEEBASE
EDCXMEM	System programming version of `malloc()`, `calloc()`, `realloc()`, `free()`, `__4kmalc()`, and `__24malc()`.	1		Member of PRD2.SCEEBASE

Notes:

1 This module must be explicitly included if you want to use the system programming version of the function.

Developing Services in the Application Service Routine Environment

The purpose of an application service routine environment is to allow the development, using LE/VSE, of services that can be developed, tested, and packaged independently of their intended users. You can:
- Isolate the service code from its user
- Specify and enforce a clearly defined Application Programming Interface (API) between the user (another application program) and the service routine
- Share server code among more than one (perhaps different) user applications simultaneously
- Enhance or maintain the service routine code with no disruption to its various user applications.

In this environment, a service application is developed as a C `main()` function together with any functions it may call, and packaged as a complete load module. This load module, if it is reentrant, can be freely installed in the SVA and shared by all of its users.

To provide the service to a user application, the developer of the service must offer small assembler language stub routines that are link edited with the user code. These stub routines use services provided by the System Programming Facilities to load or locate the server code and pass messages to it for execution. Examples of these stub routines are shown in “Constructing User-Server Stub Routines” on page 255.

Using Application Service Routine Control Flow

In this section examples are based on a service routine that manages a storage queue. This server might be used by languages that do not support dynamic memory allocation, or by applications that do not want to concern themselves with the management of such data structures. The operations supported by this service routine are:

- Initialize
- Terminate
- Add an element to the head of the queue (last in, first out)
- Add an element to the tail of the queue (first in, first out)
- Get the element at the head of the queue

The user routine shown in the example is written in C.

Service Routine User Perspective

A conversation is initiated when a user routine calls a startup routine supplied by the author of the service to establish a connection between the user and the server. This routine returns a *handle* to the user that represents the server environment. User routines may establish connections with many different services or many times with the same server as long as the needed resources, principally memory, are available in the system. Each connection has a different handle, and it is the user routine's responsibility to keep track of them.

Once the user has initialized the server, it uses other server-supplied stub routines to send requests (messages) to the server for action. One of the parameters to this routine will be the handle returned by the initialize call. These request stubs would typically return a feedback code to indicate success or failure as well as any other information requested. The server defines the parameter list to be passed and the feedback codes to be given to the user.

When the user is finished with the server, it calls yet another stub routine to terminate the server.

This structure is illustrated in a sample user routine shown in Figure 59 on page 244:

EDCJL091:

```
#pragma linkage(qmginit,OS)
#pragma linkage(qmglifo,OS)
#pragma linkage(qmgfifo,OS)
#pragma linkage(qmgget,OS)
#pragma linkage(qmgterm,OS)
 /*  Example User-Service Routine application */
void main () {

int handle;    1
int feedback, chlen, i;
char ch[100];

printf("Initializing the server\n");
qmginit(&handle);    2
printf("Server initialized. Handle is %x\n", handle);

 /*  Feed some strings to the server */    3
printf("Feeding first string to server\n");
qmglifo(handle, &feedback, 17, "2 Sample string 1");
printf("Return code is %d\n", feedback);
printf("Feeding second string to server\n");
qmglifo(handle, &feedback, 23, "1 Another sample string");
printf("Return code is %d\n", feedback);
printf("Feeding third string to server\n");
qmgfifo(handle, &feedback, 20, "3 Yet another string");
printf("Return code is %d\n", feedback);

 /*  Get the strings back, print out length and value */
printf("Getting the strings back from server\n");
for (i=0; i<3; i++){    4
 qmgget(handle, &feedback, &chlen, ch);
 printf("String is \"%.*s\".Return code is %d\n",chlen,ch,feedback);    5
}

 /*  Terminate the server */
printf("Terminating the server\n");
qmgterm(handle);    6
}
```

Figure 59. Example User Routine

The following notes refer to Figure 59.

1 The user routine sets up a variable that will be used to hold the handle returned by the server. The form taken by this handle is up to the supplier of the service, but a fullword (4 bytes) should be regarded as typical.

2 The user routine calls the initialize routine to set up the connection between the user routine and the server.

3 The user routine adds three strings to the queue. In this example, the first character of the string indicates the order in which the user expects to retrieve the strings.

4 The user enters a loop in which the strings are retrieved from the queue.

5 The user routine prints out the strings passed back by the call to the server. If there is no string remaining in the queue a null string (zero length) is returned.

6 Before ending, the user routine closes down the server.

This routine is linked normally with the server-supplied stub routines (described in "Constructing User-Server Stub Routines" on page 255).

Service Routine Perspective

A service routine is a complete, stand alone module that runs in its own C environment. Its environment is created on demand by user application routines that call it using stub routines supplied by the server. When this happens, the server code enters at its `main()` entry point and, typically, goes into a loop that contains a function call to get the next *to-do*. One possible to-do is *terminate*; when this command is received the server should `exit()` or `return` from its `main()` function. The environment created when the server was started terminates and all resources held by the server are freed (except storage acquired by `_ _24malc()` or `_ _4kmalc()`, as described in "`__24malc()` — Allocate Storage below 16MB Line" on page 266 and "`__4kmalc()` — Allocate Page-Aligned Storage" on page 267.

This structure is illustrated in a sample user routine shown in Figure 60:

EDCJL092:

```
#include <spc.h>    1
#include <stdlib.h>
#include <string.h>

#define LIFO 1    2
#define FIFO 2
#define GET  3
#define TERM -1

int main(void) {    3

  int retcode=0;

  /* data structures to manage the queue */
  struct queue_entry {    4
     struct queue_entry  *next;
     int                  length;
     char                 val[1];
  };

  struct queue_entry     *head;
  struct queue_entry     *tail;
  struct {    5
     int                 code;
     union info          *plist;
  }  *req;
```

Figure 60. Example Application Service Routine (Part 1 of 3)

```
union info {    6
    struct {
       int                *length;
       char               *string;
    }                  lifo;
    struct {
       int                *length;
       char               *string;
    }                  fifo;
    struct {
       int                *length;
       char               *string;
    }                  get;
};
/* initialize the queue pointers */
head = NULL;    7
tail = NULL;
/* the main processing loop goes on until a termination signal
   is sent */

for(;;) {    8
    union info           *info;
    int                   length;
    char                 *string;
    struct queue_entry   *ent;


    /* get a message from the user routine */
    req=__xsrvc(retcode);    9    18
    info = req->plist;    10

    switch(req->code) {    11

        case LIFO: {    12
             length=*(*info).lifo.length;
             string= (*info).lifo.string;
             ent = malloc(sizeof *ent - 1 + length);    13
             memcpy((*ent).val,string,length);
             __xsacc(0);    14
             (*ent).length=length;
             (*ent).next=head;
             head=ent;
             if (NULL==tail) tail=ent;
             break;
        }

        case FIFO: {    15
             length=*(*info).fifo.length;
             string= (*info).fifo.string;
             ent = malloc(sizeof *ent - 1 + length);
             memcpy((*ent).val,string,length);
             __xsacc(0);
             (*ent).length=length;
             (*ent).next=NULL;
             if (NULL==head) head=ent;
             else (*tail).next=ent;
             tail=ent;
             break;
        }
```

Figure 60. Example Application Service Routine (Part 2 of 3)

```
          case GET:  {    15
               if (NULL==head) {
                   *(*info).get.length=0;
                   break;
               }
               length = (*head).length;
               string = (*info).get.string;
               memcpy(string,(*head).val,length);
               *(*info).get.length=length;
               __xsacc(0);
               ent=head;
               head=(*ent).next;
               free(ent);
               if (NULL==head) tail=NULL;
               break;
          }
          case TERM:    16
               return 0;
          default:
               __xsacc(666);    17

      }
   }
}
```

Figure 60. Example Application Service Routine (Part 3 of 3)

The following notes refer to Figure 60 on page 245.

1 The server routine should include the appropriate header files. `spc.h` contains the function prototypes for the routines that are used to maintain the conversation between the server routine and the user routine. `string.h` is *required* if string or memory functions are used in the code; this header file contains the directives necessary to use these built-in functions.

2 These are the *command codes* of the requests that can be sent to this server.

3 The server begins with a `main()` function. This function gets control when the user calls `QMGINIT`.

4 This server manages an in-storage queue of unstructured elements. It does this by maintaining a linked list of elements. The structure `queue_entry` contains an individual entry; `head` and `tail` point to the first and last entries in the queue.

5 Requests come to the server in the form of a pointer to a structure containing a command code (in this case, one of `LIFO`, `FIFO`, `GET`, or `TERM`) and a pointer to a parameter list associated with the command code. The parameter list is what follows `HANDLE` and `FEEDBACK` in the calls to `QMGLIFO`, `QMGFIFO`, and `QMGGET`. Like the command codes, the structure of this parameter list is established in concert with the stub routines.

6 In this example, all the commands have exactly the same format. This may not generally be the case, so a union of the various parameter list formats is appropriate. Then the interface can be expanded without disrupting existing code.

7 Before accepting commands, required initialization is performed.

8 This server is structured as an endless loop. This loop terminates when a terminate message sends control to a `return` statement at **17**.

9 At this point, the server is ready for work. The call to `_ _xsrvc` causes the user routine to resume execution at the place it left off when it last called the server. The value passed as the parameter is made available to the stub routines for use as a feedback code. This function will not return until the user application sends a request (using one of the stub routines, in this example `QMGLIFO`, `QMGFIFO`, `QMGGET`, or `QMGTERM`).

10 Extract the parameters from the structure pointed to by the call to `_ _xsrvc`.

11 Examine the request code sent by the user application.

12 The `LIFO` request code is handled here.

13 These library functions (and many others, the complete list is given in "Using Functions in the System Programming C Environment" on page 228) are normally available in this environment even though the C Run-Time Library is not available at run time.

The amount of storage allocated is the size of the queue entry (defined at **4**) minus 1 (because the definition of the entry allowed for 1 character of value) plus the length actually required for the value.

14 This function should be used to indicate that the server has completed its use of any data structures (parameters and data areas pointed to by the parameters) belonging to the user application. The value passed to this function or the value passed by the next call to `__xsrvc()` (whichever is greater in magnitude) will be passed to the stub routine for use as a feedback code.

15 The handling of `FIFO` and `GET` is similar.

16 When a terminate request is received, the server `return`s. This terminates the loop (at **8**) and the environment set up when the server was first called.

17 If the command code is not recognized the server acknowledges the request and sets a return code that can be analyzed by the stub routine or the user application.

18 The server returns to the request for another *to-do*. The value passed as a parameter here or the last value passed to `__xsacc()`, whichever has the greater magnitude, is passed to the stub routine for use as a feedback code.

The server is built as a freestanding C application as described in "Creating Freestanding Applications" on page 229. This routine *must* be built with `EDCXSTRT` as shown in Figure 61 on page 249.

```
* $$ JOB JNM=QMGSERV,LDEST=(*,uid),PDEST=(*,uid),CLASS=Z
// JOB QMGSERV
// LIBDEF *,SEARCH=PRD2.SCEEBASE
// LIBDEF PHASE,CATALOG=MY.LIB
// OPTION CATAL,NODUMP
  ACTION NOMAP
  PHASE QMGSERV,*
  INCLUDE EDCXSTRT
  INCLUDE EDCXMEM
  INCLUDE EDCXSACC
  INCLUDE EDCXSRVC
  INCLUDE EDC0XSPC
  INCLUDE
.
.
.
(Object deck)
.
.
.
/*
// EXEC  PGM=LNKEDT
/&
* $$ EOJ
```

Figure 61. Linking and Installing the Application Service Routine

Understanding the Stub Perspective

The stub routines provide the link between the user application and the application service module. They are responsible for:
- Locating or loading the server code
- Providing the Application Programming Interface (API) seen by the user.

Many choices are available in the design of the API and how single calls in the user are mapped. For example, the initialize call could accept parameters governing the behavior of the session being established and pass them to the server as commands once the server has been initialized. In the example the interactions are straight forward, the initialize only starts up the server, and the message calls send single messages, untouched and unexamined, to the server.

There are two kinds of stubs: the initialization stub and the message stubs. Termination is a special case of a message stub. These stubs are most appropriately written in assembler so that they can run in any language environment with minimal performance cost.

The initialization stub is responsible for loading and calling the server. It can use the low-level storage management and contents supervision routines supplied in PRD2.SCEEBASE. These routines are described in “Tailoring the System Programming C Environment” on page 255. The structure of an initialization stub is shown in Figure 62 on page 250:

EDCJL093

```
QMGINIT   TITLE  'SERVER      supplied stub to initialize'
QMGINIT   CSECT  ,
          STM    R14,R12,12(R13)    1
          LR     R3,R15
          USING  QMGINIT,R3
          USING  INPARMS,R1    2
          L      R6,HANDLE@
          LA     R6,0(,R6)
          DROP   R1
          LA     R0,WALEN     length of work area, below the line    3
          L      R15,=V(EDCXGET)    GETVIS some storage
          BALR   R14,R15
          USING  WA,R1
          ST     R13,SA+4
          LR     R13,R1
          USING  WA,R13       This is now our DSA
          LA     R1,NAME    4
          L      R15,=V(EDCXLOAD)
          BALR   R14,R15      Load the server
          ST     R1,PLIST    5
          MVC    PLIST+4(12),PLISTINI
          L      R15,=V(EDCXSRVI)
          LA     R1,PLIST
          BALR   R14,R15
          MVC    0(4,R15),=CL4'QMqm'      eye-catcher    6
          ST     R13,4(,R15)    7
          ST     R15,0(,R6)   Save handle in users parameter    8
          L      R13,4(,R13)    9
          LM     R14,R12,12(R13)
          SR     R15,R15
          BR     R14
PLISTINI  DS     0D
          DC     A(0),V(EDCXGET,EDCXFREE)
NAME      DC     CL8'QMGSERV'
INPARMS   DSECT
HANDLE@   DS     F
WA        DSECT
SA        DS     18F
PLIST     DS     4F
WALEN     EQU    *-WA
*
```

Figure 62. Example Server Initialization Stub (Part 1 of 2)

```
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END
```

Figure 62. Example Server Initialization Stub (Part 2 of 2)

The following notes refer to Figure 62 on page 250.

1 Stub routines are presumed to have a save area available at the location pointed to by register 13.

2 The parameter list passed to stub routines is OS linkage; that is, register 1 points to a list of addresses. In this example, the initialization stub receives only one parameter, the handle, that gets the address of a control block representing the environment.

3 For efficiency, this routine gets a work area that will be used by *all* the stub routines. The low level storage management routine EDCXGET, (described in “EDCXGET” on page 256) is available for this purpose. This area will be the DSA for this and all other stub routines. It begins with an 18-word save area for use by routines called by this stub. It will be freed by the “terminate” stub.

4 Once a save area is available, EDCXLOAD (described in “EDCXLOAD” on page 259) is called to load the server.

5 EDCXSRVI is called to initialize the server. When control is returned from this call, the server has built a complete environment and has asked for something to do. It is waiting at **9** in Figure 60 on page 245.

6 The value returned by EDCXSRVI is the address of a control block that is used to manage the interface between the user application and the service application module. The first 3 words (12 bytes) of this control block are reserved for the exclusive use of the stub routines. The fields following the first 3 words may not be used by either the stub routines or the user, nor may their values be altered. In this example, an *eye-catcher* (often useful for debugging) is moved into the first word.

7 The address of the work area acquired for dynamic storage requirements is moved into the second word. The address of this control block is stored in the user’s handle.

8 The address of the control block from EDCXSRVI is placed in the user routine’s handle. The user routine has no knowledge of the contents or format of this field; it is simply a *token* that is passed to other stub routines to manage the conversation between the user and the service routine.

9 Having initialized the server, the stub returns to the user at **2** in Figure 59 on page 244.

Message stubs are responsible for passing requests from the user application to the service application. Like the initialization stub, they are free to use the low-level storage management and contents supervision routines supplied with the C Run-Time Library. An example message stub is shown in Figure 63.

EDCJL094

```
QMGLIFO   TITLE  'SERVER      supplied stub for feeding strings LIFO'
QMGLIFO   CSECT
          STM    R14,R12,12(R13)    1
           LR     R3,R15
          USING  QMGLIFO,R3
          LR     R5,R1
          USING  INPARMS,R5
          L      R6,HANDLE@
          L      R6,0(,R6)     Point to the handle    2
          L      R1,4(,R6)     Point to work area got by QMGINIT    3
          USING  WA,R1
          ST     R13,SA+4      Keep savearea passed into us
          LR     R13,R1        WA is new savearea
          USING  WA,R13
          LA     R7,LIFO    4
          LA     R8,INPARMS+8  User parms start at 3rd
          STM    R6,R8,PLIST   handle, LIFO, Other parms
          LA     R1,PLIST
          L      R15,=V(EDCXSRVN)    5
          BALR   R14,R15
          L      R1,FEEDBK@    6
          ST     R15,0(,R1)
          L      R13,4(,R13)    7
          L      R14,12(R13)
          LM     R0,R12,20(R13)
          BR     R14
INPARMS   DSECT
HANDLE@   DS     F
FEEDBK@   DS     F
LENGTH@   DS     F
STRING@   DS     F
WA        DSECT
SA        DS     18F
PLIST     DS     4F
WALEN     EQU    *-WA
LIFO      EQU    1
FIFO      EQU    2
GET       EQU    3
TERM      EQU    -1
          REGEQU
          END
```

Figure 63. Example Server Message Stub

The following notes refer to Figure 63.

1 Like the initialize stub, the message stubs expect a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2 The *handle* contains the value that was placed there by the initialization stub at 8 in Figure 62 on page 250. This is the address of the control block that is used to manage the interface between the user application and the server.

3 Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at 7 in Figure 62 on page 250. The save area back chain field is set according to usual conventions.

4 A parameter list consisting of the handle (as returned by EDCXSRVI at 5 in Figure 62 on page 250 in the initialization stub), code for LIFO, and the address of the remaining parameters.

5 Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at 9 in Figure 60 on page 245 in the server. The server has control until it asks for the next *to-do*, in this example at 9 in Figure 60 on page 245, again.

6 The value passed to __xsrvc() appears as the return code from EDCXSRVN. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*

7 Control is returned to the user in the normal way.

This routine uses functions supplied in to load or locate the server code and initialize its environment.

The routines in the following section are used to create and use a persistent C environment for a server co-routine, written using C and EDCXSTRT, and callable by a user application written in *any* language.

An initialization routine, EDCXSRVI, is called to start up a *server*. Control returns from the initialization call with the server code started and waiting for work.

As with the persistent C environment, the initialization call returns a *handle* that is used by EDCXSRVN for further communication with the created environment. EDCXSRVN suspends the execution of the calling routine and sends a message to the waiting server. When the server completes the function called for by the message its execution is suspended and the caller of EDCXSRVN resumes.

The server environment is terminated when a *Terminate* message is sent to the server.

Establishing a Server Environment

EDCXSRVI

This routine creates an LE/VSE environment for the server part of user-server application. It is intended that this routine be called by a stub routine supplied by the server and statically bound with the user application. The stub routine is responsible for loading the server application code.

Parameters:

1. The address of the entry point of the server code. This must be the address of the EDCXSTRT entry point.

2. The value to be in R1 when the server entry point is called. This can be used for communication between the initialization stub and the server mainline; its value can be retrieved in the server code. `EDCXREGS(1)` will return a pointer to this list of parameters.
3. The address of a low-level get-storage routine (meeting the same interface as `EDCXGET`, but not necessarily `EDCXGET`).
4. The address of a low-level free-storage routine (meeting the same interface as `EDCXFREE`, but not necessarily `EDCXFREE`).

Return: When this routine returns the server environment is fully established and waiting for a message from the user. `R15` points to a *handle* that is used in subsequent calls to `EDCXSRVN` to send messages to the server.

Initiating a Server Request

EDCXSRVN

This routine is used by the stub routines that are linked with user application routines to send a message to an active server in a user-server application.

Parameters:

1. The address of the handle returned by `EDCXSRVI`.
2. The function code for the function to be performed. The value -1 is used to indicate that the server should terminate. This value should not be used for any other purpose.
3. Other parameters, which are passed to the server code.

Return: R15 will contain the return code supplied by the server (as the parameter to `EDCXSACC`) for this service.

Accepting a Request for Service

EDCXSACC

This routine operates in the server part of a user-server application. It is used to indicate acceptance or rejection of the last-requested service.

Parameters:

1. The return code of the last-requested service 0 indicating that the request was accepted and will be processed.

For more information on `EDCXSACC`, see “`__xsacc()` — Accept Request for Service” on page 265.

Returning Control from Service

EDCXSRVC

This routine operates in the server part of a user-server application. It is used to indicate completion of the last-requested service and to get information required for the next service to be performed.

Parameters:

1. The return code for the last-requested service.

For more information on `EDCXSRVC`, see “`__xsrvc()` — Return Control from Service” on page 266.

Constructing User-Server Stub Routines

Part of building a server for use in a user-server environment is the construction of stub routines that load and initialize the server, pass messages to the server, and terminate the server. These stub routines are typically written in assembler language to allow them to be freely called from other environments without regard to the characteristics of the calling environment.

Building User-Server Environments

To build your server application, follow the rules for building a freestanding application as described in "Building Freestanding Applications" on page 230.

There are no special considerations for building user applications.

Table 39. Parts used by/with Application Server Routines

Part Name	Function	Inclusion in Executable Phase		Location
			Notes	
`EDCXSRVI`	This module is used by a server-supplied stub routine to start up a server.		In the user load module	Member of `PRD2.SCEEBASE`
`EDCXSRVN`	This module is used by a server-supplied stub routine to send a service-request message to a server.		In the user load module	Member of `PRD2.SCEEBASE`
`EDCXSRVC`	This module is used by a server to wait for the next message to process.		In the server load module	Member of `PRD2.SCEEBASE`
`EDCXSACC`	This module is used by a server to accept the last message received.		In the server load module	Member of `PRD2.SCEEBASE`
`EDCXSPRT`	System programming version of `sprintf()`	1		Member of `PRD2.SCEEBASE`
`EDCXEXIT`	System programming version of `exit()`.	1		Member of `PRD2.SCEEBASE`
`EDCXMEM`	System programming version of `malloc()`, `calloc()`, `realloc()`, `free()`, `__4kmalc()`, and `__24malc()`.	1		Member of `PRD2.SCEEBASE`

Notes:

1 This module must be explicitly included if you want to use the system programming version of the function.

Tailoring the System Programming C Environment

Depending on the environment under which you want to run your C routines, you might want to replace some of the following routines for system-specific routines. To work correctly, your routines should match the interface as documented in this section.

The routines as supplied by IBM with LE/VSE C Run-Time meet the interface as documented and are designed for VSE/ESA.

Generating Abends

EDCXABND

This routine is called to generate an abend if there is an internal error during initialization or termination of a system programming C environment.

Parameter:

R1 The address of the error code and reason code.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

This module must have the entry point name of @@XABND.

EDCJL095:

```
@@XABEND TITLE  'GENERATE AN ABEND'
EDCXABND CSECT
EDCXABND AMODE ANY
EDCXABND RMODE ANY
@@XABND  DS    0H
         ENTRY @@XABND
         USING *,R9
         LR    R9,R15
         SPACE 1
*
*        L     R1,0(,R1)         get address to codes
         USING PARMS,R1
         L     R7,ERROR_RC       get error code
         L     R8,REAS_RC        get reason code
ABEND    EDCXPIRE RC=(7),REASON=(8),DUMP=Y
*
         LTORG
         EJECT
PARMS    DSECT
ERROR_RC DS  F
REAS_RC  DS  F
DEND     DS  0H
*
R1       EQU 1
R7       EQU 7
R8       EQU 8
R9       EQU 9
R15      EQU 15
         END
```

Figure 64. Example Generate Abend Routine

Getting Storage

EDCXGET

This routine is called to get storage from the operating system.

Parameter:

R0 The requested length, in bytes. If the high-order bit is zero or if the request was made in 24-bit addressing mode, the storage will be allocated below the 16M line. If the high-order bit is on and the request is made in 31-bit addressing mode, storage will be allocated anywhere with a preference for storage above the 16M line if available.

Return:

R0 The length of the storage block acquired, in bytes.

R1 The address of the acquired area or zero if not available.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

The entry point name for this routine must be @@XGET.

If you provide your own EDCXGET routine, it will be used when C library functions explicitly get storage. Whenever the library functions invoke operating system services, there may be implicit requests for storage that cannot be tailored.

EDCJL096:

```
@@XGET   TITLE  'Obtain memory as specified in R0'
EDCXGET  CSECT
EDCXGET  AMODE ANY
EDCXGET  RMODE ANY
@@XGET   DS    0H
         ENTRY @@XGET
         SPACE 1
         BALR  R2,R0
         USING *,R2
***********************************************************************
**|   Obtain memory using GETVIS
**|      if the high bit of R1 is on
**|          turn high bit off
**|          invoke GETVIS indicating to get memory above the line
**|      else invoke GETVIS indicating to get memory below the line
***********************************************************************
         LTR   R0,R0             Memory above or below?
         BNL   BELOW
         SLL   R0,1              Want memory anywhere
         SRL   R0,1
         LTR   R2,R2             are we running above the line?
         BNL   BELOW             no, so ignore above request
         GETVIS LENGTH=(R0),LOC=ANY
         LTR   R15,R15           Was it successful?
         BZR   R14               Yes...
         SR    R1,R1             No, indicate failure
         BR    R14
BELOW    DS    0H                Get memory below the line
         GETVIS LENGTH=(R0),LOC=BELOW
         LTR   R15,R15           Was it successful?
         BZR   R14               Yes...
         SR    R1,R1             no, indicate failure in R1
         BR    R14
*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R4       EQU   4
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END
```

Figure 65. Example Get Storage Routine

Getting Page-Aligned Storage

EDCX4KGT

This routine is called to get page-aligned storage from the operating system.

Parameter:

R0	The requested length, in bytes. If the high-order bit of this register is zero or if the request was made in 24-bit addressing mode, the storage is allocated below the 16M line. If the high-order bit is on and the request is made in 31-bit addressing mode, storage is allocated above the 16M line. If this space is not available, storage is allocated elsewhere.

Return:

R0	The length of the storage block acquired, in bytes. This length may be greater than the size requested.
R1	The address of the acquired area or zero if not available.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

Its entry point must be `@@X4KGET`.

Freeing Storage

EDCXFREE

This routine is called to return storage to the operating system.

Parameters:

R0	The length of storage to be freed, in bytes
R1	The address of the area to be freed

Return:

R15	A system-dependent return code, which must be zero on success and nonzero otherwise

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

Its entry point must be `@@XFREE`.

If you provide your own `EDCXFREE` routine, it will be used when C library functions explicitly free storage. Whenever the library functions invoke operating-system services, there may be implicit requests to free storage that cannot be tailored.

EDCJL136:

```
@@XFREE  TITLE  'Free memory as specified in R1'
EDCXFREE CSECT
EDCXFREE AMODE ANY
EDCXFREE RMODE ANY
@@XFREE  DS    0H
         ENTRY @@XFREE
         SPACE 1
         DS 0H
         USING *,R15
*
**********************************************************************
**|      clear off high bit in R0 to make sure length is positive
**|      invoke FREEVIS with length in R0 and address in R1
**|      return 0 if successful, nonzero if failed
**********************************************************************
         SLL   R0,1              clear off ...
         SRL   R0,1                        ... high order bit
         FREEVIS LENGTH=(R0),ADDRESS=(R1)
         BR    R14               return
*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R4       EQU   4
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END
```

Figure 66. Example Free Storage Routine

Loading a Module

EDCXLOAD

This routine is called to load a named module into storage.

Parameter:

R1 Points to the name of the routine to be loaded.

Return:

R1 The address and AMODE of the routine, or zero if not loaded.

This routine *is* provided with a save area. Apart from the linkage registers, it must save and restore all registers used.

Its entry point must be `@@XLOAD`.

Deleting a Module

EDCXUNLD

This routine is called to delete a named module from storage.

Parameter:

R1 Points to the name of the routine to be deleted

Return:

R15 A system-dependent return code, which must be zero on success and nonzero otherwise

This routine *is* provided with a save area. Apart from the linkage registers, it must save and restore all registers used.

Its entry point must be `@@XUNLD`.

Abend Reason Codes

The following tables contain the abend codes and reason codes specific to the system programming facilities.

Table 40. Abend Codes Specific to System Programming Environments

Abend Code	Description
2100	No storage abend code
2101	Error freeing storage
2102	Error finding stack seg home
2103	Error loading library
2104	Error with heap allocation
2105	Error with system level command
2106	Error initializing statics
4000	Error when handling abend

Table 41. Reason Codes Specific to System Programming Environments

Reason Code	Description
7201	Error in initialization
7202	Error in termination
7203	Error when extending stack
7204	Error during longjmp/setjmp
7205	Can not locate static init. The routine `EDCRCINT` must be included in your module if you use the `RENT` compiler option.
7207	No initial heap allocation is specified and a heap is required.

Additional Library Routines

The following routines provide additional support that is unique to applications running in a system programming C environment. These routines are packaged as part of the run-time library.

`__xregs()`	Get registers on entry
`__xusr()`	Get address of User Word
`__xusr2()`	Get address of User Word
`__4kmalc()`	Allocate page-aligned storage
`__24malc()`	Allocate storage below 16MB line

For more information on these routines refer to Chapter 23, “Library Functions for the System Programming C Facilities,” on page 263.

Summary of Application Types

Table 42 summarizes application types, how they are called, and the module entry points.

Table 42. Summary of Types

Type of Application	How It Is Called	Module Entry Point	Run-Time Options (1) and Other Considerations
A mainline function that requires no dynamic library facilities	From the JCL	`EDCXSTRT`, which must be explicitly included at link time	Run-time options are specified by `#pragma runopts` in compilation unit for the `main()` function. The `heap` and `stack` options are honored. The `stack` defaults to be above the line.
A C subroutine called from assembler language using a pre-established persistent environment	A *handle*, the address of the subroutine and a parameter list are passed to `EDCXHOTU`.		Run-time options are specified by `#pragma runopts` in any compile unit. The `heap` and `stack` options are honored, except that the stack will default to be above the line. The runopts in the first object module in the link edit that contains runopts will prevail, even if this compilation unit is part of the calling application. The environment is established by calling `EDCXHOTC`. These functions return a value (the *handle*) which is used to call functions that use the environment.
A Server	User code includes a stub routine that calls `EDCXSRVI`. This causes the server to be loaded and control to be passed to its entry point.	`EDCXSTRT`	Run-time options are the same as for `EDCXSTRT`. The author of the server must supply stub routines which call `EDCXSRVI` and `EDCXSRVN` to initialize and communicate with the server. These are bound with the user application.
A User of an Application Server			The author of the server must supply stub routines which call `EDCXSRVI` and `EDCXSRVN` to initialize and communicate with the server.
1. The `STAE, SPIE` option is ignored if the library is not included.			

Chapter 23. Library Functions for the System Programming C Facilities

This chapter describes the library functions specific to the System Programming C (SPC) environment. The following functions are available:

```
__xhotc()
__xhott()
__xhotu()
__xregs()
__xsacc()
__xsrvc()
__xusr()
__xusr2()
__24malc()
__4kmalc()
```

These library functions are now described.

`__xhotc()` — Set Up a Persistent C Environment (No Library)

Format

```
#include <spc.h>

void *__xhotc(void *handle, int stack, int location);
```

Description

The function creates a persistent C environment with no C Run-Time Library. The parameters are fullwords (4 bytes).

1. *handle* is the field for the token (or handle) which is returned.
2. *stack* is the initial stack allocation required for the environment.
3. *location* is the location of the stack:

0	Below the line
1	Above the line

`__xhotc()` is specific to SPC. It is part of the group serving the persistent C environment.

The function is also available under the name `EDCXHOTC`.

Returned Value

`__xhotc()` returns a token (or handle) which is used in subsequent calls to `__xhotu()` and `__xhott()` to use or terminate a persistent C environment. This handle is found in both the first parameter passed and R15.

The `RENT` compiler option is not supported for routines called using this environment.

Example

For an extensive example of the use of `__xhotc()` see "Creating and Using Persistent C Environments" on page 238.

`__xhott()` — Terminate a Persistent C Environment

Format

```
#include <spc.h>

void __xhott(void *handle);
```

Description

This function terminates a persistent C environment created by `__xhotc()`.

The parameter of `__xhott()` is a handle returned by `__xhotc()`.

`__xhott()` is specific to SPC. It is part of the group serving the persistent C environment.

The function is also available under the name `EDCXHOTT`.

Example

For an extensive example of the use of `__xhott()` see "Creating and Using Persistent C Environments" on page 238.

`__xhotu()` — Run a Function in a Persistent C Environment

Format

```
#include <spc.h>

void *__xhotu(void *handle, void *function, ...);
```

Description

This function is used to run a function in a persistent C environment. The parameters are fullwords (4 bytes):

1. *handle* is a handle—returned by `__xhotc()` or `__xhotl()`
2. *function* is a function pointer, which points to the desired C function
3. First parameter to pass to the function
4. Second parameter to pass to the function

   ⋮

This routine, and the C function being called, must use `OS` linkage. C functions being invoked using `__xhotu()` must be compiled with `#pragma linkage(`*func_name*`,OS)`.

`__xhotu()` is specific to SPC. It is part of the group serving the persistent C environment.

The function is also available under the name `EDCXHOTU`.

Returned Value

The returned value from `__xhotu()` is the returned value from the function run in the persistent C environment.

Example

For an extensive example of the use of `__xhotu()` see "Creating and Using Persistent C Environments" on page 238.

__xregs — Get Registers on Entry

Format

```
#include <spc.h>
int __xregs(int register_number);
```

Description

This routine finds the value a specified register had on entry to `EDCXSTRT`, or the *main* routine of an exit routine compiled with `#pragma environment(...)`.

`__xregs()` is available in these environments only. For more information about `EDCXSTRT`, see "Creating Freestanding Applications" on page 229.

`__xregs()` is specific to SPC. It is part of the client-server group of functions.

The function is also available under the name `EDCXREGS`.

Returned Value

`__xregs()` returned the value found.

`__xsacc()` — Accept Request for Service

Format

```
#include <spc.h>

void __xsacc( int message );
```

Description

This routine operates in the server part of a user-server application. It is used to indicate acceptance or rejection of the last-requested service.

Calls to `__xsacc` are optional but, if made, should be when the request is validated and all server references to user-owned storage are complete. `__xsacc` does not cause a return of control to the user; its sole purpose is to indicate that user-owned storage is no longer required by the application server.

In the case of a request that cannot be processed, possibly because the user's command is not recognized by the server or the parameter format is invalid, the call to `__xsacc` should be omitted.

`__xsacc()` is specific to SPC. It is part of the client-server group of functions.

The function is also available under the name `EDCXSACC`.

Returned Value

The return code for the last-requested service, zero indicating that the request was accepted and will be processed.

`__xsrvc()` — Return Control from Service

Format

```
#include <spc.h>

void *__xsrvc(int message);
```

Description

This routine operates in the server part of a user-server application. It is used to indicate completion of the last-requested service and to get the information required for the next service to be performed.

message is the return code for the last-requested service.

`__xsrvc()` is specific to SPC. It is part of the client-server group of functions.

The function is also available under the name `EDCXSRVC`.

`__xusr()` - `__xusr2()` — Get Address of User Word

Format

```
#include <spc.h>

void *__xusr(void);
void *__xusr2(void);
```

Description

There are two words in an internal control block that are available for customer use. These words have an initial value of zero (that is, all bits are 0), but are otherwise ignored by compiled code. The values in these words may be freely queried or set by application code using the pointers returned by these functions.

`__xusr()` and `__xusr2()` are specific to SPC.

The `__xusr()` and `__xusr2()` functions are also available under the names `EDCXUSR` and `EDCXUSR2`, respectively.

Returned Value

`__xusr()` and `__xusr2()` return the addresses of these user words. The words, and indeed `__xusr()` and `__xusr2()` themselves, are available in *any* environment, not only the system programming environments.

`__24malc()` — Allocate Storage below 16MB Line

Format

```
#include <spc.h>

void *__24malc(size_t size);
```

Compiler Option

LANGLVL(EXTENDED)

Description

This function performs in the same manner as `malloc` except that it allocates storage below the 16MB line in z/VSE systems even when the run-time option `HEAP(ANYWHERE)` is specified.

Storage allocated by this function is not part of the heap, so you must free this storage explicitly using the `free()` function; it is not automatically freed when the environment is terminated.

The function is available under the System Programming Environment.

`__4kmalc()` — Allocate Page-Aligned Storage

Format

```
#include <spc.h>

void *__4kmalc(size_t size);
```

Compiler Option

LANGLVL(EXTENDED)

Description

This function performs in the same manner as `malloc()` except that it allocates page-aligned storage.

Storage allocated by this function is not part of the heap, so you must free this storage explicitly, using the `free()` function; it is not automatically freed when the environment is terminated.

The function is available under the System Programming Environment.

Part 5. Programming with Other Products

This part describes LE/VSE C Run-Time support for C programs using the following products:
- Customer Information Control System (CICS command level interface)
- Cross System Product (CSP)
- Data Language/I (DL/I)
- Query Management Facility (QMF)
- SQL/DS

Some of these tools do not support source code in any code page other than the default code page, 1047. To use these tools and write your code in a code page other than 1047, you must use the LE/VSE C Run-Time preprocessor `iconv` to convert your code to code page 1047. See Chapter 33, “Code Set and Locale Utilities,” on page 371 for details.

For general information about working with locale functions, including those locale functions that help with coded character set issues, see Chapter 29, “Introduction to Locale,” on page 317.

Chapter 24. Using CICS

This chapter describes how to develop C programs for the Customer Information Control System (CICS). The LE/VSE library provides support for C programs that run under the CICS Transaction Server for VSE/ESA (or CICS/VSE Version 2 Release 3 Version 2 Release 3 or later). You can find more information about the general features of LE/VSE and CICS in *LE/VSE Programming Guide*.

For information on using CSP/AD or CSP/AE under CICS, see Chapter 25, "Using CSP," on page 293.

Note: As of publication, the CICS translator does not recognize the C/VSE compiler's support for alternative locales and coded character sets. Therefore, you should write all your CICS C code in coded character set IBM-1047 (APL 293).

Developing C Programs for the CICS Environment

When developing a program to run under CICS you must:

1. Prepare CICS for use with LE/VSE.
2. Design and code the CICS program.
3. Translate and compile the translated source for reentrancy.
4. Prelink and link all object decks with the CICS stub.
5. Define the program to CICS.

Optional Tasks Related to Using CICS with LE/VSE

LE/VSE includes the C run-time environment, which is required in order to use the CICS Transaction Server.

Since LE/VSE is pre-installed on your system, you are not required to prepare the C run-time environment for using CICS.

Note: Under no circumstances should you remove the LE C component from your VSE system!

However, you might need to carry out some later customization tasks, such as:

- Including the Group (CEE) in the CSD file for LE/VSE support under the CICS/VSE Coexistence Environment.
- Tailoring the CICS Destination Control Table (DCT).
- Printing CICS-wide run-time options to the console.
- Ensuring that CICS coexistence is set up correctly.

These tasks are described in the *LE/VSE Customization Guide*.

Designing and Coding for CICS

This section describes what you must do differently when designing and coding a LE/VSE C Run-Time program for CICS, such as how to use `EXEC CICS` commands in your code, using input and output, using LE/VSE C Run-Time functions, managing storage, using interlanguage calls, and exception handling.

Using the CICS Command-Level Interface

The CICS Transaction Server provides a set of commands to access CICS. The format of a CICS command is:

```
EXEC CICS function [option[(arg)]]...;
```

In the following CICS command, the function is `SEND TEXT`. This function has 4 options: `FROM`, `LENGTH`, `RESP` and `RESP2`. Each of the options takes one argument.

```
EXEC CICS SEND TEXT FROM(mymsg)
                    LENGTH(mymsglen)
                    RESP(myresp)
                    RESP2(myresp2)
```

For further information on the `EXEC CICS` interface and a list of available CICS functions, refer to *CICS Transaction Server for VSE/ESA Application Programming Guide* and *CICS Transaction Server for VSE/ESA Application Programming Reference*.

When you are designing and coding your CICS application, remember the following:

- The `EXEC CICS` command and options should be in uppercase. The arguments follow general C conventions.
- Before any `EXEC CICS` command is issued, the EXEC Interface Block (EIB) must be addressed by the `EXEC CICS ADDRESS EIB` command.
- LE/VSE C Run-Time does not support the use of `EXEC CICS` commands in macros.

The example in Figure 67 on page 273 uses `EXEC CICS` commands to:

1	Initialize the CICS interface
2	Access the storage passed from the caller
3	Handle unexpected abends
4 and 7	Perform I/O to RRDS files
5 and 6	Request and format time

Refer to “EDCXGCI3” on page 284 for the functions `sendmsg()` and `unexpected_prob()`.

EDCXGCI1

```
#pragma runopts(rptstg(on))

 /* EDCXGCI1
    This example shows how to use EXEC CICS commands.
    Program : GETSTAT (part 1).
  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void check_4_down_status( char *status_record );

#define FILE_LEN 40
int *quiet;

main ()
{
 unsigned char status_record[41];

 long int  vsamrrn;
 signed short int  vsamlen;

 signed long int myresp;
 signed long int myresp2;

                          /* get addressability to the EIB first */
 EXEC CICS ADDRESS EIB(dfheiptr);                        1

                          /* access common area sent from caller */
 EXEC CICS ADDRESS COMMAREA(quiet);                      2

                          /* call the CATCHIT prog. if it abends */
 EXEC CICS HANDLE ABEND PROGRAM("CATCHIT");      3

 vsamrrn = 1;
 vsamlen = FILE_LEN;
                          /* read the status record from the file*/
 EXEC CICS READ FILE("STATFILE")                 4
                INTO(status_record)
                RIDFLD(vsamrrn)
                RRN
                LENGTH(vsamlen)
                RESP(myresp)
                RESP2(myresp2);
```

Figure 67. Command Level Interface Example (Part 1 of 3)

```
/* check cics response                 */
                          /*      -- non 0 implies a problem     */
 if (myresp != DFHRESP(NORMAL))
    unexpected_prob("Unable to read from file",61);

 if (memcmp(status_record,"DOWNTME ",8) == 0)
   check_4_down_status(status_record);

 if (*quiet != 1)
   sendmsg(status_record);

 exit(11);
}
/*******************************************************************/
 void check_4_down_status( char *status_record )
{
   unsigned char uptime[9];
   unsigned char update[9];
   char curabs[8];
   unsigned char curtime[9];
   unsigned char curdate[9];

   long int  vsmrrn;
   signed short int  vsmlen;
   signed long int dnresp;
   signed long int dnresp2;

   strncpy((status_record+8),update,8);
   strncpy((status_record+16),uptime,8);
   update[8] ='\0';
   uptime[8] ='\0';

                          /* get the current time/date           */
   EXEC CICS ASKTIME ABSTIME(curtime)                 5
                     RESP(dnresp)
                     RESP2(dnresp2);

   if (dnresp != DFHRESP(NORMAL))
      unexpected_prob("Unexpected prob with ASKTIME",dnresp);

                          /* format current date to YYMMDD       */
                          /* format current time to HHMMSS       */
   EXEC CICS FORMATTIME ABSTIME(curabs)               6
                        YYMMDD(curdate)
                        TIME(curtime);
```

Figure 67. Command Level Interface Example (Part 2 of 3)

```
   if (dnresp != DFHRESP(NORMAL))
      unexpected_prob("Unexpected prob with FORMATTIME",dnresp);

   curdate[8] ='\0';
   curtime[8] ='\0';

   if ((atoi(curdate) > atoi(update)) ||
      (atoi(curdate) == atoi(update) && atoi(curtime) >= atoi(uptime)))
   {
     strcpy(status_record,"OK                                     ");

     vsmrrn = 1;
     vsmlen = FILE_LEN;
                        /* update the first record to OK       */
     EXEC CICS WRITE FILE("STATFILE")           7
                    FROM(status_record)
                    RIDFLD(vsmrrn)
                    RRN
                    LENGTH(vsmlen)
                    RESP(dnresp)
                    RESP2(dnresp2);
     if (dnresp != DFHRESP(NORMAL))
        unexpected_prob("Unexpected prob with WRITE",dnresp);
     printf("%s %s Changed status from DOWNTME to OK\n",curdate,
             curtime);
   }

}
```

Figure 67. Command Level Interface Example (Part 3 of 3)

Using Input and Output

This section describes how to use LE/VSE C Run-Time I/O with CICS. It describes the file and device support and the type of I/O used with CICS.

Note: You can set up a `SIGIOERR` handler to catch read or write system errors. See Chapter 14, "Debugging I/O Programs," on page 131 for more information.

Standard Stream Support

Under CICS, `stdout` and `stderr` are assigned to transient data destinations (queues). The type of queue, intrapartition or extrapartition, is determined during CICS initialization. Intrapartition queues are used for queueing messages and data within a CICS region. Extrapartition queues are used to send data outside the CICS region or to receive data from outside the CICS region.

The transient data queues associated with `stdout` and `stderr` are CESO and CESE respectively.

Records sent to the transient data queues associated with `stdout` and `stderr` take the form of a message. The entire message record can be preceded by an ASA Standard control character. Figure 68 on page 276 illustrates the recommended message format.

ASA	Terminal ID	Transaction ID	Sp	Time Setup YYYYMMDDHHMMSS	Sp	Data
1	4	4	1	14	1	108

Figure 68. Format of Data Written to a CICS Data Queue

In Figure 68:

ASA	is the carriage-control character.
Terminal ID	is a 4-character terminal identifier.
Transaction ID	is a 4-character transaction identifier.
Sp	is a space.
Time Stamp	is the date and time displayed in the format YYYYMMDDHHMMSS.
Data	is the data outputted to the standard streams `stdout` and `stderr`.

The following are sample messages of data written to a CICS data queue:

```
SAMATST1 19960801080523 Hello World  - from transaction TST1!
BOBATST3 19960801112348 Hello World  - from transaction TST3!
TEDATST2 19960801112348 Hello World  - from transaction TST2!
```

Standard streams can only be redirected to or from memory files.

Because only one transient data queue can be associated with each of `stdout` and `stderr`, these queues can contain output written in chronological order from many C programs. This output must be sorted as necessary into the desired sequence.

Full Memory File Support

The full set of C I/O library functions is supported under CICS for memory files. Memory files are created with the parameter `type=memory` on the `fopen()` call.

Support for Disk Files and Other Devices

There is no support by the C I/O library for using disk files and other devices with CICS. I/O to access methods supported by CICS must use the CICS Application Programming Interface.

Using LE/VSE C Run-Time Library Support

This section discusses restrictions and support for the LE/VSE C Run-Time library with CICS.

Arguments to C `main()`

When a LE/VSE C Run-Time program is running under CICS, you cannot pass command line arguments to it. The values for `argc` and `argv` have the following settings:

argc	1
argv[0]	4-character CICS transaction ID

Run-Time Options

Command line run-time options cannot be passed in CICS. To specify run-time options, you must include the `#pragma runopts` directive in the code. Figure 67 on page 273 shows how to do this. See *LE/VSE Programming Guide* for information on other ways to supply run-time options when you are running under CICS.

Using Packed Decimal with CICS

The packed decimal data type is supported in CICS. However, the CICS translator does not support packed decimal. CICS usually stores packed decimal strings as arrays of characters. If you want to manipulate these arrays as a packed decimal number, you should define the array of characters in union with the appropriate packed decimal definition. Refer to *CICS Transaction Server for VSE/ESA Application Programming Reference* for information on how to define the data fields for the `EXEC CICS` commands you are using.

Locales

All locale functions are supported for locales that have been defined in the CSD. CSD definitions for the IBM-supplied locales are provided as member CEECCSD.Z in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE). `setlocale()` returns `NULL` if the locales are not defined.

Code Set Conversion Tables

The code set conversion tables that are used by the `iconv` functions must be defined in the CSD.

DL/I

There is no support for the `ctdli()` function under CICS. If you call `ctdli()` in CICS, the return value is -1. Refer to *CICS Transaction Server for VSE/ESA Application Programming Guide* for information on the CICS method to access DL/I.

Dump Functions

The dump functions `csnap()`, `cdump()`, and `ctrace()` are supported in LE/VSE. The output is sent to the CESE transient data queue. The dump can not be written if the queue does not have a sufficient LRECL. An LRECL of at least 161 is recommended.

The `fetch()` Function

The `fetch()` function is supported under CICS. Modules to be fetched must be defined to the CSD and installed in the PPT.

The `release()` Function

The `release()` function is supported under CICS.

The `system()` Function

The `system()` function is not supported in CICS. However, there are two `EXEC CICS` commands that give you similar functionality:

`EXEC CICS LINK`
: This command enables you to transfer control to another program and return to the calling program later. See Figure 69 on page 281.

`EXEC CICS XCTL`
: This command enables you to transfer control to another program. Control does not return to the caller after completion of the called program.

Time Functions

All time functions are supported except the `clock()` function, which returns the value `(time_t)(-1)` if it is used under CICS.

The `iscics()` Function

The `iscics()` function is an extension to the C library. It returns a non-zero value if your program is currently running under CICS. If your program is not running under CICS, `iscics()` returns the value 0. The following example shows how to use `iscics()` in your C program to specify non-CICS or CICS specific behavior.

```
if (iscics() == 0)
  < non-CICS behavior>
else
  < CICS-specific behavior>
```

Program Termination

A C program running under CICS will terminate when:

- An `exit()` function call or a `return` statement is issued in the C program. The `atexit()` list of functions is run when the C program terminates.

  Note: On return from a C language application, the `return` statement or values passed by C through the `exit()` function are saved in the `EIBRESP2` field of the EIB.
- An abend occurs and is not handled.
- An `EXEC CICS RETURN` is issued in your C program. The `atexit()` list of functions runs after these calls.
- The `abort()` function is started.

Storage Management

A LE/VSE C Run-Time program can acquire storage from and release storage to CICS implicitly or explicitly.

Storage is acquired and released *implicitly* by the run-time environment. This storage is used for automatic, external, and static variables. External variables are valid until program completion.

Storage is acquired and released *explicitly* by the user with the C library functions `malloc()`, `calloc()`, `realloc()`, or `free()`, with LE/VSE Callable Services (refer to *LE/VSE Programming Guide*), or with the `EXEC CICS` commands `EXEC CICS GETMAIN`, or `EXEC CICS FREEMAIN`.

- If you request the storage by using the C functions `malloc()`, `realloc()`, or `calloc()` you must deallocate it by using C functions as well.
- If you request the storage by using LE/VSE Callable Services, you must deallocate it by using LE/VSE Callable Services.
- If you request the storage by using `EXEC CICS GETMAIN`, you must deallocate it by using `EXEC CICS FREEMAIN`.

All other combinations of methods of requesting and deallocating storage are unsupported and lead to unpredictable behavior.

Partial deallocations are not supported. All storage allocated at a given time must be deallocated at the same time.

Under the LE/VSE library, LE/VSE C Run-Time uses the LE/VSE Callable Services to allocate and free storage. Refer to *LE/VSE Programming Guide* for specific information on memory and storage manipulation in CICS.

The LE/VSE C Run-Time library functions acquire all storage from the Extended Dynamic Storage Area (EDSA) unless you specify otherwise using the `ANYHEAP`, `BELOWHEAP`, `HEAP`, `STACK`, or `LIBSTACK` run-time options.

Storage that is acquired with the `EXEC CICS GETMAIN` command exists for the duration of the CICS task.

Using Interlanguage Support

The LE/VSE library supports a variety of different types of interlanguage calls (ILC) with CICS. For information on supported configurations, please refer to *LE/VSE Programming Guide*.

Exception Handling

There are three different kinds of exception handlers you can use when running C programs in a CICS environment: CICS exception handlers, LE/VSE abend handlers, and C exception handlers.

If the CICS command `EXEC CICS HANDLE ABEND PROGRAM(`*`name`*`)` was specified in the application, it will be called for any program exception that occurs (such as an operation exception or a protection exception) as well as for any `EXEC CICS ABEND ABCODE(...)` command that is run.

LE/VSE provides facilities to set up a user handler. These facilities are discussed in detail in *LE/VSE Programming Guide*.

In CICS, the C error handling facilities have almost the same behavior as discussed in Chapter 18, “Handling Error Conditions and Signals,” on page 181. A signal raised with the `raise()` function is handled by its corresponding signal handler or the default actions if no handler is installed. If a program exception such as a protection exception occurs, it is handled by the appropriate C handler if no CICS or LE/VSE handler is present.

When a C application is invoked by an `EXEC CICS LINK PROGRAM(...)`, the invoked program inherits any handlers registered by `EXEC CICS HANDLE ABEND PROGRAM(...)` in the parent program. Any handlers registered in the child override the inherited handlers. C signal handlers are *not* inherited.

The following chart shows the process for handling abends in CICS.

MAP 0020: Error Handling in CICS

001

Is this the result of a call to raise()?
Yes No

- **002**

  Has `EXEC CICS HANDLE ABEND` been issued?
  Yes No

  - **003**

    Continue at Step 005.

  004

  Call LE/VSE C Run-Time-CICS interface for termination of program. CICS turns off signal and runs program in handler.

005

Is `SIG_IGN` set for the signal?
Yes No

- **006**

  Is a LE/VSE handler registered?
  Yes No

  - **007**

    Is a C handler established?
    Yes No

    - **008**

      Default handling the program check and percolate to next stack frame.

    009

    Run C handler.

  010

  Run LE/VSE user handler. See LE/VSE Programming Guide for more details.

011

Resume at the next instruction.

Example of Error Handling in CICS

The example in Figure 69 shows how to handle errors when using LE/VSE C Run-Time with CICS. The numbers in the following list correspond to the numbers in the example code.

1 The program CATCHIT has been installed as the CICS abend handler. Because this CICS abend handler is installed, C exception handlers will only catch signals raised with the `raise()` function.

Note: The CATCHIT program has not been supplied as its function would be installation dependent.

2 Install a C signal handler to catch the user defined signal `SIGUSR1`. This handler will only be called if `raise(SIGUSR1)` is run.

3 This command causes the flow of control to shift to a child program called GETSTAT. GETSTAT will inherit CHKSTAT's CICS abend handler.

4 The C signal handler `status_not_OK` that was previously installed, will be invoked if this line is run. The `raise()` function will *not* trigger the CICS abend handler.

EDCXGCI2

```
 /* EDCXGCI2
    This example demonstrates error handling under CICS.
    Program : CHKSTAT.
    Transaction : Called stand alone from transaction CHST.
                  Is also used by other transactions to determine
                  system status.
  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>

#define FILE_LEN 40

void status_not_ok(int sig);
void unexpected_prob(char* desc, int rc);
volatile unsigned char status_record [41];
```

Figure 69. Example of Error Handling under CICS (Part 1 of 3)

```
main (int argc, char *argv [ ])
{

 int quiet;
 long int  vsamrrn;
 signed short int  vsamlen;

 signed long int myresp;
 signed long int myresp2;

 if (strcmp(argv[0],"CHST") !=0)
   quiet = 1;
 else
   quiet = 0;

 /* get addressability to the EIB first */
 EXEC CICS ADDRESS EIB(dfheiptr);

 EXEC CICS HANDLE ABEND PROGRAM("CATCHIT");     1
 signal(SIGUSR1,status_not_ok);                         2

 EXEC CICS LINK PROGRAM("GETSTAT")              3
                RESP(myresp)
                RESP2(myresp2)
                COMMAREA(quiet)
                LENGTH(4);

 /* check for failure in linked-to program */
 if (myresp != DFHRESP(NORMAL))
    unexpected_prob("CICS failure on EXEC CICS LINK\n",51);

 if (myresp2 != 11)
    unexpected_prob("Unexpected rc from GETSTAT\n",myresp2);

 vsamrrn = 1;
 vsamlen = FILE_LEN;
 EXEC CICS READ FILE("STATFILE")
                INTO(status_record)
                RIDFLD(vsamrrn)
                RRN
                LENGTH(vsamlen)
                RESP(myresp)
                RESP2(myresp2);

 /* check for cics response - non-0 implies problem */
 if (myresp != DFHRESP(NORMAL))
    unexpected_prob("Unable to read from file",52);

 if (memcmp(status_record,"OK ",3) != 0)
    raise(SIGUSR1);                                     4

 exit(11);
}
```

Figure 69. Example of Error Handling under CICS (Part 2 of 3)

```
void unexpected_prob( char* desc, int rc)
{
  long int msgresp, msgresp2;
  int msglen;

  msglen = strlen(desc);

  EXEC CICS SEND TEXT FROM(desc)
                      LENGTH(msglen)
                      RESP(msgresp)
                      RESP2(msgresp2);

  fprintf(stderr,"%s\n",desc);

  if (msgresp != DFHRESP(NORMAL))
    exit(99);
  else
    exit(rc);
}

void status_not_ok( int sig )
{

   if (memcmp(status_record,"DOWNSTR ",3) != 0)
     exit(22);
   else
     exit(33);
}
```

Figure 69. Example of Error Handling under CICS (Part 3 of 3)

ABEND Codes and Error Messages under LE/VSE C Run-Time

For information on ABEND Codes and error messages used by the LE/VSE library, refer to *LE/VSE Programming Guide* and *LE/VSE Debugging Guide and Run-Time Messages*.

Coding Hints and Tips

- Do not use `EXEC CICS` commands in macros.
- Do not use `EXEC CICS` commands in header files. This makes the translation process much simpler.
- Do not set `atexit()` routines before an `EXEC CICS XCTL`. You will get unpredictable results.
- If you call `fclose()` or `freopen()` for a standard stream, you cannot redirect or reopen the link to the transient data queue. LE/VSE C Run-Time does not provide a method of opening or reopening the transient data queues.
- The actual transient data queue is not closed when you call `fclose()` or `freopen()` for a standard stream; however, the transaction will lose access to the stream.
- You should not use the `stdin` stream unless you are redirecting it from a memory file.
- When CICS handlers (using `EXEC CICS HANDLE ABEND PROG`) are activated along with C signal handlers, the CICS handler is invoked when an abend occurs. The C signal handler that corresponds to that class of abends is ignored.

- If you do an EXEC CICS RETURN out of an atexit() routine, the resulting return code (RESP2) is undefined.

Translating and Compiling for Reentrancy

This section discusses and provides examples of using the CICS language translator and compiling for CICS. It also discusses reentrancy issues with respect to CICS.

Translating

CICS provides a utility program called the CICS language translator. This program translates the EXEC CICS statements into C code. The translator supplies a control block (DFHEIBLK) for passing information between CICS and the application program. C function references for the EXEC CICS commands are generated. The translation step is not required if you do not use EXEC CICS statements.

The CICS translator does not evaluate preprocessor statements such as #include or #define. You should ensure that all EXEC CICS statements are translated.

Translating Example

Figure 70 shows a piece of code before it is translated with the CICS language translator. Figure 71 on page 286 shows the corresponding program after translation.

EDCXGCI3

```
 /* EDCXGCI3
    This is an example of a CICS program.
    Program : GETSTAT ( part 2 - infrequent use routines ).
  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void unexpected_prob( char* desc, int rc);

void sendmsg( char* status_record )
{
  long int msgresp, msgresp2;
  char outmsg[80];
  int outlen;

  if (memcmp(status_record,"OK ",3)==0)
     strcpy(outmsg,"The system is available.");
  else if (memcmp(status_record,"DOWNTME ",8)==0)
     strcpy(outmsg,"The system is down for regular backups.");
  else
     strcpy(outmsg,"SYSTEM PROBLEM -- call help line for details.");

  outlen=strlen(outmsg);
```

Figure 70. Child Program Before Translation (Part 1 of 2)

```
  EXEC CICS SEND TEXT FROM(outmsg)                    1
                      LENGTH(outlen)
                      RESP(msgresp)
                      RESP2(msgresp2);

  if (msgresp != DFHRESP(NORMAL))
    unexpected_prob("Message output failed from sendmsg",71);

}

void unexpected_prob( char* desc, int rc)
{
  long int msgresp, msgresp2;
  int msglen;

  msglen = strlen(desc);

  EXEC CICS SEND TEXT FROM(desc)                      2
                      LENGTH(msglen)
                      RESP(msgresp)
                      RESP2(msgresp2);

  fprintf(stderr,"%s\n",desc);

  if (msgresp != DFHRESP(NORMAL))
    exit(99);
  else
    exit(rc);
}
```

Figure 70. Child Program Before Translation (Part 2 of 2)

In Figure 70 on page 284 observe the following:

1 and 2
This program contains two EXEC CICS commands to be translated by the CICS translator. A single instance of the EXEC CICS ADDRESS EIB command is required before any other call to the EXEC CICS interface. In this case, the main program (see Figure 67 on page 273) issues the ADDRESS EIB command. Since the two pieces of code make up one program there is no need to ADDRESS the EIB again.

The program once translated appears as follows:

```
#ifndef __dfheitab
  #define __dfheitab 1
     static   char      *dfhldver = "LD TABLE DFHEITAB 230." ;
     static unsigned short int  dfheib0  = 0                ;
     static   char      *dfheid0  = "\x00\x00\x00\x0c"      ;
     static   char      *dfheicb  = "        "              ;
   typedef struct  {                                 3
     unsigned char           eibtime  [4] ;
     unsigned char           eibdate  [4] ;
     unsigned char           eibtrnid [4] ;
     unsigned char           eibtaskn [4] ;
     unsigned char           eibtrmid [4] ;
     signed short int        eibfil01  ;
     signed short int        eibcposn  ;
     signed short int        eibcalen  ;
     unsigned char           eibaid    ;
     unsigned char           eibfn    [2] ;
     unsigned char           eibrcode [6] ;
     unsigned char           eibds    [8] ;
     unsigned char           eibreqid [8] ;
     unsigned char           eibrsrce [8] ;
     unsigned char           eibsync   ;
     unsigned char           eibfree   ;
     unsigned char           eibrecv   ;
     unsigned char           eibfil02  ;
     unsigned char           eibatt    ;
     unsigned char           eibeoc    ;
     unsigned char           eibfmh    ;
     unsigned char           eibcompl  ;
     unsigned char           eibsig    ;
     unsigned char           eibconf   ;
     unsigned char           eiberr    ;
     unsigned char           eiberrcd [4] ;
     unsigned char           eibsynrb  ;
     unsigned char           eibnodat  ;
     signed long  int        eibresp   ;
     signed long  int        eibresp2  ;
     unsigned char           eibrldbk  ;
   }  DFHEIBLK;
   static DFHEIBLK *dfheiptr;
#endif
```

Figure 71. Child Program After Translation (Part 1 of 4)

```
#ifndef __dfhtemps
#pragma linkage(dfhexec,OS)  /* force OS linkage */
void dfhexec();  /* Function to call CICS */
  #define __dfhtemps 1
     static signed short int  dfhb0020,   *dfhbp020 = &dfhb0020 ;
     static signed short int  dfhb0021,   *dfhbp021 = &dfhb0021 ;
     static signed short int  dfhb0022,   *dfhbp022 = &dfhb0022 ;
     static signed short int  dfhb0023,   *dfhbp023 = &dfhb0023 ;
     static signed short int  dfhb0024,   *dfhbp024 = &dfhb0024 ;
     static signed short int  dfhb0025,   *dfhbp025 = &dfhb0025 ;
     static unsigned char     dfhc0010,   *dfhcp010 = &dfhc0010 ;
     static unsigned char     dfhc0011,   *dfhcp011 = &dfhc0011 ;
     static signed short int    dfhdummy;
#endif

 /* this is an example of a CICS program                              */
 /* program : GETSTAT ( part 2 - infrequent use routines )            */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void unexpected_prob( char* desc, int rc);

void sendmsg( char* status_record )
{
  long int msgresp, msgresp2;
  char outmsg[80];
  int outlen;

  if (memcmp(status_record,"OK ",3)==0)
     strcpy(outmsg,"The system is available.");
  else if (memcmp(status_record,"DOWNTME ",8)==0)
     strcpy(outmsg,"The system is down for regular backups.");
  else
     strcpy(outmsg,"SYSTEM PROBLEM -- call help line for details.");

  outlen=strlen(outmsg);
```

Figure 71. Child Program After Translation (Part 2 of 4)

```
  /* EXEC CICS SEND TEXT FROM(outmsg)               4
                         LENGTH(outlen)
                         RESP(msgresp)
                         RESP2(msgresp2) */
  {
  dfhb0020 = outlen;
  dfhexec("\x18\x06\x60\x00\x2F\x00\x00\x00\x00\x00\x20\x04\x00\x00\x20\xF0\xF0\
\xF0\xF0\xF2\xF7\xF0\xF0",dfhdummy,outmsg,dfhbp020 );   5
   msgresp = dfheiptr->eibresp;
   msgresp2 = dfheiptr->eibresp2;
   }

  if (msgresp != 0 /* DFHRESP(NORMAL)=0 */     )
    unexpected_prob("Message output failed from sendmsg",71);

}
 void unexpected_prob( char* desc, int rc)
{
  long int msgresp, msgresp2;
  int msglen;

  msglen = strlen(desc);
```

Figure 71. Child Program After Translation (Part 3 of 4)

```
  /* EXEC CICS SEND TEXT FROM(desc)
                         LENGTH(msglen)
                         RESP(msgresp)
                         RESP2(msgresp2) */
  {
  dfhb0020 = msglen;
  dfhexec("\x18\x06\x60\x00\x2F\x00\x00\x00\x00\x00\x20\x04\x00\x00\x20\xF0\xF0\
\xF0\xF0\xF4\xF4\xF0\xF0",dfhdummy,desc,dfhbp020 );    6
   msgresp = dfheiptr->eibresp;
   msgresp2 = dfheiptr->eibresp2;
   }

  fprintf(stderr,"%s\n",desc);

  if (msgresp != 0 /* DFHRESP(NORMAL)=0 */     )
    exit(99);
  else
    exit(rc);
}
```

Figure 71. Child Program After Translation (Part 4 of 4)

In Figure 71 on page 286 observe the following:

3 The structure `DFHEIBLK` is used for passing information between CICS and the application program.

4 This is the CICS command that was interpreted by the translator. The translator comments out the `EXEC CICS` commands.

5 The translator inserts this call to the function `dfhexec()` and comments out the `EXEC CICS` commands for further processing by the C/VSE compiler. The values `msgresp` and `msgresp2` are set from the values in the `DFHEIBLK` structure.

6 This `EXEC CICS` command is similar in format to the one discussed in 4.

However, you should note that the generated call to `dfhexec()` is different. For this reason it is important that `EXEC CICS` commands are not imbedded in macros.

Compiling

CICS requires that programs be reentrant at CICS entry points. This means:

- If your program is not naturally reentrant, you must compile with the `RENT` compiler option.
- If you are compiling code that was translated by the CICS translator, you must compile with the `RENT` compiler option. The CICS translator puts external writable static in the program.
- If your program is naturally reentrant and has not been translated, you can compile and link it just as you would a non-CICS program.

Sample JCL to Translate and Compile

The sample JCL in Figure 72 on page 290 shows you how to translate and compile a C module (steps 1 and 2).

```
* $$ JOB JNM=jobname,PDEST=(*,uid),LDEST=(*,uid),PRI=prty,CLASS=class
* $$ PUN DISP=I
// JOB jobname
// LIBDEF *,SEARCH=(PRD2.SCEEBASE,...)   1
// EXEC IESINSRT
$ $$ LST DISP=D
// JOB CXLTCL
// LIBDEF *,SEARCH=(PRD2.SCEEBASE,...)   2
// LIBDEF PHASE,CATALOG=phase_lib   3
 /* --------------------------------------------------------------------
 /* STEP 2: Compile the generated source and write the object module
 /*         to SYSLNK
 /*
 /*         Note: Although step 2 appears before step 1 in this JCL
 /*               stream, it is actually executed after step 1. The JCL
 /*               comprising step 2 is first copied to SYSLNK, then step
 /*               1 (which appends the output from the translator to
 /*               SYSLNK) is executed. Only then is step 2 (and
 /*               subsequently step 3 and 4) executed.
 /* --------------------------------------------------------------------
// OPTION CATAL
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='/RENT'
* $$ END
 /* --------------------------------------------------------------------
 /* STEP 1: Translate program source
 /* --------------------------------------------------------------------
// EXEC   DFHEDP1$,PARM='CICS,...'   4
   #pragma runopts(options)
   .
   C source statements
   .
 /*
// EXEC IESINSRT
 /*
 /* --------------------------------------------------------------------
 /* STEP 3: Pre-link the object module and write the output to SYSLNK
 /* --------------------------------------------------------------------
   PHASE phase_name,*
   INCLUDE DFHELII   5
// EXEC EDCPRLK,SIZE=EDCPRLK   6
 /* --------------------------------------------------------------------
 /* STEP 4: Link-edit pre-linked program
 /* --------------------------------------------------------------------
// EXEC LNKEDT   7
 /*
$ $$ EOJ
* $$ END
/&
* $$ EOJ
```

Figure 72. JCL to Translate, Compile, Prelink, and Link-Edit

1 Code the LIBDEF search chain to include all the library names that the CICS translator needs.

2 Code the LIBDEF search chain to include all the library names that the C/VSE compiler, the prelinker, and the linkage editor needs.

3 This is the library to which the phase is written.

4 You can code the translator options (XOPTS) either on the `PARM` parameter of the `EXEC` job control statement as shown, or on a `#pragma` directive preceding the C source program.

See *CICS Transaction Server for VSE/ESA Application Programming Guide* for details of the translator options.

Prelinking and Linking All Object Decks

The sample JCL in Figure 72 on page 290 shows you how to prelink and link-edit a C module (steps 3 and 4).

5 CICS provides a stub called DFHELII, which must be link-edited with the phase. The `INCLUDE` statement for DFHELII must follow immediately after the `PHASE` statement and before the `EXEC EDCPRLK` statement. A name card should also be passed to the linkage editor.

If you have compiled your source with the `RENT` compile-time option, you must prelink all of the text decks together. The prelinker accepts one or more text decks, combines them, and generates a single output text deck which can then be linked.

For further information on the prelinker, see *LE/VSE Programming Guide*.

Note: If your program is to be installed in the SVA, ensure you linkedit the module using the SVA option on the PHASE card.

All applications *must* run `AMODE=31`. It is recommended that the text deck is linked with `AMODE(31)` and `RMODE(ANY)`. CICS does not require any other linkage editor options.

6 If it is necessary to use some of the prelinker control statements, you have to catalog the object into a sublibrary first. In this case, the compiler needs the `NAME` and `DECK` options, and prelinker statements could be used to supply the phase name and to include DFHELII, in addition to performing the other functions requested. The prelinker reads the input from `SYSIPT`. See *LE/VSE Programming Guide* for further information.

7 Weak external references (WXTRN) unresolved by the linkage editor, and their associated messages about unresolved address constants, can be ignored.

Defining and Running the CICS Program

Program Processing

In a CICS environment, a single copy of a program is used by several transactions concurrently. One section of a program can process a transaction and then be suspended (usually as a result of an `EXEC CICS` command); another transaction can then start or resume processing the same or any other section of the same application program. This behavior requires that the program be reentrant.

Link Considerations

If your program is to be installed in the SVA, the module must be linkedited using the SVA option on the PHASE card.

CSD Considerations

Before you can run a program, you must define it in the CICS CSD.

Chapter 25. Using CSP

This chapter briefly describes the interface between LE/VSE C Run-Time and applications generated through the Cross System Product/Application Development (CSP/AD) and the Cross System Product/Application Execution (CSP/AE) Version 3 Release 3 Modification 0 or later. CSP refers to both CSP/AD and CSP/AE.

CSP/AD is an interactive application generator that provides methods for interactively defining, testing, and generating application programs. It can aid in improving productivity in application development.

CSP/AE takes the generated program and executes it in a production environment. For more information on CSP, see "Where to Find More Information" on page xxi.

Calls from LE/VSE C Run-Time to CSP applications are only supported under CICS/VSE.

Common Data Types

Table 43 lists the data types common to both CSP and LE/VSE C Run-Time.

Table 43. Common Data Types Between C and CSP

LE/VSE C Run-Time	CSP
`signed short`	`BIN - 2` bytes
`signed int/long`	`BIN - 4` bytes
`struct`	`RECORD`
`char array[size]`	Characters

You must use the function `__csplist()` to receive the parameter list from a CSP application. See *LE/VSE C Run-Time Library Reference* for more information on this function.

Passing Control

You can pass control between CSP and LE/VSE C Run-Time as follows:

CALL Calls another application or subroutine to be run. When execution is completed, control is returned to the statement following the `CALL` statement in the original application.

XFER|DXFR
Transfers control and initiates execution of a CSP application or non-CSP program or transaction. The current application is terminated when the transfer statement is executed.

Under CICS, `XFER` is used to transfer control to another CICS transaction, while `DXFR` is used to transfer control to an application or program. If the target name is an application, control remains in CSP and the application is initiated immediately. If the target name is a program, CSP issues `CICS XCTL` to the program name.

Note: From a LE/VSE C Run-Time program, you can pass control to a CSP application but you cannot pass control to another LE/VSE-enabled language (C, COBOL, PL/I) from that CSP application. Only one LE/VSE-enabled language can be in the chain of calls.

Running under CICS Control

CSP-CICS Note: Because all LE/VSE C Run-Time applications running under CICS must run with `AMODE=31`, when passing parameters to CSP, you must either

- Pass parameters below the line

  or

- Relink CSP applications with `AMODE=31`

The following example program shows how parameters are received from a CSP application that uses a `CALL` statement to transfer control. The LE/VSE C Run-Time program is expecting to receive an `int` as a parameter.

Examples

EDCXGCP5

```
 /* EDCXGCP5
    This example shows how to call C from CSP under CICS, and how
    parameters are passed
  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

main()
{
   struct tag_commarea {     /* commarea passed to C from R924A1 */
       int  *ptr1 ;
       int  *ptr2 ;
       int  *ptr3 ;
   } * ca_ptr  ;             /* commarea ptr */

   int  *parm1_ptr ;
   int  *parm2_ptr ;
   int  *parm3_ptr ;
                           /* addressability to EIB control block */
                           /*  and COMMUNICATION AREA             */
   EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(ca_ptr) ;
   parm1_ptr = ca_ptr->ptr1 ;
   parm2_ptr = ca_ptr->ptr2 ;
   parm3_ptr = ca_ptr->ptr3 ;

   *parm3_ptr = (int) pow((double) *parm1_ptr,
                          (double) *parm2_ptr);

   EXEC CICS RETURN;
}
```

Figure 73. CSP CALLing LE/VSE C Run-Time under CICS

The following example program shows how parameters are received from a CSP application that uses an XFER statement to transfer control.

EDCXGCP6

```
 /* EDCXGCP6
    This example shows how to XFER control to C from CSP under CICS

           XFER           CALL
    R924A3 ====> EDCXGCP6 ====> R924A6
    R924A3 and R924A6 are CSP applications
  */

#include <math.h>
#include <string.h>
                                    /* structure passed to R924A6*/
void main()
{
 struct {
    char                       *appl_ptr;
    _Packed struct tag_a3rec   *rec3_ptr ;
 } parm_ptr ;
                                    /* Structure received R924A3*/
 struct tag_a3rec {
   char  a3ct   [ 4];
   char  a3lan  [ 4];
   char  fil1   [ 8];            /* packed fields for PLI */
   char  fil2   [ 8];            /* packed fields for PLI */
   char  fil3   [ 8];            /* packed fields for PLI */
   int   a3xbc;            /* int field 1 for LE/VSE C Run-Time */
   int   a3ybc;            /* int field 2 for LE/VSE C Run-Time */
   int   a3zbc;            /* int field 3 for LE/VSE C Run-Time */
 };
 _Packed struct tag_a3rec   a3rec ;
```

Figure 74. CSP Transferring Control to C under CICS Using the XFER Statement (Part 1 of 2)

```
char   lk_appl[16] = "USR5ALF.R924A6  " ;

struct tag_a3progx {
  char  alfx   [ 8];
  char  applx  [ 8];
};
_Packed struct tag_a3progx a3progx = {"USR5ALF.","R924A6  "};
short  length_a3rec = sizeof(a3rec) ;
char   * pa3rec ;
short  i ;

/*----- start of CSP XFER-ing to C under CICS ------------------*/

   EXEC CICS ADDRESS EIB(dfheiptr);
                              /* retrieve data from CSP */
   EXEC CICS RETRIEVE INTO(&a3rec) LENGTH(length_a3rec) ;

   a3rec.a3zbc = (int) pow((double) a3rec.a3xbc,
                           (double) a3rec.a3ybc);

/*----- end of CSP XFER-ing to C under CICS --------------------*/

                              /* call CSP to display results*/
   parm_ptr.appl_ptr = lk_appl ; /* alf.application           */
   parm_ptr.rec3_ptr = &a3rec  ;
                                  /* LINK to CSP application */
   EXEC CICS LINK  PROGRAM("DCBINIT ")
                   COMMAREA(parm_ptr)
                   LENGTH(8) ;

   if (dfheiptr->eibresp2 != 0) {
      printf("EDCXGCP6: EXEC CICS LINK  returned non zero \n");
      printf("        return code. eibresp2 =%d\n",
                         dfheiptr->eibresp2);
   }
/*----- end of C calling CSP under CICS ------------------------*/
  EXEC CICS RETURN ;
}
```

Figure 74. CSP Transferring Control to C under CICS Using the XFER Statement (Part 2 of 2)

The following example program shows how parameters are received from a CSP application that uses a DXFR statement to transfer control. You must receive a structure.

EDCXGCP7

```
 /* EDCXGCP7
    This example shows how to transfer control to C from CSP under
    CICS, using the DXFR statement

           DXFR           XCTL( equivalent to dxfr)
    R924A3 ====> EDCXGCP7 ====> DCBINIT   ( appl R924A5)
    R924A3 is a CSP application
  */

 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <math.h>

main ()
{
   struct tag_a3rec {
     char  a3ct   [ 4];
     char  a3lan  [ 4];
     char  fil1   [ 8];           /* packed fields for PLI */
     char  fil2   [ 8];           /* packed fields for PLI */
     char  fil3   [ 8];           /* packed fields for PLI */
     int   a3xbc       ;
     int   a3ybc       ;
     int   a3zbc       ;
   };
                           /* commarea passed to C from R924A3 */
   struct tag_commarea {
     char  a3ct   [ 4] ;
     char  a3lan  [ 4];
     char  fil1   [ 8];     /* packed fields for PLI */
     char  fil2   [ 8];     /* packed fields for PLI */
     char  fil3   [ 8];     /* packed fields for PLI */
     int   a3xbc       ;
     int   a3ybc       ;
     int   a3zbc       ;
   } * ca_ptr  ;                          /* commarea ptr */

   struct tag_a5progc {
     char  alfc   [ 8] ;
     char  applc  [ 8] ;
     struct tag_a3rec a3rec;
   } a5progc  = {"USR5ALF.","R924A5  "};

   short   length_a3rec   = sizeof(struct tag_a3rec) ;
   short   length_a5progc = sizeof(struct tag_a5progc) ;

                           /* addreasability to EIB control block */
                           /*  and COMMUNICATION AREA             */
```

Figure 75. CSP Transferring Control to C under CICS Using the DXFR Statement (Part 1 of 2)

```
 EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(ca_ptr) ;

    if (dfheiptr->eibcalen == length_a3rec ) {
       memcpy(&a5progc.a3rec,  ca_ptr       , length_a3rec);

                                /* calculate the pow(x,y)   */
       a5progc.a3rec.a3zbc = (int) pow((double) a5progc.a3rec.a3xbc,
                                       (double) a5progc.a3rec.a3ybc);

       EXEC CICS XCTL
                 PROGRAM("DCBINIT ")
                 COMMAREA(a5progc)
                 length(length_a5progc) ;

      if (dfheiptr->eibresp2 != DFHRESP(NORMAL)) {
        printf ("EDCXGCP7: failed on xctl call to DCBINIT\n");
        printf ("                                      \n");
      }
    }
    else {
      printf ("EDCXGCP7:"
              "length of COMMAREA is different from expected\n");
      printf ("       expected %d, actual %d\n",
                     length_a3rec,  dfheiptr->eibcalen);
      printf (" \n");
      EXEC CICS RETURN;
    }

 EXEC CICS RETURN;
}
```

Figure 75. CSP Transferring Control to C under CICS Using the DXFR Statement (Part 2 of 2)

Chapter 26. Using DL/I

This chapter explains how DL/I and C handle errors and describes the limitations to using DL/I with C.

LE/VSE C Run-Time provides the `ctdli()` C library function to invoke DL/I facilities (see *LE/VSE C Run-Time Library Reference* for more information).

You can also invoke DL/I facilities with the callable service CEETDLI which is provided by the LE/VSE. The CEETDLI interface performs essentially the same functions as `ctdli()`. If you use the CEETDLI interface instead of `ctdli()`, condition handling is improved because of the coordination between LE/VSE and DL/I condition handling facilities. For complete information on the CEETDLI interface, see *LE/VSE Programming Guide*.

For a description of writing DL/I batch and online programs in C, see the appropriate book listed in "Where to Find More Information" on page xxi.

To use DL/I from LE/VSE C Run-Time, you must keep the following in mind:

- The file ims.h must be included in the program.
- The `PLIST(OS)` and the `ENV(DLI)` suboptions of the `#pragma runopts` preprocessor directive must be used to compile DL/I C application programs. `PLIST(OS)` establishes the correct parameter list format when invoked under DL/I and `ENV(DLI)` establishes the correct operating environment.
- When you use the `PLIST(OS)` option on the `#pragma runopts` preprocessor directive, `argc` will contain 1 (one) and `argv[0]` will contain NULL.
- DL/I provides a language interface module (`DLZLI000`) that gives a common interface to DL/I. This module must be link-edited with the application program.

For the remainder of this chapter, it is assumed that you are using the `ctdli()` interface.

Handling Errors

The DL/I environments are sensitive to errors and error-handling issues. A failing DL/I transaction or program can potentially corrupt a DL/I database. DL/I must know about the failure of a transaction or program that has been updating a database so that it can back out any updates made by that failing program.

C provides extensive error-handling facilities for the programmer, but special steps are required to coordinate DL/I and C error handling so that DL/I can do its database rollbacks when a program fails.

When you are using DL/I from C:

- Run your C program with the `TRAP(ON,MAX)` option, and use DL/I interfaces by calling the `ctdli()` library function.
- The `ctdli()` library function will keep track of calls to and returns from DL/I. If an abend or program check occurs and the C error handler gets control, it can determine if the problem arose on the DL/I side of the interface or on the C side.

- If a program check or abend occurs in DL/I, when the C exception handler gets control, it immediately passes control to the DL/I partition controller. The DL/I partition controller ensures that the integrity of the database is preserved.
- If a program check occurs in the C program rather than in DL/I, all the facilities of C error handling apply, provided that you meet certain conditions when you code your program. For any error condition that arises, you must do one of the following:
  1. Resolve the error completely so that the application can continue.
  2. Have DL/I back out the program's updates by issuing a rollback call to DL/I, and then terminate the program.
  3. Make sure that the program terminates abnormally and provide an installation-modified run-time user exit that turns all abnormal terminations into operating system `ABEND`s to effect DL/I rollbacks. See *LE/VSE Programming Guide* for more information.

  The errors you most likely can fix in your program are arithmetic exception (`SIGFPE`) conditions. It is unlikely that you can resolve other types of program checks or system abends in your program.

Any program that invokes DL/I by way of some other DL/I interface should be executed with `TRAP(OFF)`. You should be sure that the program contains code to issue a rollback call to DL/I before terminating after an error. Refer to *LE/VSE Programming Reference* for more information about the limitations of using `TRAP(OFF)`.

Other Considerations

A *program communication block* (PCB) is a control block used by DL/I to describe results of a DL/I call (DB PCB) or the results of a message retrieval or insertion (I/O PCB) made by your program. A valid PCB is one that has been correctly initialized by DL/I and passed to you through your C program. For details on PCBs, refer to "Where to Find More Information" on page xxi. See also the sample C-DL/I program in *LE/VSE C Run-Time Library Reference*.

When you are running under DL/I, you should note the following effects of specifying `PLIST(OS)`, `ENV(DLI)`, and their combinations. Specifying `PLIST(OS)` results in an `argc` value of 1 (one), and `argv[0]` = NULL. Also, the following chart shows the combinations of `PLIST(OS)` and `ENV(DLI)` and the resulting PCB generated:

Table 44. PCB Generated under DL/I

Combination	**Running under DL/I**
`ENV(DLI)` only	Valid PCB
`PLIST(OS)` only	Null PCB
`ENV(DLI)` *and* `PLIST(OS)`	Valid PCB

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

Example

The following is an example of a LE/VSE C Run-Time program that makes a DL/I call and checks the return code status of the call in DL/I batch.

EDCXGIM2

```
 /* EDCXGIM2
    This is an example of how to use DL/I with C
  */

#pragma runopts(env(dli),plist(os))
#include <ims.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "edcxgim3.h"

int main(void) {
 /*******************************************************************/
 /*   Declare the database pointer control block for the database   */
 /*******************************************************************/

    PCB_STRUCT_8_TYPE *cusdb_ptr;

/*******************************************************************/
 /*   IO area used for DL/I call                                    */
 /*******************************************************************/

    auto IOA2 aio_area;
    IOA2 *io_area;

/*******************************************************************/
 /*   SSAs for DL/I calls                                           */
 /*******************************************************************/

    static char qual0[] =      "STSCCST (STQCCNO  =000001)";
    static char qual1[] =      "STSCLOC  ";
    static char qual2[] =      "STPCORD  ";
    static int six      = 6;
    static char gu[4]    = "GU  ";

    int rc;
    int failed = 0;    /* Indicate if any part of test case failed. */

 /*******************************************************************/
 /*   Get the pointer to the database from the parameter list      */
 /*******************************************************************/

    cusdb_ptr = (__pcblist[1]);
```

Figure 76. C Program Using DL/I (Part 1 of 2)

```
 /******************************************************************/
 /*   Make a call to get some data from the database               */
 /******************************************************************/

    printf("DL/I Test starting\n");

    io_area = malloc(sizeof(IOA2));

 /******************************************************************/
 /*   Issue a DL/I call with arguments below the line (using CTDLI) */
 /******************************************************************/

    rc = ctdli(six,gu,cusdb_ptr,&aio._area,qual0,qual1,qual2);

   if ((cusdb_ptr->stat_code[0] == ' ' &&; cusdb_ptr->stat_code_1¨==' ')
       && (rc == 0))
      printf("Call to CTDLI returned successfully\n");
    else
      {
       printf("Call to CTDLI returned status of %c%c.\n",
        cusdb_ptr->stat_code[0],cusdb_ptr->stat_code[1]);
       failed = 1;
      }
    if (failed == 0)
     printf("Test Successful\n");
    else printf("Test Failed");

    return(0);
}
```

Figure 76. C Program Using DL/I (Part 2 of 2)

EDCXGIM3

Figure 77. Header File for DL/I Example (Part 1 of 3)

```
 /* EDCXGIM3
    This header file is used with the DL/I example
  */

 /*-------------------*/
 /*     DB PCB        */
 /*-------------------*/
typedef struct {
    char db_name[8];
    char seg_level[2];
    char stat_code[2];
    char proc_opt[4];
    int dli;
    char seg_name[8];
    int len_kfb;
    int no_senseg;
    char key_fb[2];
} DB_PCB;

 /*-------------------*/
 /*    IO PCB         */
 /*-------------------*/
typedef struct {
    char term[8];
    char ims_res[2];
    char stat_code[2];
    char date[4];
    char time[4];
    int input_seq;
    char output_mess[8];
    char mod_nme[8];
    char user_id[8];
 } IO_AREA;
 /*-------------------*/
 /*   SPA DATA        */
 /*-------------------*/
 typedef struct {
     short int uosplth;
     char uospres1[4];
     char uosptran[8];
     char uospuser;
     char fill[85];
 } SPA_DATA;
 /*-------------------*/
 /*   INPUT MESSAGE  */
 /*-------------------*/
typedef struct {
    short int ll;
    char zz[2];
    char fill[2];
    char numb[4];
    char nme[6];
} IN_MSG;
```

Figure 77. Header File for DL/I Example (Part 2 of 3)

```
 /*-------------------*/
 /*  OUTPUT MESSAGE  */
 /*-------------------*/
typedef struct {
    short int ll;
    char z1;
    char z2;
    char fill[2];
    char sca[2];
} OUT_MSG;

 /*-------------------*/
 /*  IO AREA         */
 /*-------------------*/
typedef struct {
    char key[20];
} IOA1;

typedef struct {
    char item[40];
} IOA2;
```

Figure 77. Header File for DL/I Example (Part 3 of 3)

Chapter 27. Using QMF

LE/VSE C Run-Time's support of the Query Management Facility (QMF) interface, a query and report writing facility, enables you to write applications through the SAA callable interface. You can create applications to perform a variety of tasks such as data entry, query building, administration aids, and report analysis.

You must include the header file `dsqcommc.h` (provided with the QMF application), which contains the function and structure definitions necessary to use the QMF interface.

For information on how to write your LE/VSE C Run-Time applications with the QMF interface, see the appropriate manual listed in "Where to Find More Information" on page xxi.

Example

The following example demonstrates the interface between the QMF facility and LE/VSE C Run-Time.

EDCXGQM1

```
 /* EDCXGQM1
    This example shows how to use the interface between QMF and C
  */

#include <string.h>
#include <stdlib.h>
#include <dsqcommc.h>   /*  QMF header file  */

int main(void)
{
   struct dsqcomm communication_area;  /*  found in DSQCOMMC  */


 /*********************************************************************/
 /* Query interface command length and commands                       */
 /*********************************************************************/
   signed long command_length;
   static char start_query_interface  [] = "START";
   static char set_global_variables   [] = "SET GLOBAL";
   static char run_query              [] = "RUN QUERY Q1";
   static char print_report           [] = "PRINT REPORT (FORM=F1)";
   static char end_query_interface    [] = "EXIT";
```

Figure 78. QMF Interface Example (Part 1 of 3)

```
/*********************************************************************/
/* Query command extension, number of parameters and lengths        */
/*********************************************************************/
  signed long number_of_parameters;
  signed long keyword_lengths[10];
  signed long data_lengths[10];

/*********************************************************************/
/* Variable data type constants                                      */
/*********************************************************************/
  static char char_data_type[] = DSQ_VARIABLE_CHAR;
  static char int_data_type[]  = DSQ_VARIABLE_FINT;

/*********************************************************************/
/* Keyword parameter and value for START command                     */
/*********************************************************************/
  static char start_keywords[] = "DSQSCMD";
  static char start_keyword_values[] = "USERCMD1";

/*********************************************************************/
/* Keyword parameter and value for SET command                       */
/*********************************************************************/
  #define SIZE_VAL 8
  char set_keywords[3][SIZE_VAL];
  signed long set_values[3];

/*********************************************************************/
/* Start a Query Interface Session                                   */
/*********************************************************************/
  number_of_parameters = 1;
  command_length = sizeof(start_query_interface);
  keyword_lengths[0] = sizeof (start_keywords);
  data_lengths[0] = sizeof(start_keyword_values);
  dsqcice(&communication_area,
          &command_length,
          &start_query_interface[0],
          &number_of_parameters,
          &keyword_lengths[0],
          &start_keywords[0],
          &data_lengths[0],
          &start_keyword_values[0],
          '_data_type[0]);
```

Figure 78. QMF Interface Example (Part 2 of 3)

```
/********************************************************************/
/* Set numeric values into query using SET command                  */
/********************************************************************/
  number_of_parameters = 3;
  command_length = sizeof(set_global_variables);
  strcpy(set_keywords[0],"MYVAR01");
  strcpy(set_keywords[1],"SHORT");
  strcpy(set_keywords[2],"MYVAR03");
  keyword_lengths[0] = SIZE_VAL;
  keyword_lengths[1] = SIZE_VAL;
  keyword_lengths[2] = SIZE_VAL;
  data_lengths[0] = sizeof(long);
  data_lengths[1] = sizeof(long);
  data_lengths[2] = sizeof(long);
  set_values[0] = 20;
  set_values[1] = 40;
  set_values[2] = 84;
  dsqcice(&communication_area,
          &command_length,
          &set_global_variables[0],
          &number_of_parameters,
          &keyword_lengths[0],
          &set_keywords[0],
          &data_lengths[0],
          &set_values[0],
          &int_data_type[0]);

/********************************************************************/
/* Run a Query                                                      */
/********************************************************************/
  command_length = sizeof(run_query);
  dsqcic(&communication_area, &command_length, &run_query[0]);
/********************************************************************/
/* Print the results of the query                                   */
/********************************************************************/
  command_length = sizeof(print_report);
  dsqcic(&communication_area, &command_length, &print_report[0]);

/********************************************************************/
/* End the query interface session                                  */
/********************************************************************/
  command_length = sizeof(end_query_interface);
  dsqcic(&communication_area, &command_length,
         &end_query_interface[0]);

  exit(0);
}
```

Figure 78. QMF Interface Example (Part 3 of 3)

Chapter 28. Using DB2

Both LE/VSE and LE/VSE C Run-Time provide an interface to the DB2 Server for VSE & VM. If you are using the previous Structured Query Language/Data System (SQL/DS), the terms mentioned in this section are also valid. Refer to “Where to Find More Information” on page xxi for a list of books describing the DB2 Server.

An application program requests DB2 services using DB2 statements imbedded in the program. The DB2 preprocessor translates imbedded DB2 statements into host language statements that perform assignments and call a database language interface module.

The DB2 Server processes a request and then returns to the application. Any errors occurring during database processing are handled by the database product.

If a program is terminated, the DB2 server takes appropriate action depending on the nature of termination.

The DB2 translator does not recognize the LE/VSE C Run-Time’s support for alternate locales/codepages. Therefore, all DB2 LE/VSE C Run-Time code should be written in codepage IBM1047 (APL293).

Example

The following program creates a DB2 table called `CTAB1`, inserts values in the table, and then drops the table. You must run the program through a DB2 preprocessor, and then you can compile and link it like a regular C program.

EDCXGDB4

```
 /* EDCXGDB4
    This example demonstrates how to use DB2 with C
  */

#include <string.h>
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;
```

Figure 79. LE/VSE C Run-Time Program Using DB2 (Part 1 of 3)

```
int main(void)
{

    if (CreaTab() == -1)
    {
       printf("Test Failed in table-creation.\n");
       exit(-1);
    }

    if (DropTab() == -1)
    {
       printf("Test Failed in table-dropping.\n");
       exit(-1);
    }
    printf("Test Successful.\n");
    return(0);
}

 /*
  * This routine creates the table CTAB1 and inserts some values
  * into it
 */

int CreaTab(void)
{

    EXEC SQL CREATE TABLE CTAB1
             ( EMPNO    CHAR(6) NOT NULL,
               FIRSTNME VARCHAR(12) NOT NULL,
               LASTNME  VARCHAR(15) NOT NULL,
               WORKDEPT CHAR(3) NOT NULL,
               PHONENO  CHAR(7),
               EDUCLVL  SMALLINT,
               SALARY   FLOAT(21) ) IN DATABASE DSNUCOMP;

    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "creation of CTAB1, received %d\n",sqlca.sqlcode);
       return(-1);
    }
```

Figure 79. LE/VSE C Run-Time Program Using DB2 (Part 2 of 3)

```
 /*  Now insert some values into the table */

    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '097892','John','Adams','003','8883945',3,29500.00 );
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '000002','Joe','Smith','004','8883791',NULL,25500.00 );
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '043929','Ralph','Holland','001','8888734',1,NULL);
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '000010','Holly','Waters','001','8884590',3,29550.00 );

    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "insert into tables, received %d\n",sqlca.sqlcode);
       return(-1);
    }
    return(0);
}

 /*
  *  This routine will drop the table.
 */

int DropTab(void)
{
    EXEC SQL DROP TABLE CTAB1;
    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "drop of CTAB1 received %d??\n",sqlca.sqlcode);
       return(-1);
    }
    EXEC SQL COMMIT WORK;
    return(0);
}
```

Figure 79. LE/VSE C Run-Time Program Using DB2 (Part 3 of 3)

Part 6. Internationalization: Locales and Character Sets

This part describes internationalization and provides information on IBM Language Environment for VSE/ESA support for internationalization.

Chapter 29. Introduction to Locale

Internationalization in Programming Languages

Internationalization in programming languages is a concept that comprises *externally stored cultural data*, a set of *programming tools* to create such cultural data, a set of *programming interfaces* to access this data, and a set of *programming methods* that enable you to use provided interfaces to write programs that do not make any assumptions about the cultural environments they run in. Such programs modify their behavior according to the user's cultural environment, specified during the program's execution.

Elements of Internationalization

The typical elements of cultural environment are as follows:

Native language
The text that the executing program uses to communicate with a user or environment, that is, the natural language of the end user.

Character sets and coded character sets
Maps an alphabet, the characters used in a particular language, onto the set of hexadecimal values (code points) that uniquely identify each character. This mapping creates the coded character set, which is uniquely identified by the character set it encodes, the set of code point values, and the mapping between these two. EBCDIC coded character set IBM-273 (Germany Country Extended Code Page) and EBCDIC coded character set IBM-293 (APL-USA) are examples of two different character sets mapped onto the same set of code points. EBCDIC IBM-273 and ASCII IBM-1047 are examples of the same character set mapped differently onto the same range of code points.

Collating and ordering
The relative ordering of characters used for sorting.

Character classification
Determines the type of character (alphabetic, numeric, and so forth) represented by a code point.

Character case conversion
Defines the mapping between uppercase and lowercase characters within a single character set.

Date and time format
Defines the way date and time data (names of weekdays and months; order of month, day, and year, and so forth) are formatted.

Format of numeric and non-numeric numbers
Defines the way numbers and monetary units are formatted with commas, decimal points, and so forth.

LE/VSE C Run-Time Support for Internationalization

The LE/VSE C Run-Time support of internationalization is based on the IEEE POSIX P1003.2 and X/Open Portability Guide standards for global locales and coded character set conversion, with the following exceptions:

- Collating symbols and collating elements are not supported in one-to-many mapping in the LC_COLLATE category of the charmap file.
- The grouping arguments in the LC_NUMERIC and LC_MONETARY categories must be strings, not sets of integers.
- The use of the ellipsis (...) in the LC_COLLATE category is limited.

See "Using the charmap File" on page 321 for more information about charmap files.

Locales and Localization

A *locale* is a collection of data that encodes information about the cultural environment. *Localization* is an action that establishes the cultural environment for an application by selecting the active locale. Only one locale can be active at one time, but a program can change the active locale at any time during its execution. The active locale affects the behavior on the locale-sensitive interfaces for the entire program. This is called the *global locale model*.

Locale-Sensitive Interfaces

The LE/VSE C Run-Time library products provide many interfaces to manipulate and access locales. You can use these interfaces to write internationalized C programs.

This list summarizes all the LE/VSE C Run-Time library functions which affect or are affected by the current locale.

Selecting locale
Changing the characteristics of the user's cultural environment by changing the current locale: `setlocale()`

Querying locale
Retrieving the locale information that characterizes the user's cultural environment:

Monetary and numeric formatting conventions:
&localeconv

Date and time formatting conventions:
`localdtconv()`

User-specified information:
`nl_langinfo()`

Encoding of the variant part of the portable character set:
`getsyntx()`

Character set identifier:
`csid()`, `wcsid()`

Classification of characters:

Single-byte characters:
`isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`

Wide characters:
`iswalnum()`, `iswalpha()`, `iswblank()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `wctype()`, `iswctype()`

Character case mapping:

Single-byte characters:
`tolower()`, `toupper()`

Wide characters:
`towlower()`, `towupper()`

Multibyte character and multibyte string conversion:
`mblen()`, `mbrlen()`, `mbtowc()`, `mbrtowc()`, `wctomb()`, `wcrtomb()`, `mbstowcs()`, `mbsrtowcs()`, `wcstombs()`, `wcsrtombs()`, `mbsinit()`, `wctob()`

String conversions to arithmetic:
`strtod()`, `wcstod()`, `strtol()`, `wcstol()`, `strtoul()`, `wcstoul()`, `atof()`, `atoi()`, `atol()`

String collating:
`strcoll()`, `strxfrm()`, `wcscoll()`, `wcsxfrm()`

Character display width:
`wcswidth()`, `wcwidth()`

Date, time, and monetary formatting:
`strftime()`, `strptime()`, `wcsftime()`, `mktime()`, `ctime()`, `gmtime()`, `localtime()` `strfmon()`

Formatted input/output:
`printf()` (and family of functions), `scanf()` (and family of functions), `vswprintf()`, `swprintf()`, `swscanf()`

Processing regular expressions:
`regcomp()`, `regexec()`

Wide character unformatted input/output:
`fgetwc()`, `fgetws()`, `fputwc()`, `fputws()`, `getwc()`, `getwchar()`, `putwc()`, `putwchar()`, `ungetwc()`

Response matching:
`rpmatch()`

Collating elements:
`ismccollel()`, `strtocoll()`, `colltostr()`, `collequiv()`, `collrange()`, `collorder()`, `cclass()`, `maxcoll()`, `getmccoll()`, `getwmccoll()`

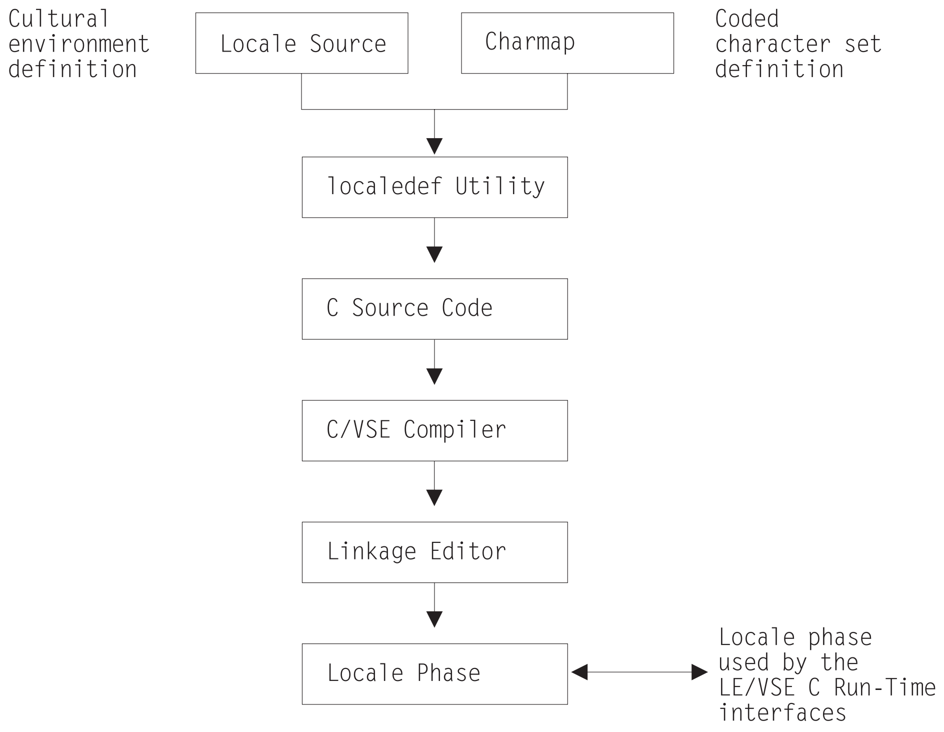
Chapter 30. Building a Locale

Cultural information is encoded in the locale source file using the locale definition language. One locale source file characterizes one cultural environment. See Appendix D, "Locales Supplied with LE/VSE C Run-Time," on page 431 for a list of the locale source and object files supplied with the LE/VSE C Run-Time.

The locale source file is processed by the locale compilation tool, called the `localedef` tool.

To enhance portability of the locale source files, certain information related to the character sets can be encoded using the symbolic names of characters. The mapping between the symbolic names and the characters they represent and its associated hexadecimal value is defined in the *character set description file* or charmap file. See Appendix E, "Charmap Files Supplied with LE/VSE C Run-Time," on page 443 for a list of the charmap files shipped with your product.

The conceptual model of the locale build process is presented below:


Cultural environment definition
Locale Source
Charmap
Coded character set definition
localedef Utility
C Source Code
C/VSE Compiler
Linkage Editor
Locale Phase
Locale phase used by the LE/VSE C Run-Time interfaces

Using the charmap File

The charmap file defines a mapping between the symbolic names of characters and the hexadecimal values associated with the character in a given coded character set. Optionally, it can provide the alternate symbolic names for characters. Characters in the locale source file can be referred to by their symbolic names or alternate symbolic names, thereby allowing for writing generic locale source files independent of the encoding of the character set they represent.

Each charmap file must contain at least the definition of the portable character set and the character symbolic names associated with each character. The characters in the portable character set and the corresponding symbolic names, and optional alternate symbolic names, are defined in Table 45.

Table 45. Characters in Portable Character Set and Corresponding Symbolic Names

Symbolic Name	Alternate Name	Character	Hex Value
<NUL>			00
<tab>	<SE10>		05
<vertical-tab>	<SE12>		0b
<form-feed>	<SE13>		0c
<carriage-return>	<SE14>		0d
<newline>	<SE11>		15
<backspace>	<SE09>		16
<alert>	<SE08>		2f
<space>	<SP01>		40
<period>	<SP11>	.	4b
<less-than-sign>	<SA03>	<	4c
<left-parenthesis>	<SP06>	(	4d
<plus-sign>	<SA01>	+	4e
<ampersand>	<SM03>	&	50
<right-parenthesis>	<SP07>	)	5d
<semicolon>	<SP14>	;	5e
<hyphen>	<SP10>	-	60
<hyphen-minus>	<SP10>	-	60
<slash>	<SP12>	/	61
<solidus>	<SP12>	/	61
<comma>	<SP08>	,	6b
<percent-sign>	<SM02>	%	6c
<underscore>	<SP09>	_	6d
<low-line>	<SP09>	_	6d
<greater-than-sign>	<SA05>	>	6e
<question-mark>	<SP15>	?	6f
<colon>	<SP13>	:	7a
<apostrophe>	<SP05>	'	7d
<equals-sign>	<SA04>	=	7e
<quotation-mark>	<SP04>	"	7f
<a>	<LA01>	a	81
<b>	<LB01>	b	82
<c>	<LC01>	c	83
<d>	<LD01>	d	84
<e>	<LE01>	e	85
<f>	<LF01>	f	86

Table 45. Characters in Portable Character Set and Corresponding Symbolic Names (continued)

Symbolic Name	Alternate Name	Character	Hex Value
<g>	<LG01>	g	87
<h>	<LH01>	h	88
<i>	<LI01>	i	89
<j>	<LJ01>	j	91
<k>	<LK01>	k	92
<l>	<LL01>	l	93
<m>	<LM01>	m	94
<n>	<LN01>	n	95
<o>	<LO01>	o	96
<p>	<LP01>	p	97
<q>	<LQ01>	q	98
<r>	<LR01>	r	99
<s>	<LS01>	s	a2
<t>	<LT01>	t	a3
<u>	<LU01>	u	a4
<v>	<LU01>	v	a5
<w>	<LW01>	w	a6
<x>	<LX01>	x	a7
<y>	<LY01>	y	a8
<z>	<LZ01>	z	a9
<A>	<LA02>	A	c1
<B>	<LB02>	B	c2
<C>	<LC02>	C	c3
<D>	<LD02>	D	c4
<E>	<LE02>	E	c5
<F>	<LF02>	F	c6
<G>	<LG02>	G	c7
<H>	<LH02>	H	c8
<I>	<LI02>	I	c9
<J>	<LJ02>	J	d1
<K>	<LK02>	K	d2
<L>	<LL02>	L	d3
<M>	<SM02>	M	d4
<N>	<LN02>	N	d5
<O>	<LO02>	O	d6
<P>	<LP02>	P	d7
<Q>	<LQ02>	Q	d8
<R>	<LR02>	R	d9

Table 45. Characters in Portable Character Set and Corresponding Symbolic Names (continued)

Symbolic Name	Alternate Name	Character	Hex Value
`<S>`	`<LS02>`	S	e2
`<T>`	`<LT02>`	T	e3
`<U>`	`<LU02>`	U	e4
`<V>`	`<LV02>`	V	e5
`<W>`	`<LW02>`	W	e6
`<X>`	`<LX02>`	X	e7
`<Y>`	`<LY02>`	Y	e8
`<Z>`	`<LZ02>`	Z	e9
`<zero>`	`<ND10>`	0	f0
`<one>`	`<ND01>`	1	f1
`<two>`	`<ND02>`	2	f2
`<three>`	`<ND03>`	3	f3
`<four>`	`<ND04>`	4	f4
`<five>`	`<ND05>`	5	f5
`<six>`	`<ND06>`	6	f6
`<seven>`	`<ND07>`	7	f7
`<eight>`	`<ND08>`	8	f8
`<nine>`	`<ND09>`	9	f9
`<vertical-line>`	`<SM13>`	|	(4f)
`<exclamation-mark>`	`<SP02>`	!	(5a)
`<dollar-sign>`	`<SC03>`	$	(5b)
`<circumflex>`	`<SD15>`	^	(5f)
`<circumflex-accent>`	`<SD15>`	^	(5f)
`<grave-accent>`	`<SD13>`		(79)
`<number-sign>`	`<SM01>`	#	(7b)
`<commercial-at>`	`<SM05>`	@	(7c)
`<tilde>`	`<SD19>`		(a1)
`<left-square-bracket>`	`<SM06>`	[	(ad)
`<right-square-bracket>`	`<SM08>`	]	(bd)
`<left-brace>`	`<SM11>`	{	(c0)
`<left-curly-bracket>`	`<SM11>`	{	(c0)
`<right-brace>`	`<SM14>`	}	(d0)
`<right-curly-bracket>`	`<SM14>`	}	(d0)
`<backslash>`	`<SM07>`	\	(e0)
`<reverse-solidus>`	`<SM07>`	\	(e0)

The portable character set is the basis for the syntactic and semantic processing of the `localedef` tool, and for most of the utilities and functions that access the locale object files. Therefore, the portable character set must always be defined. It is conceptually divided into two parts:

Invariant
Characters for which encoding must be constant among all charmap files. The required encoded values are specified in Table 45 on page 322. If any of these values change, the behavior of any locale-sensitive tool or interface on LE/VSE C Run-Time is unpredictable.

Variant
Characters for which encoding may vary from one charmap file to another. Only the following characters are allowed in this group:

```
<backslash>
<right-brace>
<left-brace>
<right-square-bracket>
<left-square-bracket>
<circumflex>
<tilde>
<exclamation-mark>
<number-sign>
<vertical-line>
<dollar-sign>
<commercial-at>
<grave-accent>
```

The default encoding of each variant character is shown by a hexadecimal value in parentheses in Table 45 on page 322.

The charmap file is divided into two main sections:

1. The charmap section, or `CHARMAP`
2. The character set identifier section, or `CHARSETID`

The following definitions can precede the two sections listed above. Each consists of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more `<blank>`s, followed by the value to be assigned to the symbol.

`<code_set_name>`
The string literal containing the name of the coded character set (IBM-1047, IBM-273, etc.)

`<mb_cur_max>`
The maximum number of bytes in a multibyte character which can be set to a value of either 1 or 4. If it is 1, each character in the character set defined in this charmap is encoded by a one-byte value. If it is 4, each character in the character set defined in this charmap is encoded by a one-, two-, three-, or four-byte value. If it is not specified, the default value of 1 is assumed. If a value of other than 1 or 4, is specified, a warning message is issued and the default value of 1 is assumed.

`<mb_cur_min>`
The minimum number of bytes in a multibyte character. Can be set to 1 only. If a value of other than 1 is specified, a warning message is issued and the default value of 1 is assumed.

`<escape_char>`
Specifies the escape character that is used to specify hexadecimal or octal

notation for numeric values. It defaults to the hexadecimal value `0xe0`, which represents the \ character in the coded character set IBM-1047.

For portability among the EBCDIC based systems, the escape character has been redefined to the / or `<slash>` character in all IBM-supplied charmap files, with the following statement:

```
<escape_char>  /
```

`<comment_char>`
Denotes the character chosen to indicate a comment within a charmap file. It defaults to the hexadecimal value `0x7b`, which represents the # character in the coded character set IBM-1047.

For portability among the EBCDIC based systems, the comment character has been redefined to the % or `<percent-sign>` character in all IBM-supplied charmap files, with the following statement:

```
<comment_char> %
```

`<shift_out>`
Specifies the value of the shift-out control character that is prepended to each double-byte value indicating the EBCDIC multibyte character. If specified, it must be the value of the EBCDIC shift-out (SO) character (hexadecimal value `0x0e`). It is ignored if the `<mb_cur_max>` value is 1.

`<shift_in>`
Specifies the value of the shift-in control character that is appended to each double-byte value indicating the EBCDIC multibyte character. If specified, it must be the value of the EBCDIC shift-in (SI) character (hexadecimal value `0x0f`). It is ignored if the `<mb_cur_max>` value is 1.

The CHARMAP Section

The `CHARMAP` section defines the values for the symbolic names representing characters in the coded character set. Each charmap file must define at least the portable character set. The character symbolic names or alternate symbolic names (or both) must be used to define the portable character set. These are shown in Table 45 on page 322.

Additional characters can be defined by the user with symbolic character names.

The `CHARMAP` section starts with the line containing the keyword `CHARMAP`, and ends with the line containing the keywords `END CHARMAP`. `CHARMAP` and `END CHARMAP` must both start in column one.

The character set mapping definitions are all the lines between the first and last lines of the `CHARMAP` section.

The formats of the character set mappings for this section are as follows:

```
"%s %s %s\n", <symbolic-name>, <encoding>, <comments>
"%s...%s %s %s\n", <symbolic-name>, <symbolic-name>, <encoding>, <comments>
```

The first format defines a single symbolic name and a corresponding encoding. A symbolic name is one or more characters with visible glyphs, enclosed between angle brackets.

For reasons of portability, a symbolic name should include only the characters from the invariant part of the portable character set. If you use variant characters or decimal or hexadecimal notation in a symbolic name, the symbolic name will not be portable. A character following an escape character is interpreted as itself; for example, the sequence <\\\>> represents the symbolic name \> enclosed within angle brackets, where the backslash (\) is the escape character. If / is the escape character, the sequence <///>> represents the symbolic name />. In the supplied charmap files, the escape character has been redefined to the forward slash (/).

The second format defines a group of symbolic names associated with a range of values. The two symbolic names are comprised of two parts, a prefix and suffix. The prefix consists of zero or more non-numeric invariant visible glyph characters and is the same for both symbolic names. The suffix consists of a positive decimal integer. The suffix of the first symbolic name must be less than or equal to the suffix of the second symbolic name. As an example, `<j0101>...<j0104>` is interpreted as the symbolic names `<j0101>,<j0102>,<j0103>,<j0104>`. The common prefix is `'j'` and the suffixes are `'0101'` and `'0104'`.

The encoding part can be written in one of two forms:

```
<escape-char><number>                       (single-byte value)
<escape-char><number><escape-char><number>  (double-byte value)
```

The number can be written using octal, decimal, or hexadecimal notation. Decimal numbers are written as a `'d'` followed by 2 or 3 decimal digits. Hexadecimal numbers are written as an `'x'` followed by 2 hexadecimal digits. An octal number is written with 2 or 3 octal digits. As an example, the single-byte value `x1F` could be written as `'\37'`, `'\x1F'`, or `'\d31'`. The double-byte value of `x1A1F` could be written as `'\13\17'`, `'\x1A\x1F'`, or `'\d10\d15'`.

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent names defined by the range have encoding values in increasing order.

When constants are concatenated for multibyte character values, they must be of the same type, and are interpreted in byte order from first to last with the least significant byte of the multibyte character specified by the last constant. Each value is then prepended by the byte value of `<shift_out>` and appended with the byte value of `<shift_in>`. Such a string represents one EBCDIC multibyte character. For example:

```
<escape_char>   /
<comment_char>  %
<mb_cur_max>    4
<mb_cur_min>    1
<shift-out>     /x0e
<shift-in>      /x0f
CHARMAP
% many definition lines
<j0101>...<j0104>           /d129/d254
%many definition lines
END CHARMAP
```

is interpreted as

```
<j0101>                     /d129/d254
<j0102>                     /d129/d255
<j0103>                     /d130/d0
<j0104>                     /d130/d1
```

It produces four 4-byte long multibyte EBCDIC characters:

```
<j0101>                 x0Ex81xFEx0F
<j0102>                 x0Ex81xFFx0F
<j0103>                 x0Ex82x00x0F
<j0104>                 x0Ex82x01x0F
```

The CHARSETID Section

The character set identifier section of the charmap file maps the symbolic names defined in the CHARMAP section to a character set identifier.

Note: The two functions csid() and wcsid() query the locales and return the character set identifier for a given character. This information is not currently used by any other library function.

The CHARSETID section starts with a line containing the keyword CHARSETID, and ends with the line containing the keywords END CHARSETID. Both CHARSETID and END CHARSETID must begin in column 1. The lines between the first and last lines of the CHARSETID section define the character set identifier for the defined coded character set.

The character set identifier mappings are defined as follows:

```
"%s %c", <symbolic-name>, <value>
"%c %c", <value>, <value>
"%s...%s %c", <symbolic-name>, <symbolic-name>, <value>
"%c...%c %c", <value>, <value>, <value>
"%s...%c %c", <symbolic-name>, <value>, <value>
"%c...%s %c", <value>, <symbolic-name>, <value>
```

The individual characters are specified by the symbolic name or the value. The group of characters are specified by two symbolic names or by two numeric values (or combination) separated by an ellipsis (...). The interpretation of ranges of values is the same as specified in the CHARMAP section. The character set identifier is specified by a numeric value.

For example:

```
<comment_char>        %
<escape_char>         /
<code_set_name>       "IBM-930"
<mb_cur_max>          4
<mb_cur_min>          1
<shift_out>           /x0e
<shift_in>            /x0f

%
%       CHARMAP
%

CHARMAP
...
<j0110>                             /x42/x5a
<j0111>...<j0112>                /x43/xbe
<judc2001>...<judc2094>          /x72/x8d
...
END CHARMAP

%
%       CHARSETID
%

CHARSETID
...
<j0110>                              1
```

```
<j0111>...<j0112>                   1
<judc2001>...<judc2094>             3
...
END CHARSETID
```

Locale Source Files

Locales are defined through the specification of a locale definition file. The locale definition contains one or more distinct locale category source definitions and not more than one definition of any category. Each category controls specific aspects of the cultural environment. A category source definition is either the explicit definition of a category or the `copy` directive, which indicates that the category definition should be copied from another locale definition file.

The definition file is composed of an optional definition section for the escape and comment characters to be used, followed by the category source definitions. Comment lines and blank lines can appear anywhere in the locale definition file. If the escape and comment characters are not defined, default code points are used (`xE0` for the escape character and `x7B` for the comment character, respectively). The definition section consists of the following optional lines:

```
escape_char     <character>
comment_char    <character>
```

where `<character>` in both cases is a single-byte character to be used, for example:

```
escape_char     /
```

defines the escape character in this file to be `'/'` (the `<slash>` character).

Locale definition files passed to the `localedef` utility are assumed to be in coded character set IBM-1047.

To ensure portability among EBCDIC systems, you should redefine these characters to characters from the invariant part of the portable character set. The suggested redefinition is:

```
      escape_char    /
      comment_char   %
```

This suggested redefinition is used in all locale definition files supplied by IBM. For reasons of portability, you should use the suggested redefinition in all your customized locale definition files. See Chapter 31, “Customizing a Locale,” on page 359 for information about customizing locales. These two redefinitions should be placed in the first lines of the locale definition source file, before any of the redefined characters are used.

Each category source definition consists of a category header, a category body, and a category trailer, in that order.

Category Header

Consists of the keyword naming the category. Each category name starts with the characters LC_. The following category names are supported: LC_CTYPE, LC_COLLATE, LC_NUMERIC, LC_MONETARY, LC_TIME, LC_MESSAGES, LC_TOD, and LC_SYNTAX.

The LC_TOD and LC_SYNTAX categories, if present, must be the last two categories in the locale definition file.

Category Body
Consists of one or more lines describing the components of the category. Each component line has the following format:

```
<identifer>   <operand1>
<identifer>   <operand1>;<operand2>;...;<operandN>
```

`<identifier>` is a keyword that identifies a locale element, or a symbolic name that identifies a collating element. `<operand>` is a character, collating element, or string literal. Escape sequences can be specified in a string literal using the `<escape_character>`. If multiple operands are specified, they must be separated by semicolons. White space can be before and after the semicolons.

Category Trailer
Consists of the keyword `END` followed by one or more `<blank>`s and the category name of the corresponding category header.

Here is an example of locale source containing the header, body, and trailer:

```
escape_char   /
comment_char  %
%
% Here is a simple locale definition file consisting of one
% category source definition, LC_CTYPE.
%
LC_CTYPE
upper <A>;...;<Z>
END LC_CTYPE
```

You do not have to define each category. Where category definitions are absent from the locale source, default definitions are used.

In each category the keyword `copy` followed by a string specifies the name of an existing locale to be used as the source for the definition of this category. If the locale is not found, an error is reported and no locale output is created.

You can continue a line in a locale definition file by placing an escape character as the last character on the line. This continuation character is discarded from the input. Even though there is no limitation on the length of each line, for portability reasons it is suggested that each line be no longer than 2048 characters (bytes). There is no limit on the accumulated length of a continued line. You cannot continue comment lines on a subsequent line by using an escaped `<newline>`.

Individual characters, characters in strings, and collating elements are represented using symbolic names, as defined below. Characters can also be represented as the characters themselves, or as octal, hexadecimal, or decimal constants. If you use non-symbolic notation, the resultant locale definition file may not be portable among systems and environments. The left angle bracket (<) is a reserved symbol, denoting the start of a symbolic name; if you use it to represent itself, you must precede it with the escape character.

The following rules apply to the character representation:

1. A character can be represented by a symbolic name, enclosed within angle brackets. The symbolic name, including the angle brackets, must exactly match a symbolic name defined in the charmap file. The symbolic name is replaced by the character value determined from the value associated with the symbolic name in the charmap file.

The use of a symbolic name not found in the charmap file constitutes an error, unless the name is in the category LC_CTYPE or LC_COLLATE, in which case it constitutes a warning. Use of the escape character or right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character. For example:

`<c>;<c-cedilla>`
: specifies two characters whose symbolic names are `"c"` and `"c-cedilla"`

`"<M><a><y>"`
: specifies a 3-character string composed of letters represented by symbolic names `"M"`, `"a"`, and `"y"`

`"<a><\>>"`
: specifies a 2-character string composed of letters represented by symbolic names `"a"` and `">"` (assuming the escape character is `\`)

If the character represented by the symbolic name is a multibyte character defined by 2 byte values in the charmap file, and the shift-out and shift-in characters are defined, the value is enclosed within shift-out and shift-in characters before the `localedef` utility processes it any further.

2. A character can represent itself. Within a string, the double quotation mark, the escape character, and the left angle bracket must be escaped (preceded by the escape character) to be interpreted as the characters themselves. For example:

   `c` `'c'` character represented by itself

   `"may"` represents a 3-character string, each character within the string represented by itself

   `"%%%"%>"`
   : represents the three character long string `"%">"`, where the escape character is defined as `%`

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value.

   For example:

   `\131 "\212\129\168" \16\66\193\17`

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character, followed by an `x`, followed by two or more hexadecimal digits. Each constant represents a byte value.

   Example: `\x83 "\xD4\x81\xA8"`

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a `d` followed by two or more decimal digits. Each constant represents a byte value.

   Example: `\d131 "\d212\d129\d168" \d14\d66\d193\d15`

For multibyte characters, the entire encoding sequence, including the shift-out and shift-in characters, must be present. Otherwise, the sequence of bytes not enclosed between the shift-out and shift-in characters are interpreted as a sequence of single byte characters.

Multibyte characters can be represented by concatenating constants specified in byte order with the last constant specifying the least significant byte of the character. If the sequence of octal, hexadecimal, or decimal constants is to represent a multibyte character, it must be enclosed in shift-out and shift-in constants.

Example: `\x0e\x42\xC1\x0f`

LC_CTYPE Category

This category defines character classification, case conversion, and other character attributes. In this category, you can represent a series of characters by using three adjacent periods as an ellipsis symbol (`...`). An ellipsis is interpreted as including all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value following the ellipsis.

An ellipsis is valid within a single encoded character set.

For example, `\x30;...;\x39;` includes in the character class all characters with encoded values from X'30' to X'39'.

The keywords recognized in the LC_CTYPE category are listed below. In the descriptions, the term "automatically included" means that it is not an error either to include or omit any of the referenced characters; they are assumed by default even if the entire keyword is missing and accepted if present. If a keyword is specified without any arguments, the default characters are assumed.

When a character is automatically included, it has an encoded value dependent on the charmap file in effect. If no charmap file is specified, the encoding of the encoded character set IBM-1047 is assumed.

`copy` Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keywords are present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`upper` Defines characters to be classified as uppercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The uppercase letters `A` through `Z` are automatically included in this class.

The `isupper()` and `iswupper()` functions test for any character and wide character, respectively, included in this class.

`lower` Defines characters to be classified as lowercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The lowercase letters `a` through `z` are automatically included in this class.

The `islower()` and `iswlower()` functions test for any character and wide character, respectively, included in this class.

`alpha` Defines characters to be classified as letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. Characters classified as either `upper` or `lower` are automatically included in this class.

The `isalpha()` and `iswalpha()` functions test for any character or wide character, respectively, included in this class.

`digit` Defines characters to be classified as numeric digits. Only the digits `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` can be specified. If they are, they must be in contiguous ascending sequence by numerical value. The digits `0` through `9` are automatically included in this class.

The `isdigit()` and `iswdigit()` functions test for any character or wide character, respectively, included in this class.

`space` Defines characters to be classified as whitespace characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, or `xdigit` can be specified for `space`. The characters `<space>`, `<form-feed>`, `<newline>`, `<carriage-return>`, `<horizontal-tab>`, and `<vertical-tab>`, and any characters defined in the class `blank` are automatically included in this class.

The functions `isspace()` and `iswspace()` test for any character or wide character, respectively, included in this class.

`cntrl` Defines characters to be classified as control characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, `punct`, `graph`, `print`, or `xdigit` can be specified for `cntrl`.

The functions `iscntrl()` and `iswcntrl()` test for any character or wide character, respectively, included in this class.

`punct` Defines characters to be classified as punctuation characters. No character defined for the keywords `upper`, `lower`, `alpha`, `digit`, `cntrl`, or `xdigit`, or as the `<space>` character, can be specified.

The functions `ispunct()` and `iswpunct()` test for any character or wide character, respectively, included in this class.

`graph` Defines characters to be classified as printing characters, not including the `<space>` character. Characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, and `punct` are automatically included. No character specified in the keyword `cntrl` can be specified for `graph`.

The functions `isgraph()` and `iswgraph()` test for any character or wide character, respectively, included in this class.

`print` Defines characters to be classified as printing characters, including the `<space>` character. Characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, `punct`, and the `<space>` character are automatically included. No character specified in the keyword `cntrl` can be specified for `print`.

The functions `isprint()` and `iswprint()` test for any character or wide character, respectively, included in this class.

`xdigit` Defines characters to be classified as hexadecimal digits. Only the characters defined for the class `digit` can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits `10` through `15`, with each set in ascending order (for example, `A, B, C, D, E, F, a, b, c, d, e, f`). The digits `0` through `9`, the uppercase letters `A` through `F`, and the lowercase letters `a` through `f` are automatically included in this class.

The functions `isxdigit()` and `iswxdigit()` test for any character or wide character, respectively, included in this class.

`blank` Defines characters to be classified as blank characters. The characters `<space>` and `<tab>` are automatically included in this class.

The functions `isblank()` and `iswblank()` test for any character or wide character, respectively, included in this class.

`toupper`
Defines the mapping of lowercase letters to uppercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed in parentheses. The first character in each pair is the lowercase letter, and the

second is the corresponding uppercase letter. Only characters specified for the keywords lower and upper can be specified for toupper. The lowercase letters a through z and their corresponding uppercase letters A through Z are automatically in this mapping, but only when the toupper keyword is omitted from the locale definition.

It affects the behavior of the toupper() and towupper() functions for mapping characters and wide characters, respectively.

tolower
Defines the mapping of uppercase letters to lowercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed by parentheses. The first character in each pair is the uppercase letter, and the second is its corresponding lowercase letter. Only characters specified for the keywords lower and upper can be specified. If the tolower keyword is omitted from the locale definition, the mapping is the reverse mapping of the one specified for the toupper.

The tolower keyword affects the behavior of the tolower() and towlower() functions for mapping characters and wide characters, respectively.

You may define additional character classes using your own keywords. A maximum of 31 classes are supported in total: the 12 standard classes, and up to 29 user-defined classes.

The defined classes affect the behavior of the wctype() and iswctype() functions.

Here is an example of the definition of the LC_CTYPE category:

```
escape_char          /
comment_char         %

%%%%%%%%%%%%
LC_CTYPE
%%%%%%%%%%%%
% upper letters are A-Z by default plus the three defined below
upper   <A-acute>;<A-grave>;<C-acute>

% lower case leters are a-z by default plus the three defined below
lower   <a-acute>;<a_grave><c-acute>

% space characters are default 6 characters plus the one defined below
space   <hyphen-minus>

cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;<NUL>;/
        <SO>;<SI>

% default graph, print,punct, digit, xdigit, blank classes

% toupper mapping defined only for the following three pairs
toupper (<a-acute),<A-acute>);/
        (<a-grave),<A-grave>);/
        (<c-acute),<C-acute>);

% default upper to lower case mapping

% user defined class
myclass  <e-ogonek>;<E-ogonek>

END LC_CTYPE
```

LC_COLLATE Category

A collation sequence definition defines the relative order between collating elements (characters and multicharacter collating elements) in the locale. This order is expressed in terms of collation values. It assigns each element one or more collation values (also known as collation weights). The collation sequence definition is used by regular expressions, pattern matching, and sorting and collating functions. The following capabilities are provided:

1. **Multicharacter collating elements.** Specification of multicharacter collating elements (sequences of two or more characters to be collated as an entity).
2. **User-defined ordering of collating elements.** Each collating element is assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching, and unless collation weights are explicitly specified, also as the collation weight to be used in sorting.
3. **Multiple weights and equivalence classes.** Collating elements can be assigned 1 to 6 collating weights for use in sorting. The first weight is referred to as the primary weight.
4. **One-to-many mapping.** A single character is mapped into a string of collating elements.
5. **Many-to-many substitution.** A string of one or more characters are mapped to another string (or an empty string). The character or characters are ignored for collation purposes.
6. **Equivalence class definition.** Two or more collating elements have the same collation value (primary weight).
7. **Ordering by weights.** When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements. Each successive pair of elements is compared according to the relative primary weights for the elements. If they are equal, and more than one weight is assigned, then the pairs of collating elements are compared again according to the relative subsequent weights, until either two collating elements are not equal or the weights are exhausted.

Collating Rules

Collation rules consist of an ordered list of collating order statements, ordered from lowest to highest. The `<NULL>` character is considered lower than any other character. The ellipsis symbol ("**...**") is a special collation order statement. It specifies that a sequence of characters collate according to their encoded character values. It causes all characters with values higher than the value of the `<collating identifier>` in the preceding line, and lower than the value for the `<collating identifier>` on the following line, to be placed in the character collation order between the previous and the following collation order statements in ascending order according to their encoded character values.

The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable among implementations.

The ellipsis symbol must be on a line by itself, *not* the first or last line, and the preceding and succeeding lines must not specify a weight.

A collating order statement describes how a collating identifier is weighted.

Each `<collating-identifier>` consists of a character, `<collating-element>`, `<collating-symbol>`, or the special symbol `UNDEFINED`. The order in which collating elements are specified determines the character order sequence, such that each collating element is considered lower than the elements following it. The <NULL> character is considered lower than any other character. Weights are expressed as characters, <collating-symbol>s, <collating-element>s, or the special symbol IGNORE. A single character, a <collating-symbol>, or a <collating-element> represents the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus, rather than assigning absolute values to weights, a particular weight is expressed using the relative "order value" assigned to a collating element based on its order in the character collation sequence.

A <collating-element> specifies multicharacter collating elements, and indicates that the character sequence specified by the <collating-element> is to be collated as a unit and in the relative order specified by its place.

A <collating-symbol> can define a position in the relative order for use in weights.

The <collating-symbol> UNDEFINED is interpreted as including all characters not specified explicitly. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their encoded character values. If no UNDEFINED symbol is specified, and the current coded character set contains characters not specified in this clause, the localedef utility issues a warning and places such characters at the end of the character collation order.

The syntax for a collation order statement is:

```
<collating-identifier> <weight1>;<weight2>;...;<weightn>
```

Collation of two collating identifiers is done by comparing their relative primary weights. This process is repeated for successive weight levels until the two identifiers are different, or the weight levels are exhausted. The operands for each collating identifier define the primary, secondary, and subsequent relative weights for the collating identifier. Two or more collating elements can be assigned the same weight. If two collating identifiers have the same primary weight, they belong to the same *equivalence class*.

The special symbol IGNORE as a weight indicates that when strings are compared using the weights at the level where IGNORE is specified, the collating element should be ignored, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are IGNOREd in their primary weight form an equivalence class.

All characters specified by an ellipsis are assigned unique weights, equal to the relative order of the characters. Characters specified by an explicit or implicit UNDEFINED special symbol are assigned the same primary weight (they belong to the same equivalence class).

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character "<ezset>" is given the string "<s><s>" as a weight, comparisons are performed as if all occurrences of the character <ezset> are replaced by <s><s> (assuming <s> has the collating weight <s>). If it is desirable to define <ezset> and <s><s> as an equivalence class, then a collating element must be defined for the string "ss".

If no weight is specified, the collating identifier is interpreted as itself.

For example, the order statement

```
<a>   <a>
```

is equivalent to

```
<a>
```

Collating Keywords

The following keywords are recognized in a collation sequence definition.

`copy` Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`collating-element`

Defines a collating-element symbol representing a multicharacter collating element. This keyword is optional.

In addition to the collating elements in the character set, the `collating-element` keyword can be used to define multicharacter collating elements. The syntax is:

```
"collating-element %s from %s\n", <collating-element>, <string>
```

The <`collating-element`> should be a symbolic name enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collate as an entity. A <`collating-element`> defined with this keyword is only recognized within the LC_COLLATE category.

For example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-element <ll> from "ll"
```

`collating-symbol`

Defines a collating symbol for use in collation order statements.

The `collating-symbol` keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight. This keyword is optional.

This construct can define symbols for use in collation sequence statements, between the `order_start` and `order_end` keywords.

The syntax is:

```
"collating-symbol %s\n", <collating-symbol>
```

The <`collating-symbol`> must be a symbolic name, enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. A <`collating-symbol`> defined with this keyword is only recognized within the LC_COLLATE category.

For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

substitute

The `substitute` keyword defines a substring substitution in a string to be collated. This keyword is optional. The following operands are supported with the `substitute` keyword:

`"substitute %s with %s\n", <`*`regular-expr`*`>, <`*`replacement`*`>`

The first operand is treated as a basic regular expression. The replacement operand consists of zero or more characters and regular expression back-references (for example, `\1` through `\9`). The back-references consist of the backslash followed by a digit from `1` to `9`. If the backslash is followed by two or three digits, it is interpreted as an octal constant.

When strings are collated according to a collation definition containing substitute statements, the collation behaves as if occurrences of substrings matching the basic regular expression are replaced by the replacement string, before the strings are compared based on the specified collation sequence. Ranges in the regular expression are interpreted according to the current character collation sequence and character classes according to the character classification specified by the LC_CTYPE environment variable at collation time. If more than one substitute statement is present in the collation definition, the collation process behaves as if the substitute statements are applied to the strings in the order they occur in the source definition. The substitution for the substitute statements are processed before any substitutions for one-to-many mappings. The support of the ″substitute″ keyword is an IBM LE/VSE C Run-Time extension to the POSIX standard.

order_start

Define collating rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

The `order_start` keyword must precede collation order entries. It defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the `order_start` keyword is:

`order_start <`*`sort-rule1`*`>;<`*`sort-rule1`*`>;...;<sort-rule`*`n`*`>`

The operands of the `order_start` keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one `forward` operand is assumed. If any is present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands are separated by semicolons (`;`). Each operand consists of one or more collation directives separated by commas (`,`). If the number of operands exceeds the limit of 6, the `localedef` utility issues a warning message.

The following directives are supported:

forward

Specifies that comparison operations for the weight level proceed from the start of the string towards its end.

backward
: Specifies that comparison operations for the weight level proceed from the end of the string toward its beginning.

no-substitute
: No substitution is performed, such that the comparison is based on collation values for collating elements before any substitution operations are performed.

Notes:

1. This is an IBM LE/VSE C Run-Time extension to the POSIX standard.
2. When the no-substitute keyword is specified, one-to-many mappings are ignored.

position
: Specifies that comparison operations for the weight level must consider the relative position of non-IGNOREd elements in the strings. The string containing a non-IGNOREd element after the fewest IGNOREd collating elements from the start of the comparison collates first. If both strings contain a non-IGNOREd character in the same relative position, the collating values assigned to the elements determine the order. If the strings are equal, subsequent non-IGNOREd characters are considered in the same manner.

order_end
: The collating order entries are terminated with an order_end keyword.

Here is an example of an LC_COLLATE category:

```
        LC_COLLATE
        % ARTIFICIAL COLLATE CATEGORY

        % collating elements
 1        collating-element   <ch>  from "<c><h>"
        collating-element   <Ch>  from "<C><h>"
        collating-element   <eszet> from "<s><z>"

        %collating symbols for relative order definition

        collating-symbol    <LOW>
 2        collating-symbol    <UPPER-CASE>
        collating-symbol    <LOWER-CASE>
        collating-symbol    <NONE>


 3        order_start forward;backward;forward
        <NONE>
 4        <LOW>
        <UPPER-CASE>
        <LOWER-CASE>

 5        UNDEFINED IGNORE;IGNORE;IGNORE

        <space>
 6        ....
        <quotation-mark>
 7        <a>         <a>;<NONE>;<LOWER-CASE>
10        <a-acute>   <a>;<a-acute>;<LOWER-CASE>
11        <a-grave>   <a>;<a-grave>;<LOWER-CASE>
 8        <A>         <a>;<NONE>;<UPPER-CASE>
11        <A-acute>   <a>;<a-acute>;<UPPER-CASE>
11        <A-grave>   <a>;<a-grave>;<UPPER-CASE>
11        <ch>        <ch>;<NONE>;<LOWER-CASE>
```

```
11          <Ch>          <ch>;<NONE>;<UPPER-CASE>
 9          <s>           <s>;<s>;<LOWER-CASE>
12          <eszet>       "<s><s>";"<eszet><s>";<LOWER-CASE>
 9          <z>           <z>;<NONE>;<LOWER-CASE>
          order_end
```

The example is interpreted as follows:

1. Collating elements
   - Character `<c>` followed by `<h>` collate as one entity named `<ch>`
   - Character `<C>` followed by `<h>` collate as one entity named `<Ch>`
   - Character `<s>` followed by `<z>` collate as one entity named `<eszet>`
2. Collating symbols `<LOW>`, `<UPPER-CASE>`, `<LOWER-CASE>` and `<NONE>` are defined to be used in relative order definition.
3. Up to 3 string comparisons are defined:
   - First pass starts from the beginning of the strings
   - Second pass starts from the end of the strings, and
   - Third pass starts from the beginning of the strings
4. The collating weights are defined such that
   - `<LOW>` collates before `<UPPER-CASE>`
   - `<UPPER-CASE>` collates before `<LOWER-CASE>`
   - `<LOWER-CASE>` collates before `<NONE>`
5. All characters for which collation is not specified here are ordered after `<NONE>`, and before `<space>` in ascending order according to their encoded values.
6. All characters with an encoded value larger than the encoded value of `<space>` and lower than the encoded value of `<quotation-mark>` in the current encoded character set, collate in ascending order according to their values.
7. `<a>` has a:
   - Primary weight of `<a>`
   - Secondary weight `<NONE>`
   - Tertiary weight of `<LOWER-CASE>`
8. `<A>` has a:
   - Primary weight of `<a>`
   - Secondary weight of `<NONE>`
   - Tertiary weight of `<UPPER-CASE>`
9. The weights of `<s>` and `<z>` are determined in a similar fashion to `<a>` and `<A>`.
10. `<a-acute>` has a:
    - Primary weight of `<a>`
    - Secondary weight of `<a-acute>` itself
    - Tertiary weight of `<LOWER-CASE>`
11. The weights of `<a-grave>`, `<A-acute>`, `<A-grave>`, `<ch>` and `<Ch>` are determined in a similar fashion to `<a-acute>`.
12. `<eszet>` has a:
    - Primary weight determined by replacing each occurrence of `<eszet>` with the sequence of two `<s>`'s and using the weight of `<s>`
    - Secondary weight determined by replacing each occurrence of `<eszet>` with the sequence of `<eszet>` and `<s>` and using their weights

- Tertiary weight is the relative position of `<LOWER-CASE>`

Comparison of Strings

Compare the strings `s1="aAch"` and `s2="AaCh"` using the above LC_COLLATE definition:

1. `s1=> "aA<ch>"`, and `s2=> "Aa<Ch>"`
2. First pass:
   a. Substitute the elements of the strings with their primary weights:
      `s1=> "<a><a><ch>", s2=> "<a><a><ch>"`
   b. Compare the two strings starting with the first element—they are equal.
3. Second pass:
   a. Substitute the elements of the strings with their secondary weights:
      `s1=> "<NONE><NONE><NONE>", s2=>"<NONE><NONE><NONE>"`
   b. Compare the two strings from the last element to the first—they are equal.
4. Third pass:
   a. Substitute the elements of the strings with their third level weights:
      `s1=> "<LOWER-CASE><UPPER-CASE><LOWER-CASE>", s2=> "<UPPER-CASE><LOWER-CASE><UPPER-CASE>"`
   b. Compare the two strings starting from the beginning of the strings:
      `s2` compares lower than `s1` because `<UPPER-CASE>` is before `<LOWER-CASE>`.

Compare the strings `s1="a1sz"` and `s2=>"a2ss"`:

1. `s1=> "a1<eszet>"` and `s2= "a2ss"`
2. First pass:
   a. Substitute the elements of the strings with their primary weights:
      `s1=> "<a><s><s>", s2=> "<a><s><s>"`
   b. Compare the two strings starting with the first element—they are equal.
3. Second pass:
   a. Substitute the elements of the strings with their secondary weights:
      `s1=> "<a-acute><eszet><s>", s2=>"<a-grave><s><s>"`
   b. Compare the two strings from the last element to the first—`<s>` is before `<ezset>`.

LC_MONETARY Category

This category defines the rules and symbols used to format monetary quantities. The operands are strings or integers. The following keywords are supported:

`copy` Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`int_curr_symbol`
Specifies the international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in ISO4217 *Codes for the Representation of Currency and Funds* . The fourth character is the character used to separate the international currency symbol from the monetary quantity.

If not defined, it defaults to the empty string ("").

currency_symbol
Specifies the string used as the local currency symbol. If not defined, it defaults to the empty string ("").

mon_decimal_point
The string used as a decimal delimiter to format monetary quantities.

If not defined it defaults to the empty string ("").

mon_thousands_sep
Specifies the string used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. If not defined, it defaults to the empty string ("").

mon_grouping
Defines the size of each group of digits in formatted monetary quantities. The operand is a string representing a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not -1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is -1, then no further grouping is performed. If not defined, `mon_grouping` defaults to -1 which indicates no grouping. An empty string is interpreted as -1.

positive_sign
A string used to indicate a formatted monetary quantity with a non-negative value. If not defined, it defaults to the empty string ("").

negative_sign
Specifies a string used to indicate a formatted monetary quantity with a negative value. If not defined, it defaults to the empty string ("").

int_frac_digits
Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `int_curr_symbol`. If not defined, it defaults to -1.

frac_digits
Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `currency_symbol`. If not defined, it defaults to -1.

p_cs_precedes
Specifies an integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a non-negative formatted monetary quantity, and set to `0` if the symbol succeeds the value. If not defined, it defaults to -1.

p_sep_by_space
Specifies an integer set to `0` if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a non-negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to -1.

n_cs_precedes
An integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a negative formatted monetary quantity, and set to `0` if the symbol succeeds the value. If not defined, it defaults to -1.

`n_sep_by_space`
: An integer set to `0` if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to `-1`.

`p_sign_posn`
: An integer set to a value indicating the positioning of the positive_sign for a non-negative formatted monetary quantity. The following integer values are recognized:

0	Parentheses surround the quantity and the `currency_symbol` or `int_curr_symbol`.
1	The sign string precedes the quantity and the `currency_symbol` or `int_curr_symbol`.
2	The sign string succeeds the quantity and the `currency_symbol` or `int_curr_symbol`.
3	The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.
4	The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

The following value may also be specified, though it is not part of the POSIX standard.

5	Use `debit-sign` or `credit-sign` for `p_sign_posn` or `n_sign_posn`.

If not defined, it defaults to `-1`.

`n_sign_posn`
: An integer set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity. The recognized values are the same as for `p_sign_posn`. If not defined, it defaults to `-1`.

`left_parenthesis`
: The symbol of the locale's equivalent of `(` to form a negative-valued formatted monetary quantity together with `right_parenthesis`. If not defined, it defaults to the empty string ("").

Note: This is an IBM-specific extension.

`right_parenthesis`
: The symbol of the locale's equivalent of `)` to form a negative-valued formatted monetary quantity together with `left_parenthesis`. If not defined, it defaults to the empty string ("");

Note: This is an IBM-specific extension.

`debit_sign`
: The symbol of locale's equivalent of `DB` to indicate a non-negative-valued formatted monetary quantity. If not defined, it defaults to the empty string ("");

Note: This is an IBM-specific extension.

`credit_sign`
: The symbol of locale's equivalent of `CR` to indicate a negative-valued formatted monetary quantity. If not defined, it defaults to the empty string ("");

Note: This is an IBM-specific extension.

Here is an example of the definition of the LC_MONETARY category:

```
escape_char          /
comment_char         %

%%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%%

int_curr_symbol   "<J><P><Y><space>"
currency_symbol   "<yen>"
mon_decimal_point "<period>"
mon_thousands_sep "<comma>"
mon_grouping      "3;0"
positive_sign     ""
negative_sign     "<hyphen-minus>"
int_frac_digits   0
frac_digits       0
p_cs_precedes     1
p_sep_by_space    0
n_cs_precedes     1
n_sep_by_space    0
p_sign_posn       1
n_sign_posn       1
debit_sign        "<D><B>"
credit_sign       "<C><R>"
left_parenthesis  "<left-parenthesis>"
right_parenthesis "<right-parenthesis>"

END LC_MONETARY
```

LC_NUMERIC Category

This category defines the rules and symbols used to format non-monetary numeric information. The operands are strings. The following keywords are recognized:

copy
Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

decimal_point
Specifies a string used as the decimal delimiter in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string.

thousands_sep
Specifies a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary, formatted quantities.

grouping
Defines the size of each group of digits in formatted non-monetary quantities. The operand is a string representing a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not -1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is -1, then no further grouping is performed. An empty string is interpreted as -1.

Here is an example of how to specify the LC_NUMERIC category:

```
escape_char          /
comment_char         %

%%%%%%%%%%%%
LC_NUMERIC
%%%%%%%%%%%%

decimal_point     "<comma>"
thousands_sep     "<space>"
grouping          "3;0"

END LC_NUMERIC
```

LC_TIME Category

The LC_TIME category defines the interpretation of the field descriptors used for parsing, then formatting, the date and time. The descriptors identify the replacement portion of the string, while the rest of a string is constant. The definition of descriptors is included in *LE/VSE C Run-Time Library Reference*. All these descriptors can be used in the format specifier in the time formatting function `strftime()`.

The following keywords are supported:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category.

If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

abday Defines the abbreviated weekday names, corresponding to the `%a` field descriptor. The operand consists of seven semicolon-separated strings. The first string is the abbreviated name corresponding to Sunday, the second string corresponds to Monday, and so forth.

day Defines the full weekday names, corresponding to the `%A` field descriptor. The operand consists of seven semicolon-separated strings. The first string is the full name corresponding to Sunday, the second string to Monday, and so forth.

abmon Defines the abbreviated month names, corresponding to the `%b` field descriptor. The operand consists of twelve strings separated by semicolons. The first string is an abbreviated name that corresponds to January, the second corresponds to February, and so forth.

mon Defines the full month names, corresponding to the `%B` field descriptor. The operand consists of twelve strings separated by semicolons. The first string is an abbreviated name that corresponds to January, the second corresponds to February, and so forth.

d_t_fmt
Defines the appropriate date and time representation, corresponding to the `%c` field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

d_fmt Defines the appropriate date representation, corresponding to the `%x` field descriptor. The operand consists of a string, and may contain any combination of characters and field descriptors.

t_fmt Defines the appropriate time representation, corresponding to the `%X` field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

am_pm Defines the appropriate representation of the ante meridian and post meridian strings, corresponding to the `%p` field descriptor. The operand consists of two strings, separated by a semicolon. The first string represents the ante meridian designation, the last string the post meridian designation.

t_fmt_ampm
Defines the appropriate time representation in the 12-hour clock format with am_pm, corresponding to the `%r` field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors.

era Defines how the years are counted and displayed for each era (or emperor's reign) in a locale.

No `era` is needed if the `%E` field descriptor modifier is not used for the locale. See the description of the `strftime()` function in *LE/VSE C Run-Time Library Reference* for information about this field descriptor.

For each `era`, there must be one string in the following format:

```
direction:offset:start_date:end_date:name:format
```

where

direction
Either a `+` or `-` character. The `+` character indicates the time axis should be such that the years count in the positive direction when moving from the starting date towards the ending date. The `-` character indicates the time axis should be such that the years count in the negative direction when moving from the starting date towards the ending date.

offset A number of the first year of the era.

start_date
A date in the form yyyy/mm/dd where yyyy, mm and dd are the year, month and day numbers, respectively, of the start of the era. Years prior to the year AD `0` are represented as negative numbers. For example, an era beginning March 5th in the year `100` BC would be represented as `-100/3/5`.

end_date
The ending date of the era in the same form as the start_date above or one of the two special values `-*` or `+*`. A value of `-*` indicates the ending date of the era extends to the beginning of time while `+*` indicates it extends to the end of time. The ending date may be either before or after the starting date of an era. For example, the strings for the Christian eras AD and BC would be:

```
+:0:0000/01/01:+*:AD:%EC %Ey
+:1:-0001/12/31:-*:BC:%EC %Ey
```

name A string representing the name of the era which is substituted for the `%EC` field descriptor.

format A string for formatting the `%EY` field descriptor. This string is usually a function of the `%EC` and `%Ey` field descriptors.

The operand consists of one string for each era. If there is more than one era, strings are separated by semicolons.

era_year
: Defines the format of the year in alternate era format, corresponding to the %EY field descriptor.

era_d_fmt
: Defines the format of the date in alternate era notation, corresponding to the %Ex field descriptor.

alt_digits
: Defines alternate symbols for digits, corresponding to the %O field descriptor modifier. The operand consists of semicolon-separated strings. The first string is the alternate symbol corresponding to zero, the second string is the symbol corresponding to one, and so forth. A maximum of 100 alternate strings may be specified. The %O modifier indicates that the string corresponding to the value specified by the field descriptor is used instead of the value.

For the definitions of the time formatting descriptors, see the description of the strftime() function in *LE/VSE C Run-Time Library Reference*.

LC_MESSAGES Category

The LC_MESSAGES category defines the format and values for positive and negative responses.

The following keywords are recognized:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present in this category.

If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

yesexpr
: The operand consists of an extended regular expression that describes the acceptable **affirmative** response to a question that expects an affirmative or negative response.

noexpr The operand consists of an extended regular expression that describes the acceptable **negative** response to a question that expects an affirmative or negative response.

Here is an example that shows how to define the LC_MESSAGES category:

```
%%%%%%%%%%%%
LC_MESSAGES
%%%%%%%%%%%%
% yes expression is a string that starts with
% "SI", "Si" "sI" "si" "s" or "S"
yesexpr "<circumflex><left-parenthesis><left-square-bracket><s><S>/
<right-square-bracket><left-square-bracket><i><I><right-square-bracket>/
<vertical-line><left-square-bracket><s><S><right-square-bracket>/
<right-parenthesis>"

% no expression is a string that starts with
% "NO", "No" "nO" "no" "N" or "n"
noexpr "<circumflex><left-parenthesis><left-square-bracket><n><N>/
<right-square-bracket><left-square-bracket><o><O><right-square-bracket>/
```

```
<vertical-line><left-square-bracket><n><N><right-square-bracket>/
<right-parenthesis>"

END LC_MESSAGES
```

LC_TOD Category

The LC_TOD category defines the rules used to define the beginning, end, and duration of daylight savings time, and the difference between local time and Greenwich Mean time. This is an IBM extension.

The following keywords are recognized:

copy
Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category.

If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

Note: If you specify this keyword, no other keyword should be present in this category.

timezone_difference
An integer specifying the time zone difference expressed in minutes. If the local time zone is west of the Greenwich Meridian, this value must be positive. If the local time zone is east of the Greenwich Meridian, this value must be negative. An absolute value greater than `1440` (the number of minutes in a day) for this keyword indicates that LE/VSE is to get the time zone difference from the system.

timezone_name
A string specifying the time zone name such as `"PST"` (Pacific Standard Time) specified within quotation marks. The default for this field is a `NULL` string.

daylight_name
A string specifying the Daylight Saving Time zone name, such as `"PDT"` (Pacific Daylight Time), if there is one available. The string must be specified within quotation marks. If DST information is not available, this is set to `NULL`, which is also the default. This field must be filled in if DST information as provided by the other fields is to be taken into account by the &mktime and `localtime()` functions. These functions ignore DST if this field is `NULL`.

start_month
An integer specifying the month of the year when Daylight Saving Time comes into effect. This value ranges from 1 through 12 inclusive, with 1 corresponding to January and 12 corresponding to December. If DST is not applicable to a locale, `start_month` is set to `0`, which is also the default.

end_month
An integer specifying the month of the year when Daylight Saving Time ceases to be in effect. The specifications are similar to those for `start_month`.

start_week
An integer specifying the week of the month when DST comes into effect. Acceptable values range from `-4` to `+4`. A value of `4` means the fourth week of the month, while a value of `-4` means fourth week of the month,

counting from the end of the month. Sunday is considered to be the start of the week. If DST is not applicable to a locale, `start_week` is set to `0`, which is also the default.

`end_week`
An integer specifying the week of the month when DST ceases to be in effect. The specifications are similar to those for `start_week`.

Note: The `start_week` and `end_week` need not be used. The `start_day` and `end_day` fields can specify either the day of the week or the day of the month. If day of month is specified, `start_week` and `end_week` become redundant.

`start_day`
An integer specifying the day of the week or the day of the month when DST comes into effect. The value depends on the value of `start_week`. If `start_week` is not equal to `0`, this is the day of the week when DST comes into effect. It ranges from `0` through `6` inclusive, with `0` corresponding to Sunday and `6` corresponding to Saturday. If `start_week` equals `0`, `start_day` is the day of the month (for the current year) when DST comes into effect. It ranges from `1` through to the last day of the month inclusive. The last day of the month is `31` for January, March, May, July, August, October, and December. It is `30` for April, June, September, and November. For February, it is `28` in non-leap years and `29` in leap years. If DST is not applicable to a locale, `start_day` is set to `0`, which is also the default.

`end_day`
An integer specifying the day of the week or the day of the month when DST ceases to be in effect. The specifications are similar to those for `start_day`.

`start_time`
An integer specifying the number of seconds after 12 midnight, local standard time, when DST comes into effect. For example, if DST is to start at 2 a.m., `start_time` is assigned the value `7200`; for 12 midnight, `start_time` is `0`; for 1 a.m., it is `3600`.

`end_time`
An integer specifying the number of seconds after 12 midnight, local standard time, when DST ceases to be in effect. The specifications are similar to those for `start_time`.

`shift` An integer specifying the DST time shift, expressed in seconds. The default is `3600`, for 1 hour.

`uctname`
A string specifying the name to be used for Coordinated Universal Time. If this keyword is not specified, the `uctname` will default to `"UTC"`.

Here is an example of how to define the LC_TOD category:

```
escape_char   /
comment-char  %

%%%%%%%%%%%%
LC_TOD
%%%%%%%%%%%%
% the time zone difference is 8hrs; the name of the daylight saving
% time is PDT, and it starts on the first Sunday of April at 2 a.m.
% and ends on the second Sunday of October at 2 a.m.
timezone_difference +480
timezone_name       "<P><S><T>"
daylight_name       "<P><D><T>"
start_month         4
end_month           10
start_week          1
end_week            2
start_day           1
end_day             30
start_time          7200
end_time            3600
shift               3600
END LC_TOD
```

LC_SYNTAX Category

The LC_SYNTAX category defines the variant characters from the portable character set. LC_SYNTAX is an IBM-specific extension. This category can be queried by the C library function `getsyntx()` to determine the encoding of a variant character if needed.

Attention: Customizing the LC_SYNTAX category is not recommended. You should use the LC_SYNTAX values obtained from the charmap file when you use the `localedef` utility.

The operands for the characters in the LC_SYNTAX category accept the single byte character specification in the form of a symbolic name, the character itself, or the decimal, octal, or hexadecimal constant. The characters must be specified in the LC_CTYPE category as a *punct* character. The values for the LC_SYNTAX characters must be unique. If symbolic names are used to define the encoding, only the symbolic names listed for each character should be used.

The code points for the LC_SYNTAX characters are set to the code points specified. Otherwise, they default to the code points for the respective characters from the charmap file, if the file is present, or to the code points of the respective characters in the IBM-1047 code page.

The following keywords are recognized:

`copy` Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present.

If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

`backslash`
Specifies a string that defines the value used to represent the backslash

character. If this keyword is not specified, the value from the charmap file for the character `<backslash>`, `<reverse-solidus>`, or `<SM07>` is used, if it is present.

`right_brace`
Specifies a string that defines the value used to represent the right brace character. If this keyword is not specified, the value from the charmap file for the character `<right-brace>`, `<right-curly-bracket>`, or `<SM14>` is used, if it is present.

`left_brace`
Specifies a string that defines the value used to represent the left brace character. If this keyword is not specified, the value from the charmap file for the character `<left-brace>`, `<left-curly-bracket>`, or `<SM11>` is used, if it is present.

`right_bracket`
Specifies a string that defines the value used to represent the right bracket character. If this keyword is not specified, the value from the charmap file for the character `<right-square-bracket>`, or `<SM08>` is used, if it is present.

`left_bracket`
Specifies a string that defines the value used to represent the left bracket character. If this keyword is not specified, the value from the charmap file for the character `<left-square-bracket>`, or `<SM06>` is used, if it is present.

`circumflex`
Specifies a string that defines the value used to represent the circumflex character. If this keyword is not specified, the value from the charmap file for the character `<circumflex>`, `<circumflex-accent>`, or `<SD15>` is used, if it is present.

`tilde` Specifies a string that defines the value used to represent the tilde character. If this keyword is not specified, the value from the charmap file for the character `<tilde>`, or `<SD19>` is used, if it is present.

`exclamation_mark`
Specifies a string that defines the value used to represent the exclamation mark character. If this keyword is not specified, the value from the charmap file for the character `<exclamation-mark>`, or `<SP02>` is used, if it is present.

`number_sign`
Specifies a string that defines the value used to represent the number sign character. If this keyword is not specified, the value from the charmap file for the character `<number-sign>`, or `<SM01>` is used, if it is present.

`vertical_line`
Specifies a string that defines the value used to represent the vertical line character. If this keyword is not specified, the value from the charmap file for the character `<vertical-line>`, or `<SM13>` is used, if it is present.

`dollar_sign`
Specifies a string that defines the value used to represent the dollar sign character. If this keyword is not specified, the value from the charmap file for the character `<dollar-sign>`, or `<SC03>` is used, if it is present.

`commercial_at`
Specifies a string that defines the value used to represent the commercial at character. If this keyword is not specified, the value from the charmap file for the character `<commercial-at>`, or `<SM05>` is used, if it is present.

grave_accent
Specifies a string that defines the value used to represent the grave accent character. If this keyword is not specified, the value from the charmap file for the character `<grave-accent>`, or `<SD13>` is used, if it is present.

Here is an example of how the LC_SYNTAX category is defined:

```
escape_char   /
comment-char  %

%%%%%%%%%%%%%
LC_SYNTAX
%%%%%%%%%%%%%

backslash        "<backslash>"
right_brace      "<right-brace>"
left_brace       "<left-brace>"
right_bracket    "<right-square-bracket>"
left_bracket     "<left-square-bracket>"
circumflex       "<circumflex>"
tilde            "<tilde>"
exclamation_mark "<exclamation-mark>"
number_sign      "<number-sign>"
vertical_line    "<vertical-line>"
dollar_sign      "<dollar-sign>"
commercial_at    "<commercial-at>"
grave_accent     "<grave-accent>"

END LC_SYNTAX
```

Using the `localedef` Utility

The locale objects or locales are generated using the `localedef` utility and the C/VSE compiler. The `localedef` utility:

1. Reads the *locale definition file*.
2. Resolves all the character symbolic names to the values of characters defined in the specified *character set definition file*.
3. Produces a C source file.

The C source file created by the `localedef` utility must be compiled using the C/VSE compiler and the resulting text deck link-edited to produce a locale phase.

The locale phase can be loaded by the `setlocale()` function and then accessed by the LE/VSE C Run-Time functions that are sensitive to the cultural information, or that can query the locales. For a list of all the library functions sensitive to locale, see “Locale-Sensitive Interfaces” on page 318. For detailed information on how to invoke the `localedef` utility, see “The `localedef` Utility” on page 392.

Locale Naming Conventions

The `setlocale()` library function that selects the active locale maps the descriptive locale name into the name of the locale object before loading the locale and making it accessible.

In LE/VSE C Run-Time programs, the locale phases are referred to by descriptive locale names. The locale names themselves are not case sensitive. They follow these conventions:

Language-Territory.Codeset@Modifier

Where:

Language
: is a two-letter uppercase abbreviation for the language name. The abbreviations come from the ISO 639 standard.

Territory
: is a two-letter uppercase abbreviation for the territory name. The abbreviation comes from the ISO 3166 standard.

Codeset
: is the name registered by the MIT X Consortium that identifies the registration authority that owns the specific encoding. `Codeset` is not required. If it is not specified, it defaults to the codeset described in Table 46 on page 354.

Modifier
: In general, this is used to select a specific instance of localization data within a single category. The modifier `euro` is used to select euro-currency specific data in the `LC_MONETARY` category. The modifier `preeuro` is used to select the previous (local) currency specific data in the `LC_MONETARY` category. See the note on page 431. `Modifier` is not required. If it is not specified, it defaults to nothing.

The mapping between the descriptive locale name and the eight-character name of the locale object is performed as follows:

1. The `Language-Territory` part is mapped into a two-letter LT code.
2. The `Codeset` part is mapped into a two-letter CC code.
3. If the `Modifier` is not specified, the object name is built from the characters `EDC$`, the two-letter LT code, and the two-letter CC code.
4. If the `Modifier` is `euro`, the object name is built from the characters `EDC@` [2] , the two-letter LT code, and the two-letter CC code.
5. If the `Modifier` is `preeuro`, the object name is built from the characters `EDC3`, the two-letter LT code, and the two-letter CC code.

Here are some examples that illustrate the above rules:
`Fr_BE.IBM-1148` maps to `EDC$FBHO`
`Fr_BE.IBM-1148@euro` maps to `EDC@FBHO`
`Fr_BE.IBM-1148@preeuro` maps to `EDC3FBHO`

The mapping between `Language-Territory` and the two-letter LT code is defined in the LT conversion table `EDC$LCNM`, built with assembler macros as follows:

```
EDC$LCNM TITLE 'LOCALE NAME CONVERSION TABLE'
EDC$LCNM CSECT
         EDCLOCNM TYPE=ENTRY,LOCALE='DA_DK',CODESET='IBM-1047',CODE='DA'
         EDCLOCNM TYPE=ENTRY,LOCALE='DE_CH',CODESET='IBM-1047',CODE='DC'
         EDCLOCNM TYPE=ENTRY,LOCALE='DE_DE',CODESET='IBM-1047',CODE='DD'
         EDCLOCNM TYPE=ENTRY,LOCALE='JA_JP',CODESET='IBM-939',CODE='JA'
⋮
         EDCLOCNM TYPE=END
         END   EDC$LCNM
```

`LOCALE` specifies the `Language-Territory` name, while `CODE` specifies the respective LT code.

2. The @-sign in the locale names always has Latin-1/Open Systems encoding. See IBM-1047 CHARMAP.

You can customize this table by adding new LOCALE name mappings. LE/VSE C Run-Time reserves alphabetic LT codes, but you can use codes containing numeric values for your own customized names.

The following Language-Territory names and their mappings into LT codes are provided:

Table 46. Supported Language-Territory Names and LT Codes

Locale Name	Language	Country or Region	Default Codeset	2-byte LT Code
BG_BG	Bulgarian	Bulgaria	IBM-1025	BG
C			IBM-1047	CC
CA_ES	Catalan	Spain	IBM-924	CS
CS_CZ	Czech	Czech Republic	IBM-870	CZ
DA_DK	Danish	Denmark	IBM-1047	DA
DE_AT	German	Austria	IBM-924	DT
DE_CH	German	Switzerland	IBM-1047	DC
DE_DE	German	Germany	IBM-1047 [1]	DD
DE_LU	German	Luxembourg	IBM-924	DL
EL_GR	Greek	Greece	IBM-875 [2]	EL
EN_BE	English	Belgium	IBM-924	EB
EN_GB	English	United Kingdom	IBM-1047	EK
EN_IE	English	Ireland	IBM-924	EI
EN_JP	English	Japan	IBM-1027	EJ
EN_US	English	United States	IBM-1047	EU
ES_ES	Spanish	Spain	IBM-1047 [3]	ES
ET_EE	Estonian	Estonia	IBM-1122	EE
FI_FI	Finish	Finland	IBM-1047 [4]	FI
FR_BE	French	Belgium	IBM-1047 [5]	FB
FR_CA	French	Canada	IBM-1047	FC
FR_CH	French	Switzerland	IBM-1047	FS
FR_FR	French	France	IBM-1047 [6]	FF
FR_LU	French	Luxembourg	IBM-924	FL
HR_HR	Croatian	Croatia	IBM-870	HR
HU_HU	Hungarian	Hungary	IBM-870	HU
IS_IS	Icelandic	Iceland	IBM-1047	IS
IT_IT	Italian	Italy	IBM-1047 [7]	IT
IW_IL	Hebrew	Israel	IBM-424	IL
JA_JP	Japanese	Japan	IBM-939	JA
KO_KR	Korean	Korea	IBM-933	KR
LT_LT	Lithuanian	Lithuania	IBM-1112	LT
MK_MK	Macedonian	Macedonia	IBM-1025	MM
NL_BE	Dutch	Belgium	IBM-1047 [8]	NB
NL_NL	Dutch	Netherlands	IBM-1047 [9]	NN

Table 46. Supported Language-Territory Names and LT Codes (continued)

Locale Name	Language	Country or Region	Default Codeset	2-byte LT Code
NO_NO	Norwegian	Norway	IBM-1047	NO
PL_PL	Polish	Poland	IBM-870	PL
PT_BR	Portugese	Brazil	IBM-1047	BR
PT_PT	Portugese	Portugal	IBM-1047 [10]	PT
RO_RO	Romanian	Romania	IBM-870	RO
RU_RU	Russian	Russia	IBM-1025	RU
SH_SP	Serbian (Latin)	Serbia	IBM-870	SL
SI_SI	Slovene	Slovenia	IBM-870	SI
SK_SK	Slovak	Slovakia	IBM-870	SK
SQ_AL	Albanian	Albania	IBM-1047	SA
SR_SP	Serbian (Cyrillic)	Serbia	IBM-1025	SC
SV_SE	Swedish	Sweden	IBM-1047	SV
TH_TH	Thai	Thailand	IBM-838	TH
TR_TR	Turkish	Turkey	IBM-1026	TR
ZH_CN	Chinese (simplified)	China	IBM-935	ZC
ZH_TW	Chinese (traditional)	Taiwan	IBM-937	ZT

Notes:
1. Germany should use the codeset IBM-924 or IBM-1141. Also see Note 11 below.
2. Greece should use the codeset IBM-4971. Also see Note 11 below.
3. Spain should use the codeset IBM-924 or IBM-1145. Also see Note 11 below.
4. Finland should use the codeset IBM-924 or IBM-1143. Also see Note 11 below.
5. Belgium should use the codeset IBM-924 or IBM-1148. Also see Note 11 below.
6. France should use the codeset IBM-924 or IBM-1147. Also see Note 11 below.
7. Italy should use the codeset IBM-924 or IBM-1144. Also see Note 11 below.
8. Belgium should use the codeset IBM-924 or IBM-1148. Also see Note 11 below.
9. The Netherlands should use the codeset IBM-924 or IBM-1140. Also see Note 11 below.
10. Portugal should use the codeset IBM-924 or IBM-1140. Also see Note 11 below.

Note: 11. If used with the default codeset, this locale does NOT support the Euro currency.

The mapping between `Codeset` and the two-letter CC code is defined in the CC conversion table `EDCUCSNM`. This table is built with assembler macros as follows:

```
EDCUCSNM TITLE 'CODE SET NAME CONVERSION TABLE'
EDCUCSNM CSECT
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-037',CODE='EA'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-273',CODE='EB'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-274',CODE='EC'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-277',CODE='ED'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-278',CODE='EE'
```

```
⋮
         EDCCSNAM TYPE=END
         END   EDCUCSNM
```

CODESET specifies the Codeset name; CODE specifies the respective CC code.

You can customize this table by adding new Codeset names. The alphabetic codes in the first byte of each CC name are reserved by IBM for future use, but you can use codes starting with numeric values for your own customized names.

The following Codeset names and their mappings into CC codes are provided:

Table 47. Supported Codeset Names and CC Codes

Codeset	Country or Region	2-byte CC Code
EBCDIC Codesets		
IBM-037	USA, Canada, Brazil	EA
IBM-273	Germany, Austria	EB
IBM-274	Belgium	EC
IBM-275	Brazil	ED
IBM-277	Denmark, Norway	EE
IBM-278	Finland, Sweden	EF
IBM-280	Italy	EG
IBM-281	Japan (Latin-1)	EH
IBM-282	Portugal	EI
IBM-284	Spain, Latin America	EJ
IBM-285	United Kingdom	EK
IBM-290	Japan (Katakana)	EL
IBM-297	France	EM
IBM-424	Israel	FB
IBM-500	International	EO
IBM-838	Thailand	EP
IBM-870	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia (Latin), Slovakia, Slovenia	EQ
IBM-871	Iceland	ER
IBM-875	Greece	ES
IBM-924	Latin 9/Open Systems	EZ
IBM-930	Japan Katakana Extended (combined with DBCS)	EU
IBM-933	Korea	GZ
IBM-935	China	GY
IBM-937	Taiwan	GW
IBM-939	Japan (latin) Extended (combined with DBCS)	EV
IBM-1025	Bulgaria, Macedonia, Russia, Serbia (Cyrillic)	FE
IBM-1026	Turkey	EW
IBM-1027	Japan (Latin) Extended	EX

Table 47. Supported Codeset Names and CC Codes (continued)

Codeset	Country or Region	2-byte CC Code
IBM-1047	Latin-1/Open Systems	EY
IBM-1112	Lithuania	GD
IBM-1122	Estonia	FD
IBM-1124	Ukraine	AU
IBM-1140	USA, Canada, Brazil (Euro)	HA
IBM-1141	Austria, Germany (Euro)	HB
IBM-1142	Denmark, Norway (Euro)	HE
IBM-1143	Finland, Sweden (Euro)	HF
IBM-1144	Italy (Euro)	HG
IBM-1145	Spain, Latin America (Euro)	HJ
IBM-1146	United Kingdom (Euro)	HK
IBM-1147	France (Euro)	HM
IBM-1148	International (Euro)	HO
IBM-1149	Iceland (Euro)	HR
IBM-1153	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia (Latin), Slovakia, Slovenia (Euro)	MB
IBM-1154	Bulgaria, Macedonia, Russia, Serbia/Cyrillic (Euro)	HT
IBM-1155	Turkey (Euro)	HW
IBM-1156	Lithuania (Euro)	HZ
IBM-1157	Estonia (Euro)	HD
IBM-1160	Thailand (Euro)	HP
IBM-1371	Taiwan (Euro)	KA
IBM-4971	Greece (Euro)	HS
IBM-5123	Japan (Latin Extended with Euro)	HX
IBM-8482	Japan (Katakana with Euro)	HL
IBM-12712	Israel (Euro)	HH
ASCII Codesets (IBM-PC and AIX)		
IBM-850	IBM PC - International	AA
IBM-932	IBM PC - Japanese	AB
IBM-eucJP	Japanese	AC
ISO8859 Codesets		
ISO8859-1	ISO Standard ASCII	I1
ISO8859-7	ISO ASCII - Greece	I7
ISO8859-9	ISO ASCII - Turkey	I9

The exceptions to the rule above are the following special locale names, which are already recognized:
- `C`
- `POSIX`
- `SAA`
- `S370`

The special names C, POSIX, SAA, and S370 always refer to the built-in locales, which cannot be modified.
- GERM
- FRAN
- UK
- ITAL
- SPAI
- USA

These names are for locales in the old format, created with assembler macros rather than with the localedef utility.

Note: These locales are not up-to-date, and do **not** support the Euro currency.

You can use the following macros, defined in the locale.h header file, as synonyms for the special locale names above.

Macro	Locale	Compiled locale
C	C	Not applicable
POSIX	POSIX	EDC$POSX
SAA	SAA	EDC$SAAC
S370	S370	EDC$S370
LC_C_GERMANY	"GERM"	EDC$GERM
LC_C_FRANCE	"FRAN"	EDC$FRAN
LC_C_UK	"UK"	EDC$UK
LC_C_ITALY	"ITAL"	EDC$ITAL
LC_C_SPAIN	"SPAI"	EDC$SPAI
LC_C_USA	"USA"	EDC$USA

The predefined name for the built-in locale in the old format is S370.

The rest of the special names refer to the locale objects whose names are built by prepending the letters EDC$ to the special name, as for EDC$FRAN.

Chapter 31. Customizing a Locale

This chapter describes how you can create your own locales, based on the locale definition files supplied by IBM. The information in this chapter applies to the format of locales based on the `localedef` utility.

In this example you will build a locale named TEXAN using the charmap file representing the IBM-1047 encoded character set. The locale is derived from the locale representing the English language and the cultural conventions of the United States.

1. Determine the source of the locale you are going to use from the Table 53 on page 437. In this case, it is the English language in the United States locale, the source for which is the member EDC$EUEY.L in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE).
2. Copy the member EDC$EUEY.L from the LE/VSE installation sublibrary (default is PRD2.SCEEBASE) to a sublibrary named LOCALE.WRK (which has been predefined using the VSE/Librarian) as member name TEXAN.L.
3. In your new file, change the locale variables to the desired values. For example, change

   ```
   d_t_fmt "%a %b %e %H:%M:%S %Z %Y
   ```

   to

   ```
   d_t_fmt "Howdy Pardner %a %b %e %H:%M:%S %Z %Y"
   ```
4. Using the `localedef` utility, and subsequently the C/VSE compiler and the linkage editor, generate a new locale as member EDC$1TEY.PHASE in the LOCALE.WRK sublibrary. See “The `localedef` Utility” on page 392 for detailed information about how to use the `localedef` utility. Specifically, the example on page 394 shows how the above phase can be produced.

   The member name in the LOCALE.WRK sublibrary has the predefined prefix EDC$. The next two characters must consist of a numerical character (alphabetics are reserved for IBM use) followed by an alphanumeric character. For this example, the letters 1T defines the TEXAN locale (defined in the next step). You can determine the last two characters which identify the `CodesetRegistry-CodesetEncoding` from Table 47 on page 356. In this case they should be the value of the CC code for the coded character set IBM-1047, which is EY. If you are using your own charmap file you must define its two-letter CC code (starting with a numeric value) in the table EDCUCSNM. This is done in a similar way to defining EDC$LCNM, as described in the next step.
5. Copy the member EDC$LCNM.A from the LE/VSE installation sublibrary (default is PRD2.SCEEBASE) to the LOCALE.WRK sublibrary. LE/VSE C Run-Time uses this table to map locale code registry prefixes into two-character codes. For this example, insert a new line into the assembler table before the last `EDCLOCNM TYPE=END` entry:

   ```
   EDCLOCNM TYPE=ENTRY,LOCALE='TEXAN',CODESET='IBM-1047',CODE='1T'
   ```
6. Assemble the EDC$LCNM.A member and link-edit it into the LOCALE.WRK sublibrary with the member name EDC$LCNM.PHASE.

Using the Customized Locale

The customized locale is now ready to be used in these ways:

- Explicitly referenced by name in LE/VSE C Run-Time application code that uses `setlocale()` calls referring to the locale descriptive name (recommended) such as:

  ```
  setlocale(LC_ALL, "TEXAN.IBM-1047");
  ```

  or by a short internal name (not recommended) such as:

  ```
  setlocale(LC_ALL, "1TEY");
  ```
- Explicitly referenced in the LE/VSE C Run-Time initialization exit, using customized setup code in CEEBINT.
- Implicitly specified in each user environment with environment variables.
- Passed via the `PARM` parameter of the `EXEC` statement to the compiler as an argument on the `LOCALE` compiler option. For example,

  ```
     PARM='locale("TEXAN.IBM-1047")'
  ```

  tells the compiler to use the TEXAN.IBM-1047 locale at compile time and generate output in code page IBM-1047. For more information, refer to "Converting Coded Character Sets at Compile Time" on page 408.

Note: You cannot customize the built-in locales, C, POSIX, SAA, or S370. The locale source files EDC$POSX and EDC$SAAC are provided for reference only.

Referring Explicitly to a Customized Locale

Here is a program with an explicit reference to the TEXAN locale.

EDCXGCL1

```
 /* EDCXGCL1
    This example shows how to get the local time formatted by the
    current locale
  */

#include <stdio.h>
#include <time.h>
#include <locale.h>

int main(void){
    char dest[80];
    int ch;
    time_t temp;
    struct tm *timeptr;
    temp = time(NULL);
    timeptr = localtime(&temp);
    /* Fetch default locale name */
    printf("Default locale is %s\n",setlocale(LC_ALL,""));
    ch = strftime(dest,sizeof(dest)-1,
      "Local C datetime is %c", timeptr);
    printf("%s\n",  dest);

    /* Set new Texan locale name */
    printf("New locale is %s\n", setlocale(LC_ALL,"Texan.IBM-1047"));
    ch = strftime(dest,sizeof(dest)-1,
      "Texan datetime is %c ", timeptr);
    printf("%s\n", dest);

    return(0);
}
```

Figure 80. Referring Explicitly to a Customized Locale

Compile the above program. Before you execute it, ensure the VSE/Librarian sublibrary containing the TEXAN locale and updated table is available.

The output should be similar to:

```
Default locale is S370
Local C datetime is 96/08/14 03:12:14
New locale is Texan.IBM-1047
Texan datetime is Howdy Pardner Wed Aug 14 15:12:14  1996
```

Note that if the second operand to `setlocale()` had been `NULL`, rather than `""`, the default locale name returned would have been ″C″.

```
setlocale(LC_ALL,"")    returns "S370"
setlocale(LC_ALL,NULL)  returns "C"
```

Note: For `setlocale(LC_ALL,"")`, `"S370"` is returned unless the locale-related environment variables are set. See Chapter 32, “Definition of S370 C, SAA C, and POSIX C Locales,” on page 363 for more information about the definition of the S370 locale.

Referring Implicitly to a Customized Locale

An installation may require that a global mechanism should be used for all C programs. The exit CEEBINT may be used for this purpose. Users can insert a `setlocale()` call inside the routines referencing the locale required. Here is an example:

EDCXGCL2

```
 /* EDCXGCL2
    This example refers implicitly to a customized locale
  */

#pragma linkage(CEEBINT,OS)

#pragma map(CEEBINT,"CEEBINT")
void CEEBINT(int, int, int, int, void**, int, void**);

#include <locale.h>
#include <stdio.h>

int main(void){
   printf("Default NULL locale = %s\n", setlocale(LC_ALL,NULL));
   printf("Default \"\" locale = %s\n", setlocale(LC_ALL,""));
 }

void CEEBINT(int number, int retcode, int rsncode, int fnccode,
             void **a_main, int userwd, void **a_exits)
 {  /* user code goes here */
   printf("CEEBINT entry. number = %i\n", number);
   printf("Locale = %s\n", setlocale(LC_ALL,"Texan.IBM-1047"));
   }
```

Figure 81. Referring Implicitly to a Customized Locale

If the above example is compiled and executed with the TEXAN locale, the results are as follows:

```
CEEBINT entry. number = 7
Locale = Texan.IBM-1047
Default NULL locale = Texan.IBM-1047
Default "" locale = S370
```

The exit CEEBINT may provide a uniform way of restricting the use of customized locales across an installation. To do this, a system programmer can compile CEEBINT separately, and link it with the application program that will use it. The disadvantage to this approach is that CEEBINT must be link-edited into each user phase explicitly. See Chapter 20, “Using Run-Time User Exits,” on page 201 for more information about user exits.

Customizing Your Installation

When LE/VSE C Run-Time initializes the environment, it uses the C locale as its default locale. The only values that may be customized when Language Environment is installed are those associated with the LC_TOD category. Details on this customization are provided in *LE/VSE Customization Guide*.

Chapter 32. Definition of S370 C, SAA C, and POSIX C Locales

The default C locales for POSIX SAA, and S370 are pre-built into the run-time library. The SAA C locale provides compatibility with previous releases of C/370. The POSIX C locale provides consistency with POSIX requirements.

The POSIX definition of the C locale is described below, with the IBM extensions LC_SYNTAX and LC_TOD showing their default values.

The SAA and S370 definitions of the C locale are different from the POSIX definition; consistency with previous releases of LE/VSE C Run-Time is provided for migration compatibility. The differences are described in “Differences Between SAA C and POSIX C Locales” on page 369.

The relationship between the POSIX C and SAA C locales is as follows.

1. The SAA C locale definition is the default. `"C"`, `"SAA"`, and `"S370"` are synonyms for the SAA C locale definition, which is pre-built into the library.

   The source file EDC$SAAC.L is provided for reference, but cannot be used to alter the definition of this pre-built locale.
2. Issuing `setlocale(`*`category`*`, "")` has the following effect:
   - Locale-related environment variables are checked to find the name of locales) to use to set the *category* specified. Querying the locale with `setlocale(`*`category`*`, NULL)` returns the name of the locales specified by the appropriate environment variables.
   - If no non-null environment variable is present, then it is the equivalent of having issued `setlocale(`*`category`*`, "S370")` That is, the locale chosen is the SAA C locale definition, and querying the locale with `setlocale(`*`category`*`, NULL)` returns `"S370"` as the locale name.
3. If no `setlocale()` function is issued, or `setlocale(LC_ALL, "C")`, then the locale chosen is the pre-built SAA C locale, and querying the locale with `setlocale(`*`category`*`, NULL)` returns ″C″ as the locale name.
4. For `setlocale(LC_ALL, "SAA")`, the locale chosen is the pre-built SAA C locale, and querying the locale with `setlocale(`*`category`*`, NULL)` returns ″SAA″ as the locale name.
5. For `setlocale(LC_ALL, "S370")`, the locale chosen is the pre-built SAA C locale, and querying the locale with `setlocale(`*`category`*`, NULL)` returns `"S370"` as the locale name.
6. For `setlocale(LC_ALL, "POSIX")`, the locale chosen is the pre-built POSIX C locale, and querying the locale with `setlocale(`*`category`*`, NULL)` returns `"POSIX"` as the locale name.

The `setlocale()` function supports locales built using the `localedef` utility, as well as locales built using the assembler source and produced by the EDCLOC macro.

The LC_TOD category for the SAA C and POSIX C locales can be customized during installation of the library by your system programmer. See “Customizing Your Installation” on page 362 for more information. The supplied default will obtain the time zone difference from the operating system. However, it will not define the daylight savings time.

The LC_SYNTAX category for the SAA C and POSIX C locales is set to the IBM-1047 definition of the variant characters.

The other locale categories for the POSIX C locale are as follows.

```
escape_char   /
comment_char  %

%%%%%%%%%%%%
LC_CTYPE
%%%%%%%%%%%%

% "alpha" is by default "upper" and "lower"
% "alnum" is by definition "alpha" and "digit"
% "print" is by default "alnum", "punct" and <space> character
% "punct" is by default "alnum" and "punct"

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;/
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;/
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>

digit   <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>

space   <tab>;<newline>;<vertical-tab>;<form-feed>;/
        <carriage-return>;<space>
cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;/
        <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;/
        <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;/
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;/
        <IS1>;<DEL>
punct   <exclamation-mark>;<quotation-mark>;<number-sign>;/
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;/
        <left-parenthesis>;<right-parenthesis>;<asterisk>;/
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;/
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;/
        <greater-than-sign>;<question-mark>;<commercial-at>;/
        <left-square-bracket>;<backslash>;<right-square-bracket>;/
        <circumflex>;<underscore>;<grave-accent>;/
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;<tilde>
xdigit  <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>;/
        <A>;<B>;<C>;<D>;<E>;<F>;/
        <a>;<b>;<c>;<d>;<e>;<f>
blank   <space>;/
        <tab>
toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);/
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);/
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);/
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);/
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);/
        (<z>,<Z>)
tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);/
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);/
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);/
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);/
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);/
        (<Z>,<z>)

END LC_CTYPE
```

```
%%%%%%%%%%%%
LC_COLLATE
%%%%%%%%%%%%

order_start
% ASCII Control characters
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
```

```
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A>
<B>
<C>
<D>
<E>
<F>
<G>
<H>
<I>
<J>
<K>
<L>
<M>
<N>
<O>
<P>
<Q>
<R>
<S>
<T>
<U>
<V>
<W>
<X>
<Y>
<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a>
<b>
<c>
<d>
<e>
<f>
<g>
<h>
<i>
<j>
<k>
<l>
<m>
<n>
<o>
<p>
<q>
<r>
<s>
<t>
<u>
<v>
<w>
<x>
<y>
<z>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
```

```
<DEL>
order_end

END LC_COLLATE

%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%

int_curr_symbol   ""
currency_symbol   ""
mon_decimal_point ""
mon_thousands_sep ""
mon_grouping      ""
positive_sign     ""
negative_sign     ""
int_frac_digits   -1
frac_digits       -1
p_cs_precedes     -1
p_sep_by_space    -1
n_cs_precedes     -1
n_sep_by_space    -1
p_sign_posn       -1
n_sign_posn       -1

END LC_MONETARY

%%%%%%%%%%%%
LC_NUMERIC
%%%%%%%%%%%%

decimal_point     "<period>"
thousands_sep     ""
grouping          ""

END LC_NUMERIC

%%%%%%%%%%%%
LC_TIME
%%%%%%%%%%%%

abday   "<S><u><n>";/
        "<M><o><n>";/
        "<T><u><e>";/
        "<W><e><d>";/
        "<T><h><u>";/
        "<F><r><i>";/
        "<S><a><t>"

day     "<S><u><n><d><a><y>";/
        "<M><o><n><d><a><y>";/
        "<T><u><e><s><d><a><y>";/
        "<W><e><d><n><e><s><d><a><y>";/
        "<T><h><u><r><s><d><a><y>";/
        "<F><r><i><d><a><y>";/
        "<S><a><t><u><r><d><a><y>"

abmon   "<J><a><n>";/
        "<F><e><b>";/
        "<M><a><r>";/
        "<A><p><r>";/
        "<M><a><y>";/
        "<J><u><n>";/
        "<J><u><l>";/
        "<A><u><g>";/
        "<S><e><p>";/
        "<O><c><t>";/
        "<N><o><v>";/
        "<D><e><c>"
```

```
mon     "<J><a><n><u><a><r><y>";/
        "<F><e><b><r><u><a><r><y>";/
        "<M><a><r><c><h>";/
        "<A><p><r><i><l>";/
        "<M><a><y>";/
        "<J><u><n><e>";/
        "<J><u><l><y>";/
        "<A><u><g><u><s><t>";/
        "<S><e><p><t><e><m><b><e><r>";/
        "<O><c><t><o><b><e><r>";/
        "<N><o><v><e><m><b><e><r>";/
        "<D><e><c><e><m><b><e><r>"

% equivalent of AM/PM (%p)
am_pm      "<A><M>";"<P><M>"

% appropriate date and time representation (%c) "%a %b %e %H:%M:%S %Y"
d_t_fmt    "<percent-sign><a><space><percent-sign><b><space><percent-sign><e>/
<space><percent-sign><H><colon><percent-sign><M>/
<colon><percent-sign><S><space><percent-sign><Y>"

% appropriate date representation (%x) "%m/%d/%y"
d_fmt      "<percent-sign><m><slash><percent-sign><d><slash><percent-sign><y>"

% appropriate time representation (%X) "%H:%M:%S"
t_fmt      "<percent-sign><M><colon><percent-sign><M><colon><percent-sign><S>"

% appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon><percent-sign><S>/
<space><percent-sign><p>"

END LC_TIME

%%%%%%%%%%%%
LC_MESSAGES
%%%%%%%%%%%%

yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"

END LC_MESSAGES
```

Differences Between SAA C and POSIX C Locales

In fact, there are three built-in locales, S370 C, SAA C, and POSIX C. The default locale at your site depends on the system that is running the application. Issuing `setlocale(LC_ALL, "")` sets the default, based on the current environment. Issuing `setlocale(LC_ALL, "SAA")` sets the SAA C locale. Likewise, `setlocale(LC_ALL, "POSIX")` sets the POSIX locale.

If you are running in a C locale, one way you can determine whether the SAA C or the POSIX locale is in effect is to check whether the cent sign (¢ at X'4A') is defined as a punctuation character. Under the default POSIX support, the cent sign is not part of the POSIX portable character set. The following code illustrates how to perform this test:

EDCXGDL1

```
 /* EDCXGDL1
    This example shows how to determine whether the SAA C or POSIX
    locale is in effect
  */

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    if (ispunct(0x4A)) {
        printf(" cent sign is punct\n");
        printf(" current locale is SAA- or S370-like\n");
    }
    else {
        printf(" cent sign is not punct\n");
        printf(" default locale is POSIX-like\n");
    }

return(0);
}
```

Figure 82. Determining Which Locale is in Effect

Alternatively, the collating sequence may be tested: under the SAA or System/370 default locales, the lowercase letters collate before the uppercase letters, whereas under the POSIX definition, the lowercase letters collate after the uppercase letters. This difference may be tested using the string collate function `strcoll()`. The locale "" is the same locale as the one obtained from `setlocale(LC_ALL, "")`. For more detail on these special environment variables, see Chapter 21, “Using Environment Variables,” on page 219.

Other differences between the SAA C locale and the POSIX C locale are as follows:

<mb_cur_max>
: The POSIX C locale is built using coded character set IBM-1047, with `<mb_cur_max>` as 1.

: The SAA C locale is built using coded character set IBM-1047, with `<mb_cur_max>` as 4.

The cent sign
In the default POSIX support, the cent sign (¢) is *not* part of the POSIX portable character set, but in the SAA locale it *is* defined as a punctuation character.

Collation weight by case
In the POSIX definition, the lowercase letters collate *after* the uppercase letters, whereas in the SAA or System/370 default locales, the lowercase letters collate *before* the uppercase letters.

LC_CTYPE category
The SAA C locale has all the EBCDIC control characters defined in the `'cntrl'` class. The POSIX C locale has only the ASCII control characters in the `'cntrl'` class.

The SAA C locale includes ¢ (the cent character) and ¦ (the broken vertical line) as `'punct'` characters. The POSIX C locale does not group these characters as `'punct'` characters.

LC_COLLATE category
The default collation for the SAA C locale is the EBCDIC sequence. The POSIX C locale uses the ASCII collation sequence; the first 128 ASCII characters are defined in the collation sequence, and the remaining EBCDIC characters are at the end of the collating sequence.

LC_TIME category
The SAA C locale uses the date and time format `(d_t_fmt)` as `"%Y/%M/%D %X"`, whereas the POSIX C locale uses `"%a %b %d %H/%M/%S %Y"`.

The SAA C locale uses the strings `"am"` and `"pm"`, whereas the POSIX C locale uses `"AM"` and `"PM"`.

Chapter 33. Code Set and Locale Utilities

This chapter describes the code set conversion utilities which help you convert a file from one code set to another and the `localedef` utility which allows you to define the language and cultural conventions used in your environment.

Code Set Conversion Utilities

This section describes the code set conversion utilities provided with LE/VSE C Run-Time. These utilities are as follows:

The `genxlt` utility
: Generates a translation table for use by the `iconv` utility and `iconv` functions to perform code set conversion. It can be used to build code set conversion tables for existing code pages, whether or not these are supplied with LE/VSE C Run-Time.

The `uconvdef` utility
: Generates a UCS-2 translation table for use by the `iconv` utility and `iconv` functions to perform code set conversion between a multibyte code set and UCS-2.

Note: UCS-2 (or Unicode) is the Universal Multiple-Octet Coded Character Set defined by ISO/IEC 10646-1:1993(EE), while multibyte code sets consist of one or more bytes per character.

The `iconv` utility
: Converts a file from one code set encoding to another. It can be used to convert C source code before compilation or to convert data files.

The `iconv` functions
: Performs code set translation. These functions are `iconv_open()`, `iconv()`, and `iconv_close()`. They are used by the `iconv` utility and may be called from any LE/VSE C Run-Time program requiring code set translation.

See *LE/VSE C Run-Time Library Reference* for descriptions of the `iconv` functions.

The `genxlt` Utility

The `genxlt` utility reads a source translation file as specified using the `IFILE` option described below and writes the compiled version to `SYSLNK` or `SYSPCH` depending on the `DECK`, `LINK`, and `CATAL` JCL options. The source translation file contains directives that are acted upon by the `genxlt` utility to produce the compiled version of the translation table. The source input to the `genxlt` utility is assumed to be implicitly specified in code page IBM-1047.

The output from the `genxlt` utility must be link-edited to produce a phase whose name adheres to the following naming convention:

- The name must start with the constant four-letter prefix EDCU.
- The prefix is followed by the two-letter CC code that corresponds to the “from” code set defined in Table 47 on page 356.
- The first CC code is followed by the two-letter CC code than corresponds to the “to” code set defined in Table 47 on page 356.

To generate your own conversions, you must modify the code set name table EDCUCSNM with the macros described in “Locale Naming Conventions” on page 352. In addition, if you plan to use your own conversions under CICS, you must add the name of your phase to your CSD file.

The `genxlt` utility has the following options. If the same option is specified more than once, the last option specified is used. The options are specified on the `EXEC PARM`, and may be separated by spaces or commas.

DBCS | NODBCS

Specifies whether the DBCS characters within shift-out and shift-in characters will be converted. The `DBCS` option should only be specified when an EBCDIC code page is being converted to a different EBCDIC code page.

If the `DBCS` option is specified, when a shift-out character is encountered in the input, the characters up to the shift-in character are copied to the output, and not converted. There must be an even number of characters between the shift-out and shift-in characters, and the characters must be valid DBCS characters.

If the `NODBCS` option is specified (or by default), all the characters are converted, and no checking of DBCS characters is performed.

IFILE(...)

Specifies the source translation file containing the character conversion information as follows:

genxlt IFILE Option

```
►►──IFILE──(─┬─────────────────────┬─┬───────────────────────────┬──────────►
             │ ┌─DD:SYS001-SYSUT1─┐ │ │              ┌─80──────┐  │
             └─┴─i_name───────────┴─┘ └─,──lrecl──=──┴─i_lrecl─┴──┘

►─┬──────────────────────────┬─┬───────────────────────────────┬─)──────────►◄
  │              ┌─FB──────┐ │ │                ┌─4000──────┐  │
  └─,──recfm──=──┴─i_recfm─┴─┘ └─,──blksize──=──┴─i_blksize─┴──┘
```

where

i_name
: is the file specification in any of the formats supported by the `fopen()` function. (See the “Opening Files” sections in this book for additional information regarding file specification formats.) Default is `DD:SYS001-SYSUT1`.

i_lrecl
: is the input file logical record length in bytes. Default is `80`.

i_recfm
: is the input file record format. Default is `FB`.

i_blksize
: is the input file block size in bytes. Default is `4000`.

Notes:

1. All of the above parameters are optional.
2. If more than one parameter is specified, a comma (,) must be used as a separator between each parameter.
3. No spaces are allowed.
4. If *i_name* is specified, it must be the first parameter. Other parameters are non-positional.

Example:

```
IFILE(dd:file1,recfm=vb)
```

NAME(*obj_name***) | NONAME**
Specifies the name *obj_name* to be used on one of the following:

- The PHASE card if the VSE JCL option LINK and/or CATAL is in effect as follows:

  ```
  PHASE obj_name,*,SVA
  ```

- The CATALOG card if the VSE JCL option DECK is in effect as follows:

  ```
  CATALOG obj_name.OBJ REPLACE=YES
  ```

The PHASE or CATALOG card is written as the first record to the output file from the genxlt utility in order to produce a valid input stream for the linkage editor or the VSE Librarian respectively.

Messages from the genxlt utility are written to stdout/stderr.

Example

In the following example, the input source translation file is member EDCUEAEY.X in the default LE/VSE installation sublibrary PRD2.SCEEBASE. The output from the genxlt utility is link-edited and a phase FRED01 is placed in the VSE/Librarian sublibrary FRED.LIB.

```
// JOB GXLTSAMP
// LIBDEF *,SEARCH=PRD2.SCEEBASE
// LIBDEF PHASE,CATALOG=FRED.LIB
// OPTION LINK,CATAL
// EXEC EDCGNXLT,PARM='IFILE(DD:PRD2.SCEEBASE(EDCUEAEY.X)),NODBCS,     X
               NAME(FRED01)'
 ENTRY TABLENAM
/*
// EXEC LNKEDT
/*
// EXEC LISTLOG
/&
```

Note: The ’X’ at the end of statement ’// EXEC EDCGNXLT...’ is in column 72.

The uconvdef Utility

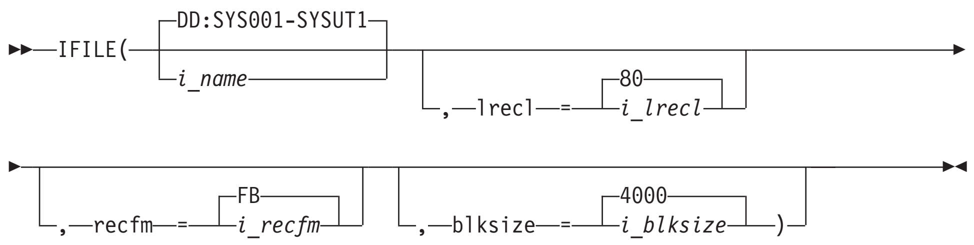
The uconvdef utility reads a source file that defines a mapping between UCS-2 and a multibyte code set, as specified using the IFILE option described below, and writes the compiled version to SYSLNK or SYSPCH depending on the DECK, LINK, and CATAL JCL options. The source file contains directives that are acted upon by the uconvdef utility to produce the compiled version of the translation table for use by the iconv utility and iconv functions. Refer to “UCMAP Source Format” on page 389 for information on these directives.

The output from the uconvdef utility must be link-edited to produce a phase whose name is of the form EDCUUccU, where cc is a two-letter CC code that corresponds to a code set defined in Table 47 on page 356. To generate your own conversions, you must modify the code set name table EDCUCSNM with the macros described in “Locale Naming Conventions” on page 352.

The uconvdef utility is invoked by specifying EDCUCDEF as the program name on the EXEC statement, and has the following options. If the same option is specified more than once, the last option specified is used. The options are specified on the EXEC PARM, and may be separated by spaces or commas.

IFILE(...)
Specifies the source file defining the mapping between UCS-2 and a multibyte code set, as follows:

uconvdef IFILE Option



where

i_name
is the file specification in any of the formats supported by the `fopen()` function. (See the "Opening Files" sections in this book for additional information regarding file specification formats.) Default is `DD:SYS001-SYSUT1`.

i_lrecl
is the input file logical record length in bytes. Default is `80`.

i_recfm
is the input file record format. Default is `FB`.

i_blksize
is the input file block size in bytes. Default is `4000`.

Notes:

1. All of the above parameters are optional.
2. If more than one parameter is specified, a comma (,) must be used as a separator between each parameter.
3. No spaces are allowed.
4. If *i_name* is specified, it must be the first parameter. Other parameters are non-positional.

Example:

```
IFILE(dd:file2,recfm=vb)
```

NAME(*obj_name*) | <u>NONAME</u>
Specifies the name *obj_name* to be used on one of the following:

- The `PHASE` card if the VSE JCL option `LINK` and/or `CATAL` is in effect as follows:

  `PHASE` *`obj_name`*`,*,SVA`

- The `CATALOG` card if the VSE JCL option `DECK` is in effect as follows:

  `CATALOG` *`obj_name`*`.OBJ REPLACE=YES`

The `PHASE` or `CATALOG` card is written as the first record to the output file from the `uconvdef` utility in order to produce a valid input stream for the linkage editor or the VSE Librarian respectively.

VERBOSE | <u>NOVERBOSE</u>
Specifies whether the source file statements will be written to `stdout`.

If the `VERBOSE` option is specified, the source file statements from the input file will be written to `stdout` as they are encountered.

If the NOVERBOSE option is specified (or by default), the source file statements will not be written to stdout.

Messages from the uconvdef utility are written to stdout/stderr.

Example

In the following example, the input source file is member EDCUUEYU.UCMAP in the default LE/VSE installation sublibrary PRD2.SCEEBASE. The output from the uconvdef utility is link-edited and a phase FRED02 is placed in the VSE/Librarian sublibrary FRED.LIB.

```
// JOB UCDESAMP
// LIBDEF *,SEARCH=PRD2.SCEEBASE
// LIBDEF PHASE,CATALOG=FRED.LIB
// OPTION LINK,CATAL
// EXEC EDCUCDEF,PARM='IFILE(DD:PRD2.SCEEBASE(EDCUUEYU.UCMAP)),        X
               NAME(FRED02)'
/*
// EXEC LNKEDT
/*
// EXEC LISTLOG
/&
```

The iconv Utility

The iconv utility converts the characters in the input file from one coded character set (code set) definition to another code set definition, and writes the characters to the output file.

The conversion is performed according to the tables generated by the genxlt utility. The tables used are determined by the CC codes of the “from” and “to” code sets, appended to the four-character string EDCU. See “The genxlt Utility” on page 371 for more information.

The iconv utility uses the iconv_open(), iconv(), and iconv_close() functions to convert the input file records from the coded character set definition for the input code page to the coded character set definition for the output code page. There is one record in the output file for each record in the input file. No padding or truncation of records is performed.

When conversions are performed between single-byte code pages, the output records are the same length as the input records. When conversions are performed between double-byte code pages, the output records may be longer or shorter than the input records because the shift-out and shift-in characters may be added or removed.

The iconv utility has the following options. If the same option is specified more than once, the last option specified is used. The options are specified on the EXEC PARM, and must be separated by spaces or commas.

IFILE(...)
Specifies the input file as follows:

iconv IFILE Option

```
►►──IFILE──(──────────────────────────────────────────────────────────────────►
              ┌─DD:SYS001-SYSUT1─┐   ┌────────────────────┌─80──────┐──┐
              └─i_name───────────┘   └─,──lrecl──=──────└─i_lrecl─┘──┘
►─┬────────────────────────────┬─┬───────────────────────────────────┬─)─────────►◄
  │                 ┌─FB──────┐ │ │                    ┌─4000──────┐  │
  └─,──recfm──=─────┴─i_recfm─┴─┘ └─,──blksize──=──────┴─i_blksize─┴──┘
```

where

i_name
: is the input file specification in any of the formats supported by the `fopen()` function. (See the “Opening Files” sections in this book for additional information regarding file specification formats.) Default is `DD:SYS001-SYSUT1`.

i_lrecl
: is the input file logical record length in bytes. Default is `80`.

i_recfm
: is the input file record format. Default is `FB`.

i_blksize
: is the input file block size in bytes. Default is `4000`.

Notes:

1. All of the above parameters are optional.
2. If more than one parameter is specified, a comma (,) must be used as a separator between each parameter.
3. No spaces are allowed.
4. If *i_name* is specified, it must be the first parameter. Other parameters are non-positional.

Example:

```
IFILE(dd:file1,recfm=vb)
```

OFILE(...)
Specifies the output file as follows:

iconv OFILE Option

```
►►──OFILE──(─┬─────────────────────┬─┬─────────────────────────────┬──────────►
             ├─DD:SYS002-SYSUT2──┤ │ │                ┌─i_lrecl─┐  │
             └─o_name────────────┘ │ └─,──lrecl──=────┴─o_lrecl─┴──┘

►─┬────────────────────────────────┬─┬───────────────────────────────────────┬─)─────────►◄
  │                 ┌─i_recfm─┐    │ │                    ┌─i_blksize─┐      │
  └─,──recfm──=─────┴─o_recfm─┴────┘ └─,──blksize──=──────┴─o_blksize─┴──────┘
```

where

o_name
: is the output file specification in any of the formats supported by the `fopen()` function. (See the “Opening Files” sections in this book for additional information regarding file specification formats.) Default is `DD:SYS002-SYSUT2`.

o_lrecl
: is the output file logical record length in bytes. Default is the explicit or implicit input file logical record length.

o_recfm
: is the output file record format. Default is the explicit or implicit input file record format.

o_blksize
: is the output file block size in bytes. Default is the explicit or implicit input file block size.

Notes:

1. All of the above parameters are optional.
2. If more than one parameter is specified, a comma (,) must be used as a separator between each parameter.
3. No spaces are allowed.
4. If *o_name* is specified, it must be the first parameter. Other parameters are non-positional.

Example:

```
OFILE('my.file',lrecl=121,blksize=4840)
```

FROMCODE(*from_code_set***)**
Specifies the name *from_code_set*, of the code set in which the input data is encoded.

TOCODE(*to_code_set***)**
Specifies the name *to_code_set*, of the code set to which the output data is to be converted.

Messages from the `iconv` utility are written to `stdout/stderr`.

Example

In the following example, the input file is INPUT.FILE in code page IBM-037 and the output file is OUTPUT.FILE in code page IBM-1047.

```
// JOB ICNVSAMP
// LIBDEF *,SEARCH=PRD2.SCEEBASE
// EXEC EDCICONV,PARM='IFILE(''INPUT.FILE''),OFILE(''OUTPUT.FILE''),   X
               FROMCODE(IBM-037),TOCODE(IBM-1047)'
/*
// EXEC LISTLOG
/&
```

Code Conversion Functions

The `iconv_open()`, `iconv()`, and `iconv_close()` library functions can be called from C source to initialize and perform the character conversions from one character set encoding to another.

See *LE/VSE C Run-Time Library Reference* for additional information regarding these library functions.

Code Set Converters Supplied

There is a set of code set converters that are provided in the base component of LE/VSE.

The converters are as follows:

- Code set converters between Latin-1 and some non-Latin-1 coded character sets and coded character set IBM-1047. The code set conversions between the non-Latin-1 coded character sets and IBM-1047 use the “Round Trip Conversion” that follows the direction of the IBM CDRA.
- Code set converters to convert to and from IBM-1047, IBM-850, and ISO8859-1.
- Code set converters between the Japanese coded character sets. These conversions will use the “Enforced subset match technique” according to IBM CDRA direction.

The code set converters provided as phases are shown in Table 48. Also shipped are the source files for these converters. The converters, and their source files whose member name extension is ".X", are in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE).

Because the "Round Trip Integrity" method is used, the string after conversion may contain characters that were not in the original string.

The converters that have the source code supplied (*GENXLT source* column marked "Yes") can be modified by the users. The converters for which conversions are performed by the library code (*GENXLT source* column marked "No") cannot be modified.

Table 48. Coded Character Set Conversion Table

FromCode	ToCode	GENXLT source	Phase Name
IBM-037	IBM-924	Yes	EDCUEAEZ
IBM-037	IBM-1047	Yes	EDCUEAEY
IBM-273	IBM-924	Yes	EDCUEBEZ
IBM-273	IBM-1047	Yes	EDCUEBEY
IBM-274	IBM-1047	Yes	EDCUECEY
IBM-274	IBM-1148	Yes	EDCUECHO
IBM-275	IBM-1047	Yes	EDCUEDEY
IBM-275	IBM-1148	Yes	EDCUEDHO
IBM-277	IBM-1047	Yes	EDCUEEEY
IBM-278	IBM-924	Yes	EDCUEFEZ
IBM-278	IBM-1047	Yes	EDCUEFEY
IBM-280	IBM-924	Yes	EDCUEGEZ
IBM-280	IBM-1047	Yes	EDCUEGEY
IBM-281	IBM-1047	Yes	EDCUEHEY
IBM-281	IBM-1148	Yes	EDCUEHHO
IBM-282	IBM-1047	Yes	EDCUEIEY
IBM-282	IBM-1148	Yes	EDCUEIHO
IBM-284	IBM-924	Yes	EDCUEJEZ
IBM-284	IBM-1047	Yes	EDCUEJEY
IBM-285	IBM-924	Yes	EDCUEKEZ
IBM-285	IBM-1047	Yes	EDCUEKEY
IBM-290	IBM-1027	Yes	EDCUELEX
IBM-290	IBM-1047	Yes	EDCUELEY
IBM-290	IBM-1148	Yes	EDCUELHO
IBM-297	IBM-924	Yes	EDCUEMEZ
IBM-297	IBM-1047	Yes	EDCUEMEY
IBM-437	IBM-1047	Yes	EDCUAVEY
IBM-500	IBM-924	Yes	EDCUEOEZ
IBM-500	IBM-1047	Yes	EDCUEOEY
IBM-500	IBM-1140	Yes	EDCUEOHA

Table 48. Coded Character Set Conversion Table (continued)

FromCode	ToCode	GENXLT source	Phase Name
IBM-500	IBM-1141	Yes	EDCUEOHB
IBM-500	IBM-1142	Yes	EDCUEOHE
IBM-500	IBM-1143	Yes	EDCUEOHF
IBM-500	IBM-1144	Yes	EDCUEOHG
IBM-500	IBM-1145	Yes	EDCUEOHJ
IBM-500	IBM-1146	Yes	EDCUEOHK
IBM-500	IBM-1147	Yes	EDCUEOHM
IBM-500	IBM-1149	Yes	EDCUEOHR
IBM-850	IBM-1047	Yes	EDCUAAEY
IBM-850	IBM-1140	Yes	EDCUAAHA
IBM-850	IBM-1141	Yes	EDCUAAHB
IBM-850	IBM-1142	Yes	EDCUAAHE
IBM-850	IBM-1143	Yes	EDCUAAHF
IBM-850	IBM-1144	Yes	EDCUAAHG
IBM-850	IBM-1145	Yes	EDCUAAHJ
IBM-850	IBM-1146	Yes	EDCUAAHK
IBM-850	IBM-1147	Yes	EDCUAAHM
IBM-850	IBM-1148	Yes	EDCUAAHO
IBM-850	IBM-1149	Yes	EDCUAAHR
IBM-858	IBM-1047	Yes	EDCUAIEY
IBM-858	IBM-1140	Yes	EDCUAIHA
IBM-858	IBM-1141	Yes	EDCUAIHB
IBM-858	IBM-1142	Yes	EDCUAIHE
IBM-858	IBM-1143	Yes	EDCUAIHF
IBM-858	IBM-1144	Yes	EDCUAIHG
IBM-858	IBM-1145	Yes	EDCUAIHJ
IBM-858	IBM-1146	Yes	EDCUAIHK
IBM-858	IBM-1147	Yes	EDCUAIHM
IBM-858	IBM-1148	Yes	EDCUAIHO
IBM-858	IBM-1149	Yes	EDCUAIHR
IBM-871	IBM-924	Yes	EDCUEREZ
IBM-871	IBM-1047	Yes	EDCUEREY
IBM-875	IBM-1047	Yes	EDCUESEY
IBM-875	ISO8859-7	Yes	EDCUESI7
IBM-924	IBM-037	Yes	EDCUEZEA
IBM-924	IBM-273	Yes	EDCUEZEB
IBM-924	IBM-278	Yes	EDCUEZEF
IBM-924	IBM-280	Yes	EDCUEZEG
IBM-924	IBM-284	Yes	EDCUEZEJ
IBM-924	IBM-285	Yes	EDCUEZEK

Table 48. Coded Character Set Conversion Table (continued)

FromCode	ToCode	GENXLT source	Phase Name
IBM-924	IBM-297	Yes	EDCUEZEM
IBM-924	IBM-500	Yes	EDCUEZEO
IBM-924	IBM-871	Yes	EDCUEZER
IBM-924	IBM-1047	Yes	EDCUEZEY
IBM-924	IBM-1140	Yes	EDCUEZHA
IBM-924	IBM-1141	Yes	EDCUEZHB
IBM-924	IBM-1142	Yes	EDCUEZHE
IBM-924	IBM-1143	Yes	EDCUEZHF
IBM-924	IBM-1144	Yes	EDCUEZHG
IBM-924	IBM-1145	Yes	EDCUEZHJ
IBM-924	IBM-1146	Yes	EDCUEZHK
IBM-924	IBM-1147	Yes	EDCUEZHM
IBM-924	IBM-1148	Yes	EDCUEZHO
IBM-924	IBM-1149	Yes	EDCUEZHR
IBM-924	IBM-4971	Yes	EDCUEZHS
IBM-930	IBM-932	No	EDCUEUAB
IBM-930	IBM-eucJP	No	EDCUEUAC
IBM-932	IBM-930	No	EDCUABEU
IBM-932	IBM-939	No	EDCUABEV
IBM-939	IBM-932	No	EDCUEVAB
IBM-939	IBM-eucJP	No	EDCUEVAC
IBM-1026	IBM-1047	Yes	EDCUEWEY
IBM-1027	IBM-290	Yes	EDCUEXEL
IBM-1027	IBM-1047	Yes	EDCUEXEY
IBM-1027	IBM-1148	Yes	EDCUEXHO
IBM-1047	IBM-037	Yes	EDCUEYEA
IBM-1047	IBM-273	Yes	EDCUEYEB
IBM-1047	IBM-274	Yes	EDCUEYEC
IBM-1047	IBM-275	Yes	EDCUEYED
IBM-1047	IBM-277	Yes	EDCUEYEE
IBM-1047	IBM-278	Yes	EDCUEYEF
IBM-1047	IBM-280	Yes	EDCUEYEG
IBM-1047	IBM-281	Yes	EDCUEYEH
IBM-1047	IBM-282	Yes	EDCUEYEI
IBM-1047	IBM-284	Yes	EDCUEYEJ
IBM-1047	IBM-285	Yes	EDCUEYEK
IBM-1047	IBM-290	Yes	EDCUEYEL
IBM-1047	IBM-297	Yes	EDCUEYEM
IBM-1047	IBM-437	Yes	EDCUEYAV
IBM-1047	IBM-500	Yes	EDCUEYEO

Table 48. Coded Character Set Conversion Table (continued)

FromCode	ToCode	GENXLT source	Phase Name
IBM-1047	IBM-850	Yes	EDCUEYAA
IBM-1047	IBM-858	Yes	EDCUEYAI
IBM-1047	IBM-871	Yes	EDCUEYER
IBM-1047	IBM-875	Yes	EDCUEYES
IBM-1047	IBM-924	Yes	EDCUEYEZ
IBM-1047	IBM-1026	Yes	EDCUEYEW
IBM-1047	IBM-1027	Yes	EDCUEYEX
IBM-1047	IBM-1140	Yes	EDCUEYHA
IBM-1047	IBM-1141	Yes	EDCUEYHB
IBM-1047	IBM-1142	Yes	EDCUEYHE
IBM-1047	IBM-1143	Yes	EDCUEYHF
IBM-1047	IBM-1144	Yes	EDCUEYHG
IBM-1047	IBM-1145	Yes	EDCUEYHJ
IBM-1047	IBM-1146	Yes	EDCUEYHK
IBM-1047	IBM-1147	Yes	EDCUEYHM
IBM-1047	IBM-1148	Yes	EDCUEYHO
IBM-1047	IBM-1149	Yes	EDCUEYHR
IBM-1047	ISO8859-1	Yes	EDCUEYI1
IBM-1140	IBM-500	Yes	EDCUHAEO
IBM-1140	IBM-850	Yes	EDCUHAAA
IBM-1140	IBM-858	Yes	EDCUHAAI
IBM-1140	IBM-924	Yes	EDCUHAEZ
IBM-1140	IBM-1047	Yes	EDCUHAEY
IBM-1140	IBM-1148	Yes	EDCUHAHO
IBM-1140	ISO8859-1	Yes	EDCUHAI1
IBM-1141	IBM-500	Yes	EDCUHBEO
IBM-1141	IBM-850	Yes	EDCUHBAA
IBM-1141	IBM-858	Yes	EDCUHBAI
IBM-1141	IBM-924	Yes	EDCUHBEZ
IBM-1141	IBM-1047	Yes	EDCUHBEY
IBM-1141	IBM-1148	Yes	EDCUHBHO
IBM-1141	ISO8859-1	Yes	EDCUHBI1
IBM-1142	IBM-500	Yes	EDCUHEEO
IBM-1142	IBM-850	Yes	EDCUHEAA
IBM-1142	IBM-858	Yes	EDCUHEAI
IBM-1142	IBM-924	Yes	EDCUHEEZ
IBM-1142	IBM-1047	Yes	EDCUHEEY
IBM-1142	IBM-1148	Yes	EDCUHEHO
IBM-1142	ISO8859-1	Yes	EDCUHEI1
IBM-1143	IBM-500	Yes	EDCUHFEO

Table 48. Coded Character Set Conversion Table (continued)

FromCode	ToCode	GENXLT source	Phase Name
IBM-1143	IBM-850	Yes	EDCUHFAA
IBM-1143	IBM-858	Yes	EDCUHFAI
IBM-1143	IBM-924	Yes	EDCUHFEZ
IBM-1143	IBM-1047	Yes	EDCUHFEY
IBM-1143	IBM-1148	Yes	EDCUHFHO
IBM-1143	ISO8859-1	Yes	EDCUHFI1
IBM-1144	IBM-500	Yes	EDCUHGEO
IBM-1144	IBM-850	Yes	EDCUHGAA
IBM-1144	IBM-858	Yes	EDCUHGAI
IBM-1144	IBM-924	Yes	EDCUHGEZ
IBM-1144	IBM-1047	Yes	EDCUHGEY
IBM-1144	IBM-1148	Yes	EDCUHGHO
IBM-1144	ISO8859-1	Yes	EDCUHGI1
IBM-1145	IBM-500	Yes	EDCUHJEO
IBM-1145	IBM-850	Yes	EDCUHJAA
IBM-1145	IBM-858	Yes	EDCUHJAI
IBM-1145	IBM-924	Yes	EDCUHJEZ
IBM-1145	IBM-1047	Yes	EDCUHJEY
IBM-1145	IBM-1148	Yes	EDCUHJHO
IBM-1145	ISO8859-1	Yes	EDCUHJI1
IBM-1146	IBM-500	Yes	EDCUHKEO
IBM-1146	IBM-850	Yes	EDCUHKAA
IBM-1146	IBM-858	Yes	EDCUHKAI
IBM-1146	IBM-924	Yes	EDCUHKEZ
IBM-1146	IBM-1047	Yes	EDCUHKEY
IBM-1146	IBM-1148	Yes	EDCUHKHO
IBM-1146	ISO8859-1	Yes	EDCUHKI1
IBM-1147	IBM-500	Yes	EDCUHMEO
IBM-1147	IBM-850	Yes	EDCUHMAA
IBM-1147	IBM-858	Yes	EDCUHMAI
IBM-1147	IBM-924	Yes	EDCUHMEZ
IBM-1147	IBM-1047	Yes	EDCUHMEY
IBM-1147	IBM-1148	Yes	EDCUHMHO
IBM-1147	ISO8859-1	Yes	EDCUHMI1
IBM-1148	IBM-274	Yes	EDCUHOEC
IBM-1148	IBM-275	Yes	EDCUHOED
IBM-1148	IBM-281	Yes	EDCUHOEH
IBM-1148	IBM-282	Yes	EDCUHOEI
IBM-1148	IBM-290	Yes	EDCUHOEL
IBM-1148	IBM-850	Yes	EDCUHOAA

Table 48. Coded Character Set Conversion Table (continued)

FromCode	ToCode	GENXLT source	Phase Name
IBM-1148	IBM-858	Yes	EDCUHOAI
IBM-1148	IBM-924	Yes	EDCUHOEZ
IBM-1148	IBM-1027	Yes	EDCUHOEX
IBM-1148	IBM-1047	Yes	EDCUHOEY
IBM-1148	IBM-1140	Yes	EDCUHOHA
IBM-1148	IBM-1141	Yes	EDCUHOHB
IBM-1148	IBM-1142	Yes	EDCUHOHE
IBM-1148	IBM-1143	Yes	EDCUHOHF
IBM-1148	IBM-1144	Yes	EDCUHOHG
IBM-1148	IBM-1145	Yes	EDCUHOHJ
IBM-1148	IBM-1146	Yes	EDCUHOHK
IBM-1148	IBM-1147	Yes	EDCUHOHM
IBM-1148	IBM-1149	Yes	EDCUHOHR
IBM-1148	ISO8859-1	Yes	EDCUHOI1
IBM-1149	IBM-500	Yes	EDCUHREO
IBM-1149	IBM-850	Yes	EDCUHRAA
IBM-1149	IBM-858	Yes	EDCUHRAI
IBM-1149	IBM-924	Yes	EDCUHREZ
IBM-1149	IBM-1047	Yes	EDCUHREY
IBM-1149	IBM-1148	Yes	EDCUHRHO
IBM-1149	ISO8859-1	Yes	EDCUHRI1
IBM-4909	IBM-4971	Yes	EDCUIAHS
IBM-4971	IBM-924	Yes	EDCUHSEZ
IBM-4971	IBM-4909	Yes	EDCUHSIA
IBM-eucJP	IBM-930	No	EDCUACEU
IBM-eucJP	IBM-939	No	EDCUACEV
ISO8859-1	IBM-1047	Yes	EDCUI1EY
ISO8859-1	IBM-1140	Yes	EDCUI1HA
ISO8859-1	IBM-1141	Yes	EDCUI1HB
ISO8859-1	IBM-1142	Yes	EDCUI1HE
ISO8859-1	IBM-1143	Yes	EDCUI1HF
ISO8859-1	IBM-1144	Yes	EDCUI1HG
ISO8859-1	IBM-1145	Yes	EDCUI1HJ
ISO8859-1	IBM-1146	Yes	EDCUI1HK
ISO8859-1	IBM-1147	Yes	EDCUI1HM
ISO8859-1	IBM-1148	Yes	EDCUI1HO
ISO8859-1	IBM-1149	Yes	EDCUI1HR
ISO8859-7	IBM-875	Yes	EDCUI7ES
ISO8859-9	IBM-1026	Yes	EDCUI9EW

The following code set converters are also supplied. These converters are used by the code set converters between the codesets IBM-930, IBM-932, IBM-939, and IBM-eucJP.

FromCode	ToCode	GENXLT source	Phase Name
IBM-290	IBM-932	Yes	EDCUELAB
IBM-290	IBM-eucJP	No	EDCUELAC
IBM-300	IBM-eucJP	No	EDCUENAC
IBM-300	IBM-932	No	EDCUENAB
IBM-932	IBM-290	Yes	EDCUABEL
IBM-932	IBM-300	No	EDCUABEN
IBM-932	IBM-1027	Yes	EDCUABEX
IBM-1027	IBM-932	Yes	EDCUEXAB
IBM-1027	IBM-eucJP	No	EDCUEXAC
IBM-eucJP	IBM-290	No	EDCUACEL
IBM-eucJP	IBM-300	No	EDCUACEN
IBM-eucJP	IBM-1027	No	EDCUACEX

Universal Coded Character Set Converters

You can use the name `UCS-2` to request setup for conversion to and from UCS-2. For example, `iconv_open("UCS-2", "IBM-1047")` requests setup for conversion from IBM-1047 character encoding to UCS-2 character encoding. You can also use the name `UTF-8` to request setup for conversion to and from Transform Format 8, UTF-8. This is specified in the Unicode Standard, Version 2.1, Appendixes A-7 and A-8. For example, `iconv_open("UTF-8", "IBM-1047")` requests setup for conversion from IBM-1047 character encoding to UTF-8 character encoding.

The code set converters provided as phases are shown in Table 49. The converter names are of the form `EDCUUccU`; where cc is the CC code associated with a particular coded character set name. The converters are used for conversions to and from UTF-8 as well as UCS-2, and are in the LE/VSE installation sublibrary (the default is PRD2.SCEEBASE).

The `uconvdef` utility produces the phases required by `iconv_open()` from UCS-2 source files.

Table 49. UCS-2 Converters

Codeset Name	CC code	Phase Name
IBM-850	AA	EDCUUAAU
IBM-4946	AA	EDCUUAAU
IBM-301	AB	EDCUUABU
IBM-942	AB	EDCUUABU
IBM33722	AC	EDCUUACU
IBM-EUCJP	AC	EDCUUACU
IBM-922	AD	EDCUUADU
IBM-1046	AF	EDCUUAFU
IBM-859	AK	EDCUUAKU
IBM-1124	AU	EDCUUAUU
IBM-437	AV	EDCUUAVU
IBM-921	BD	EDCUUBDU
IBM-866	BE	EDCUUBEU
IBM-862	BH	EDCUUBHU
IBM-874	BU	EDCUUBUU
IBM-964	BW	EDCUUBWU
IBM-1383	BY	EDCUUBYU
IBM-EUCKR	BZ	EDCUUBZU
IBM-970	BZ	EDCUUBZU
IBM-861	CA	EDCUUCAU
IBM-852	CB	EDCUUCBU
IBM-855	CE	EDCUUCEU
IBM-864	CF	EDCUUCFU
IBM-869	CG	EDCUUCGU
IBM-856	CH	EDCUUCHU
IBM-1115	CL	EDCUUCLU

Table 49. UCS-2 Converters (continued)

Codeset Name	CC code	Phase Name
IBM-1380	CM	EDCUUCMU
IBM-904	CN	EDCUUCNU
IBM-927	CO	EDCUUCOU
IBM-1088	CP	EDCUUCPU
IBM-951	CQ	EDCUUCQU
IBM-1363	CU	EDCUUCUU
IBM-938	CW	EDCUUCWU
IBM-948	CW	EDCUUCWU
IBM-1381	CY	EDCUUCYU
IBM-949	CZ	EDCUUCZU
IBM-1252	DA	EDCUUDAU
IBM-1250	DB	EDCUUDBU
IBM-1251	DE	EDCUUDEU
IBM-1256	DF	EDCUUDFU
IBM-1253	DG	EDCUUDGU
IBM-1255	DH	EDCUUDHU
IBM-5348	DJ	EDCUUDJU
IBM-5349	DK	EDCUUDKU
BIG5	DW	EDCUUDWU
IBM-947	DW	EDCUUDWU
IBM-950	DW	EDCUUDWU
IBM-928	DY	EDCUUDYU
IBM-936	DY	EDCUUDYU
IBM-946	DY	EDCUUDYU
IBM-037	EA	EDCUUEAU
IBM-28709	EA	EDCUUEAU
IBM-273	EB	EDCUUEBU
IBM-274	EC	EDCUUECU
IBM-275	ED	EDCUUEDU
IBM-277	EE	EDCUUEEU
IBM-278	EF	EDCUUEFU
IBM-280	EG	EDCUUEGU
IBM-282	EI	EDCUUEIU
IBM-284	EJ	EDCUUEJU
IBM-290	EL	EDCUUELU
IBM-297	EM	EDCUUEMU
IBM-300	EN	EDCUUENU
IBM-4396	EN	EDCUUENU
IBM-500	EO	EDCUUEOU
IBM-838	EP	EDCUUEPU

Table 49. UCS-2 Converters (continued)

Codeset Name	CC code	Phase Name
IBM-870	EQ	EDCUUEQU
IBM-871	ER	EDCUUERU
IBM-875	ES	EDCUUESU
IBM-880	ET	EDCUUETU
IBM-930	EU	EDCUUEUU
IBM-5026	EU	EDCUUEUU
IBM-939	EV	EDCUUEVU
IBM-5035	EV	EDCUUEVU
IBM-1026	EW	EDCUUEWU
IBM-1027	EX	EDCUUEXU
IBM-1047	EY	EDCUUEYU
IBM-924	EZ	EDCUUEZU
IBM-424	FB	EDCUUFBU
IBM-1122	FD	EDCUUFDU
IBM-1025	FE	EDCUUFEU
IBM-420	FF	EDCUUFFU
IBM-1112	GD	EDCUUGDU
IBM-836	GL	EDCUUGLU
IBM-837	GM	EDCUUGMU
IBM-835	GO	EDCUUGOU
IBM-833	GP	EDCUUGPU
IBM-834	GQ	EDCUUGQU
IBM-1364	GU	EDCUUGUU
IBM-937	GW	EDCUUGWU
IBM-935	GY	EDCUUGYU
IBM-5031	GY	EDCUUGYU
IBM-933	GZ	EDCUUGZU
IBM-1140	HA	EDCUUHAU
IBM-1141	HB	EDCUUHBU
IBM-16804	HC	EDCUUHCU
IBM-1157	HD	EDCUUHDU
IBM-1142	HE	EDCUUHEU
IBM-1143	HF	EDCUUHFU
IBM-1144	HG	EDCUUHGU
IBM-12712	HH	EDCUUHHU
IBM-1145	HJ	EDCUUHJU
IBM-1146	HK	EDCUUHKU
IBM-8482	HL	EDCUUHLU
IBM-1147	HM	EDCUUHMU
IBM-1148	HO	EDCUUHOU

Table 49. UCS-2 Converters (continued)

Codeset Name	CC code	Phase Name
IBM-1160	HP	EDCUUHPU
IBM-1149	HR	EDCUUHRU
IBM-4971	HS	EDCUUHSU
IBM-1154	HT	EDCUUHTU
IBM-1155	HW	EDCUUHWU
IBM-5123	HX	EDCUUHXU
IBM-1156	HZ	EDCUUHZU
IBM-819	I1	EDCUUI1U
IBM-912	I2	EDCUUI2U
IBM-914	I4	EDCUUI4U
IBM-915	I5	EDCUUI5U
IBM-1089	I6	EDCUUI6U
IBM-813	I7	EDCUUI7U
IBM-916	I8	EDCUUI8U
IBM-920	I9	EDCUUI9U
IBM-4909	IA	EDCUUIAU
IBM-1371	KA	EDCUUKAU
IBM-1370	LA	EDCUULAU
IBM-902	LD	EDCUULDU
IBM-872	LE	EDCUULEU
IBM-808	LF	EDCUULFU
IBM-9061	LG	EDCUULGU
IBM-901	LH	EDCUULHU
IBM-9238	LI	EDCUULIU
IBM-867	LJ	EDCUULJU
IBM-1161	LU	EDCUULUU
IBM-1153	MB	EDCUUMBU
IBM-5346	NB	EDCUUNBU
IBM-5347	NE	EDCUUNEU
IBM-5352	NF	EDCUUNFU
IBM-9044	NG	EDCUUNGU
IBM-5351	NH	EDCUUNHU
IBM-5350	NI	EDCUUNIU
IBM-17248	NJ	EDCUUNJU

Codeset Conversion Using UCS-2

LE/VSE `iconv` supports use of UCS-2 as an intermediate code set for conversion of characters encoded in one code set to another. The `_ICONV_UCS2` environment variable instructs `iconv_open("Y", "X")` whether or not to set up indirect conversion from code set X to code set Y using UCS-2 as an intermediate code set. Values `iconv_open()` recognizes for `_ICONV_UCS2` are:

1 Set up indirect conversion using UCS-2 first. If this fails, try to set up direct conversion.

2 Set up direct conversion first. If this fails, try to set up indirect conversion using UCS-2. This is the default.

O Only set up indirect conversion using UCS-2. If required converters cannot be found, the `iconv_open()` request is not successful.

N Never set up indirect conversion using UCS-2. If a direct converter cannot be found, the `iconv_open()` request fails.

Notes:

1. If the value of the `_ICONV_UCS2` environment variable allows `iconv_open("Y", "X")` to use UCS-2 as an intermediate code set when it cannot find a direct converter from X to Y, `iconv_open()` will attempt to do so even if X and Y are not compatible code sets. That is, even if character sets encoded by X and Y are not the same, `iconv_open()` will set up conversion from X to UCS-2 to Y.
2. The application must specify compatible source and target code set names on various `iconv_open()` requests. For example, this can be accomplished by using a code set registry such as is used by DCE (Distributed Computing Environment) to prevent `iconv` setup for conversion from incompatible code sets.

UCMAP Source Format

A UCMAP source file defines UCS-2 conversion mappings for input to the `uconvdef` utility. Conversion mapping values are defined using UCS-2 symbolic character names followed by character encoding (code point) values for the multibyte code set. For example:

```
<U0020> \x20
```

represents the mapping between the <U0020> UCS-2 symbolic character name for the space character and the \x20 hexadecimal code point for the space character in ASCII.

In addition to the code set mappings, directives are interpreted by the `uconvdef` utility to produce the compiled table. These directives must precede the code set mapping section. They consist of the following keywords surrounded by <> (angle brackets), starting in column 1, followed by white space and the value to be assigned to the symbol:

<comment_char>
Character used to denote start of escape sequence. Default escape character is <number_sign> (#). In UCMAP source shipped with LE/VSE, <percent_sign> (%) is specified for <comment_char>.

<escape_char>
Character used to denote start of escape sequence. Default escape character is <backslash> (\). In UCMAP source shipped with LE/VSE, <slash> (/) is specified for <escape_char>.

<code_set_name>
The name of the coded character set, enclosed in quotation marks ("), for which the character set description file is defined.

<mb_cur_max>
The maximum number of bytes in a multibyte character. The default value is 1.

<mb_cur_min>
An unsigned positive integer value that defines the minimum number of bytes in a character for the encoded character set. The value is less than or equal to <mb_cur_max>. If not specified, the minimum number is equal to <mb_cur_max>.

<char_name_mask>
A quoted string consisting of format specifiers for the UCS-2 symbolic names. This must be a value of `Uxxxx`, where xxxx is 4 hexadecimal digits which represent the UCS-2 code point for the character. An example of a symbolic character name based on this mask is <U0020>, which is the UCS-2 space character.

<uconv_class>
Specifies the type of the code set. It must be one of the following:

SBCS Single-byte encoding

DBCS Stateless double-byte, single-byte, or mixed encodings

EBCDIC_STATEFUL
Stateful double-byte, single-byte, or mixed encodings

MBCS Stateless multibyte encoding

This type is used to direct `uconvdef` on the type of table to build. It is also stored in the table to indicate the type of processing algorithm in the UCS conversion methods.

<locale>
Specifies the default locale name to be used if locale information is needed.

<subchar>
Specifies the encoding of the default substitute character in the multibyte code set.

The mapping definition section consists of a sequence of mapping definition lines preceded by a CHARMAP declaration and terminated by an END CHARMAP declaration. Empty lines and lines containing <comment_char> in the first column are ignored.

Symbolic character names in mapping lines must follow the pattern specified in the <char_name_mask>, except for the reserved symbolic name, <unassigned>, that indicates the associated code points are unassigned.

Each noncomment line of the character set mapping definition must be in one of the following formats:

1. <symbolic_name> <encoding> <comments>

   For example:

   ```
   <U3004> \x81\x57
   ```

   This format defines a single symbolic character name and a corresponding encoding.

   The encoding part is expressed as one or more concatenated decimal, hexadecimal, or octal constants in the following formats:

   - <escape_char> <decimal byte value>
   - <escape_char> <hexadecimal byte value>
   - <escape_char> <octal byte value>

Decimal constants are represented by two or more decimal digits preceded by the escape character and the lowercase letter d, as in \d97 or \d143. Hexadecimal constants are represented by two or more hexadecimal digits preceded by an escape character and the lowercase letter x, as in \x61 or \x8f. Octal constants are represented by two or more octal digits preceded by an escape character.

Each constant represents a single-byte value. When constants are concatenated for multibyte character values, the last value specifies the least significant octet and preceding constants specify successively more significant octets.

2. <symbolic-name> <symbolic_name> <encoding> <comments>

   For example:

   ```
   <U3003><U3006> \x81\x56
   ```

   This format defines a range of symbolic character names and corresponding encodings. The range is interpreted as a series of symbolic names formed from the alphabetic prefix and all the values in the range defined by the numeric suffixes.

   The listed encoding value is assigned to the first symbolic name, and subsequent symbolic names in the range are assigned corresponding incremental values. For example, the line:

   ```
   <U3003>...<U3006> \x81\x56
   ```

   is interpreted as:

   ```
   <U3003> \x81\x56
   <U3004> \x81\x57
   <U3005> \x81\x58
   <U3006> \x81\x59
   ```

3. <unassigned> <encoding> <comments>

   This format defines a range of one or more unassigned encodings. For example, the line:

   ```
   <unassigned> \x9b...\x9c
   ```

   is interpreted as:

   ```
   <unassigned> \x9b
   <unassigned> \x9c
   ```

The `localedef` Utility

A *locale* is the definition of the subset of your environment that depends on language and cultural conventions. The locale objects contain the rules and pointers to methods used to implement the language and cultural conventions. A locale object is made up of a number of categories, identified by name, that control specific aspects of the behavior of components of the system.

The locale objects are generated by first executing the `localedef` utility according to the rules defined in the locale definition file, and then compiling and link-editing the C source deck produced. The locale object is a phase that can be loaded via the operating system load and used by the locale specific library functions.

The options for the `localedef` utility are as follows. The options are specified on the `EXEC PARM` and are separated by spaces or commas. If the same option is specified more than once, the last specification of the option is used.

CHARMAP(*mbr_name***)**
Specifies a coded character set name. This name is converted to the name of a VSE/Librarian member containing the mapping of the character symbols to actual character encodings. The member name is the character set name with the - (dash) converted to an @ (at) sign and a ".K" extension. For example, IBM-1047 maps to member name IBM@1047.K.

If this option is not specified, the `localedef` utility defaults to use the coded character set IBM-1047.

FLAG(W|E)
The `FLAG` option controls whether warning messages are issued. If `FLAG(W)` is specified (or by default), warning and error messages are issued. If `FLAG(E)` is specified, only the error messages are issued.

BLDERR|NOBLDERR
If the `BLDERR` option is specified, the locale is generated even if errors are detected. If the `NOBLDERR` option is specified (or by default), the locale is not generated if an error is detected.

IFILE(...)

Specifies the input file as follows:

localedef IFILE Option

```
►►──IFILE──(──┬──────────────────┬──┬────────────────────────────┬────────►
              │ ┌─DD:SYS001-SYSUT1─┐│  │                ┌─80──────┐ │
              └─┴─i_name───────────┘┘  └─,──lrecl──=───┴─i_lrecl─┴─┘

►──┬────────────────────────────┬──┬─────────────────────────────────┬──)──────►◄
   │               ┌─FB──────┐  │  │                  ┌─4000──────┐  │
   └─,──recfm──=───┴─i_recfm─┴──┘  └─,──blksize──=────┴─i_blksize─┴──┘
```

where

i_name
: is the input file specification in any of the formats supported by the `fopen()` function. (See the "Opening Files" sections in this book for additional information regarding file specification formats.) Default is `DD:SYS001-SYSUT1`.

i_lrecl
: is the input file logical record length in bytes. Default is `80`.

i_recfm
: is the input file record format. Default is `FB`.

i_blksize
: is the input file block size in bytes. Default is `4000`.

Notes:

1. All of the above parameters are optional.
2. If more than one parameter is specified, a comma (,) must be used as a separator between each parameter.
3. No spaces are allowed.
4. If *i_name* is specified, it must be the first parameter. Other parameters are non-positional.

Example:

```
IFILE(dd:file1,recfm=vb)
```

OFILE(...)

Specifies the output file as follows:

localedef OFILE Option

```
►►──OFILE──(──┬──────────────────┬──┬────────────────────────────┬────────►
              │ ┌─DD:SYS002-SYSUT2─┐│  │                ┌─i_lrecl─┐ │
              └─┴─o_name───────────┘┘  └─,──lrecl──=───┴─o_lrecl─┴─┘

►──┬────────────────────────────┬──┬─────────────────────────────────┬──)──────►◄
   │               ┌─i_recfm─┐  │  │                  ┌─i_blksize─┐  │
   └─,──recfm──=───┴─o_recfm─┴──┘  └─,──blksize──=────┴─o_blksize─┴──┘
```

where

o_name
: is the output file specification in any of the formats supported by the `fopen()` function. (See the "Opening Files" sections in this book for additional information regarding file specification formats.) Default is `DD:SYS002-SYSUT2`.

o_lrecl
: is the output file logical record length in bytes. Default is the explicit or implicit input file logical record length.

o_recfm
: is the output file record format. Default is the explicit or implicit input file record format.

o_blksize
: is the output file block size in bytes. Default is the explicit or implicit input file block size.

Notes:

1. All of the above parameters are optional.
2. If more than one parameter is specified, a comma (,) must be used as a separator between each parameter.
3. No spaces are allowed.
4. If *o_name* is specified, it must be the first parameter. Other parameters are non-positional.

Example:

```
OFILE(DD:SYS006-MYFILE,lrecl=121,blksize=4840)
```

Messages from the `localedef` utility are written to `stdout/stderr`.

Defining Your Own Locales

Some of the locales shipped from LE/VSE 1.4.1 onwards are very large. As a result, if you plan to use locales as the basis for defining your own locales:

1. You will require a large amount of virtual storage in the partition in which the object code is to be generated. This applies to the:
   - `localedef` utility itself.
   - C compiler which is used to compile the code that is generated by the `localedef` utility.
2. You will probably need to:
   - Increase the size of the work files used by the C compiler.
   - Adjust some of the LE/VSE run-time options that control the amount of virtual storage used in the partition.
3. The C source code that is generated by the `localedef` utility might contain records with a column-size of more than 80. Therefore, you should use a variable-length record file in which to store the C source code.
4. As the C compiler will only accept variable-length records from a VSAM-managed SAM dataset, you should use VSAM-managed SAM as the intermediate C source file.

Examples

In the first example, the input source is a member TEXAN.L in a VSE/Librarian sublibrary LOCALE.WRK. Having created the C source file using the `localedef` utility, the C/VSE compiler and the linkage editor is used to produce a locale phase as member EDC$1TEY.PHASE in the LOCALE.WRK sublibrary.

```
* $$ JOB JNM=jobname,PDEST=(*,uid),LDEST=(*,uid),PRI=prty,CLASS=class
* $$ PUN DISP=I
// JOB jobname
// LIBDEF *,SEARCH=(PRD2.DBASE,PRD2.SCEEBASE,...)   1
// LIBDEF PHASE,CATALOG=LOCALE.WRK  2
// OPTION CATAL
/* ------------------------------------------------------------------
/* Step 1: Create the C source using the localedef utility
/* ------------------------------------------------------------------
// EXEC EDCLLDEF,PARM='IFILE(DD:LOCALE.WRK(TEXAN.L)),                X
            OFILE(DD:LOCALE.WRK(EDC$1TEY.C))'
/* ------------------------------------------------------------------
/* Step 2: Compile the generated C source and write the object module
/*         to SYSLNK
/* ------------------------------------------------------------------
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='/LIST,SOURCE,NAME(EDC$1TEY)'
#include "edc$1tey.c"
/*
/* ------------------------------------------------------------------
/* Step 3: Link-edit the object module to produce the phase
/* ------------------------------------------------------------------
   ENTRY INSTANTI  3
// EXEC LNKEDT
/*
/&
* $$ EOJ
```

Figure 83. Sample 1 `localedef` JCL

In the first example, please notice the following:

1 Code the LIBDEF search chain to include all the sublibrary names that the `localedef` utility and the C/VSE compiler needs.

2 This is the sublibrary to which the phase is written.

3 The entry point in the locale phase must be INSTANTI.

The second example shows how to generate the EDC$ZCGY locale to create the relevant locale phase.

```
// JOB     LOCALED
// LIBSDEF *,SEARCH=(PRD2.BASE,PRD2.SCEEBASE)
/. C -----------------------------------------------------------------
/. C Step 1 : Generate the C source using the localedef utility
/. C -----------------------------------------------------------------
// DLBL    CSOURCE,'%LOCALE.CSOURCE',,VSAM,DISP=(NEW,KEEP),    1      X
               RECORDS=(150000,500),RECSIZE=60
// EXEC    EDCLLDEF,PARM='/CHARMAP(IBM-935),IFILE(DD:SYSIPT),         X
               OFILE(DD:CSOURCE,RECFM=VB,LRECL=128,BLKSIZE=6000)'  2
   ... (locale source) ...
/*
/. C -----------------------------------------------------------------
/. C Step 2 : Compile the C source
/. C -----------------------------------------------------------------
// DLBL    IJSYS05,'%DOS.WORKFILE.SYS005',0,VSAM,DISP=(NEW,DELETE), 3 X
               RECORDS=(1500,200),RECSIZE=4096
// DLBL    IJSYS06,'%DOS.WORKFILE.SYS006',0,VSAM,DISP=(NEW,DELETE),   X
               RECORDS=(1500,200),RECSIZE=4096
// DLBL    CSOURCE,'%LOCALE.CSOURCE',,VSAM,DISP=(OLD,DELETE)    4
// LIBDEF  PHASE,CATALOG=...
// OPTION  CATAL
   PHASE EDC$ZCGY,*
// EXEC    EDCCOMP,SIZE=AUTO,PARM='BE(2K,512),H(32K,32K,ANY,FREE,1K,5',X    5
               PARM='12),LIBS(2K,512),STAC(4K,2K,BELOW,FREE),STO(,,,0',X
               PARM='),TRACE(,0)/NOLIST,SPILL(3900),IFILE(DD:CSOURCE)'  6
/. C -----------------------------------------------------------------
/. C Step 3 : Link edit
/. C -----------------------------------------------------------------
   ENTRY INSTANTI
// EXEC    LNKEDT
/&
```

Figure 84. Sample 2 `localedef` JCL

In this second example, please note the following:

1 The DLBL statement defines a VSAM-managed SAM data set to contain the C source. The C program generated from the EDC$ZHCN locale will consist of approximately 150,000 lines of code. The `RECSIZE=60` parameter defines the average record size for VSAM space calculation.

Note that this is a VSAM *implicit* definition. If you wish to predefine the file (*explicit* definition), you should use the following IDCAMS parameters:

```
 DEFINE CLUSTER(NAME(%LOCALE.CSOURCE) -
               NONINDEXED -
               RECORDFORMAT(VB) -
               RECORDSIZE(60 128) -
               RECORDS(150000 500) -
              )
```

2 The OFILE parameter defines where the `localedef` utility is to place the C source file that it generates. The `DD:CSOURCE` option relates it to the DLBL statement that defines the VSAM-managed SAM file. The `RECFM`, `LRECL` and `BLKSIZE` options define the format of the data set. These are required for an implicitly-defined file, and are ignored for an explicitly-defined file.

3 The work files that the C compiler uses need to be large enough to process the C source file. If your standard labels define smaller work files, you will need to provide DLBL statements to override them.

4 The DLBL statement defines the data set that contains the C source program. If the data set is explicitly defined, the DISP parameter should not be specified.

5 You need to specify a number of LE/VSE run-time options for the C compile step to reduce the amount of storage used by LE/VSE functions and allow more for the compiler itself. The JCL in this example was run in a partition with a virtual storage allocation of 25600K, and the run-time options that relate to storage allocations were specified as shown to reduce LE/VSE's own virtual storage requirements.

Note that, in order to use this form of the EXEC statement that allows up to 300 characters of information in the PARM parameter, your z/VSE system must have the PTF or the relevant APAR applied. The APAR numbers are:
(for VSE/ESA Version 1 - DY44232)
for VSE/ESA Version 2 - DY44173

6 The options after the slash are options for the compiler itself rather than for LE/VSE. The `IFILE(DD:CSOURCE)` option relates to the DLBL statement that defines the input source file.

Chapter 34. Coded Character Set Considerations with Locale Functions

Each EBCDIC *coded character set* consists of a mapping of all the available glyphs to their respective hex encodings and also to their unique Graphic Character Global Identifiers (GCGIDs). GCGIDs are unique identifiers assigned to each character in the Unicode standard. A *glyph* is the printed appearance of a character. Each coded character set is intended to serve one linguistic environment.

There is a wide variation among coded character sets: many glyphs do not appear in all coded character sets, and hexadecimal encodings for some glyphs differ from one coded character set to another. You may have trouble when you export a file from a system running in one coded character set to a system running in another coded character set. For example, a left bracket ([) entered under the APL-293 or Open Systems IBM-1047 coded character set appears as the capitalized Y-acute (Ý) in such common coded character sets as International 500, France 297, Germany 273, and US/Canada 037.

LE/VSE C Run-Time now contains the following extensions to prevent such problems:
- The `??=pragma filetag` directive (see "The `??=pragma filetag` Directive" on page 407).
- The `locale` compile-time option (see "Converting Coded Character Sets at Compile Time" on page 408).

These new facilities cause the compiler to respect your code page. Thus, you can enter source code with what appears to you to be the correct characters, and the compiler will recognize those characters.

The rest of this chapter discusses other ways to work efficiently in different locales.

Variant Character Detail

The POSIX Portable Character Set (PPCS) identifies the core set of 128 characters that are needed to write code and run applications. Of these, 13 characters are variant among the EBCDIC coded character sets.

Table 50 on page 400 lists these 13 characters and shows how they appear when the Open Systems coded character set IBM-1047 hexadecimal values are entered, on systems where different Country Extended Coded Character Sets are installed. These hex values are the ones expected by LE/VSE C Run-Time, and are consistent with the use of the APL-293 coded character set. Table 51 on page 400 lists the hexadecimal values assigned across some of the EBCDIC coded character sets for the 13 variant characters from the PPCS. Appendix C, "LE/VSE C Run-Time Code Point Mappings," on page 429 gives more information about the mapping of glyphs. Appendix A, "POSIX Character Set," on page 417 lists the full PPCS.

Table 50. Mappings of 13 PPCS Variant Characters

Character	Open Systems Hex Value (Default)	Open Systems IBM-1047 View	APL IBM-293 View	International IBM-500 View	France IBM-297 View	Germany IBM-273 View	US/ Canada IBM-037 View
left bracket	AD	[	[	Ý	Ý	Ý	Ý
right bracket	BD	]	]	ü	~	¨	¨
left brace	C0	{	{	{	é	ä	{
right brace	D0	}	}	}	è	ü	}
backslash	E0	\	\	\	ç	Ö	\
circumflex	5F	^	¬	^	^	^	¬
tilde	A1	~	~	~	ü	ß	~
exclamation mark	5A	!	!	]	§	Ü	!
pound (number) sign	7B	#	#	#	£	#	#
vertical bar	4F	|	|	!	!	!	|
accent grave	79	`	`	`	µ	`	`
dollar sign	5B	$	$	$	$	$	$
commercial "at"	7C	@	@	@	á	§	@

Two tables are available to show the full code point mappings for Open Systems coded character set IBM-1047 (Figure 91 on page 429) and for the APL coded character set IBM-293 (Figure 92 on page 430). If you look at those coded character sets, you will notice that coded character set 1047 is a "latinized" coded character set IBM-293, in the sense that all the APL code points have been replaced by Latin-1 code points, thus allowing a one-to-one mapping among coded character set IBM-1047 and all the other coded character sets in the Latin-1 group.

Although the official current coded character set for LE/VSE C Run-Time is now coded character set IBM-1047 (Open Systems), the coded character set IBM-293 *syntax* points are still being honored. Those points are the ones with syntactic relevance to the C/VSE compiler; they are listed in both Table 50 and Table 51.

Table 51. Mappings of Hex Encoding of 13 PPCS Variant Characters

Character Name	Glyph	GCGID	Open Systems IBM-1047 View	APL IBM-293 View	International IBM-500 View	France IBM-297 View	Germany IBM-273 View	US/ Canada IBM-037 View
left bracket	[	SM060000	AD	AD	4A	90	63	BA
right bracket	]	SM080000	BD	BD	5A	B5	FC	BB
left brace	{	SM110000	C0	C0	C0	51	43	C0
right brace	}	SM140000	D0	D0	D0	54	DC	D0
backslash	\	SM070000	E0	E0	E0	48	EC	E0
circumflex	^	SD150000	5F	5F	5F	5F	5F	B0
tilde	~	SD190000	A1	A1	A1	BD	59	A1
exclamation mark	!	SP020000	5A	5A	4F	4F	4F	5A
pound (number) sign	#	SM010000	7B	7B	7B	B1	7B	7B
vertical bar	|	SM130000	4F	4F	BB	BB	BB	4F

Table 51. Mappings of Hex Encoding of 13 PPCS Variant Characters (continued)

Character Name	Glyph	GCGID	Open Systems IBM-1047 View	APL IBM-293 View	International IBM-500 View	France IBM-297 View	Germany IBM-273 View	US/ Canada IBM-037 View
accent grave	`	SD130000	79	79	79	A0	79	79
dollar sign	$	SC030000	5B	5B	5B	5B	5B	5B
commercial "at"	@	SM050000	7C	7C	7C	44	B5	7C

Alternate Code Points

All syntactic code points that were supported in previous versions of LE/VSE C Run-Time will continue to be supported *if* you are compiling with the `nolocale` option.

The following four alternate code points are *not* supported with the `locale` compiler option. If your code relies on these alternate code points, IBM recommends that you start using the `??=pragma filetag` directive and the `locale` option instead, and stop using the alternate code points.

- The French open and close double quotation marks (« at X'8B' and » at X'9B'). These symbols no longer serve as alternates for the left and right braces ({ at X'C0' and } at X'D0').
- The APL-293 left brace and right brace (⇃ at X'8B' and ⊂ at X'9B'). These symbols no longer serve as alternates for the left and right braces either. These alternate code points were supported by the C/370 and AD/Cycle C/370 compilers (the `nolocale` option was required if you were using the AD/Cycle C/370 Version 1 Release 2 compiler).

For reasons of compatibility, the vertical bar character can be represented by two encodings, if you are not using a locale compiler option or if you are using the `nolocale` option. These two encodings are X'4F' and X'6A' If you do specify the `locale` option, each of these characters is represented by a unique value as specified in the LC_SYNTAX category of the selected locale.

Coding without Locale Support

If you want to avoid using the `locale` option of the compiler, you must use a hybrid coded character set. A *hybrid* piece of code is one where the data is in the local coded character set but the syntax is written *as if* it was in coded character set IBM-1047.

Using a Hybrid Coded Character Set

You can continue coding in the local coded character set, writing the syntax *as if* it was in coded character set IBM-1047. This solution uses the existing behavior of the compiler and will continue to be supported. However, this method is not recommended. The code becomes difficult to read and may not even look like C code anymore. There may be ambiguities in the code. Finally, exporting code to another site can be difficult because the mapping between the hybrid characters actually used and the target coded character set may not be exact.

The following example illustrates these difficulties.

EDCXGCC1

```
 /* EDCXGCC1
    This example uses a hybrid coded character set.

    This has strings in codepage 273 with APL 293 syntax, and is a
    pre-locale source file for a user in Germany.
  */

#define MAX_NAMES          20
#define MAX_NAME_LEN       80
#define STR(num)           #num
#define SCAN_FORMAT(len)   "%"STR(len)"s %"STR(len)"s"

struct NameList ä 1
  char firstÝMAX_NAME_LEN+1¨; 2 3
  char surnameÝMAX_NAME_LEN+1¨; 2 3
ü; 4

int compareNames(const void *elem1, const void *elem2) ä 1
  struct NameList *name1 = (struct NameList *) elem1;
  struct NameList *name2 = (struct NameList *) elem2;
  int surnameComp = strcoll(name1->surname,
                            name2->surname);
  int firstComp   = strcoll(name1->first,
                            name2->first);

  return(surnameComp ? surnameComp : firstComp);
ü 4

main() ä 1

  int i, rc, numEntries;
  struct NameList curName;
  struct NameList nameListÝMAX_NAMES¨; 2 3

  printf("Bitte geben Sie die Namen ein, "
         "im Format <Familienname> <Vorname> "
         "(Maximum %d Namen!)Ön", 8 5
         MAX_NAMES);
  for (i=0; i<MAX_NAMES; ++i) ä 1
    printf("Name (oder EOF wenn fertig):Ön"); 5
    rc = scanf(SCAN_FORMAT(MAX_NAME_LEN),
               curName.surname, curName.first);
    if (rc Ü= 2) ä 6 1
      break;
    ü 4
    nameListÝi¨ = curName; 2 3
  ü 4
```

Figure 85. Hybrid Coded Character Set Example (Part 1 of 2)

```
  numEntries = i;
  qsort(nameList, numEntries, sizeof(struct NameList),
        compareNames);
  for (i=0; i<numEntries; ++i) ä 1
    printf("Name %d:<%s, %s>Ön", i+1, 5
           nameListÝi¨.surname, 2 3
           nameListÝi¨.first); 2 3
  ü 4
  i != (MAX_NAMES << sizeof(int)/2); 7
  return(i);
ü 4
```

Figure 85. Hybrid Coded Character Set Example (Part 2 of 2)

The code points in the example above with different glyphs in character code set IBM-273 and APL-293 are highlighted in the previous example and described here:

1	This is the code point for the { character. In coded character set 273 this is the character ä.
2	This is the code point for the [character. In coded character set 273 this is the character Ý.
3	This is the code point for the] character. In coded character set 273 this is the character ¨.
4	This is the code point for the } character. In coded character set 273 this is the character ü.
5	This is the code point for the \ character. In coded character set 273 this is the character Ö.
6	This is the code point for the ! character. In coded character set 273 this is the character Ü.
7	This is the code point for the | character. In coded character set 273 this is the character !. This particular code point mapping is unfortunate because the | and the ! character are both valid C syntax characters. Note that the ! used in the `printf()` call at 8 will appear as ! on a terminal displaying in coded character set 273.

This example illustrates some of the problems with hybrid files. To write this code would require the following steps:

1. The programmer looks up each variant character in coded character set IBM-1047 to find out what the compiler expects. For example, LE/VSE C Run-Time expects the character [to have a byte value of X'AD'.
2. The programmer determines which glyph is at X'AD' in her own coded character set so that she can code that character in her application.
3. The programmer takes care to always use the appropriate substitution. For example, for a needed [in Germany, we look up X'AD' in the German IBM-273 coded character set, and we find the character Ý.

Converting Existing Work

This section describes some issues in conversion and illustrates some conversions. We assume that existing source code and libraries will not be quickly converted from mixed coded character sets to a common coded character set, so we suggest a staged approach:

- Code your new source in one coded character set, preferably IBM-1047. Tag all new source files to make them more portable: put the `??=pragma filetag` directive at the top of each one.

- If you need to interface with existing code, compile your new code using the locale that the existing code was written in.
- If you wish to write code in a coded character set that does not have a one-to-one mapping to coded character set IBM-1047 (that is, a coded character set which is not Latin-1), you can create your own conversion table and compile it with the `genxlt` utility. Such a compiled table can then be used with the `iconv` utility to convert your source code to coded character set IBM-1047.

Converting Hybrid Code

Existing code that was written in a hybrid coded character set will continue to be accepted.

Appendix G, "Converting Code from Coded Character Set IBM-1047," on page 457 shows you a program you can use to convert the hybrid code to another coded character set.

Writing Source Code in Coded Character Set IBM-1047

It is recommended for two reasons that you write source in coded character set IBM-1047.

First, even though LE/VSE C Run-Time provides support for multiple coded character sets, other tools may not do so. Tools such as CICS and SQL/DS may not support source code in any coded character set other than the default coded character set, IBM-1047. If you are using these tools, and you write your code in a code page other than IBM-1047, then you need to use the LE/VSE C Run-Time `iconv` utility to convert your code to coded character set IBM-1047 before you can use the tool.

Second, older versions of the C/370 product do not support source in coded character sets other than IBM-1047. This makes it difficult to share code with a site using an older compiler.

Exporting Source Code to Other Sites

This section deals with *exporting* of code from one Latin-1 coded character set to another. That is, it deals with how to write code that will be run in a locale that uses a different coded character set than the one used to write the source.

The most simple way to export code is to use the `iconv` utility to convert each source file, header file, and data file to the target coded character set, then to send all files to the target location for compilation. You should ensure that your code runs with the same locale that it was compiled under before you try running it with any other locales.

1. Use the `??=pragma filetag` directive to tag each source file, header file, and data file.
2. To write truly portable code, you should use message files for all external strings, such as prompts, help screens, and error messages. Convert these strings to the run-time coded character set in your application code.
3. Use the `setlocale()` function so that the library functions are sensitive to the run-time coded character set.

   Be sure that locale-sensitive information, such as decimal points, is displayed appropriately. Use either `nl_langinfo()` or `localeconv()` to obtain this information.

The `setlocale()` function does not change the CEE functions under Language Environment in such areas as date, time, currency, and time zones. Internationalization is specific to LE/VSE C Run-Time applications. Also, the LE/VSE CEE callable services do not change the LE/VSE C Run-Time locales. For a list of the Language Environment callable services, see *LE/VSE Programming Guide*.

4. Compile with `locale` specifying coded character set IBM-1047.

If you specify `locale("`*`locale-name`*`")`, your code will run correctly with libraries running in the same coded character set. However, if you compile with a different `locale` than you run under, you have to ensure that your code has no internal data, and also that all libraries you use are run-time locale sensitive. Consider the following code fragment:

```
int main() {
  setlocale(LC_ALL, "");
   ⋮
  rc = scanf("%[1234567890abcdefABCDEF]", hexNum);
   ⋮
}
```

For example, if you compile with `locale("De_DE.IBM-273")`, the square brackets will be converted to the hex values X'63' and X'FC'. If the default locale you then run under is not ″De_DE.IBM-273″, but instead ″En_US.IBM-1047″, and you have not used `setlocale()`, then the square brackets will be interpreted as Ä and Ü, and the call to `scanf()` will not do what you intended.

If you only need to run your code locally or export your code to a site that has your locale environment, you can solve this problem by coding:

```
int main() {
  setlocale(LC_ALL, __LOCALE__);
   ⋮
  rc = scanf("%[1234567890abcdefABCDEF]", hexNum);
   ⋮
}
```

This ensures that your code runs with the same locale it was compiled under. Library functions such as `printf()`, `scanf()`, `strfmon()`, and `regcomp()` are sensitive to the current coded character set. The `__LOCALE__` macro is described in "Using Predefined Macros" on page 410.

If you are generating code to export to a site that may not have your locale environment, you should write your code in IBM-1047.

Coded Character Set Independence in Developing Applications

To work effectively with the locale functionality, you may need to use functions, macros, and tools. Here is a summary of the compile-edit work flow, showing what functions you can use where.

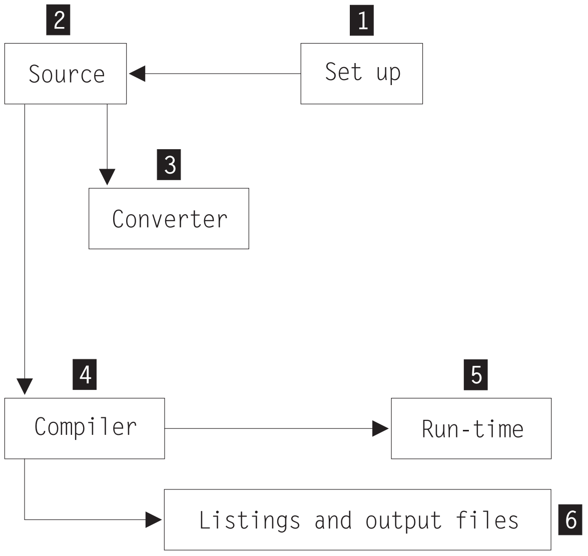


Figure 86. Compile-Edit, Related to Locale Function

The illustration in Figure 86 refers to the following functions:

1 **Setup**.

The `localedef` information (see overview in Chapter 31, "Customizing a Locale," on page 359 and details in "Locale Source Files" on page 329)

2 **Coded character set of source, header files, and data**.

The coded character set used to create a source file must be understood by the compiler so that it will recognize the variant C syntax characters correctly.

- The `??=pragma filetag` directive identifies the coded character set of the source file as well as in library or user's *include* files (see overview in "The `??=pragma filetag` Directive" on page 407)
- Predefined macros __LOCALE__, __FILETAG__, and __CODESET__ (see overview in "Using Predefined Macros" on page 410)
- The function `setlocale()` (see *LE/VSE C Run-Time Library Reference*)

3 **Coded character set conversion utilities & functions**.

The coded character set of a file, or a stream of data, can be converted to another coded character set using the utilities `genxlt` and `iconv` (see "Code Set Conversion Utilities" on page 371 for details) as well as the functions in the run-time library.

4 **Coded character set conversion at compile time**.

This is determined by the compile-time locale, and supported by the compile-time options, `locale` and `nolocale` (see overview in "Converting Coded Character Sets at Compile Time" on page 408; details in *IBM C for VSE/ESA User's Guide*),

5 **Run-time environment**.

During run time, the `setlocale()` function has an effect on such run-time functions, such as `printf()`, `scanf()`, and `regcomp()`, which use variant characters.

6 **Listings and output files**.

Listings, pre-processed source code, and text decks (see overview in "Working With Listings and Output Files" on page 412) may be affected by the coded character set used to create or to convert source files. Your application can, however, include logic, using the following functions, to minimize the impact:
- Use of __LOCALE__, __FILETAG__, and __CODESET__
- Use of locale functions such as `setlocale()`

Coded Character Set of Source Code and Header Files

There are three locale-related activities that can take place from source code:
1. You may tag your source code, and other associated files, with the `??=pragma filetag` directive to specify the coded character set that was used while entering the file. Having done so, you can run compiles being sure that all variant characters in your file are respected.
2. You may use the three macros: __LOCALE__, __FILETAG__, and __CODESET__. These C/VSE compiler macros expand to provide information about the `??=pragma filetag` directive of the current source, and the locale and target coded character set used by the compiler at compile time. Refer to the description of predefined macros in your LE/VSE-conforming C compiler's *Language Reference* for more information.
3. You may use the `setlocale()` function to set the run-time locale to be the same as the locale used to compile the application. This can be used when your application contains dependencies on the coded character set, as it would when comparing constants with external data. Using the macros forces the run-time locale to be the same as the one used to compile your code.

The `??=pragma filetag` Directive

By using `??=pragma filetag` directive, you may write your programs in any convenient supported coded character set (see Appendix D, "Locales Supplied with LE/VSE C Run-Time," on page 431 for a list of coded character set names). The `??=pragma filetag` directive instructs the C/VSE compiler how to "read" the source. As long as you *tag* the source files, the header files, and all data files (including messages) with the `??=pragma filetag` directive, you will be keeping the information about the coded character set used to create each source file in the source file itself. This information can be helpful when moving source files to systems with different coded character sets. Here is the syntax.

??=pragma filetag

```
►►──??=pragma filetag──(─"──code-page-name──"─)──────────────────────────────►◄
```

Here is an example tag that uses the German coded character set IBM-273:

```
??=pragma filetag("IBM-273")
```

Because the # character is variant in different coded character sets, you must use the trigraph `??=` instead for the `pragma filetag` directive.

The ??=pragma filetag directive specifies the coded character set in which the source or data was entered. The coded character set specified in the ??=pragma filetag directive is in effect for the entire source file, but not for any other source file. This applies also to header files and data files.

This directive may appear at most once per file. It must appear before the first statement in a program. If it is encountered anywhere else, a warning is issued and the directive is ignored. Comments which contain variant characters, and which appear before the directive, will not be translated.

Attention: If you use the iconv utility on a file tagged with the ??=pragma filetag directive, you must update the file manually to change the filetag to the correct converted coded character set. The iconv utility does not update the pragma in source files.

Converting Coded Character Sets at Compile Time

The compiler option locale enables the user to tell the compiler what locale to use at compile time; specifically, in what coded character set to generate output. The output affected consists of:

- Pre-processed source code
- Listings
- Object deck

The syntax is:

Compiler Option local

```
►►──┬─locale─(─"──locale-name──"─)─┬──────────────────────────────────────►◄
    └─nolocale─────────────────────┘
```

Further detail on this option is available in *IBM C for VSE/ESA User's Guide*

Examples

Run the compiler using the following compiler option on the PARM parameter of the EXEC statement:

```
LOCALE("De_DE.IBM–273")
```

The compiler recognizes "De_DE.IBM-273" as a valid locale and automatically converts the source code to coded character set IBM-273, for its own use. The compiler would then generate listings in the German coded character set 273.

Here are the input files that are affected:
- The primary source file
- Library header files
- User header files

To generate a pre-processed file that can be sent to other sites, at which different coded character sets are used, use the compiler options:

```
LOCALE("De_DE.IBM–273") PPONLY
```

The compiler will insert the ??=pragma filetag directive at the start of the pre-processed file, using the coded character set specified in the locale option. In this example, ??=pragma filetag("IBM-273") is inserted.

Since the pre-processed file has been tagged, it can be compiled using the C/VSE compiler at any site, regardless of the locale used.

Usage

If no `??=pragma filetag` directive was specified for the source file, and the locale compile-time option is used, then no conversion is performed. The compiler assumes that the file is in the correct target coded character set already.

The *locale-name* is a string that represents the locale the user wants to compile source with; this will determine the characteristics of output, including the coded character set used for variant characters in the source. Usually, a *locale-name* consists of two components, the *territory name* and the *coded character set*. For example, the German locale for coded character set 273 is `De_DE.IBM-273`. The *territory name* is `De_DE` and the *coded character set* is `IBM-273`. To determine the coded character set of a given locale, use the function `nl_langinfo(CODESET)`.

The special *locale-name* `""` gives you the default locale, which can be set using environment variables. The locale name `"C"` specifies the C default locale. Full detail about the C locale is found in Chapter 32, "Definition of S370 C, SAA C, and POSIX C Locales," on page 363.

The default option setting is `nolocale`. It instructs the compiler to do no conversion of text for input or for output. With `nolocale`, no conversion is performed on source files being read. A warning message is issued if a `??=pragma filetag` directive is encountered.

You can create your own locales by using the `localedef` utility. See "Locale Source Files" on page 329 for details.

Summary of Source and Compile Use

The following list shows the results from different combinations of the `??=pragma filetag` directive and the `locale` compiler option.

`locale` option specified:
In this case, the compiler does the following:

- Converts the source code from the coded character set specified with the `??=pragma filetag` directive to the code set specified by the `locale` option.
- If no `??=pragma filetag` directive is specified, the compiler assumes the source is in the same coded character set as specified by the `locale`, and does not perform any conversion.
- Converts compiler error messages from coded character set IBM-1047 to the coded character set specified in `locale`.
- Generates compiler output in the same coded character set as that of the locale specified in the `locale` option.
- Inserts the `??=pragma filetag` directive, using the coded character set specified in the `locale` option, at the start of the preprocessor file, if `PPONLY` is specified.

`nolocale` specified:
In this case, the compiler does the following:

- Does not convert text in the input or output file, and uses the default coded character set IBM-1047 to interpret syntactic characters.

- If a `??=pragma filetag` directive is specified, the compiler suppresses the `??=pragma filetag` directive in the preprocessor file. The compiler issues warnings if the `??=pragma filetag` directive specifies a coded character set other than IBM-1047, and uses IBM-1047 anyway.

Using Predefined Macros

There are three macros for the C/VSE compiler that relate to locale.

__LOCALE__

This macro expands to a string literal representing the locale of the `locale` compiler option. This macro can be used to set the run-time locale to be the same as the compiled locale:

```
main() {
  setlocale(LC_ALL, __LOCALE__);

   ⋮
}
```

The value of this macro is defined per compilation. If no `locale` compiler option was supplied, the macro is undefined.

__FILETAG__

This macro expands to a string literal representing the character coded character set of the `??=pragma filetag` directive associated with the current file. For example, to convert to the coded character set specified by the `locale` option from the coded character set specified by the `??=pragma filetag` directive, you would use the `iconv_open()` function:

```
iconv_open(__FILETAG__,variable);
```

The value of this macro is defined per source file. If no `??=pragma filetag` directive is present, the macro is undefined.

__CODESET__

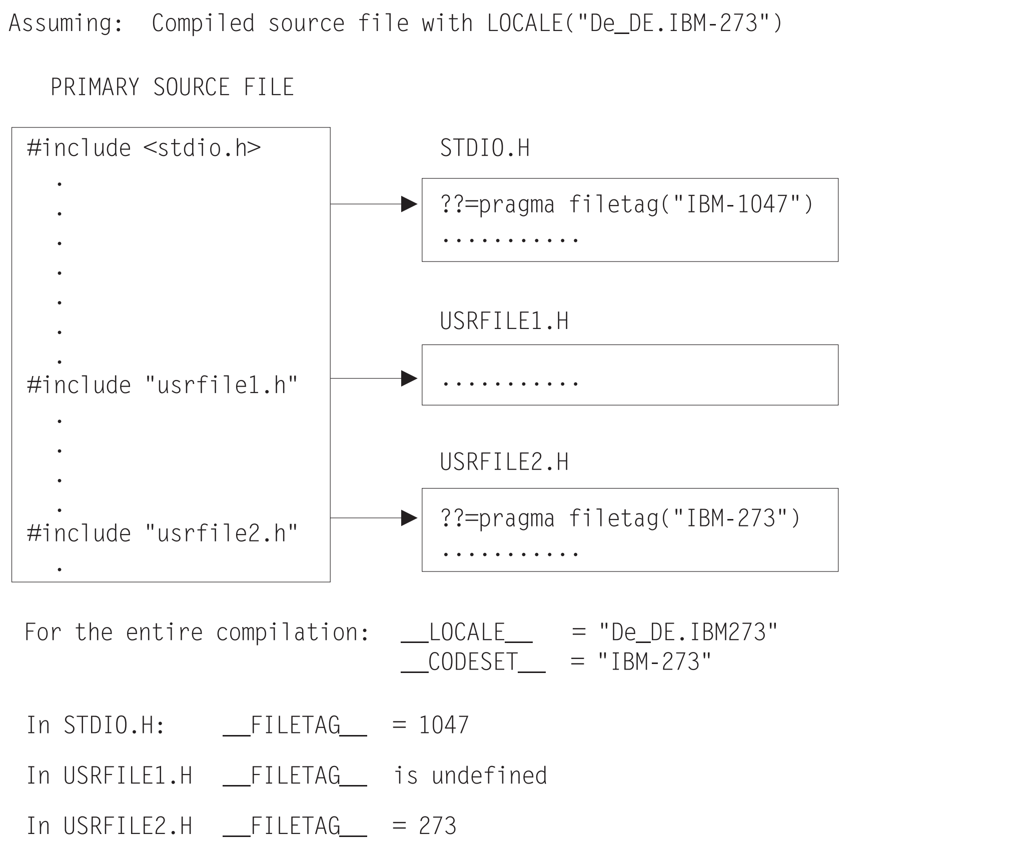
This macro expands to a string literal representing the character coded character set of the `locale` compiler option. If a value was not supplied at compilation, the macro is undefined.

EDCXGCC2

```
 /* EDCXGCC2
    This example shows how to use the __CODESET__ macro
  */

#include <iconv.h>
#include <string.h>
#include <stdio.h>

 /* The following function could be in a header file  */
#ifdef __CODESET__
  static int convstr(iconv_t convInfo, char *in, int inSize,
                     char *out, int outSize) {
     return(iconv(convInfo, in, inSize, out, outSize))
  }
#else
  static int convstr(iconv_t convInfo, char *in, int inSize,
                     char *out, int outSize) {
     memcpy(out, in, outSize > inSize ? inSize : outSize);
     return(outSize > inSize ? -1 : 0);
  }
#endif

iconv_t convInfo;

int main() {
#ifdef __CODESET__
  char *runtimeCodeSet;
  setlocale(LC_ALL, ""); /* set locale to default locale */
  runtimeCodeSet = nl_langinfo(CODESET);
  convInfo = iconv_open(runtimeCodeSet, __CODESET__);
#endif
  char intro[] = "Welcome to my variant world!\n";
  char nlIntro[sizeof(intro)];
  convstr(convInfo, intro, sizeof(intro),
          nlIntro, sizeof(nlIntro));
  puts(nlIntro);  /* string will print appropriately */
#ifdef __CODESET__
  iconv_close(convInfo);
#endif

return(0);
}
```

Figure 87. Example of __CODESET__ Macro

The illustration below shows the values that these macros will take on, emphasizing that for __FILETAG__, a value is assigned for each source file, but for __LOCALE__ and __CODESET__, a value is assigned for a compilation.



```
Assuming:  Compiled source file with LOCALE("De_DE.IBM-273")

   PRIMARY SOURCE FILE

#include <stdio.h>                 STDIO.H
  .
  .                          ??=pragma filetag("IBM-1047")
  .                          ...........
  .
  .                                USRFILE1.H
  .
  .
#include "usrfile1.h"        ...........
  .
  .                                USRFILE2.H
  .
  .
#include "usrfile2.h"        ??=pragma filetag("IBM-273")
  .                          ...........

For the entire compilation:  __LOCALE__   = "De_DE.IBM273"
                             __CODESET__  = "IBM-273"

In STDIO.H:    __FILETAG__  = 1047

In USRFILE1.H  __FILETAG__  is undefined

In USRFILE2.H  __FILETAG__  = 273
```

Figure 88. Values of Macros __FILETAG__, __LOCALE__, and __CODESET__

Using a Predefined Locale

You can change the run-time locale to any one of the other predefined locales listed in Table 52 on page 431. To use a defined locale, refer to it by its `setlocale()` parameter.

To define a new locale, copy the source file provided, edit it, and assemble it (see Chapter 31, "Customizing a Locale," on page 359).

Working With Listings and Output Files

The compiler respects the locale specified by the `locale` compile option in generating the listing. If the `nolocale` compiler option is in effect, no locale information is used and no conversion is performed on any of the output files.

The *output* files affected:
- Object Decks
- Pre-processed source code
- Listings

Object Decks

If the `locale` option is specified, the object deck is generated in the coded character set of your current locale. Otherwise, the object deck is generated in the coded character set IBM-1047.

Code will run correctly if the run-time locale is the same as the locale of the object deck.

If the object was generated with a different locale from the one you run under, you must ensure that your code can run under different locales. Refer to Chapter 31, "Customizing a Locale," on page 359 for more information.

For information about exporting code to other sites, see "Exporting Source Code to Other Sites" on page 404.

Listings

You can use the compiler option `locale` to ensure that listings are sensitive to a specified locale. For example, here is the result from compiling the source file `HELLO` with the compiler options:

```
LIST SOURCE LOCALE("De_DE.IBM-273")
```

```
15686A01 V1 R1 M00 IBM C/VSE                                       DD:SYSIPT                 1  12.04.01 10:19:41   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
                                     : *NOXREF     *NOAGGR     *NOPPONLY   *NOEXPMAC   *NOSHOWINC  *NOOFFSET   *NOMEMORY   *NOSSCOMM
                                     : *NOCSECT    *NOLONGNAME *START      *EXECOPS    *NOEVENTS   *NOINFILE
                                     : *TARGET()   *FLAG(I)    *NOTEST(SYM,BLOCK,LINE,NOPATH)      *OPTIMIZE(0)*SPILL(128)
                                     : *NOINLINE(AUTO,NOREPORT,250,1000)   *NESTINC(16)
                                     : *NOCHECKOUT(NOPPTRACE,PPCHECK,GOTO,ACCURACY,PARM,NOENUM,
                                     : *           NOEXTERNAL,TRUNC,INIT,NOPORT,GENERAL)
                                     : *NOSEARCH
                                     : *NOLSEARCH
                                     : *OBJECT     *NOHWOPTS   *LOCALE
     LANGUAGE LEVEL. . . . . . . . . : *EXTENDED
     SOURCE MARGINS. . . . . . . . . :
       VARYING LENGTH. . . . . . . . : 1 - 32767
       FIXED LENGTH  . . . . . . . . : 1 - 72
     SEQUENCE COLUMNS. . . . . . . . :
       VARYING LENGTH. . . . . . . . : NONE
       FIXED LENGTH. . . . . . . . . : 73 - 80
     LOCALE NAME . . . . . . . . . . : DE_DE.IBM-273  2
     CODE SET. . . . . . . . . . . . : IBM-273
15686A01 V1 R1 M00 IBM C/VSE                                       DD:SYSIPT                    12.04.01 10:19:41   PAGE     2

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

  LINE  STMT                                                                                                          SEQNBR INCNO
             *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.......*
     1      |??=pragma filetag("IBM-1047")                                                                  |       1
     2      |int main(void) ä                                                                               |       2
     3     1|  printf("Hello worldÖn");                                                                     |       3
     4      |ü                                                                                              |       4
                                    * * * * *   E N D   O F   S O U R C E   * * * * *
   ⋮
                              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         7  25,93  3        LA        4  14,81           BALR      3  11,11           BCR       2   7,41
             BC        2   7,41           ST        2   7,41           LR        2   7,41           STM       2   7,41
             CL        1   3,70           MVI       1   3,70           LM        1   3,70
```

Figure 89. Example of Output When Locale Option Used

In the listing above, notice the locale-specific information:

1 The date at the top right.

2 The name of the locale and the Code Set.

3 The currency decimal-separator.

Considerations With Other Products and Tools

Note

Any software tool that scans source code or compiler listings is affected by the introduction of the locale functionality. Tools that read or generate source code now need to recognize the `??=pragma filetag` directive. Tools that read listings need to recognize the coded character set in the title header.

Since the following tools scan source code, they may be affected.

- Debug Tool for VSE/ESA does not support code written in any coded character set other than IBM-1047.
- Translators such as CICS and SQL/DS read source files and generate new source files. If they do not recognize the `??=pragma filetag` directive, then follow these steps:
  1. Convert the source file to coded character set IBM-1047 using the `iconv` utility.
  2. Remove the `??=pragma filetag` directive from the source file, or change it to `??=pragma filetag("IBM-1047")`. Run the source that is in the IBM-1047 coded character set through the appropriate translator, if needed.

Appendix A. POSIX Character Set

POSIX 1003.2, section 2.4, specifies the characters that are in the portable character set. The following table lists the characters in the portable character set with their symbolic name, the GCGID, and the graphic symbol for the character. Some of the characters (the hyphen, for example) also have alternate symbolic names.

The input files for the `localedef` utility, the charmap file and the locale definition file, are coded using the characters in the portable character set.

Symbolic Name	Alternate Name	Character
<NUL>		
<alert>	<SE08>	
<backspace>	<SE09>	
<tab>	<SE10>	
<newline>	<SE11>	
<vertical-tab>	<SE12>	
<form-feed>	<SE13>	
<carriage-return>	<SE14>	
<space>	<SP01>	
<exclamation-mark>	<SP02>	!
<quotation-mark>	<SP04>	"
<number-sign>	<SM01>	#
<dollar-sign>	<SC03>	$
<percent-sign>	<SM02>	%
<ampersand>	<SM03>	&
<apostrophe>	<SP05>	'
<left-parenthesis>	<SP06>	(
<right-parenthesis>	<SP07>	)
<asterisk>	<SM04>	*
<plus-sign>	<SA01>	+
<comma>	<SP08>	,
<hyphen>	<SP10>	-
<hyphen-minus>	<SP10>	-
<period>	<SP11>	.
<slash>	<SP12>	/
<zero>	<ND10>	0
<one>	<ND01>	1
<two>	<ND02>	2
<three>	<ND03>	3
<four>	<ND04>	4
<five>	<ND05>	5
<six>	<ND06>	6
<seven>	<ND07>	7
<eight>	<ND08>	8
<nine>	<ND09>	9
<colon>	<SP13>	:
<semicolon>	<SP14>	;
<less-than-sign>	<SA03>	<
<equals-sign>	<SA04>	=
<greater-than-sign>	<SA05>	>
<question-mark>	<SP15>	?
<commercial-at>	<SM05>	@

Symbolic Name	Alternate Name	Character
<A>	<LA02>	A
<B>	<LB02>	B
<C>	<LC02>	C
<D>	<LD02>	D
<E>	<LE02>	E
<F>	<LF02>	F
<G>	<LG02>	G
<H>	<LH02>	H
<I>	<LI02>	I
<J>	<LJ02>	J
<K>	<LK02>	K
<L>	<LL02>	L
<M>	<SM02>	M
<N>	<LN02>	N
<O>	<LO02>	O
<P>	<LP02>	P
<Q>	<LQ02>	Q
<R>	<LR02>	R
<S>	<LS02>	S
<T>	<LT02>	T
<U>	<LU02>	U
<V>	<LV02>	V
<W>	<LW02>	W
<X>	<LX02>	X
<Y>	<LY02>	Y
<Z>	<LZ02>	Z
<left-square-bracket>	<SM06>	[
<backslash>	<SM07>	\
<reverse-solidus>	<SM07>	\
<right-square-bracket>	<SM08>	]
<circumflex>	<SD15>	^
<circumflex-accent>	<SD15>	^
<underscore>	<SP09>	_
<low-line>	<SP09>	_
<grave-accent>	<SD13>	`
<a>	<LA01>	a
<b>	<LB01>	b
<c>	<LC01>	c
<d>	<LD01>	d
<e>	<LE01>	e
<f>	<LF01>	f
<g>	<LG01>	g
<h>	<LH01>	h
<i>	<LI01>	i
<j>	<LJ01>	j
<k>	<LK01>	k
<l>	<LL01>	l
<m>	<LM01>	m
<n>	<LN01>	n
<o>	<LO01>	o
<p>	<LP01>	p
<q>	<LQ01>	q
<r>	<LR01>	r
<s>	<LS01>	s

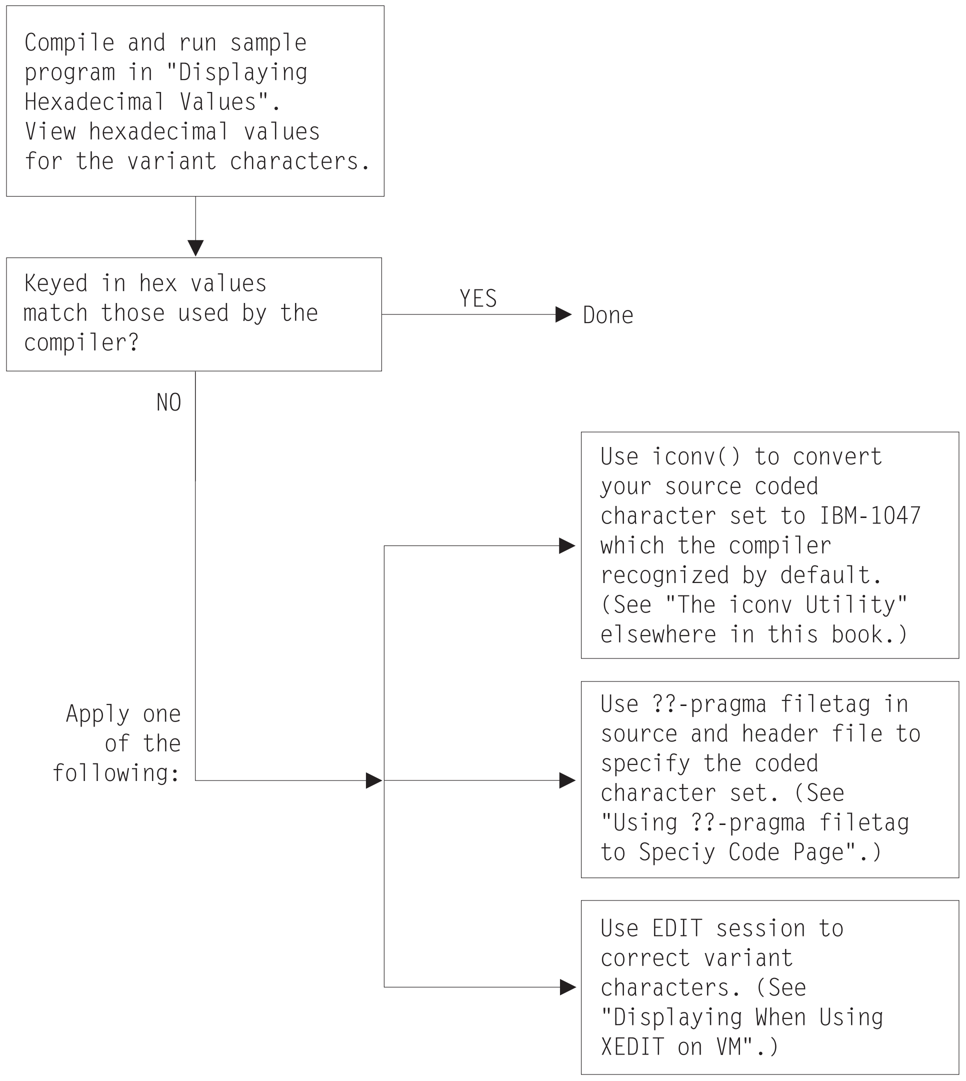
Symbolic Name	Alternate Name	Character
<t>	<LT01>	`t`
<u>	<LU01>	`u`
<v>	<LU01>	`v`
<w>	<LW01>	`w`
<x>	<LX01>	`x`
<y>	<LY01>	`y`
<z>	<LZ01>	`z`
<left-brace>	<SM11>	`{`
<left-curly-bracket>	<SM11>	`{`
<vertical-line>	<SM13>	`|`
<right-brace>	<SM14>	`}`
<right-curly-bracket>	<SM14>	`}`
<tilde>	<SD19>	`~`

With LE/VSE C Run-Time, the `localedef` utility uses code page IBM-1047 as the definition of the code points for the characters in the *Portable Character Set*. Therefore the default values for the `escape-char` and `comment-char` are the code points from the IBM-1047 code page.

There are some coded character sets, such as the Japanese Katakana coded character set 290, that have code points for the lowercase characters different from the code points for the lowercase characters in the set IBM-1047. A charmap file or locale definition file cannot be coded using these coded character sets.

Appendix B. Mapping Variant Characters for C/VSE

This appendix describes how you can enter and display the variant characters such as the square brackets [and], and the caret character, ^, for the host environment. If you use a programmable workstation or a 3270 terminal, you can follow the documented procedures to map the keys on your keyboard to send the correct variant character hexadecimal values to the host system for the C/VSE compiler. In the following sections, square brackets are used to illustrate the suggested procedures.



Compile and run sample program in "Displaying Hexadecimal Values". View hexadecimal values for the variant characters.
Keyed in hex values match those used by the compiler?
YES
Done
NO
Apply one of the following:
Use iconv() to convert your source coded character set to IBM-1047 which the compiler recognized by default. (See "The iconv Utility" elsewhere in this book.)
Use ??-pragma filetag in source and header file to specify the coded character set. (See "Using ??-pragma filetag to Speciy Code Page".)
Use EDIT session to correct variant characters. (See "Displaying When Using XEDIT on VM".)

Note: If you are running a programmable workstation using host emulation software, apply your host emulation software's keyboard remapping first. If this allows correct hexadecimal values for the variant characters to be sent to the host, then you have completed the task.

Displaying Hexadecimal Values

If you are not sure whether your current keys generate correct hexadecimal values for the C/VSE compiler and LE/VSE C Run-Time, you can use the following program to show their hexadecimal values on the display. This program displays the hexadecimal values for the variant characters that your current setup uses and the values that the compiler and library expect.

Note: See the appropriate section of *IBM C for VSE/ESA User's Guide* for information on the `LOCALE/NOLOCALE` option and the list of IBM-supported locales that can be used at compile or run time. The default C locale is encoded in code page IBM-1047; therefore the default encoding of variant characters is as in IBM-1047.

Example

The sample program reads the ten characters from the input file `MYFILE.DAT` and displays the character values in hexadecimal notation. The program also queries the current compile time locale for the character values that compiler would expect. These ten variant characters are selected because they are syntactically important to the C compiler. You must type them in `MYFILE.DAT` in this order on a single line, without spaces between them:

- backslash \
- right square bracket]
- left square bracket [
- right brace }
- left brace {
- circumflex ^
- tilde ~
- exclamation mark !
- number sign #
- vertical line |

You can use the sample program to display the character values and then reset your environment to generate the codes as shown in the column `EXPECTED BY COMPILER`. After re-editing your input file, you can run this program again. Consult your system programmer for the coded character set that your installation uses.

EDCXGMV1

```
 /* EDCXGMV1
    This example will display hexadecimal values for the variant
    characters
  */

#include <stdio.h>
#include <locale.h>
#include <variant.h>
#include <stdlib.h>

void read_user_data(char *, int);
void main() {
  char  *user_char, *compiler_char;

  struct variant *compiler_var_char;
  int num_var_char, index;
  char *code_set;
  char *char_names[]={"backslash",
                      "right bracket",
                      "left bracket",
                      "right brace",
                      "left_brace",
                      "circumflex",
                      "tilde",
                      "exclamation mark",
                      "number sign",
                      "vertical line"};

  num_var_char=sizeof(char_names)/sizeof(char *);
  if ((user_char=(char*)calloc(num_var_char, 1)) == NULL)
  {
    printf("Error: Unable to allocate the storage\n");
    exit(99);
  }

  read_user_data(user_char, num_var_char);
  /* managed to read the users' characters from the file */

  code_set="default IBM-1047";
  compiler_char="\xe0\xbd\xad\xd0\xc0\x5f\xa1\x5a\x7b\x4f";
                                   /* standard compiler code page */

  printf("Compiler and library code page is : %s\n\n", code_set);
  printf("                           Variant character values:\n");
  printf(" %16s     expected by compiler    your current\n", "");
  for (index=0; index<num_var_char; index++)
    printf(" %16s  :      %X                      %X\n",
             char_names[index], compiler_char[index], user_char[index]);
  exit(0);
}
```

Figure 90. Example of Displaying Hexadecimal Values (Part 1 of 2)

```
void read_user_data(char* char_array, int num_var_char)
{
  FILE  *stream;
  int num;

  if (stream = fopen ("myfile.dat", "rb"))
    if(!(num = fread(char_array, 1, num_var_char, stream)))
    {
      printf("Error: Unable to read from the file\n");
      exit(99);
    }
    else { ;}
  else
  {
    printf("Error: Unable to open the file\n");
    exit(99);
  }
  fclose(stream);
  return;
}
```

Figure 90. Example of Displaying Hexadecimal Values (Part 2 of 2)

Note: After executing this program you can use some of the procedures described above to make sure that your special characters on the keyboard generate the hexadecimal values expected by the LE/VSE C Run-Time.

Using `??=pragma filetag` To Specify Code Page

Add the following `??=pragma filetag` in the source and header file to specify the code page that a C program uses:

```
??=ifdef __COMPILER_VER__
  ??=pragma filetag ("codepage")
??=endif
```

codepage is the codepage that the source code is written in.

Note: If you are running standard 3270 emulation in the U.S., your workstation software most likely uses code page 37. You can then use this alternative by specifying IBM-037 as *codepage*.

Displaying When Using XEDIT on VM

If you know that the hexadecimal values of the square brackets that you key in are not those accepted by the C/VSE compiler as square brackets, then you can add `SET INPUT` commands in your XEDIT profile to convert the hexadecimal values of the keyed-in square brackets to ones that the compiler recognizes. This conversion will then happen during your XEDIT session. If you know that the square brackets that you key in are written correctly into the source file but are not displayed as square brackets, you can add `SET OUTPUT` commands to display the keyed-in square brackets correctly during the XEDIT session. The following are three samples of XEDIT profile changes you can make to make sure that XEDIT writes out hexadecimal values recognizable by the compiler for the square brackets, and displays the [and] characters correctly during the XEDIT session.

Example One (For Keyboard with [and] Keys)

The following can be added to your XEDIT PROFILE if:
1. Your keyboard has [and] keys and it generates code page 37 characters
2. The [key generates X'BA'
3. The] key generates X'BB'
4. The C/VSE compiler recognizes X'AD' as the [(code page 1047)
5. The C/VSE compiler recognizes X'BD' as the] (code page 1047)
6. XEDIT displays X'BA' as the [
7. XEDIT displays X'BB' as the]

Note: You can use the program "Displaying Hexadecimal Values" on page 422 to find out the hexadecimal values that the special keys on your keyboard generate.

```
/*-----------------------------------------------------------------*/
/* Display: read x'AD' and x'BD' in the source file and display    */
/*          them as square brackets                                */
/*-----------------------------------------------------------------*/
/*                                                                 */
/* XEDIT displays x'BA' which is a left square bracket when it     */
/* encounters x'AD' in the source file                             */
   Address XEDIT 'SET OUTPUT AD' 'BA'x   /* Left square bracket */
/*                                                                 */
/* XEDIT displays x'BB' which is a right square bracket when it    */
/* encounters x'BD' in the source file                             */
   Address XEDIT 'SET OUTPUT BD' 'BB'x   /* Right square bracket*/
/*-----------------------------------------------------------------*/
/* Write: gets x'BA' and x'BB' from keyboard and writes them in the */
/*        file as the hexadecimal values that the C                */
/*        compiler recognizes as the left and right square         */
/*        brackets                                                 */
/*-----------------------------------------------------------------*/
/*                                                                 */
/* XEDIT writes out x'AD' when it gets x'BA' from the keyboard     */
   Address XEDIT 'SET INPUT BA AD'       /* Left square bracket */
   Address XEDIT 'SET INPUT BB BD'       /* Right square bracket*/
/*                                                                 */
/*-----------------------------------------------------------------*/
/* Turn off text and APL characters for this XEDIT session         */
/*-----------------------------------------------------------------*/
/*                                                                 */
   Address XEDIT 'SET TEXT OFF'
   Address XEDIT 'SET APL OFF'
/*                                                                 */
/*-----------------------------------------------------------------*/
/*Turn off text and APL characters at CP level for this logon session*/
/*-----------------------------------------------------------------*/
/*                                                                 */
   Address COMMAND 'CP TERMINAL TEXT OFF'
   Address COMMAND 'CP TERMINAL APL OFF'
```

Example Two (For Keyboard with No [and] Keys)

The following can be added to your XEDIT PROFILE if:
1. Your keyboard has no [and] keys
2. You have not remapped @ and $ [3] by using your host emulation software
3. You want the @ and $ keys to generate square brackets X′AD′ and X′BD′ during XEDIT
4. The C/VSE compiler recognizes X′AD′ as the [(code page 1047)
5. The C/VSE compiler recognizes X′BD′ as the] (code page 1047)
6. XEDIT displays X′BA′ as the [
7. XEDIT displays X′BB′ as the]

```
/*------------------------------------------------------------------*/
/* Display: read x'AD' and x'BD' in the source file and display      */
/*          them as square brackets                                  */
/*------------------------------------------------------------------*/
/*                                                                   */
/* XEDIT displays x'BA' which is a left square bracket when it       */
/* encounters x'AD' in the source file                               */
   Address XEDIT 'SET OUTPUT AD' 'BA'x   /* Left square bracket */
/*                                                                   */
/* XEDIT displays x'BB' which is a right square bracket when it      */
/* encounters x'BD' in the source file                               */
   Address XEDIT 'SET OUTPUT BD' 'BB'x   /* Right square bracket*/
/*------------------------------------------------------------------*/
/* Write: gets x'AD' and x'BD' from keyboard and writes them in the  */
/*        file as correct hexadecimal values that the C              */
/*        compiler recognizes as the left and right square           */
/*        brackets                                                   */
/*------------------------------------------------------------------*/
/*                                                                   */
/* XEDIT writes out x'AD' when it gets x'7C' from the keyboard       */
   Address XEDIT 'SET INPUT 7C AD'       /* Left square bracket */
/* XEDIT writes out x'BD' when it gets x'5B' from the keyboard       */
   Address XEDIT 'SET INPUT 5B BD'       /* Right square bracket*/
/*                                                                   */
/*------------------------------------------------------------------*/
/* Turn off text and APL characters for this XEDIT session           */
/*------------------------------------------------------------------*/
/*                                                                   */
   Address XEDIT 'SET TEXT OFF'
   Address XEDIT 'SET APL OFF'
/*                                                                   */
/*------------------------------------------------------------------*/
/*Turn off text and APL characters at CP level for this logon session*/
/*------------------------------------------------------------------*/
/*                                                                   */
   Address COMMAND 'CP TERMINAL TEXT OFF'
   Address COMMAND 'CP TERMINAL APL OFF'
```

3. These characters are not used in the C language.

Example Three (For Keyboard with No [and] Keys)

The following can be added to your XEDIT PROFILE if:
1. Your keyboard has no [and] keys
2. You have remapped @ and $, respectively, using your host emulation software to generate X'AD' and X'BD' which are [and]
3. The C/VSE compiler recognizes X'AD' as the [(code page 1047)
4. The C/VSE compiler recognizes X'BD' as the] (code page 1047)
5. XEDIT displays X'BA' as the [
6. XEDIT displays X'BB' as the]

```
/*-----------------------------------------------------------------*/
/* Display: read x'AD' and x'BD' in the source file                */
/*          or gets them from the keyboard and displays            */
/*          them as left and right square brackets                 */
/*-----------------------------------------------------------------*/
/*                                                                 */
/* XEDIT displays x'BA' which is a left square bracket when it     */
/* encounters x'AD' in the source file                             */
   Address XEDIT 'SET OUTPUT AD' 'BA'x   /* Left square bracket */
/*                                                                 */
/* XEDIT displays x'BB' which is a right square bracket when it    */
/* encounters x'BD' in the source file                             */
   Address XEDIT 'SET OUTPUT BD' 'BB'x   /* Right square bracket*/
/*                                                                 */
/*-----------------------------------------------------------------*/
/* Set Input is not necessary                                      */
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
/* Turn off text and APL characters for this XEDIT session         */
/*-----------------------------------------------------------------*/
/*                                                                 */
   Address XEDIT 'SET TEXT OFF'
   Address XEDIT 'SET APL OFF'
/*                                                                 */
/*-----------------------------------------------------------------*/
/*Turn off text and APL characters at CP level for this logon session*/
/*-----------------------------------------------------------------*/
/*                                                                 */
   Address COMMAND 'CP TERMINAL TEXT OFF'
   Address COMMAND 'CP TERMINAL APL OFF'
```

Note: You can use `QUERY INPUT` and `QUERY OUTPUT` to find out the existing settings for your XEDIT session. You can also create your own EXEC with `SET INPUT` and `SET OUTPUT` to clear settings and/or reset them.

Appendix C. LE/VSE C Run-Time Code Point Mappings

The tables below show the code point mappings for Latin-1/Open Systems coded character set 1047 (Figure 91) and for the APL coded character set 293 (Figure 92 on page 430).

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¬ SM660000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	¢ SC040000	! SP020000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	Ý LY120000	(SHY) SP320000	^{1} ND011000	^{2} ND021000	^{3} ND031000
-B	. SP110000	\$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	¨ SD170000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	@ SM050000	ð LD630000	æ LA510000	Đ LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	[SM060000	] SM080000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	| SM130000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 01047

Figure 91. Coded Character Set for Latin-1/Open Systems

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	– SL690000	◇ SL370000	~ SL460000	⎕ SL360000	¯ SL630000	⍺ SL710000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	*A̲* LA480000	*J̲* LJ480000	/ SL760000	∧ SL510000	a LA010000	j LJ010000	~ SD190000	∊ SL720000	A LA020000	J LJ020000	≡ SL300000	1 ND010000
-2	*B̲* LB480000	*K̲* LK480000	*S̲* LS480000	¨ SL450000	b LB010000	k LK010000	s LS010000	⍳ SL730000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	*C̲* LC480000	*L̲* LL480000	*T̲* LT480000	⌼ SL270000	c LC010000	l LL010000	t LT010000	⍴ SL740000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	*D̲* LD480000	*M̲* LM480000	*U̲* LU480000	⍸ SL860000	d LD010000	m LM010000	u LU010000	⍵ SL750000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	*E̲* LE480000	*N̲* LN480000	*V̲* LV480000	⍷ SL870000	e LE010000	n LN010000	v LV010000		E LE020000	N LN020000	V LV020000	5 ND050000
-6	*F̲* LF480000	*O̲* LO480000	*W̲* LW480000	⊢ SL340000	f LF010000	o LO010000	w LW010000	× SL550000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	*G̲* LG480000	*P̲* LP480000	*X̲* LX480000	⊣ SL350000	g LG010000	p LP010000	x LX010000	\ SL640000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	*H̲* LH480000	*Q̲* LQ480000	*Y̲* LY480000	∨ SL500000	h LH010000	q LQ010000	y LY010000	÷ SL540000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	*I̲* LI480000	*R̲* LR480000	*Z̲* LZ480000	` SD130000	i LI010000	r LR010000	z LZ010000		I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	¢ SC040000	! SP020000	¦ SM650000	: SL830000	↑ SL610000	⊃ SL430000	∩ SL400000	∇ SL030000	⍲ SL170000	⌶ SL240000	⌿ SL150000	
-B	. SL840000	$ SC030000	, SL850000	# SM010000	↓ SL620000	⊂ SL420000	∪ SL410000	∆ SL060000	⍱ SL180000	! SL580000	⍀ SL160000	⍫ SL040000
-C	< SL520000	* SL650000	% SM020000	@ SM050000	≤ SL560000		⊥ SL230000	⊤ SL220000	⌷ SL260000	⍒ SL050000	∵ SL320000	⍙ SL330000
-D	(SL670000	) SL680000	_ SL440000	' SL660000	⌈ SL010000	○ SL080000	[SL770000	] SL780000	⌽ SL090000	⍋ SL070000	⊖ SL120000	⍟ SL110000
-E	+ SL790000	; SL800000	> SL530000	= SL810000	⌊ SL020000		≥ SL570000	≠ SL820000	⍂ SL280000	⍞ SL130000	⌹ SL140000	⍎ SL190000
-F	| SM130000	¬ SM660000	? SL700000	" SP040000	→ SL600000	← SL590000	∘ SL250000	| SL380000	⍉ SL100000	⍝ SL210000	⍕ SL200000	(EO)

Code Page 00293

Figure 92. Coded Character Set for APL

Appendix D. Locales Supplied with LE/VSE C Run-Time

The following table lists the compiled locales supported by default with LE/VSE C Run-Time. All of these locale files are provided with the base feature of LE/VSE.

The table lists each `setlocale()` parameter and its corresponding language, country, codeset, and actual phase name. The S370 C, POSIX C and SAA C locales do not have locale modules associated with them. They are built-in locales that cannot be modified, and are always present. Their names cannot be changed. These locales are based on the coded character set IBM-1047. The new versions of the `POSIX C` and SAA C locales can be provided, but to refer to them, you must specify the full name of the requested locale, including the `CodesetRegistry-CodesetEncoding` names. For example,

```
"SAA.IBM-037"
```

refers to the SAA C locale built from the coded character set IBM-037.

Notes:

1. Prior to LE/VSE Version 1 Release 4 Modification Level 4 (V1R4.4), the default currency for the European Economic Community was set to “local currency” in the `LC_MONETARY` category of the base locale. If you wanted to set the Euro as currency, you had to use `setlocale()` to set the `@euro` locales. However, from LE/VSE V1R4.4 onwards the `LC_MONETARY` category in the base locale is now set to use the Euro. If you set the base locale, you now have the Euro as your default currency. If you wish to use your previous (local) currency, you must issue `setlocale()` to set the `@preeuro` locales.
2. Locales using a euro-currency codeset (for example, IBM-114x) use the symbolic name `<euro-sign>` instead of `<currency>`. Locales used when the `@euro` modifier is specified have `int_curr_symbol` and `currency_symbol` in the `LC_MONETARY` category set to euro-currency symbols.

Table 52. Compiled Locales Supplied With LE/VSE C Run-Time

Local name as in `setlocale()` argument	Language	Country or Region	Codeset	Phase name
Bg_BG.IBM-1025	Bulgarian	Bulgaria	IBM-1025	EDC$BGFE
Bg_BG.IBM-1154	Bulgarian	Bulgaria	IBM-1154	EDC$BGHT
Ca_ES.IBM-924	Catalan	Spain	IBM-924	EDC$CSEZ
Ca_ES.IBM-924@euro	Catalan	Spain	IBM-924	EDC@CSEZ
Ca_ES.IBM-924@preeuro [1]	Catalan	Spain	IBM-924	EDC3CSEZ
Cs_CZ.IBM-870	Czech	Czech Republic	IBM-870	EDC$CZEQ
Cs_CZ.IBM-1153	Czech	Czech Republic	IBM-1153	EDC$CZMB
Da_DK.IBM-1142@euro	Danish	Denmark	IBM-1142	EDC@DAHE
Da_DK.IBM-277	Danish	Denmark	IBM-277	EDC$DAEE
Da_DK.IBM-924	Danish	Denmark	IBM-924	EDC$DAEZ
Da_DK.IBM-924@euro	Danish	Denmark	IBM-924	EDC@DAEZ
Da_DK.IBM-1047	Danish	Denmark	IBM-1047	EDC$DAEY
Da_DK.IBM-1142	Danish	Denmark	IBM-1142	EDC$DAHE
Da_DK.IBM-1142@euro	Danish	Denmark	IBM-1142	EDC@DAHE

Table 52. Compiled Locales Supplied With LE/VSE C Run-Time (continued)

Local name as in `setlocale()` argument	Language	Country or Region	Codeset	Phase name
De_AT.IBM-924	German	Austria	IBM-924	EDC$DTEZ
De_AT.IBM-924@euro	German	Austria	IBM-924	EDC@DTEZ
De_AT.IBM-924@preeuro [1]	German	Austria	IBM-924	EDC3DTEZ
De_CH.IBM-500	German	Switzerland	IBM-500	EDC$DCEO
De_CH.IBM-1047	German	Switzerland	IBM-1047	EDC$DCEY
De_CH.IBM-1148	German	Switzerland	IBM-1148	EDC$DCHO
De_CH.IBM-1148@euro	German	Switzerland	IBM-1148	EDC@DCHO
De_DE.IBM-273 [1]	German	Germany	IBM-273	EDC$DDEB
De_DE.IBM-924	German	Germany	IBM-924	EDC$DDEZ
De_DE.IBM-924@euro	German	Germany	IBM-924	EDC@DDEZ
De_DE.IBM-924@preeuro [1]	German	Germany	IBM-924	EDC3DDEZ
De_DE.IBM-1047 [1]	German	Germany	IBM-1047	EDC$DDEY
De_DE.IBM-1141	German	Germany	IBM-1141	EDC$DDHB
De_DE.IBM-1141@euro	German	Germany	IBM-1141	EDC@DDHB
De_DE.IBM-1141@preeuro [1]	German	Germany	IBM-1141	EDC3DDHB
De_LU.IBM-924	German	Luxembourg	IBM-924	EDC$DLEZ
De_LU.IBM-924@euro	German	Luxembourg	IBM-924	EDC@DLEZ
De_LU.IBM-924@preeuro [1]	German	Luxembourg	IBM-924	EDC3DLEZ
El_GR.IBM-875 [1]	Greek	Greece	IBM-875	EDC$ELES
El_GR.IBM-4971	Greek	Greece	IBM-4971	EDC$ELHS
El_GR.IBM-4971@euro	Greek	Greece	IBM-4971	EDC@ELHS
El_GR.IBM-4971@preeuro [1]	Greek	Greece	IBM-4971	EDC3ELHS
En_BE.IBM-924	English	Belgium	IBM-924	EDC$EBEZ
En_BE.IBM-924@euro	English	Belgium	IBM-924	EDC@EBEZ
En_BE.IBM-924@preeuro [1]	English	Belgium	IBM-924	EDC3EBEZ
En_GB.IBM-285	English	United Kingdom	IBM-285	EDC$EKEK
En_GB.IBM-924	English	United Kingdom	IBM-924	EDC$EKEZ
En_GB.IBM-924@euro	English	United Kingdom	IBM-924	EDC@EKEZ
En_GB.IBM-1047	English	United Kingdom	IBM-1047	EDC$EKEY
En_GB.IBM-1146	English	United Kingdom	IBM-1146	EDC$EKHK
En_GB.IBM-1146@euro	English	United Kingdom	IBM-1146	EDC@EKHK
En_IE.IBM-924	English	Ireland	IBM-924	EDC$EIEZ
En_IE.IBM-924@euro	English	Ireland	IBM-924	EDC@EIEZ
En_IE.IBM-924@preeuro [1]	English	Ireland	IBM-924	EDC3EIEZ
En_JP.IBM-1027	English	Japan	IBM-1027	EDC$EJEX

Table 52. Compiled Locales Supplied With LE/VSE C Run-Time (continued)

Local name as in `setlocale()` argument	Language	Country or Region	Codeset	Phase name
En_JP.IBM-5123	English	Japan	IBM-5123	EDC$EJHX
En_US.IBM-037	English	United States	IBM-037	EDC$EUEA
En_US.IBM-1047	English	United States	IBM-1047	EDC$EUEY
En_US.IBM-1140	English	United States	IBM-1140	EDC$EUHA
En_US.IBM-1140@euro	English	United States	IBM-1140	EDC@EUHA
Es_ES.IBM-284 [1]	Spanish	Spain	IBM-284	EDC$ESEJ
Es_ES.IBM-924	Spanish	Spain	IBM-924	EDC$ESEZ
Es_ES.IBM-924@euro	Spanish	Spain	IBM-924	EDC@ESEZ
Es_ES.IBM-924@preeuro [1]	Spanish	Spain	IBM-924	EDC3ESEZ
Es_ES.IBM-1047 [1]	Spanish	Spain	IBM-1047	EDC$ESEY
Es_ES.IBM-1145	Spanish	Spain	IBM-1145	EDC$ESHJ
Es_ES.IBM-1145@euro	Spanish	Spain	IBM-1145	EDC@ESHJ
Es_ES.IBM-1145@preeuro [1]	Spanish	Spain	IBM-1145	EDC3ESHJ
Et_EE.IBM-1122	Estonian	Estonia	IBM-1122	EDC$EEFD
Et_EE.IBM-1157	Estonian	Estonia	IBM-1157	EDC$EEHD
Fi_FI.IBM-278 [1]	Finnish	Finland	IBM-278	EDC$FIEF
Fi_FI.IBM-924	Finnish	Finland	IBM-924	EDC$FIEZ
Fi_FI.IBM-924@euro	Finnish	Finland	IBM-924	EDC@FIEZ
Fi_FI.IBM-924@preeuro [1]	Finnish	Finland	IBM-924	EDC3FIEZ
Fi_FI.IBM-1047 [1]	Finnish	Finland	IBM-1047	EDC$FIEY
Fi_FI.IBM-1143	Finnish	Finland	IBM-1143	EDC$FIHF
Fi_FI.IBM-1143@euro	Finnish	Finland	IBM-1143	EDC@FIHF
Fi_FI.IBM-1143@preeuro [1]	Finnish	Finland	IBM-1143	EDC3FIHF
Fr_BE.IBM-500 [1]	French	Belgium	IBM-500	EDC$FBEO
Fr_BE.IBM-924	French	Belgium	IBM-924	EDC$FBEZ
Fr_BE.IBM-924@euro	French	Belgium	IBM-924	EDC@FBEZ
Fr_BE.IBM-924@preeuro [1]	French	Belgium	IBM-924	EDC3FBEZ
Fr_BE.IBM-1047 [1]	French	Belgium	IBM-1047	EDC$FBEY
Fr_BE.IBM-1148	French	Belgium	IBM-1148	EDC$FBHO
Fr_BE.IBM-1148@euro	French	Belgium	IBM-1148	EDC@FBHO
Fr_BE.IBM-1148@preeuro [1]	French	Belgium	IBM-1148	EDC3FBHO
Fr_CA.IBM-037	French	Canada	IBM-037	EDC$FCEA
Fr_CA.IBM-1047	French	Canada	IBM-1047	EDC$FCEY
Fr_CA.IBM-1140	French	Canada	IBM-1140	EDC$FCHA
Fr_CA.IBM-1140@euro	French	Canada	IBM-1140	EDC@FCHA
Fr_CH.IBM-500	French	Switzerland	IBM-500	EDC$FSEO
Fr_CH.IBM-1047	French	Switzerland	IBM-1047	EDC$FSEY
Fr_CH.IBM-1148	French	Switzerland	IBM-1148	EDC$FSHO
Fr_CH.IBM-1148@euro	French	Switzerland	IBM-1148	EDC@FSHO

Table 52. Compiled Locales Supplied With LE/VSE C Run-Time (continued)

Local name as in `setlocale()` argument	Language	Country or Region	Codeset	Phase name
Fr_FR.IBM-297 ¹	French	France	IBM-297	EDC$FFEM
Fr_FR.IBM-924	French	France	IBM-924	EDC$FFEZ
Fr_FR.IBM-924@euro	French	France	IBM-924	EDC@FFEZ
Fr_FR.IBM-924@preeuro ¹	French	France	IBM-924	EDC3FFEZ
Fr_FR.IBM-1047 ¹	French	France	IBM-1047	EDC$FFEY
Fr_FR.IBM-1147	French	France	IBM-1147	EDC$FFHM
Fr_FR.IBM-1147@euro	French	France	IBM-1147	EDC@FFHM
Fr_FR.IBM-1147@preeuro ¹	French	France	IBM-1147	EDC3FFHM
Fr_LU.IBM-924	French	Luxembourg	IBM-924	EDC$FLEZ
Fr_LU.IBM-924@euro	French	Luxembourg	IBM-924	EDC@FLEZ
Fr_LU.IBM-924@preeuro ¹	French	Luxembourg	IBM-924	EDC3FLEZ
Hr_HR.IBM-870	Croatian	Croatia	IBM-870	EDC$HREQ
Hr_HR.IBM-1153	Croatian	Croatia	IBM-1153	EDC$HRMB
Hu_HU.IBM-870	Hungarian	Hungary	IBM-870	EDC$HUEQ
Hu_HU.IBM-1153	Hungarian	Hungary	IBM-1153	EDC$HUMB
Is_IS.IBM-871	Iceland	Iceland	IBM-871	EDC$ISER
Is_IS.IBM-1047	Iceland	Iceland	IBM-1047	EDC$ISEY
Is_IS.IBM-1149	Icelandic	Iceland	IBM-1149	EDC$ISHR
Is_IS.IBM-1149@euro	Icelandic	Iceland	IBM-1149	EDC@ISHR
It_IT.IBM-280 ¹	Italian	Italy	IBM-280	EDC$ITEG
It_IT.IBM-924	Italian	Italy	IBM-924	EDC$ITEZ
It_IT.IBM-924@euro	Italian	Italy	IBM-924	EDC@ITEZ
It_IT.IBM-924@preeuro ¹	Italian	Italy	IBM-924	EDC3ITEZ
It_IT.IBM-1047 ¹	Italian	Italy	IBM-1047	EDC$ITEY
It_IT.IBM-1144	Italian	Italy	IBM-1144	EDC$ITHG
It_IT.IBM-1144@euro	Italian	Italy	IBM-1144	EDC@ITHG
It_IT.IBM-1144@preeuro ¹	Italian	Italy	IBM-1144	EDC3ITHG
Iw_IL.IBM-424	Hebrew	Israel	IBM-424	EDC$ILFB
Iw_IL.IBM-12712	Hebrew	Israel	IBM-12712	EDC$ILHH
Ja_JP.IBM-290	Japanese	Japan	IBM-290	EDC$JAEL
Ja_JP.IBM-930	Japanese	Japan	IBM-930	EDC$JAEU
Ja_JP.IBM-939	Japanese	Japan	IBM-939	EDC$JAEV
Ja_JP.IBM-1027	Japanese	Japan	IBM-1027	EDC$JAEX
Ja_JP.IBM-5123	Japanese	Japan	IBM-5123	EDC$JAHX
Ja_JP.IBM-8482	Japanese	Japan	IBM-8482	EDC$JAHL
Ko_KR.IBM-933	Korean	Korea	IBM-933	EDC$KRGZ
Lt_LT.IBM-1112	Lithuanian	Lithuania	IBM-1112	EDC$LTGD
Lt_LT.IBM-1156	Lithuanian	Lithuania	IBM-1156	EDC$LTHZ
Mk_MK.IBM-1025	Macedonian	Macedonia	IBM-1025	EDC$MMFE

Table 52. Compiled Locales Supplied With LE/VSE C Run-Time (continued)

Local name as in `setlocale()` argument	Language	Country or Region	Codeset	Phase name
Mk_MK.IBM-1154	Macedonian	Macedonia	IBM-1154	EDC$MMHT
Nl_BE.IBM-500 [1]	Dutch	Belgium	IBM-500	EDC$NBEO
Nl_BE.IBM-924	Dutch	Belgium	IBM-924	EDC$NBEZ
Nl_BE.IBM-924@euro	Dutch	Belgium	IBM-924	EDC@NBEZ
Nl_BE.IBM-924@preeuro [1]	Dutch	Belgium	IBM-924	EDC3NBEZ
Nl_BE.IBM-1047 [1]	Dutch	Belgium	IBM-1047	EDC$NBEY
Nl_BE.IBM-1148	Dutch	Belgium	IBM-1148	EDC$NBHO
Nl_BE.IBM-1148@euro	Dutch	Belgium	IBM-1148	EDC@NBHO
Nl_BE.IBM-1148@preeuro [1]	Dutch	Belgium	IBM-1148	EDC3NBHO
Nl_NL.IBM-037 [1]	Dutch	Netherlands	IBM-037	EDC$NNEA
Nl_NL.IBM-924	Dutch	Netherlands	IBM-924	EDC$NNEZ
Nl_NL.IBM-924@euro	Dutch	Netherlands	IBM-924	EDC@NNEZ
Nl_NL.IBM-924@preeuro [1]	Dutch	Netherlands	IBM-924	EDC3NNEZ
Nl_NL.IBM-1047 [1]	Dutch	Netherlands	IBM-1047	EDC$NNEY
Nl_NL.IBM-1140	Dutch	Netherlands	IBM-1140	EDC$NNHA
Nl_NL.IBM-1140@euro	Dutch	Netherlands	IBM-1140	EDC@NNHA
Nl_NL.IBM-1140@preeuro [1]	Dutch	Netherlands	IBM-1140	EDC3NNHA
No_NO.IBM-277	Norwegian	Norway	IBM-277	EDC$NOEE
No_NO.IBM-1047	Norwegian	Norway	IBM-1047	EDC$NOEY
No_NO.IBM-1142	Norwegian	Norway	IBM-1142	EDC$NOHE
No_NO.IBM-1142@euro	Norwegian	Norway	IBM-1142	EDC@NOHE
Pl_PL.IBM-870	Polish	Poland	IBM-870	EDC$PLEQ
Pl_PL.IBM-1153	Polish	Poland	IBM-1153	EDC$PLMB
Pt_BR.IBM-037	Portugese	Brazil	IBM-037	EDC$BREA
Pt_BR.IBM-1047	Portugese	Brazil	IBM-1047	EDC$BREY
Pt_BR.IBM-1140	Portugese	Brazil	IBM-1140	EDC$BRHA
Pt_BR.IBM-1140@euro	Portugese	Brazil	IBM-1140	EDC@BRHA
Pt_PT.IBM-037 [1]	Portugese	Portugal	IBM-037	EDC$PTEA
Pt_PT.IBM-924	Portugese	Portugal	IBM-924	EDC$PTEZ
Pt_PT.IBM-924@euro	Portugese	Portugal	IBM-924	EDC@PTEZ
Pt_PT.IBM-924@preeuro [1]	Portugese	Portugal	IBM-924	EDC3PTEZ
Pt_PT.IBM-1047 [1]	Portugese	Portugal	IBM-1047	EDC$PTEY
Pt_PT.IBM-1140	Portugese	Portugal	IBM-1140	EDC$PTHA
Pt_PT.IBM-1140@euro	Portugese	Portugal	IBM-1140	EDC@PTHA
Pt_PT.IBM-1140@preeuro [1]	Portugese	Portugal	IBM-1140	EDC3PTHA
Ro_RO.IBM-870	Romanian	Romania	IBM-870	EDC$ROEQ
Ro_RO.IBM-1153	Romanian	Romania	IBM-1153	EDC$ROMB
Ru_RU.IBM-1025	Russian	Russia	IBM-1025	EDC$RUFE
Ru_RU.IBM-1154	Russian	Russia	IBM-1154	EDC$RUHT

Table 52. Compiled Locales Supplied With LE/VSE C Run-Time (continued)

Local name as in `setlocale()` argument	Language	Country or Region	Codeset	Phase name
Sh_SP.IBM-870	Serbian (Latin)	Serbia	IBM-870	EDC$SLEQ
Sh_SP.IBM-1153	Serbian (Latin)	Serbia	IBM-1153	EDC$SLMB
Sr_SP.IBM-1154	Serbian (Cyrillic)	Serbia	IBM-1154	EDC$SCHT
Si_SI.IBM-870	Slovene	Slovenia	IBM-870	EDC$SIEQ
Si_SI.IBM-1153	Slovene	Slovenia	IBM-1153	EDC$SIMB
Sk_SK.IBM-870	Slovak	Slovakia	IBM-870	EDC$SKEQ
Sk_SK.IBM-1153	Slovak	Slovakia	IBM-1153	EDC$SKMB
Sq_AL.IBM-1047	Albanian	Albania	IBM-1047	EDC$SAEY
Sq_AL.IBM-1148	Albanian	Albania	IBM-1148	EDC$SAHO
Sq_AL.IBM-1148@euro	Albanian	Albania	IBM-1148	EDC@SAHO
Sr_SP.IBM-1025	Serbian (Cyrillic)	Serbia	IBM-1025	EDC$SCFE
Sv_SE.IBM-278	Swedish	Sweden	IBM-278	EDC$SVEF
Sv_SE.IBM-924	Swedish	Sweden	IBM-924	EDC$SVEZ
Sv_SE.IBM-924@euro	Swedish	Sweden	IBM-924	EDC@SVEZ
Sv_SE.IBM-1047	Swedish	Sweden	IBM-1047	EDC$SVEY
Sv_SE.IBM-1143	Swedish	Sweden	IBM-1143	EDC$SVHF
Sv_SE.IBM-1143@euro	Swedish	Sweden	IBM-1143	EDC@SVHF
Th_TH.IBM-838	Thai	Thailand	IBM-838	EDC$THEP
Th_TH.IBM-1160	Thai	Thailand	IBM-1160	EDC$THHP
Tr_TR.IBM-1026	Turkish	Turkey	IBM-1026	EDC$TREW
Tr_TR.IBM-1155	Turkish	Turkey	IBM-1155	EDC$TRHW
Zh_CN.IBM-935	Simplified Chinese	China	IBM-935	EDC$ZCGY
Zh_TW.IBM-937	Traditional Chinese	Taiwan	IBM-937	EDC$ZTGW
Zh_TW.IBM-1371	Traditional Chinese	Taiwan	IBM-1371	EDC$ZTKA

Note:

1. This locale does NOT support the Euro currency. Only the “non euro” locales for countries that adapted the Euro as their legal currency are flagged with this note.

The locale source files are supplied to enable you to build locales in coded character sets other than those supplied. The locale sources supplied are listed in the following table. Under VSE, the source files are in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE).

The “Applicable Codesets” column indicates which charmap files can be used with the source files to build the locales. The values in this column indicate the following:

All The locale source contains only the portable character set and can be used to build a locale with any of the supplied charmap files.

Latin-1
The locale source contains characters from the Latin-1 character set, and can be used to build a locale from any of the supplied Latin-1 charmap files. See Appendix E, “Charmap Files Supplied with LE/VSE C Run-Time,” on page 443 for a list of Latin-1 charmap files.

Other The locale source is specific to the specified coded character set, and can only be used to build a locale with the specified charmap file.

Table 53. Locale Source Files Supplied With LE/VSE C Run-Time

Language	Country or Region	Source name	Applicable Codesets
POSIX (built-in)		EDC$POSX	All
SAA (built-in)		EDC$SAAC	Latin-1
Bulgarian	Bulgaria	EDC$BGFE	IBM-1025
Bulgarian	Bulgaria	EDC$BGHT	IBM-1154
Portugese	Brazil	EDC$BREY	Latin-1
Portugese	Brazil	EDC$BRHA	IBM-1140
Portugese (Euro)	Brazil	EDC@BRHA	IBM-1140
Catalan (Euro)	Spain	EDC$CSEZ	IBM-924
Catalan (Euro)	Spain	EDC@CSEZ	IBM-924
Catalan [1]	Spain	EDC3CSEZ	IBM-924
Czech	Czech Republic	EDC$CZEQ	IBM-870
Czech	Czech Republic	EDC$CZMB	IBM-1153
Danish	Denmark	EDC$DAEY	Latin-1
Danish	Denmark	EDC$DAEZ	IBM-924
Danish (Euro)	Denmark	EDC@DAEZ	IBM-924
Danish	Denmark	EDC$DAHE	IBM-1142
Danish (Euro)	Denmark	EDC@DAHE	IBM-1142
German	Switzerland	EDC$DCEY	Latin-1
German	Switzerland	EDC$DCHO	IBM-1148
German (Euro)	Switzerland	EDC@DCHO	IBM-1148
German [1]	Germany	EDC$DDEY	Latin-1
German (Euro)	Germany	EDC$DDEZ	IBM-924
German (Euro)	Germany	EDC@DDEZ	IBM-924
German [1]	Germany	EDC3DDEZ	IBM-924
German (Euro)	Germany	EDC$DDHB	IBM-1141
German (Euro)	Germany	EDC@DDHB	IBM-1141
German [1]	Germany	EDC3DDHB	IBM-1141
German (Euro)	Luxembourg	EDC$DLEZ	IBM-924
German (Euro)	Luxembourg	EDC@DLEZ	IBM-924
German [1]	Luxembourg	EDC3DLEZ	IBM-924
German (Euro)	Austria	EDC$DTEZ	IBM-924
German (Euro)	Austria	EDC@DTEZ	IBM-924

Table 53. Locale Source Files Supplied With LE/VSE C Run-Time (continued)

Language	Country or Region	Source name	Applicable Codesets
German ¹	Austria	EDC3DTEZ	IBM-924
English (Euro)	Belgium	EDC$EBEZ	IBM-924
English (Euro)	Belgium	EDC@EBEZ	IBM-924
English ¹	Belgium	EDC3EBEZ	IBM-924
Estonian	Estonia	EDC$EEFD	IBM-1122
Estonian	Estonia	EDC$EEHD	IBM-1157
English (Euro)	Ireland	EDC$EIEZ	IBM-924
English (Euro)	Ireland	EDC@EIEZ	IBM-924
English ¹	Ireland	EDC3EIEZ	IBM-924
English	Japan	EDC$EJEX	IBM-1027
English	Japan	EDC$EJHX	IBM-5123
English	United Kingdom	EDC$EKEY	Latin-1
English	United Kingdom	EDC$EKEZ	IBM-924
English (Euro)	United Kingdom	EDC@EKEZ	IBM-924
English	United Kingdom	EDC$EKHK	IBM-1146
English (Euro)	United Kingdom	EDC@EKHK	IBM-1146
Greek ¹	Greece	EDC$ELES	IBM-875
Greek (Euro)	Greece	EDC$ELHS	IBM-4971
Greek (Euro)	Greece	EDC@ELHS	IBM-4971
Greek ¹	Greece	EDC3ELHS	IBM-4971
Spanish ¹	Spain	EDC$ESEY	Latin-1
Spanish (Euro)	Spain	EDC$ESEZ	IBM-924
Spanish (Euro)	Spain	EDC@ESEZ	IBM-924
Spanish ¹	Spain	EDC3ESEZ	IBM-924
Spanish (Euro)	Spain	EDC$ESHJ	IBM-1145
Spanish (Euro)	Spain	EDC@ESHJ	IBM-1145
Spanish ¹	Spain	EDC3ESHJ	IBM-1145
English	United States	EDC$EUEY	Latin-1
English	United States	EDC$EUHA	IBM-1140
English (Euro)	United States	EDC@EUHA	IBM-1140
Finnish ¹	Finland	EDC$FIEY	Latin-1
Finnish (Euro)	Finland	EDC$FIEZ	IBM-924
Finnish (Euro)	Finland	EDC@FIEZ	IBM-924
Finnish ¹	Finland	EDC3FIEZ	IBM-924
Finnish (Euro)	Finland	EDC$FIHF	IBM-1143
Finnish (Euro)	Finland	EDC@FIHF	IBM-1143
Finnish ¹	Finland	EDC3FIHF	IBM-1143
French ¹	Belgium	EDC$FBEY	Latin-1
French (Euro)	Belgium	EDC$FBEZ	IBM-924
French (Euro)	Belgium	EDC@FBEZ	IBM-924

Table 53. Locale Source Files Supplied With LE/VSE C Run-Time (continued)

Language	Country or Region	Source name	Applicable Codesets
French [1]	Belgium	EDC3FBEZ	IBM-924
French (Euro)	Belgium	EDC$FBHO	IBM-1148
French (Euro)	Belgium	EDC@FBHO	IBM-1148
French [1]	Belgium	EDC3FBHO	IBM-1148
French	Canada	EDC$FCEY	Latin-1
French	Canada	EDC$FCHA	IBM-1140
French (Euro)	Canada	EDC@FCHA	IBM-1140
French [1]	France	EDC$FFEY	Latin-1
French (Euro)	France	EDC$FFEZ	IBM-924
French (Euro)	France	EDC@FFEZ	IBM-924
French [1]	France	EDC3FFEZ	IBM-924
French (Euro)	France	EDC$FFHM	IBM-1147
French (Euro)	France	EDC@FFHM	IBM-1147
French [1]	France	EDC3FFHM	IBM-1147
French (Euro)	Luxembourg	EDC$FLEZ	IBM-924
French (Euro)	Luxembourg	EDC@FLEZ	IBM-924
French [1]	Luxembourg	EDC3FLEZ	IBM-924
French	Switzerland	EDC$FSEY	Latin-1
French	Switzerland	EDC$FSHO	IBM-1148
French (Euro)	Switzerland	EDC@FSHO	IBM-1148
Croatian	Croatia	EDC$HREQ	IBM-870
Croatian	Croatia	EDC$HRMB	IBM-1153
Hungarian	Hungary	EDC$HUEQ	IBM-870
Hungarian	Hungary	EDC$HUMB	IBM-1153
Hebrew	Israel	EDC$ILFB	IBM-424
Hebrew	Israel	EDC$ILHH	IBM12712
Icelandic	Iceland	EDC$ISEY	Latin-1
Icelandic	Iceland	EDC$ISHR	IBM-1149
Icelandic (Euro)	Iceland	EDC@ISHR	IBM-1149
Italian [1]	Italy	EDC$ITEY	Latin-1
Italian (Euro)	Italy	EDC$ITEZ	IBM-924
Italian (Euro)	Italy	EDC@ITEZ	IBM-924
Italian [1]	Italy	EDC3ITEZ	IBM-924
Italian (Euro)	Italy	EDC$ITHG	IBM-1144
Italian (Euro)	Italy	EDC@ITHG	IBM-1144
Italian [1]	Italy	EDC3ITHG	IBM-1144
Japanese	Japan	EDC$JAEL	IBM-290
Japanese	Japan	EDC$JAEU	IBM-930
Japanese	Japan	EDC$JAEV	IBM-939
Japanese	Japan	EDC$JAEX	IBM-1027

Table 53. Locale Source Files Supplied With LE/VSE C Run-Time (continued)

Language	Country or Region	Source name	Applicable Codesets
Japanese	Japan	EDC$JAHL	IBM-8482
Japanese	Japan	EDC$JAHX	IBM-5123
Korean	Korea	EDC$KRGZ	IBM-933
Lithuanian	Lithuania	EDC$LTGD	IBM-1112
Lithuanian	Lithuania	EDC$LTHZ	IBM-1156
Macedonian	Macedonia	EDC$MMFE	IBM-1025
Macedonian	Macedonia	EDC$MMHT	IBM-1154
Dutch ¹	Belgium	EDC$NBEY	Latin-1
Dutch (Euro)	Belgium	EDC$NBEZ	IBM-924
Dutch (Euro)	Belgium	EDC@NBEZ	IBM-924
Dutch ¹	Belgium	EDC3NBEZ	IBM-924
Dutch (Euro)	Belgium	EDC$NBHO	IBM-1148
Dutch (Euro)	Belgium	EDC@NBHO	IBM-1148
Dutch ¹	Belgium	EDC3NBHO	IBM-1148
Dutch ¹	Netherlands	EDC$NNEY	Latin-1
Dutch (Euro)	Netherlands	EDC$NNEZ	IBM-924
Dutch (Euro)	Netherlands	EDC@NNEZ	IBM-924
Dutch ¹	Netherlands	EDC3NNEZ	IBM-924
Dutch (Euro)	Netherlands	EDC$NNHA	IBM-1140
Dutch (Euro)	Netherlands	EDC@NNHA	IBM-1140
Dutch ¹	Netherlands	EDC3NNHA	IBM-1140
Norwegian	Norway	EDC$NOEY	Latin-1
Norwegian	Norway	EDC$NOHE	IBM-1142
Norwegian (Euro)	Norway	EDC@NOHE	IBM-1142
Polish	Poland	EDC$PLEQ	IBM-870
Polish	Poland	EDC$PLMB	IBM-1153
Portuguese ¹	Portugal	EDC$PTEY	Latin-1
Portuguese (Euro)	Portugal	EDC$PTEZ	IBM-924
Portuguese (Euro)	Portugal	EDC@PTEZ	IBM-924
Portuguese ¹	Portugal	EDC3PTEZ	IBM-924
Portuguese (Euro)	Portugal	EDC$PTHA	IBM-1140
Portuguese (Euro)	Portugal	EDC@PTHA	IBM-1140
Portuguese ¹	Portugal	EDC3PTHA	IBM-1140
Romanian	Romania	EDC$ROEQ	IBM-870
Romanian	Romania	EDC$ROMB	IBM-1153
Russian	Russia	EDC$RUFE	IBM-1025
Russian	Russia	EDC$RUHT	IBM-1154
Albanian	Albania	EDC$SAEY	Latin-1
Albanian	Albania	EDC$SAHO	IBM-1148
Albanian (Euro)	Albania	EDC@SAHO	IBM-1148

Table 53. Locale Source Files Supplied With LE/VSE C Run-Time (continued)

Language	Country or Region	Source name	Applicable Codesets
Serbian (Cyrillic)	Serbia	EDC$SCFE	IBM-1025
Serbian (Cyrillic)	Serbia	EDC$SCHT	IBM-1154
Slovak	Slovakia	EDC$SKEQ	IBM-870
Slovak	Slovakia	EDC$SKMB	IBM-1153
Slovene	Slovenia	EDC$SIEQ	IBM-870
Slovene	Slovenia	EDC$SIMB	IBM-1153
Serbian (Latin)	Serbia	EDC$SLEQ	IBM-870
Serbian (Latin)	Serbia	EDC$SLMB	IBM-1153
Swedish	Sweden	EDC$SVEY	Latin-1
Swedish	Sweden	EDC$SVEZ	IBM-924
Swedish (Euro)	Sweden	EDC@SVEZ	IBM-924
Swedish	Sweden	EDC$SVHF	IBM-1143
Swedish (Euro)	Sweden	EDC@SVHF	IBM-1143
Thai	Thailand	EDC$THEP	IBM-838
Thai	Thailand	EDC$THHP	IBM-1160
Turkish	Turkey	EDC$TREW	IBM-1026
Turkish	Turkey	EDC$TRHW	IBM-1155
Simplified Chinese	China	EDC$ZCGY	IBM-935
Traditional Chinese	Taiwan	EDC$ZTGW	IBM-937
Traditional Chinese	Taiwan	EDC$ZTKA	IBM-1371

Note:

1. This locale does NOT support the Euro currency. Only the “non euro” locales for countries that adapted the Euro as their legal currency are flagged with this note.

Appendix E. Charmap Files Supplied with LE/VSE C Run-Time

All the locales supplied were built using the appropriate charmap file that represents the coded character sets described by the `CodesetRegistry-CodesetEncoding` element of the locale name.

All of these charmap files are provided with the base feature of LE/VSE in the LE/VSE installation sublibrary (default is PRD2.SCEEBASE). The `-` sign is converted to the @ character.

The following table lists the coded character set name, which is the same as the name of the corresponding charmap file, and the national language each code set represents.

The column marked *Latin-1* indicates whether the charmap file is for a coded character set that contains the Latin-1 character set.

Table 54. Coded Character Set Names and Corresponding National Languages

Codeset	Country or Region	Latin-1
IBM-037	USA, Canada, Brazil	Yes
IBM-273	Germany, Austria	Yes
IBM-274	Belgium	Yes
IBM-275	Brazil	Yes
IBM-277	Denmark, Norway	Yes
IBM-278	Finland, Sweden	Yes
IBM-280	Italy	Yes
IBM-281	Japan (Latin-1)	Yes
IBM-282	Portugal	Yes
IBM-284	Spain, Latin America	Yes
IBM-285	United Kingdom	Yes
IBM-290	Japan (Katakana)	No
IBM-297	France	Yes
IBM-424	Israel	No
IBM-500	International	Yes
IBM-838	Thailand	No
IBM-870	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia (Latin), Slovakia, Slovenia	No
IBM-871	Iceland	Yes
IBM-875	Greece	No
IBM-924	Latin 9/Open Systems	No
IBM-930	Japan (Katakana, combined with DBCS)	No
IBM-933	Korea	No
IBM-935	China	No
IBM-937	Taiwan	No

Table 54. Coded Character Set Names and Corresponding National Languages (continued)

Codeset	Country or Region	Latin-1
IBM-939	Japan (Latin, combined with DBCS)	No
IBM-1025	Bulgaria, Macedonia, Russia, Serbia (Cyrillic)	No
IBM-1026	Turkey	No
IBM-1027	Japan (Latin) extended	No
IBM-1047	Latin-1/Open Systems	Yes
IBM-1112	Lithuania	No
IBM-1122	Estonia	No
IBM-1124	Ukraine	No
IBM-1140	USA, Canada, Brazil (Euro)	Yes
IBM-1141	Germany, Austria (Euro)	Yes
IBM-1142	Denmark, Norway (Euro)	Yes
IBM-1143	Finland, Sweden (Euro)	Yes
IBM-1144	Italy (Euro)	Yes
IBM-1145	Spain, Latin America (Euro)	Yes
IBM-1146	United Kingdom (Euro)	Yes
IBM-1147	France (Euro)	Yes
IBM-1148	International (Euro)	Yes
IBM-1149	Iceland (Euro)	Yes
IBM-1153	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia (Latin), Slovakia, Slovenia	No
IBM-1154	Bulgaria, Macedonia, Russia, Serbia, (Cyrillic)	No
IBM-1155	Turkey	No
IBM-1156	Lithuania	No
IBM-1157	Estonia	No
IBM-1160	Thailand	No
IBM-1371	Taiwan	No
IBM-4971	Greece	No
IBM-5123	Japan	No
IBM-8482	Japan	No
IBM-12712	Israel	No

Only the charmap files for IBM-930 and IBM-939 specify `<mb_cur_max>` as 4 and include the definition of the double-byte characters. All other charmap files define the single-byte character sets, and specify the `<mb_cur_max>` as 1.

Note: The SAA C locale is built with the charmap IBM-1047, but has `<mb_cur_max>` set to 4 in order to maintain compatibility with old releases of C/370.

Any of these charmaps that represent the same character set, even though they represent different encoding of the same character sets, can be used with the any locale source that uses the same character set, to build a new locale and charmap combination. See Chapter 30, “Building a Locale,” on page 321 for information about building your own locales.

Appendix F. Examples of the Charmap and Locale Definition Source Files

Following are examples of the charmap source and locale definition source files.

Charmap File

This example shows the charmap file for the encoded character set IBM-1047.

Charmap File

```
<code_set_name>     "IBM-1047"
<mb_cur_max>        1
<mb_cur_min>        1
<escape_char>       /
<comment_char>      %

CHARMAP
<NUL>                  /x00
<SOH>                  /x01
<STX>                  /x02
<ETX>                  /x03
<SEL>                  /x04
<tab>                  /x05
<HT>                   /x05
<RNL>                  /x06
<DEL>                  /x07
<GE>                   /x08
<SPS>                  /x09
<RPT>                  /x0a
<vertical-tab>         /x0b
<VT>                   /x0b
<form-feed>            /x0c
<FF>                   /x0c
<carriage-return>      /x0d
<CR>                   /x0d
<SO>                   /x0e
<SI>                   /x0f
<DLE>                  /x10
<DC1>                  /x11
<DC2>                  /x12
<DC3>                  /x13
<RES>                  /x14
<newline>              /x15
<backspace>            /x16
<BS>                   /x16
<POC>                  /x17
<CAN>                  /x18
<EM>                   /x19
<UBS>                  /x1a
<CU1>                  /x1b
<IFS>                  /x1c    % file separator
<IS4>                  /x1c
<FS>                   /x1c
<IGS>                  /x1d    % group separator
<IS3>                  /x1d
<GS>                   /x1d
<IRS>                  /x1e    % record separator
<IS2>                  /x1e
<RS>                   /x1e
<IUS>                  /x1f    % unit separator
<IS1>                  /x1f
```

```
<US>                    /x1f
<ITB>                   /x1f
<DS>                    /x20
<SOS>                   /x21
<FS>                    /x22    % field separator
<WUS>                   /x23
<BYP>                   /x24
<LF>                    /x25
<ETB>                   /x26
<ESC>                   /x27
<SA>                    /x28
<SFE>                   /x29
<SM>                    /x2a
<CSP>                   /x2b
<MFA>                   /x2c
<ENQ>                   /x2d
<ACK>                   /x2e
<alert>                 /x2f
<BEL>                   /x2f
<SYN>                   /x32
<IR>                    /x33
<PP>                    /x34
<TRN>                   /x35
<NBS>                   /x36
<EOT>                   /x37
<SBS>                   /x38
<IT>                    /x39
<RFF>                   /x3a
<CU3>                   /x3b
<DC4>                   /x3c
<NAK>                   /x3d
<SUB>                   /x3f
<space>                 /x40
<SP01>                  /x40
<RSP>                   /x41
<SP30>                  /x41
<a-circumflex>          /x42
<LA15>                  /x42
<a-diaeresis>           /x43
<LA17>                  /x43
<a-grave>               /x44
<LA13>                  /x44
<a-acute>               /x45
<LA11>                  /x45
<a-tilde>               /x46
<LA19>                  /x46
<a-ring>                /x47
<LA27>                  /x47
<c-cedilla>             /x48
<LC41>                  /x48
<n-tilde>               /x49
<LN19>                  /x49
<cent>                  /x4a
<SC04>                  /x4a
<period>                /x4b
<SP11>                  /x4b
<less-than-sign>        /x4c
<SA03>                  /x4c
<left-parenthesis>      /x4d
<SP06>                  /x4d
<plus-sign>             /x4e
<SA01>                  /x4e
<vertical-line>         /x4f
<SM13>                  /x4f
<ampersand>             /x50
<SM03>                  /x50
<e-acute>               /x51
```

```
<LE11>                  /x51
<e-circumflex>          /x52
<LE15>                  /x52
<e-diaeresis>           /x53
<LE17>                  /x53
<e-grave>               /x54
<LE13>                  /x54
<i-acute>               /x55
<LI11>                  /x55
<i-circumflex>          /x56
<LI15>                  /x56
<i-diaeresis>           /x57
<LI17>                  /x57
<i-grave>               /x58
<LI13>                  /x58
<s-sharp>               /x59
<LS61>                  /x59
<exclamation-mark>      /x5a
<SP02>                  /x5a
<dollar-sign>           /x5b
<SC03>                  /x5b
<asterisk>              /x5c
<SM04>                  /x5c
<right-parenthesis>     /x5d
<SP07>                  /x5d
<semicolon>             /x5e
<SP14>                  /x5e
<circumflex>            /x5f
<circumflex-accent>     /x5f
<SD15>                  /x5f
<hyphen>                /x60
<hyphen-minus>          /x60
<SP10>                  /x60
<slash>                 /x61
<SP12>                  /x61
<A-circumflex>          /x62
<LA16>                  /x62
<A-diaeresis>           /x63
<LA18>                  /x63
<A-grave>               /x64
<LA14>                  /x64
<A-acute>               /x65
<LA12>                  /x65
<A-tilde>               /x66
<LA20>                  /x66
<A-ring>                /x67
<LA28>                  /x67
<C-cedilla>             /x68
<LC42>                  /x68
<N-tilde>               /x69
<LN20>                  /x69
<broken-bar>            /x6a
<SM65>                  /x6a
<comma>                 /x6b
<SP08>                  /x6b
<percent-sign>          /x6c
<SM02>                  /x6c
<underscore>            /x6d
<SP09>                  /x6d
<greater-than-sign>     /x6e
<SA05>                  /x6e
<question-mark>         /x6f
<SP15>                  /x6f
<o-slash>               /x70
<LO61>                  /x70
<E-acute>               /x71
<LE12>                  /x71
```

```
<E-circumflex>         /x72
<LE16>                 /x72
<E-diaeresis>          /x73
<LE18>                 /x73
<E-grave>              /x74
<LE14>                 /x74
<I-acute>              /x75
<LI12>                 /x75
<I-circumflex>         /x76
<LI16>                 /x76
<I-diaeresis>          /x77
<LI18>                 /x77
<I-grave>              /x78
<LI14>                 /x78
<grave-accent>         /x79
<SD13>                 /x79
<colon>                /x7a
<SP13>                 /x7a
<number-sign>          /x7b
<SM01>                 /x7b
<commercial-at>        /x7c
<SM05>                 /x7c
<apostrophe>           /x7d
<SP05>                 /x7d
<equals-sign>          /x7e
<SA04>                 /x7e
<quotation-mark>       /x7f
<SP04>                 /x7f
<O-slash>              /x80
<LO62>                 /x80
<a>                    /x81
<LA01>                 /x81
<b>                    /x82
<LB01>                 /x82
<c>                    /x83
<LC01>                 /x83
<d>                    /x84
<LD01>                 /x84
<e>                    /x85
<LE01>                 /x85
<f>                    /x86
<LF01>                 /x86
<g>                    /x87
<LG01>                 /x87
<h>                    /x88
<LH01>                 /x88
<i>                    /x89
<LI01>                 /x89
<left-angle-quotes>    /x8a
<guillemot-left>       /x8a
<SP17>                 /x8a
<right-angle-quotes>   /x8b
<guillemot-right>      /x8b
<SP18>                 /x8b
<eth>                  /x8c
<LD63>                 /x8c
<y-acute>              /x8d
<LY11>                 /x8d
<thorn>                /x8e
<LT63>                 /x8e
<plus-minus>           /x8f
<SA02>                 /x8f
<degree>               /x90
<SM19>                 /x90
<j>                    /x91
<LJ01>                 /x91
<k>                    /x92
```

```
<LK01>                /x92
<l>                   /x93
<LL01>                /x93
<m>                   /x94
<LM01>                /x94
<n>                   /x95
<LN01>                /x95
<o>                   /x96
<LO01>                /x96
<p>                   /x97
<LP01>                /x97
<q>                   /x98
<LQ01>                /x98
<r>                   /x99
<LR01>                /x99
<feminine>            /x9a
<SM21>                /x9a
<masculine>           /x9b
<SM20>                /x9b
<ae>                  /x9c
<LA51>                /x9c
<cedilla>             /x9d
<SD41>                /x9d
<AE>                  /x9e
<LA52>                /x9e
<currency>            /x9f
<SC01>                /x9f
<mu>                  /xa0
<SM17>                /xa0
<tilde>               /xa1
<SD19>                /xa1
<s>                   /xa2
<LS01>                /xa2
<t>                   /xa3
<LT01>                /xa3
<u>                   /xa4
<LU01>                /xa4
<v>                   /xa5
<LV01>                /xa5
<w>                   /xa6
<LW01>                /xa6
<x>                   /xa7
<LX01>                /xa7
<y>                   /xa8
<LY01>                /xa8
<z>                   /xa9
<LZ01>                /xa9
<exclamation-down>    /xaa
<SP03>                /xaa
<question-down>       /xab
<SP16>                /xab
<Eth>                 /xac
<LD62>                /xac
<left-square-bracket> /xad
<SM06>                /xad
<Thorn>               /xae
<LT64>                /xae
<registered>          /xaf
<SM53>                /xaf
<not>                 /xb0
<SM66>                /xb0
<sterling>            /xb1
<SC02>                /xb1
<yen>                 /xb2
<SC05>                /xb2
<dot>                 /xb3
<SD63>                /xb3
```

```
<copyright>            /xb4
<SM52>                 /xb4
<section>              /xb5
<SM24>                 /xb5
<paragraph>            /xb6
<SM25>                 /xb6
<one-quarter>          /xb7
<NF04>                 /xb7
<one-half>             /xb8
<NF01>                 /xb8
<three-quarters>       /xb9
<NF05>                 /xb9
<Y-acute>              /xba
<LY12>                 /xba
<diaeresis>            /xbb
<SD17>                 /xbb
<macron>               /xbc
<SM15>                 /xbc
<right-square-bracket> /xbd
<SM08>                 /xbd
<acute>                /xbe
<SD11>                 /xbe
<multiply>             /xbf
<SA07>                 /xbf
<left-brace>           /xc0
<left-curly-bracket>   /xc0
<SM11>                 /xc0
<A>                    /xc1
<LA02>                 /xc1
<B>                    /xc2
<LB02>                 /xc2
<C>                    /xc3
<LC02>                 /xc3
<D>                    /xc4
<LD02>                 /xc4
<E>                    /xc5
<LE02>                 /xc5
<F>                    /xc6
<LF02>                 /xc6
<G>                    /xc7
<LG02>                 /xc7
<H>                    /xc8
<LH02>                 /xc8
<I>                    /xc9
<LI02>                 /xc9
<syllable-hyphen>      /xca
<SP32>                 /xca
<o-circumflex>         /xcb
<LO15>                 /xcb
<o-diaeresis>          /xcc
<LO17>                 /xcc
<o-grave>              /xcd
<LO13>                 /xcd
<o-acute>              /xce
<LO11>                 /xce
<o-tilde>              /xcf
<LO19>                 /xcf
<right-brace>          /xd0
<right-curly-bracket>  /xd0
<SM14>                 /xd0
<J>                    /xd1
<LJ02>                 /xd1
<K>                    /xd2
<LK02>                 /xd2
<L>                    /xd3
<LL02>                 /xd3
<M>                    /xd4
```

```
<LM02>                  /xd4
<N>                     /xd5
<LN02>                  /xd5
<O>                     /xd6
<LO02>                  /xd6
<P>                     /xd7
<LP02>                  /xd7
<Q>                     /xd8
<LQ02>                  /xd8
<R>                     /xd9
<LR02>                  /xd9
<one-superior>          /xda
<ND011>                 /xda
<u-circumflex>          /xdb
<LU15>                  /xdb
<u-diaeresis>           /xdc
<LU17>                  /xdc
<u-grave>               /xdd
<LU13>                  /xdd
<u-acute>               /xde
<LU11>                  /xde
<y-diaeresis>           /xdf
<LY17>                  /xdf
<backslash>             /xe0
<reverse-solidus>       /xe0
<SM07>                  /xe0
<divide>                /xe1
<division>              /xe1
<SA06>                  /xe1
<S>                     /xe2
<LS02>                  /xe2
<T>                     /xe3
<LT02>                  /xe3
<U>                     /xe4
<LU02>                  /xe4
<V>                     /xe5
<LV02>                  /xe5
<W>                     /xe6
<LW02>                  /xe6
<X>                     /xe7
<LX02>                  /xe7
<Y>                     /xe8
<LY02>                  /xe8
<Z>                     /xe9
<LZ02>                  /xe9
<two-superior>          /xea
<ND021>                 /xea
<O-circumflex>          /xeb
<LO16>                  /xeb
<O-diaeresis>           /xec
<LO18>                  /xec
<O-grave>               /xed
<LO14>                  /xed
<O-acute>               /xee
<LO12>                  /xee
<O-tilde>               /xef
<LO20>                  /xef
<zero>                  /xf0
<ND10>                  /xf0
<one>                   /xf1
<ND01>                  /xf1
<two>                   /xf2
<ND02>                  /xf2
<three>                 /xf3
<ND03>                  /xf3
<four>                  /xf4
<ND04>                  /xf4
```

```
<five>                 /xf5
<ND05>                 /xf5
<six>                  /xf6
<ND06>                 /xf6
<seven>                /xf7
<ND07>                 /xf7
<eight>                /xf8
<ND08>                 /xf8
<nine>                 /xf9
<ND09>                 /xf9
<three-superior>       /xfa
<ND031>                /xfa
<U-circumflex>         /xfb
<LU16>                 /xfb
<U-diaeresis>          /xfc
<LU18>                 /xfc
<U-grave>              /xfd
<LU14>                 /xfd
<U-acute>              /xfe
<LU12>                 /xfe
<eo>                   /xff
END CHARMAP


CHARSETID
<NUL>...<SUB>                0
<space>...<U-acute>          1
END CHARSETID
```

The Locale Definition Source File

This example shows the typical locale definition file representing the cultural and language conventions in the United States of America. For this example (LC_COLLATE), please note the following:

- The digits (0...9) sort before the letters.
- Upper case and lowercase letters have the same primary sorting weight.
- For each letter, the uppercase letter sorts before the equivalent lowercase letter.

Locale Definition File

```
escape_char   /
comment-char  %

%%%%%%%%%%%%%
LC_CTYPE
%%%%%%%%%%%%%

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;/
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;/
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>

space   <tab>;<newline>;<vertical-tab>;<form-feed>;/
        <carriage-return>;<space>

cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;<NUL>;<SOH>;<STX>;/
        <ETX>;<SEL>;<RNL>;<DEL>;<GE>;<SPS>;<RPT>;<SI>;<SO>;<DLE>;<DC1>;/
        <DC2>;<DC3>;<RES>;<POC>;<CAN>;<EM>;<UBS>;<CU1>;<IFS>;/
        <IGS>;<IRS>;<ITB>;<DS>;<SOS>;<fs>;<WUS>;<BYP>;<LF>;/
        <ETB>;<ESC>;<SA>;<SM>;<CSP>;<MFA>;<ENQ>;<ACK>;/
        <SYN>;<IR>;<PP>;<TRN>;<NBS>;<EOT>;<SBS>;<IT>;<RFF>;/
        <CU3>;<DC4>;<NAK>;<SUB>
```

```
punct   <exclamation-mark>;<quotation-mark>;<number-sign>;<dollar-sign>;/
        <percent-sign>;<ampersand>;<apostrophe>;<left-parenthesis>;/
        <right-parenthesis>;<asterisk>;<plus-sign>;<comma>;/
        <hyphen-minus>;<period>;<slash>;<colon>;<semicolon>;/
        <less-than-sign>;<equals-sign>;<greater-than-sign>;/
        <question-mark>;<commercial-at>;<left-square-bracket>;/
        <backslash>;<right-square-bracket>;<circumflex>;/
        <underscore>;<grave-accent>;<left-curly-bracket>;/
        <vertical-line>;<right-curly-bracket>;<tilde>

digit   <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>

xdigit  <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>;/
        <A>;<B>;<C>;<D>;<E>;<F>;/
        <a>;<b>;<c>;<d>;<e>;<f>

blank   <space>;<tab>

END LC_CTYPE

%%%%%%%%%%%%%
LC_COLLATE
%%%%%%%%%%%%%

order_start forward;forward

<NUL>
...
<SUB>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen-minus>
<period>
<slash>
<zero>
...
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A> <A>;<A>
<B> <B>;<B>
<C> <C>;<C>
<D> <D>;<D>
<E> <E>;<E>
<F> <F>;<F>
<G> <G>;<G>
<H> <H>;<H>
<I> <I>;<I>
<J> <J>;<J>
```

```
<K> <K>;<K>
<L> <L>;<L>
<M> <M>;<M>
<N> <N>;<N>
<O> <O>;<O>
<P> <P>;<P>
<Q> <Q>;<Q>
<R> <R>;<R>
<S> <S>;<S>
<T> <T>;<T>
<U> <U>;<U>
<V> <V>;<V>
<W> <W>;<W>
<X> <X>;<X>
<Y> <Y>;<Y>
<Z> <Z>;<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a> <A>;<a>
<b> <B>;<b>
<c> <C>;<c>
<d> <D>;<d>
<e> <E>;<e>
<f> <F>;<f>
<g> <G>;<g>
<h> <H>;<h>
<i> <I>;<i>
<j> <J>;<j>
<k> <K>;<k>
<l> <L>;<l>
<m> <M>;<m>
<n> <N>;<n>
<o> <O>;<o>
<p> <P>;<p>
<q> <Q>;<q>
<r> <R>;<r>
<s> <S>;<s>
<t> <T>;<t>
<u> <U>;<u>
<v> <V>;<v>
<w> <W>;<w>
<x> <X>;<x>
<y> <Y>;<y>
<z> <Z>;<z>
UNDEFINED
order_end

END LC_COLLATE

%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%

int_curr_symbol   "<U><S><D><space>"
currency_symbol   "<dollar-sign>"
mon_decimal_point "<period>"
mon_thousands_sep "<comma>"
mon_grouping      "3;0"
positive_sign     ""
negative_sign     "<hyphen-minus>"
int_frac_digits   2
frac_digits       2
p_cs_precedes     1
```

```
p_sep_by_space    0
n_cs_precedes     1
n_sep_by_space    0
p_sign_posn       2
n_sign_posn       2
debit_sign        "<D><B>"
credit_sign       "<C><R>"
left_parenthesis  "<left-parenthesis>"
right_parenthesis "<right-parenthesis>"

END LC_MONETARY

%%%%%%%%%%%%
LC_NUMERIC
%%%%%%%%%%%%

decimal_point     "<period>"
thousands_sep     "<comma>"
grouping          "3;0"

END LC_NUMERIC

%%%%%%%%%%%%
LC_TIME
%%%%%%%%%%%%
abday   "<S><u><n>";/
        "<M><o><n>";/
        "<T><u><e>";/
        "<W><e><d>";/
        "<T><h><u>";/
        "<F><r><i>";/
        "<S><a><t>"

day     "<S><u><n><d><a><y>";/
        "<M><o><n><d><a><y>";/
        "<T><u><e><s><d><a><y>";/
        "<W><e><d><n><e><s><d><a><y>";/
        "<T><h><u><r><s><d><a><y>";/
        "<F><r><i><d><a><y>";/
        "<S><a><t><u><r><d><a><y>"

abmon   "<J><a><n>";/
        "<F><e><b>";/
        "<M><a><r>";/
        "<A><p><r>";/
        "<M><a><y>";/
        "<J><u><n>";/
        "<J><u><l>";/
        "<A><u><g>";/
        "<S><e><p>";/
        "<O><c><t>";/
        "<N><o><v>";/
        "<D><e><c>"

mon     "<J><a><n><u><a><r><y>";/
        "<F><e><b><r><u><a><r><y>";/
        "<M><a><r><c><h>";/
        "<A><p><r><i><l>";/
        "<M><a><y>";/
        "<J><u><n><e>";/
        "<J><u><l><y>";/
        "<A><u><g><u><s><t>";/
        "<S><e><p><t><e><m><b><e><r>";/
        "<O><c><t><o><b><e><r>";/
        "<N><o><v><e><m><b><e><r>";/
        "<D><e><c><e><m><b><e><r>"
```

```
d_t_fmt "%a %b %e %H:%M:%S %Z %Y"

d_fmt   "%m//%d//%y"

t_fmt   "%H:%M:%S"

am_pm   "<A><M>";"<P><M>"

END LC_TIME

%%%%%%%%%%%%%
LC_MESSAGES
%%%%%%%%%%%%%

yesexpr "<circumflex><left-parenthesis><left-square-bracket><y><Y>/
<right-square-bracket><left-square-bracket><e><E><right-square-bracket>/
<left-square-bracket><s><S><right-square-bracket><vertical-line>/
<left-square-bracket><y><Y><right-square-bracket><right-parenthesis>"
noexpr "<circumflex><left-parenthesis><left-square-bracket><n><N>/
<right-square-bracket><left-square-bracket><o><O><right-square-bracket>/
<vertical-line><left-square-bracket><n><N><right-square-bracket>/
<right-parenthesis>"

END LC_MESSAGES

%%%%%%%%%%%%%
LC_SYNTAX
%%%%%%%%%%%%%

backslash        "<backslash>"
right_brace      "<right-brace>"
left_brace       "<left-brace>"
right_bracket    "<right-square-bracket>"
left_bracket     "<left-square-bracket>"
circumflex       "<circumflex>"
tilde            "<tilde>"
exclamation_mark "<exclamation-mark>"
number_sign      "<number-sign>"
vertical_line    "<vertical-line>"
dollar_sign      "<dollar-sign>"
commercial_at    "<commercial-at>"
grave_accent     "<grave-accent>"

END LC_SYNTAX

%%%%%%%%%%%%%
LC_TOD
%%%%%%%%%%%%%

timezone_difference +480
timezone_name       "<P><S><T>"
daylight_name       "<P><D><T>"
start_month         0
end_month           0
start_week          0
end_week            0
start_day           0
end_day             0
start_time          0
end_time            0
shift               3600
END LC_TOD
```

Appendix G. Converting Code from Coded Character Set IBM-1047

The following program shows you how to convert hybrid code to a specified code page. Hybrid code is code in which the data is in the local coded character set but the syntax is written as if the code were in IBM-1047.

EDCXGHC1

```
 /*
  * EDCXGHC1: Sample code to convert all C syntax from code page 1047
  *           to the coded character set the user specifies.
  *           Comments, string literals and character constants are
  *           left alone. The escape character in an escape sequence
  *           is changed, since it is variant.
  *
  * Usage:    EDCXGHC1 <coded character set>
  *           The input file is read from stdin and the output is
  *           written to stdout.
  *
  * Example:  If you want to convert all C syntax, written in coded
  *           character set 1047, in a file (test1047.c) to coded
  *           character set 500, you can use EDCXGHC1 by invoking it
  *           with the following EXEC PARM:
  *
  *           PARM='<'test1047.c'' >''test1047.gen'' IBM-500'
  *
  *           The result will be stored in the "test1047.gen" file.
  */

#include <stdio.h>
#include <stdlib.h>
#include <iconv.h>
#include <errno.h>

enum boolean   { false=0, False=0, FALSE=0, true=1, True=1, TRUE=1 };

 /*
  * CharState - state that the FSM is in. Initial State is CodeState
  */
enum CharState { CodeState, SQuoteState, DQuoteState, CommentState,
                 DBCSState, EscState, EOFState };

 /*
  * CharVal - characters that can change the state of the FSM
  */
enum CharVal   { SlashChar='/',   SQuoteChar='\', DQuoteChar='"',
                 StarChar='*',    SOChar='\x0E',   SIChar='\x0F',
                 BSlashChar='\\', EOFChar= -1 };
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 1 of 10)

```
 /*
  * XlateTable - type of translation table
  */
typedef iconv_t  XlateTable;

static char *Initialize(int argc, char *argv[]);
static int Convert(char *codeset);
static int InitConv(char **inBuff, char **outBuff, int *maxRecSize,
                    char *codeSet, XlateTable *xlateTable);
static void ConvBuff(int start, int end,
                     char *buff, XlateTable xlateTable);
static enum CharVal LookAhead(char *inBuff, char *outBuff,
                              int *recSize, int *curPos,
                              int maxRecSize, int *codeStartPos,
                              enum CharState state,
                              XlateTable xlateTable);
static enum CharVal GetNextChar(char *inBuff, char *outBuff,
                                int *recSize, int maxRecSize,
                                int *curPos, int *codeStartPos,
                                enum CharState state,
                                XlateTable xlateTable);
static int UpdateAndRead(char *inBuff, char *outBuff,
                         int *recSize, int maxRecSize,
                         int codeStartPos, enum CharState state,
                         XlateTable xlateTable);
static int ReadAndCopy(char *inBuff,char *outBuff, int maxRecSize);

#pragma inline(LAST_POS)
#pragma inline(NEXT_TO_LAST_POS)
#pragma inline(LookAhead)
#pragma inline(GetNextChar)
#pragma inline(ConvBuff)

 /*
  * Initialize the environment, and if everything is ok, convert input
  */
main(int argc, char *argv[]) {
  char *codeset = Initialize(argc, argv);
  if (codeset == NULL) {
    return(8);
  }
  return(Convert(codeset));
}
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 2 of 10)

```
 /*
  * Check that 1 parameter was specified - the coded character set to
  * convert the the syntax to.
  * Re-open stdin and stdout as binary files for record IO.
  * Return the code set if everything is ok, NULL otherwise
  */
static char *Initialize(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Expected %d argument but got %d\n",
            1, argc-1);
    return(NULL);
  }
  stdin = freopen("", "rb,type=record", stdin);
  stdout= freopen("", "wb,type=record", stdout);
  if (stdin == NULL || stdout == NULL) {
    fprintf(stderr, "Could not re-open standard streams\n");
    return(NULL);
  }

  return(argv[1]);
}

 /*
  * Return the last position in a record
  */
static int LAST_POS(int recSize) {
  return(recSize-1);
}

 /*
  * Return the next to last position in a record
  */
static int NEXT_TO_LAST_POS(int recSize) {
  return(recSize-2);
}
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 3 of 10)

```
 /*
  * Convert the stdin file using codeset and write to stdout.
  * Set up the translation table.
  * Read the first record and copy it into the output buffer.
  * Go through the FSM, starting in the Code State and leaving
  * when EOFState is reached (End Of File).
  * Close the translation table.
  */
static int Convert(char *codeset) {
  enum CharVal   c;
  int            recSize;
  enum CharState prvState;
  int            rc;

  int            codeStartPos = 0;
  int            curPos  = 0;
  enum boolean   high    = FALSE;
  enum CharState state   = CodeState;

  char *         inBuff;
  char *         outBuff;
  int            maxRecSize;
  XlateTable     xlateTable;

  rc = InitConv(&inBuff, &outBuff, &maxRecSize, codeset, &xlateTable);
  if (rc) {
    if (inBuff)  free(inBuff);
    if (outBuff) free(outBuff);
    return(rc);
  }

  recSize = ReadAndCopy(inBuff, outBuff, maxRecSize);

  while (state != EOFState) {
    c = GetNextChar(inBuff, outBuff, &recSize, maxRecSize,
                    &curPos, &codeStartPos, state, xlateTable);
    if (c == EOFChar) {
      state = EOFState;
    }
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 4 of 10)

```
    switch(state) {
      case CodeState:
        switch (c) {
          case BSlashChar:
            curPos = LAST_POS(recSize);
            break;
          case SlashChar:
            if (LookAhead(inBuff, outBuff, &recSize,
                          &curPos, maxRecSize, &codeStartPos,
                          state, xlateTable)
                          == StarChar) {
              state = CommentState;
            }
            break;
          case SQuoteChar:
            state = SQuoteState;
            break;
          case DQuoteChar:
            state = DQuoteState;
            break;
        }
        if (state != CodeState || curPos == NEXT_TO_LAST_POS(recSize)) {
          if (curPos == NEXT_TO_LAST_POS(recSize)) {
            ++curPos;
          }
          else {
            ConvBuff(codeStartPos, curPos, outBuff, xlateTable);
          }
        }
        break;

      case CommentState:
        switch(c) {
          case BSlashChar:
            curPos = LAST_POS(recSize);
            break;
          case StarChar:
            if (LookAhead(inBuff, outBuff, &recSize,
                          &curPos, maxRecSize, &codeStartPos,
                          state, xlateTable)
                          == SlashChar) {
              state = CodeState;
              codeStartPos = curPos;
            }
            break;
        }
        break;
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 5 of 10)

```
    case DQuoteState:
      switch(c) {
        case DQuoteChar:
          state = CodeState;
          codeStartPos = curPos;
          break;
        case SOChar:
          prvState = state;
          state    = DBCSState;
          break;
        case BSlashChar:
          ConvBuff(curPos, curPos, outBuff, xlateTable);
          if (curPos != LAST_POS(recSize)) {
            prvState = state;
            state = EscState;
          }
          break;
      }
      break;

    case SQuoteState:
      switch(c) {
        case SQuoteChar:
          state = CodeState;
          codeStartPos = curPos;
          break;
        case SOChar:
          prvState = state;
          state    = DBCSState;
          break;
        case BSlashChar:
          ConvBuff(curPos, curPos, outBuff, xlateTable);
          if (curPos != LAST_POS(recSize)) {
            prvState = state;
            state = EscState;
          }
          break;
      }
      break;
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 6 of 10)

```
      case DBCSState:
        high ¬= 1;  /* TRUE -> FALSE or FALSE -> TRUE */
        if (high && (c == SIChar)) {
          state = prvState;
          high  = FALSE;
        }
        break;

      case EscState:
        state = prvState; /* really, this is ok */
        break;

      case EOFState:
        break;

      default:
        fprintf(stderr, "Internal error - ended up in state %d\n",
                state);
        return(16);

    } /* end of switch statement */
    ++curPos;
  }
  rc = TermConv(inBuff, outBuff, xlateTable);
  return(0);
}

/*
  * Initialize the translation table and allocate the input and
  * output buffers to use.
  * Return 0 if successful.
  */
static int InitConv(char **inBuff, char **outBuff, int *maxRecSize,
                    char *codeset, XlateTable* xlateTable) {

  static char fileNameBuff[FILENAME_MAX+1];
  fldata_t info;
  int rc;

  *outBuff = *inBuff = NULL;

  rc = fldata(stdin, fileNameBuff, &info);
  if (rc) {
    return(rc);
  }

  *maxRecSize = info.__maxreclen;
  *inBuff     = malloc(*maxRecSize);
  *outBuff    = malloc(*maxRecSize);

 if ((*xlateTable = iconv_open(codeset,"IBM-1047")) == (iconv_t)(-1)) {
    fprintf(stderr,"Cannot open convertor from IBM-1047 to %s",codeset);
    return (8);
  }

  return(!inBuff || !outBuff);
}
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 7 of 10)

```
 /*
  * Convert the buffer from start to end using the translation table
  */
static void ConvBuff(int start, int end,
                     char *buff, XlateTable xlateTable) {
   int rc;
   size_t inleft, outleft, org;
   char *inptr, *outptr;

   outleft = inleft = end-start+1;
   inptr = outptr = &buff[start];

   while (1) {
     rc = iconv(xlateTable,&inptr,&inleft,&outptr,&outleft);

     if (rc == -1) {
       switch (errno) {
                       /* Skip the invalid character */
       case EILSEQ:  if (--inleft == 0) return;
                      ++inptr;
                      ++outptr;
                      --outleft;
                      break;

       default: fprintf(stderr,"iconv() fails with errno = %d\n",errno);
                exit(8);
         }
     } else
       return;
   }
}

 /*
  * Look ahead to the next character. If the current position
  * is the last character of the input record, write the current
  * output record and read in the next record.
  * Return the 'character' read, which may be EOF if the end of
  * the file was reached.
  */
static enum CharVal LookAhead(char *inBuff, char *outBuff,
                              int *recSize, int *curPos,
                              int maxRecSize, int *codeStartPos,
                              enum CharState state,
                              XlateTable xlateTable) {

  if (*curPos == LAST_POS(*recSize)) {
    if (UpdateAndRead(inBuff, outBuff, recSize, maxRecSize,
                      *codeStartPos, state, xlateTable)) {
      return(EOFChar);
    }
    *curPos = 0;
    *codeStartPos = 0;
  }
  else {
    (*curPos)++;
  }
  return(inBuff[*curPos]);
}
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 8 of 10)

```
 /*
  * Similar to LookAhead(), but return the current character
  */
static enum CharVal GetNextChar(char *inBuff, char *outBuff,
                                int *recSize, int maxRecSize,
                                int *curPos, int *codeStartPos,
                                enum CharState state,
                                XlateTable xlateTable) {

  if (*curPos > LAST_POS(*recSize)) {
    if (UpdateAndRead(inBuff, outBuff, recSize, maxRecSize,
                      *codeStartPos, state, xlateTable)) {
      return(EOFChar);
    }
    *curPos = 0;
    *codeStartPos = 0;
  }
  return(inBuff[*curPos]);
}

/*
  * If the current state is the code state, translate the remaining
  * part of the record.
  * Write out the record to stdout
  * Read in the next record and copy it to the output buffer.
  */
static int UpdateAndRead(char *inBuff, char *outBuff,
                         int *recSize, int maxRecSize,
                         int codeStartPos, enum CharState state,
                         XlateTable xlateTable) {

  if (state == CodeState) {
    ConvBuff(codeStartPos, LAST_POS(*recSize), outBuff, xlateTable);
  }
  fwrite(outBuff, 1, *recSize, stdout);
  *recSize = ReadAndCopy(inBuff, outBuff, maxRecSize);
  return((*recSize == 0) ? 1 : 0);
}
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 9 of 10)

```
 /*
  * Read in a record from stdin and copy it to the output buffer.
  * Return the number of bytes read.
  */
static int ReadAndCopy(char *inBuff, char *outBuff,
    int maxRecSize) {
  int recSize;

  recSize = fread(inBuff, 1, maxRecSize, stdin);
  if (feof(stdin) && recSize == 0) {
    return(0);
  }
  else {
    memcpy(outBuff, inBuff, recSize);
    return(recSize);
  }
}

 /*
  * Free allocated storage and close the translation table.
  */
static int TermConv(char *inBuff,
                    char *outBuff, XlateTable xlateTable) {
  iconv_close(xlateTable);
  free(inBuff);
  free(outBuff);
  return(0);
}
```

Figure 93. Converting Hybrid Code to a Specific Character Set (Part 10 of 10)

Appendix H. Using Built-In Functions

The following is a list of all the functions which are built into the C/VSE compiler. The C/VSE compiler generates inline code for these functions at compile time.

Built-In Function	Header File
`abs()`	`stdlib.h`
`cds()`	`stdlib.h`
`cs()`	`stdlib.h`
`decabs()`	`decimal.h`
`decchk()`	`decimal.h`
`decfix()`	`decimal.h`
`fabs()`	`math.h`
`memchr()`	`string.h`
`memcpy()`	`string.h`
`memcmp()`	`string.h`
`memset()`	`string.h`
`strcat()`	`string.h`
`strchr()`	`string.h`
`strcmp()`	`string.h`
`strcpy()`	`string.h`
`strlen()`	`string.h`
`strrchr()`	`string.h`
`tsched()`	`mtf.h`

Note: Built-in functions are not associated with inline functions resulting from the use of the compile-time option `INLINE` and the `#pragma inline` directive. Refer to *IBM C for VSE/ESA User's Guide* for more information.

Appendix I. DSECT Conversion Utility

This chapter describes how to use the DSECT conversion utility.

The DSECT conversion utility generates a C structure to map an assembler DSECT. This utility is used when a C program calls or is called by an Assembler program and a C structure is required to map the area passed.

The source for the assembler DSECT is assembled using High Level Assembler specifying the ADATA option. (See *IBM High Level Assembler Programmer's Guide* , for a description of the ADATA option.) The DSECT utility then reads the SYSADAT file produced by High Level Assembler and produces a file containing the C structure according to the options specified.

The SYSADAT file must be RECFM=VB, BLKSIZE=8192, and LRECL=8188.

The file to which the C structure is written is specified using the OUTPUT option—the attributes are specified using the RECFM, LRECL, and BLKSIZE options.

DSECT Utility Options

The options that you can use to control the generation of the C structure are as follows. You can specify them in upper- or lowercase, separating them by spaces or commas.

Table 55. DSECT Utility Options, Abbreviations, and IBM-Supplied Defaults

DSECT Utility Option	Abbreviated Name	IBM Supplied Default
BITF0XL | NOBITF0XL	BITF | NOBITF	NOBITF0XL
BLKSIZE[(*blksize*)]	None	C Library defaults
COMMENT[(*delim,...*)] | NOCOMMENT	COM | NOCOM	COMMENT
DECIMAL | NODECIMAL	None	NODECIMAL
DEFSUB | NODEFSUB	DEF | NODEF	DEFSUB
EQUATE[(*suboptions,...*)] | NOEQUATE	EQU | NOEQU	NOEQUATE
HDRSKIP[(*length*)] | NOHDRSKIP	HDR | NOHDR	NOHDRSKIP
INDENT[(*count*)] | NOINDENT	IN | NOIN	INDENT(2)
LOCALE(*name*) | NOLOCALE	LOC | NOLOC	NOLOCALE
LOWERCASE | NOLOWERCASE	LC | NOLC	LOWERCASE
LRECL[(*lrecl*)]	None	C Library defaults
OPTFILE(*filename*) | NOOPTFILE	OPTF | NOOPTF	NOOPTFILE
OUTPUT[(*filename*)]	OUT	OUTPUT(DD:SYSPCH)
PPCOND[(*switch*)] | NOPPCOND	PP | NOPP	NOPPCOND
RECFM[(*recfm*)]	None	C Library defaults
SEQUENCE | NOSEQUENCE	SEQ | NOSEQ	NOSEQUENCE
SECT[(*name,...*)]	None	SECT(ALL)
UNIQUE | NOUNIQUE	None	NOUNIQUE
UNNAMED | NOUNNAMED	UNN | NOUNN	NOUNNAMED

Table 55. DSECT Utility Options, Abbreviations, and IBM-Supplied Defaults (continued)

DSECT Utility Option	Abbreviated Name	IBM Supplied Default
Note: [] surrounding a suboption indicates that the suboption is optional.		

BITF0XL | NOBITF0XL

DEFAULT: NOBITF0XL

Specify the BITF0XL option when the bit fields are mapped into a flag byte as in the following example:

```
FLAGFLD   DS   F
          ORG  FLAGFLD+0
B1FLG1    DC   0XL(B'10000000')'00'    Definition for bit 0 of 1st byte
B1FLG2    DC   0XL(B'01000000')'00'    Definition for bit 1 of 1st byte
B1FLG3    DC   0XL(B'00100000')'00'    Definition for bit 2 of 1st byte
B1FLG4    DC   0XL(B'00010000')'00'    Definition for bit 3 of 1st byte
B1FLG5    DC   0XL(B'00001000')'00'    Definition for bit 4 of 1st byte
B1FLG6    DC   0XL(B'00000100')'00'    Definition for bit 5 of 1st byte
B1FLG7    DC   0XL(B'00000010')'00'    Definition for bit 6 of 1st byte
B1FLG8    DC   0XL(B'00000001')'00'    Definition for bit 7 of 1st byte
          ORG  FLAGFLD+1
B2FLG1    DC   0XL(B'10000000')'00'    Definition for bit 0 of 2nd byte
B2FLG2    DC   0XL(B'01000000')'00'    Definition for bit 1 of 2nd byte
B2FLG3    DC   0XL(B'00100000')'00'    Definition for bit 2 of 2nd byte
B2FLG4    DC   0XL(B'00010000')'00'    Definition for bit 3 of 2nd byte
```

When the bit fields are mapped as shown in the above example, the bit fields can be tested using the following code:

```
TM   FLAGFLD,L'B1FLG          Test bit 0 of byte 1
Bx   label                    Branch if set/not set
```

When you specify the BITF0XL option, the length attribute of the following fields is used to provide the mapping for the bits within the flag bytes.

The length attribute of the following fields is used to map the bit fields if a field conforms to the following rules:
- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and does not have a bit length.
- Does not have more than 1 nominal value.

and the following fields conform to the following rules:
- Has a Type attribute of ’B’, ’C’ or ’X’.
- Has the same offset as the field (or consecutive fields have overlapping offsets).
- Has a duplication factor of zero.
- Does not have more than 1 nominal value.
- Has a length attribute between 1 and 255, and does not have a bit length.
- The length attribute maps one bit or consecutive bits. For example, B’10000000’ or B’11000000’, but not B’10100000’.

The fields must be on consecutive lines and must overlap a named field. If the fields above are used to define the bits for a field, any EQU statements following the field are not used to define the bit fields.

The following fields are used to define the bit fields as long as they map consecutive bits. If two consecutive fields are equivalent, the second field is skipped.

You can abbreviate the option to BITF or NOBITF.

BLKSIZE

DEFAULT: C Library default

The BLKSIZE option specifies the block size for the file to be produced. The block size specified must not be greater than 32767. This option is required if the file specified using the OUTPUT option is a SAM file or a member of a VSE/Librarian sublibrary. If it is not specified, the C library defaults will be used.

COMMENT | NOCOMMENT

DEFAULT: COMMENT

The COMMENT[(*delim*,...)] option specifies whether the comments on the line where the field is defined will be placed in the C structure produced.

If you specify the COMMENT option without a delimiter, the entire comment is placed in the C structure.

If you specify a delimiter, any comments following the delimiter are skipped and are not placed in the C structure. You can remove changes that are flagged with a particular delimiter. The delimiter cannot contain imbedded spaces or commas. The case of the delimiter and comment text is not significant. You can specify up to 10 delimiters, and they can contain up to 10 characters each.

You can abbreviate the option to COM or NOCOM.

DECIMAL | NODECIMAL

DEFAULT: NODECIMAL

The DECIMAL option will instruct the DSECT utility to convert all SYSADATA DC/DS records of type P to datatype decimal(w,0), where w is the number of digits. The parameter w is computed as follow:

1. Multiply the byte size of the P-type data by the value 2.
2. Subtract the value 1 from the result above.

This can be written as: (byte_size * 2) -1 . You can find the byte size of the P-type data in the SYSADATA DC/DS record.

The precision will always be "left as zero", since there is no way to calculate the value from the SYSADATA DC/DS record. The zero will be outputted rather than simply the digit size (that is, decimal(w,0) rather than decimal(w)). This allows you to easily edit the DSECT utility output, and adjust for the level of precision you require.

If the DECIMAL option is enabled and type P records are found, the utility will also include the following code at the beginning of the output file:

```
#ifndef __decimal_found
#define __decimal_found
#include <decimal.h>
#endif
```

DEFSUB | NODEFSUB

DEFAULT: `DEFSUB`

The `DEFSUB` option specifies whether `#define` directives will be built for fields that are part of a union or substructure.

If the `DEFSUB` option is in effect, fields within a substructure or union have the field names prefixed by an underscore. A `#define` directive is written at the end of the structure to allow the field name to be specified directly as in the following example.

```
_Packed struct dsect_name {
  int           field1;
  _Packed struct {
    int           _subfld1;
    short int     _subfld2;
    unsigned char _subfld3[4];
    } field2;
  }
#define subfld1  field2._subfld1
#define subfld2  field2._subfld2
#define subfld3  field2._subfld3
```

If the `DEFSUB` option is in effect, the fields prefixed by an underscore may match the name of another field within the structure. No warning is issued.

You can abbreviate the option to `DEF` or `NODEF`.

EQUATE | NOEQUATE

DEFAULT: `NOEQUATE`

The `EQUATE`[(*suboptions*,...)] option specifies whether the `EQU` statements following a field are to be used to define bit fields, to generate `#define` directives, or are to be ignored.

The suboptions specify how the `EQU` statement is used. You can specify one or more of the suboptions, separating them by spaces or commas. If you specify more than one suboption, the `EQU` statements following a field are checked to see if they are valid for the first suboption. If so, they are formatted according to that option. Otherwise, the subsequent suboptions are checked to see if they are applicable.

If you specify the `EQUATE` option without suboptions, `EQUATE(BIT)` is used. If you specify `NOEQUATE` (or select it by default), the `EQU` statements following a field is ignored.

You can specify the following suboptions for the `EQUATE` option:

`BIT` indicates that the value for an `EQU` statement is used to define the bits for a field where the field conforms to the following rules:
- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- Does not have more than 1 nominal value.

and the `EQU` statements following the field conform to the following rules:
- The value for the `EQU` statements following the field mask consecutive bits (for example, X’80’ followed by X’40’).

- The value for an EQU statement masks one bit or consecutive bits. For example, B'10000000' or B'11000000', but not B'10100000'.
- Where the length of the field is greater than 1 byte, the bits for the remaining bytes can be defined by providing the EQU statements for the second byte after the EQU statement for the first byte.
- The value for the EQU statement is not a relocatable value.

When you specify EQUATE(BIT), the EQU statements are converted as in the following example:

```
FLAGFLD  DS   H
FLAG21   EQU  X'80'
FLAG22   EQU  X'40'
FLAG23   EQU  X'20'
FLAG24   EQU  X'10'
FLAG25   EQU  X'08'
FLAG26   EQU  X'04'
FLAG27   EQU  X'02'
FLAG28   EQU  X'01'
FLAG2A   EQU  X'80'
FLAG2B   EQU  X'40'
```

```
_Packed struct dsect_name {
  unsigned int flag21   : 1,
               flag22   : 1,
               flag23   : 1,
               flag24   : 1,
               flag25   : 1,
               flag26   : 1,
               flag27   : 1,
               flag28   : 1,
               flag2a   : 1,
               flag2b   : 1,
                        : 6;
  };
```

BITL indicates that the length attribute for an EQU statement is used to define the bits for a field where the field conforms to the following rules:
- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- Does not have more than 1 nominal value.

and the EQU statements following the field conform to the following rules:
- The value specified for the EQU statement has the same or overlapping offset as the field.
- The length attribute for the EQU statement is between 1 and 255.
- The length attribute for the EQU statement masks one bit or consecutive bits. For example, B'10000000' or B'11000000', but not B'10100000'.
- The value for the EQU statement is a relocatable value.

When you specify EQUATE(BITL), the EQU statements are converted as in the following example:

```
BYTEFLD   DS   F
B1FLG1    EQU  BYTEFLD+0,B'10000000'
B1FLG2    EQU  BYTEFLD+0,B'01000000'
B1FLG3    EQU  BYTEFLD+0,B'00100000'
B1FLG4    EQU  BYTEFLD+0,B'00010000'
B1FLG5    EQU  BYTEFLD+0,B'00001000'
B1FLG6    EQU  BYTEFLD+0,B'00000100'
B1FLG7    EQU  BYTEFLD+0,B'00000010'
B1FLG8    EQU  BYTEFLD+0,B'00000001'
```

```
B2FLG1    EQU  BYTEFLD+1,B'10000000'
B2FLG2    EQU  BYTEFLD+1,B'01000000'
B2FLG3    EQU  BYTEFLD+1,B'00100000'
B2FLG4    EQU  BYTEFLD+1,B'00010000'
```

```
_Packed struct dsect_name {
  unsigned int b1flg1  : 1,
               b1flg2  : 1,
               b1flg3  : 1,
               b1flg4  : 1,
               b1flg5  : 1,
               b1flg6  : 1,
               b1flg7  : 1,
               b1flg8  : 1,
               b2flg1  : 1,
               b2flg2  : 1,
               b2flg3  : 1,
               b2flg4  : 1,
                       : 20;
  };
```

DEF indicates that the EQU statements following a field are used to build #define directives to define the possible values for a field. The #define directives are placed after the end of the C structure. The EQU statements should not specify a relocatable value.

When you specify EQUATE(DEF), the EQU statements are converted as in the following example:

```
FLAGBYTE DS   X
FLAG1    EQU  X'80'
FLAG2    EQU  X'20'
FLAG3    EQU  X'10'
FLAG4    EQU  X'08'
FLAG5    EQU  X'06'
FLAG6    EQU  X'01'
```

```
_Packed struct dsect_name {
  unsigned char flagbyte;
  }
 /* Values for flagbyte field */
#define flag1 0x80
#define flag2 0x20
#define flag3 0x10
#define flag4 0x08
#define flag5 0x06
#define flag6 0x01
```

You can abbreviate the option to EQU or NOEQU.

HDRSKIP | NOHDRSKIP

DEFAULT: NOHDRSKIP

The HDRSKIP[(*length*)] option specifies that the fields within the specified number of bytes from the start of the section are to be skipped. Use this option where a section has a header that is not required in the C structure produced.

The value specified on the HDRSKIP option indicates the number of bytes at the start of the section that are to be skipped. HDRSKIP(0) is equivalent to NOHDRSKIP.

In the following example, if you specify HDRSKIP(8), the first two fields are skipped and only the remaining two fields are built into the structure.

```
SECTNAME DSECT
PREFIX1  DS   CL4
PREFIX2  DS   CL4
FIELD1   DS   CL4
FIELD2   DS   CL4
```

```
_Packed struct sectname {
  unsigned char field1[4];
  unsigned char field2[4];
  }
```

If the value specified for the HDRSKIP option is greater than the length of the section, the C structure is not be produced for that section.

You can abbreviate the option to HDR or NOHDR.

INDENT | NOINDENT

DEFAULT: INDENT(2)

The INDENT[(*count*)] option specifies the number of character positions that the fields, unions, and substructures are indented. Turn off indentation by specifying INDENT(0) or NOINDENT. The maximum value that you can specify for the INDENT option is 32767.

You can abbreviate the option to IN or NOIN.

LOCALE | NOLOCALE

DEFAULT: `NOLOCALE`

The `LOCALE`(*name*) option specifies the name of a locale to be passed to the setlocale() function. Specifying `LOCALE` without the *name* parameter is equivalent to passing the NULL string to the setlocale() function.

The structure produced contains the left and right brace, and left and right square bracket, backslash, and number sign which have different code point values for the different code pages. When the `LOCALE` option is specified, and these characters are written to the output file, the code point from the LC_SYNTAX category for the specified locale is used.

You can abbreviate the option to `LOC` or `NOLOC`.

LOWERCASE | NOLOWERCASE

DEFAULT: `LOWERCASE`

The `LOWERCASE` option specifies whether the field names within the C structure are to be converted to lowercase or left as entered. If you specify `LOWERCASE`, all the field names are converted to lowercase. If you specify `NOLOWERCASE`, the field names are built into the structure in the case in which they were entered in the assembler section.

You can abbreviate the option to `LC` or `NOLC`.

LRECL

DEFAULT: C Library default

The `LRECL` option specifies the logical record length for the file to be produced. The logical record length specified must not be greater than 32767. This option is required if the file specified using the `OUTPUT` option is a SAM file or a member of a VSE/Librarian sublibrary. If it is not specified, the C library defaults will be used.

OPTFILE | NOOPTFILE

DEFAULT: NOOPTFILE

The OPTFILE(*filename*) option specifies the filename containing the records that specify the options to be used for processing the sections. The filename can be any of the file specification formats supported by the fopen() function. (See the "Opening Files" sections in this book for additional information regarding file specification formats.) The filename will be passed to fopen() as entered. The records must be as follows:

- The lines must begin with the SECT option, with only one section name specified. The options following determine how the structure is produced for the specified section. The section name must only be specified once.
- The lines may contain the options BITF0XL, COMMENT, DEFSUB, EQUATE, HDRSKIP, INDENT, LOWERCASE, PPCOND, and UNNAMED, separated by spaces or commas. These override the options specified on the command line for the section.

The OPTFILE option is ignored if the SECT option is also specified on the command line.

You can abbreviate the option to OPTF or NOOPTF.

OUTPUT

DEFAULT: OUTPUT(DD:SYSPCH)

The C structures produced are, by default, written to SYSPCH. You can use the OUTPUT option to specify a different filename for writing the C structures using any of the file specification formats supported by the fopen() function. (See the "Opening Files" sections in this book for additional information regarding file specification formats.) The filename will be passed to fopen() as entered.

You can abbreviate the option to OUT.

PPCOND | NOPPCOND

DEFAULT: NOPPCOND

The PPCOND[(*switch*)] option specifies whether preprocessor directives will be built around the structure definition to prevent duplicate definitions.

If you specify PPCOND, the following are built around the structure definition.

```
#ifndef switch
#define switch
   ⋮
  structure definition for section
   ⋮
#endif
```

where *switch* is the switch specified on the PPCOND option or the section name prefixed and suffixed by two underscores, for example, _ _name_ _.

If you specify a switch, the #ifndef and #endif directives are placed around all structures that are produced. If you do not specify a switch, the #ifndef and #endif directives are placed around each structure produced.

You can abbreviate the option to PP or NOPP.

SECT

DEFAULT: SECT(ALL)

The SECT option specifies the section names for which C structures are to produced. The section names can be either CSECT or DSECT names. They must exist in the SYSADAT file produced by the Assembler. If you do not specify the SECT option or if you specify SECT(ALL), C structures are produced for all CSECTs and DSECTs defined in the SYSADAT file, except for private code and unnamed DSECTs.

If High Level Assembler is run with the BATCH option, only the section names defined within the first program can be specified on the SECT option. If you specify SECT(ALL) (or select it by default), only the sections from the first program are selected.

SEQUENCE | NOSEQUENCE

DEFAULT: NOSEQUENCE

The SEQUENCE option specifies whether sequence numbers will be placed in columns 73 to 80 of the output record. If you specify the SEQUENCE option, the C structure is built into columns 1 to 72 of the output record and sequence numbers are placed in columns 73 to 80. If you specify NOSEQUENCE (or select it by default), sequence numbers are not generated and the C structure is built within all available columns in the output record.

If the record length for the output file is less than 80 characters, the SEQUENCE option is ignored.

You can abbreviate the option to SEQ or NOSEQ.

UNIQUE | NOUNIQUE

DEFAULT: NOUNIQUE

The UNIQUE option tells the DSECT utility to use a unique string as though it does not occur in any of the field names contained in the SYSADATA input. This user wishes to guarantee that if the DSECT utility were to use the unique string to map national characters, no conflict would occur with any other field name. Given this guarantee, the DSECT utility will map the national characters as follows:

```
# = unique string + 'n' + unique string
@ = unique string + 'a' + unique string
$ = unique string + 'd' + unique string
```

For example, if the default "_" unique string was used, the national characters would be mapped as:

```
# = _n_
@ = _a_
$ = _d_
```

If the default NOUNIQUE option is enabled, the DSECT utility converts all national characters to a single underscore *even if* there is a conflict between the resulting label names.

UNNAMED | NOUNNAMED

DEFAULT: NOUNNAMED

The UNNAMED option specifies that names are not generated for the unions and substructures within the main structure.

You can abbreviate the option to UNN or NOUNN.

RECFM

DEFAULT: C Library default

The RECFM option specifies the record format for the file to be produced. You can specify up to 10 characters. This option is required if the file specified using the OUTPUT option is a SAM file or a member of a VSE/Librarian sublibrary. If it is not specified, the C library defaults are used.

Generation of C Structures

The C structure is produced as follows according to the options in effect.

- The section name is used as the structure name. The structure is generated with the _Packed attribute to ensure it matches the assembler section.

  Whenever you specify the structure name, you should also specify the _Packed attribute.
- Any nonalphanumeric characters in the section or field names are converted to underscores. Duplicate names may be generated when the field names are identical except for the national character. No warning is issued.
- Where fields overlap, a substructure or union is built within the main structure. A substructure is produced where possible. When substructures and unions are built, the structure and unions names are generated by the DSECT utility.
- The substructures and unions within the main structure are indented according to the INDENT option unless the record length is too small to permit any further indentation.
- Fillers are added within the structure when required. A filler name is generated by the DSECT utility.
- Where there is no direct equivalent for an assembler definition within the C language, the field is defined as a character field.
- If a field has a duplication factor of zero, but cannot be used as a structure name, the field is defined as though the duplication factor of zero was eliminated.
- Where a line within the assembler input consists of an operand with a duplication factor of zero (for alignment), followed by the field definition, the first operand is skipped. For example:

  ```
  FIELDA    DS  OF,CLB
  ```

  is treated as though the following was specified.

  ```
  FIELDA    DS  CLB
  ```
- When the COMMENT option is in effect, the comment on the line following the definition of the field is placed in the C structure. The comment is placed on the same line as the field definition where possible, or on the following line.

  /* is removed from the beginning of comments and */ is removed from the end of comments. Any remaining instances of */ in the comment are converted to **.

Each field within the section is converted to a field within the C structure as shown in the following examples:

- Bit length fields

  If the field has a bit length that is not a multiple of 8, it is converted as follows. Otherwise, it is converted according to the field type.

DS CL.n	`unsigned int   name : n;` where n is from 1 to 31.
DS CL.n	`unsigned char  name[x];` where n is greater than 32. x will be the number of bytes required (that is, the bit length / 8 + 1).
DS 5CL.n	`unsigned char  name[x];` where x will be the number of bytes required (that is, the duplication factor * bit length / 8 + 1).

- Characters

DS C	`unsigned char name;`
DS CL2	`unsigned char name[2];`
DS 4CL2	`unsigned char name[4][2];`

- Graphic Characters

DS G	`wchar_t       name;`
DS GL1	`unsigned char name;`
DS GL2	`wchar_t       name;`
DS GL3	`unsigned char name[3];`
DS 4GL1	`unsigned char name[4];`
DS 4GL2	`wchar_t       name[4];`
DS 4GL3	`unsigned char name[4][3];`

- Hexadecimal Characters

DS X	`unsigned char name;`
DS XL2	`unsigned char name[2];`
DS 4XL2	`unsigned char name[4][2];`

- Binary fields

DS B	`unsigned char  name;`
DS BL2	`unsigned char  name[2];`
DS 4BL2	`unsigned char  name[4][2];`

- Half and Fullword Fixed-point

DS F	`int            name;`
DS H	`short int      name;`
DS FL1 or HL1	`char           name;`
DS FL2 or HL2	`short int      name;`
DS FL3 or HL3	`int            name : 24;`
DS FLn or HLn	`unsigned char  name[n];` where n is greater than 4.
DS 4F	`int            name[4];`
DS 4H	`short int      name[4];`
DS 4FL1 or 4HL1	`char           name[4];`
DS 4FL2 or 4HL2	`short int      name[4];`

DS 4FL3 or 4HL3	`unsigned char  name[4][3];`
DS 4FLn or 4HLn	`unsigned char  name[4][n];`

where n is greater than 4.

- Floating Point

DS E	`float          name;`
DS D	`double         name;`
DS L	`long double    name;`
DS 4E	`float          name[4];`
DS 4D	`double         name[4];`
DS 4L	`long double    name[4];`
DS EL4 or DL4 or LL4	`float          name;`
DS EL8 or DL8 or LL8	`double         name;`
DS LL16	`long double    name;`
DS E, D or L	`unsigned char  name[n];`

where n is other than 4, 8 or 16.

- Packed Decimal

DS P	`unsigned char  name;`
DS PL2	`unsigned char  name[2];`
DS 4PL2	`unsigned char  name[4][2];`

- Zoned Decimal

DS Z	`unsigned char  name;`
DS ZL2	`unsigned char  name[2];`
DS 4ZL2	`unsigned char  name[4][2];`

- Address

DS A	`void          *name;`
DS AL1	`unsigned char  name;`
DS AL2	`unsigned short name;`
DS AL3	`unsigned int   name : 24;`
DS 4A	`void          *name[4];`
DS 4AL1	`unsigned char  name[4];`
DS 4AL2	`unsigned short name[4];`
DS 4AL3	`unsigned char  name[4][3];`

- Y-type Address

DS Y	`unsigned short name;`
DS YL1	`unsigned char  name;`
DS 4Y	`unsigned short name[4];`
DS 4YL1	`unsigned char  name[4];`

- S-type Address (Base and displacement)

DS S	`unsigned short name;`
DS SL1	`unsigned char  name;`
DS 4S	`unsigned short name[4];`
DS 4SL1	`unsigned char  name[4];`

- External Symbol Address

DS V	`void          *name;`
DS VL3	`unsigned int   name : 24;`
DS 4V	`void          *name[4];`
DS 4VL3	`unsigned char  name[4][3];`

- External Dummy Section Offset

DS Q	`unsigned int   name;`

DS QL1	`unsigned char  name;`
DS QL2	`unsigned short name;`
DS QL3	`unsigned int   name : 24;`
DS 4Q	`unsigned int   name[4];`
DS 4QL1	`unsigned char  name[4];`
DS 4QL2	`unsigned short name[4];`
DS 4QL3	`unsigned char  name[4][3];`

- Channel Command Words

  When a CCW, CCW0, or CCW1 assembler instruction is present within the section, a typedef ccw0_t or ccw1_t is defined to map the format of the CCW.

  The CCW, CCW0 or CCW1 is built into the C structure as follows:

CCW cc,addr,flags,count	`ccw0_t`	`name;`
CCW0 cc,addr,flags,count	`ccw0_t`	`name;`
CCW1 cc,addr,flags,count	`ccw1_t`	`name;`

Under VSE Batch

Under VSE, you can execute the DSECT utility as shown in the following example:

```
// JOB DSECTXMP
// LIBDEF *,SEARCH=(PRD2.SCEEBASE,hlasm.lib)
// DLBL SYSADAT,'file.name',0,VSAM,RECORDS=100,RECSIZE=80,DISP=OLD
// OPTION NOLINK,NODUMP
// EXEC ASMA90,SIZE=512K,PARM='ADATA'

   ⋮
  Assembler source code

   ⋮
 /*
// EXEC EDCDSECT,PARM='OUT(DD:SYSLST)'
 /*
// EXEC LISTLOG
/&
```

Figure 94. Running the DSECT Utility under VSE Batch

In the above example, High Level Assembler is invoked to assemble the source provided with the ADATA option. The DSECT utility is then executed to produce the C structure. The C structure is written to the file specified by the OUTPUT option. A report is written to SYSLST indicating the options in effect and any error messages.

Glossary

This glossary defines terms and abbreviations that are used in this book. Included are terms and definitions from the following sources:

- *American National Standard Dictionary for Information Systems* , ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *ANSI/ISO* after the definition.
- *IBM Dictionary of Computing* , SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.
- *X/Open CAE Specification. Commands and Utilities, Issue 4. July, 1992* . These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990* These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

abend. Abnormal end of task. Synonym for abnormal termination.

abstract code unit. See *ACU*.

access mode. The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*.

ACU (abstract code unit). A measurement used by C compilers for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

addressing mode. See *AMODE*.

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM*.

American National Standards Institute. See *ANSI/ISO*.

AMODE (addressing mode). In VSE, a program attribute that refers to the address length that a program is prepared to handle upon entry. In VSE, addresses may be 24 or 31 bits in length. *IBM*.

ANSI/ISO (American National Standards Institute). 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. *ANSI/ISO*.

API (application program interface). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM*.

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM*. (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM*.

application program. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM*.

argument. (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*.

array. In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. *IBM*.

array element. A data item in an array. *IBM*.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM*.

Note: IBM has defined an extension to ASCII code (characters 128-255).

assembler language. A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM*.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

backslash. The character \. This character is named <backslash> in the portable character set.

batch processing. (1) Serial processing of computer programs. (2) Pertaining to the technique of processing a set of computer programs in such a way that each is completed before the next program of the set is started.

batch program. A program that is processed in series with other programs and therefore normally processes data without user interaction.

binary stream. (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM*.

blank character. (1) A graphic representation of the space character. *ANSI/ISO*. (2) A character that represents an empty position in a graphic character string. *ISO Draft*. (3) One of the characters that belong to the *blank* character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character. *X/Open*.

block. (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

blocking. The process of combining (or cutting) records into blocks.

boundary alignment. The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM*.

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase “enclosed in (square) brackets” the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

built-in. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with predefined. *IBM*.

C

call. To transfer control to a procedure, program, routine, or subroutine. *IBM*.

callable services. A set of services that can be invoked by a Language Environment-conforming high level language using the conventional Language Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

Use of these services helps to decrease an application’s dependence on the specific form and content of the services delivered by any single operating system.

call chain. A trace of all active routines and subroutines.

caller. A routine that calls another routine.

cast. In the C language, an expression that converts the type of the operand to a specified data type (the operator). *IBM*.

catalog. (1) A directory of files and libraries, with reference to their locations. A catalog may contain other information such as the types of devices in which the files are stored, passwords, blocking factors. (2) To store a library member such as a phase, module, or book in a sublibrary.

chained sublibraries. A facility that allows sublibraries to be chained by specifying the sequence in which they must be searched for a certain library member.

chaining. A logical connection of sublibraries to be searched by the system for members of the same type (phases or object modules, for example).

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI/ISO*. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multibyte character), where a single-byte character is a special case of the multibyte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open*. *ISO.1*.

character array. An array of type char. *X/Open*.

character class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open*.

character constant. (1) A constant with a character value. *IBM*. (2) A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM*. (3) In some languages, a character enclosed in apostrophes. *IBM*.

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft*. (2) All the valid characters for a programming language or for a computer system. *IBM*. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM*. (4) See also *portable character set*.

character special file. (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM*. (2) A file that refers to a device. *X/Open*. *ISO.1*.

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open*.

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

CICS (Customer Information Control System). Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM*.

CICS system definition (CSD) file. See CSD.

CKD device. Count-key-data device.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

COBOL (Common Business-Oriented Language). A high-level language, based on English, that is primarily used for business applications.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*. (3) Loosely, a code. *ANSI/ISO*.

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

code point. (1) A 1-byte code representing one of 256 potential characters. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

codeset. Synonym for code element set. *IBM*.

collating element. The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open*.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI/ISO*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO*. (3) The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting or multiple collating elements. *X/Open*.

collection. (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

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.

COMMAREA. A communication area made available to applications running under CICS.

common anchor area (CAA). Dynamically acquired storage that represents a LE/VSE thread. Thread-related storage/resources are anchored off of the CAA. This area acts as a central communications area for the program, holding addresses of various storage and error-handling routines, and control blocks. The CAA is anchored by an address in register 12.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

compile. To translate a source program into an executable program (object program). See also *assembler*.

compiler. A program used to compile.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM*. (2) An exception that has been enabled, or recognized, by the Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

constant. (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM*.

control character. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft*. (2) Synonymous with nonprinting character. *IBM*. (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open*.

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1*. (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) 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.

coordinated universal time (UTC). Equivalent to Greenwich Mean Time (GMT)

count-key-data (CKD) device. A disk device that stores data in the record format: count field, key field, data field. The count field contains, among others, the address of the record in the format: cylinder, head (track), record number and the length of the data field. The key field, if present, contains the record's key or search argument. CKD disk space is allocated by tracks and cylinders. Contrast with *FBA disk device*.

Cross System Product. See *CSP*.

CSD (CICS system definition) file. A component of CICS resource definition online (RDO). It keeps a permanent record of resource information, independently of the active CICS system. The information held in the CSD is used for installing new resources and at a CICS restart.

CSP (Cross System Product). A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development (CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM*.

Customer Information Control System. See *CICS*.

D

DATABASE 2 (DB2). An IBM relational database management system.

data object. (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM*.

data set. Under VSAM, a named collection of related data records that is stored and retrieved by an assigned name.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM*.

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

DBCS (double-byte character set). 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, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

DB2. See DATABASE 2.

DCT. Destination control table.

declaration. (1) In the C language, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

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 locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

degree. The number of children of a node.

demangling. The conversion of mangled names back to their original source code names. During C compilation, identifiers such as functions are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

device. A computer peripheral or an object that appears to the application as such. *X/Open*. *ISO.1*.

difference. Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if *m*>*n*, the difference contains that element *m-n* times. If *m*≤*n*, the difference contains that element zero times.

directory. A type of file containing the names and controlling information for other files or other directories. *IBM*.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open*.

DLBL. Disk Label Information job control statement.

double-byte character set. See *DBCS*.

DTF (Define the File). Generalized term used for various VSE Define the File macros. For example, DTFCD, DTFSD and DTFPR.

dump. To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM*.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM*.

dynamic storage. Synonym for *automatic storage*.

E

EBCDIC (extended binary-coded decimal interchange code). A coded character set of 256 8-bit characters. *IBM*.

E-format. Floating-point format, consisting of a number in scientific notation. *IBM*.

element. The component of an array, subrange, enumeration, or set.

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

enclave. In LE/VSE, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

entry point. In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

exception. (1) 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 (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1*.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

extension. (1) An element or function not included in the standard language. (2) File name extension.

F

FBA disk device. Fixed-block architecture disk device.

feature test macro. A macro (`#define`) used to determine whether a particular set of features will be included from a header. *X/Open*. *ISO.1*.

file. A named set of records stored or processed as a unit. Synonymous with *data set*.

file descriptor. (1) A small positive integer that the system uses instead of the file name to identify an open file. *IBM*. (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1*.

The value of a file descriptor is from zero to {OPEN_MAX}—which is defined in limits.h. A process can have no more than {OPEN_MAX} file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open*.

file-ID. The unique name associated with a file on a volume.

file mode. The mode in which a file is opened (read, write, update, etc.).

file offset. The byte position in the file where the next I/O operation begins. Each open file description associated with a regular file, block special file, or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a FIFO. *X/Open*. *ISO.1*.

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.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

fixed-block architecture (FBA) disk device. A disk device that stores data in blocks of fixed size. These blocks are addressed by block number relative to the beginning of the file. Contrast with *CKD device*.

flat collection. A collection that has no hierarchical structure.

for statement. A looping statement that contains the word *for* followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM*.

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 values. *IBM*.

G

global. Pertaining to information available to more than one program or subroutine. *IBM*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

glyph. (1) An image, usually of a character, in a font. (2) A graphic symbol whose appearance conveys information; for example, the vertical and horizontal arrows on cursor keys that indicate the directions in which they control cursor movement, the sunburst symbol on the screen illumination control of a display device.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated.

Greenwich Mean Time. See GMT.

H

header file. A text file that contains declarations used by a group of functions, programs, or users.

High Level Assembler. An IBM licensed program. Translates symbolic assembler language into binary machine language.

I

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

if statement. A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM*.

ILC (interlanguage call). A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

ILC (interlanguage communication). The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. See *header file*.

initializer. An expression used to initialize data objects. In the C language, there are two types of initializers:
1. An expression followed by an assignment operator is used to initialize fundamental data type objects
2. An expression enclosed in braces ({ }) is used to initialize aggregates.

input stream. A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM*.

inlined function. A function whose actual code replaces a function call. The function must be declared inline using the `#pragma inline` directive.

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.

interlanguage call. See *ILC*. (1)

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

I/O. Input/output.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

J

JCL (job control language). A control language used to identify a job to an operating system and to describe the job's requirements. *IBM*.

job control. A facility that allows users to selectively stop (suspend) the execution of a process and continue (resume) their execution at a later point. *X/Open*. *ISO.1*.

K

key function. (1) When used on a flat collection, a function that returns a reference to the key of an element. (2) In general, a function, called by a member function, that manipulates the keys of a class.

key set. An unordered flat collection that uses keys and does not allow duplicate elements.

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

label. (1) An identifier within or attached to a set of data elements. *ISO Draft*. (2) An identification record for a tape, disk, or diskette volume or for a file on such a volume.

LE/VSE. Abbreviated form of IBM Language Environment for VSE/ESA. Pertaining to an IBM software product that provides a common run-time environment and run-time services to applications compiled by LE/VSE-conforming compilers.

librarian. The set of programs that maintains, services, and organizes the system and private libraries.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

library member. The smallest unit of data to be stored in and retrieved from a sublibrary.

line. A sequence of zero or more non-newline characters plus a terminating newline character. *X/Open*.

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 computer program for creating phases from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM*.

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, "APRIL" represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1*. (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM*. (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM*.

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1*. (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI/ISO*.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open*.

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.

macro instruction. Synonym for *macro*.

main function. An external function with the identifier `main` that is the first user function, aside from exit routines, 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 prelinker uses these mangled names to ensure type-safe linkage. See also *demangling*.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. *ISO-JTC1*. *ANSI/ISO*.

member. (1) A data object or function in a structure or union. Members can also be enumerations, bit fields, and type names. (2) The smallest unit of data to be stored in and retrieved from a sublibrary.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM*.

mode. A collection of attributes that specifies a file's type and its access permissions. *X/Open*. *ISO.1*.

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.

N

name. In the C language, a name is commonly referred to as an identifier. However, syntactically, a name can be an identifier, operator function name, conversion function name, or qualified name.

node. In a tree structure, a point at which subordinate items of data originate. *ANSI/ISO*.

NULL. In the C language, a pointer that does not point to a data object. *IBM*.

null character (NUL). The ASCII or EBCDIC character '\0' with the hex value `00`, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, `(void *) 0`.

The C language guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open*.

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

null value. A parameter position for which no value is specified. *IBM*.

null wide-character code. A wide-character code with all bits set to zero. *X/Open*.

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

O

object. A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance*.)

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

open file. A file that is currently associated with a file descriptor. *X/Open*. *ISO.1*.

open file description. A record of how a process or a group of processes are accessing a file. Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor. A file offset, file status, and file access modes are attributes of an open file description. *X/Open*. *ISO.1*.

operand. An entity on which an operation is performed. *ISO-JTC1*. *ANSI/ISO*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1*.

orientation of a stream. After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM*.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

parameter. (1) In the C language, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes will be treated as a single slash. The interpretation of the path name is described in *pathname resolution*. *ISO.1*. (2) A file name specifying all directories leading to the file.

path name resolution. Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. *X/Open*.

path prefix. A path name, with an optional ending slash, that refers to a directory. *ISO.1*.

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

period. The character (.). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

phase. All or part of a computer program in a form suitable for loading into main storage for execution. A phase is usually the output of a linkage editor.

pointer. In the C language, a variable that holds the address of a data object or a function. *IBM*.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

<NUL>	
<alert>	
<backspace>	
<tab>	
<newline>	
<vertical-tab>	
<form-feed>	
<carriage-return>	
<space>	
<exclamation-mark>	!
<quotation-mark>	"
<number-sign>	#
<dollar-sign>	$
<percent-sign>	%
<ampersand>	&
<apostrophe>	'
<left-parenthesis>	(
<right-parenthesis>	)
<asterisk>	*
<plus-sign>	+
<comma>	,
<hyphen>	–
<hyphen-minus>	–
<period>	.
<slash>	/
<zero>	0
<one>	1
<two>	2
<three>	3
<four>	4
<five>	5
<six>	6
<seven>	7
<eight>	8
<nine>	9
<colon>	:
<semicolon>	;
<less-than-sign>	<
<equals-sign>	=
<greater-than-sign>	>
<question-mark>	?
<commercial-at>	@
<A>	A
<B>	B
<C>	C
<D>	D
<E>	E
<F>	F
<G>	G
<H>	H
<I>	I
<J>	J
<K>	K
<L>	L
<M>	M
<N>	N
<O>	O
<P>	P
<Q>	Q
<R>	R
<S>	S
<T>	T
<U>	U
<V>	V
<W>	W
<X>	X
<Y>	Y
<Z>	Z
<left-square-bracket>	[
<backslash>	\
<reverse-solidus>	\
<right-square-bracket>	]
<circumflex>	^
<circumflex-accent>	^
<underscore>	_
<low-line>	_
<grave-accent>	`
<a>	a
<b>	b
<c>	c
<d>	d
<e>	e
<f>	f
<g>	g
<h>	h
<i>	i
<j>	j
<k>	k
<l>	l
<m>	m
<n>	n
<o>	o
<p>	p
<q>	q
<r>	r
<s>	s
<t>	t
<u>	u
<v>	v
<w>	w
<x>	x
<y>	y
<z>	z
<left-brace>	{
<left-curly-bracket>	{
<vertical-line>	|
<right-brace>	}
<right-curly-bracket>	}
<tilde>	~

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

positional parameter. A parameter that must appear in a specified location relative to other positional parameters. *IBM*.

precedence. The priority system for grouping different types of operators with their operands.

predefined macros. Frequently used routines provided by an application or language for the programmer.

preinitialization. A process by which an environment or library is initialized and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker. A utility provided with LE/VSE that can be used for application programs that are reentrant, or have external symbol names that are longer than what the linkage editor supports. The prelinker is invoked before the linkage editor.

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.

printable character. One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open*.

programmer logical unit. A logical unit available primarily for user-written programs.

Q

Query Management Facility (QMF). Pertaining to a query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis.

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

R

radix character. The character that separates the integer part of a number from the fractional part. *X/Open*.

reason code. A code that identifies the reason for a detected error. *IBM*.

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

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.

refresh. To ensure that the information on the user's terminal screen is up-to-date. *X/Open*.

regular expression. (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open*. *ISO.1*.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution*. *IBM*.

root. A node that has no parent. All other nodes of a tree are descendants of the root.

run-time library. A compiled collection of functions whose members can be referred to by an application program during run-time execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The run-time library itself is not statically bound into the application modules.

S

SAM. Sequential access method.

SAM ESDS file. A SAM file managed in VSE/VSAM space, so it can be accessed by both SAM and VSE/VSAM macros.

scope. (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

sequence. A sequentially ordered flat collection.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM*.

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership. There can be multiple process groups in the same session. *X/Open*. *ISO.1*.

shared virtual area (SVA). In VSE, a high address area that contains a system directory list (SDL) of frequently used phases, resident programs shared between partitions, and an area for system support.

signal. (1) 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. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open*. *ISO.1*.

signal handler. A function to be called when the signal is reported.

slash. The character `/`, also known as *solidus*. This character is named <slash> in the portable character set.

S-name. An external name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

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 high-level language programs and data description specifications. *IBM*.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM*.

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open*.

spanned record. A logical record contained in more than one block. *IBM*.

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). A language designed to create, access, update and free data tables.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open*.

standard input. (1) An input stream usually intended to be used for primary data input. *X/Open*. (2) The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM*.

standard output. An output stream usually intended to be used for primary data output. *X/Open*.

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.

stream. See *data stream*.

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

string constant. Zero or more characters enclosed in double quotation marks.

struct. An aggregate of elements, having arbitrary types.

structure. A construct (a class data type) 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.

stub routine. Within run-time libraries, contains the minimum lines of code required to locate a given routine at run time.

sublibrary. In VSE, a subdivision of a library. Members can only be accessed in a sublibrary.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM*.

SVA. See *shared virtual area*.

system logical unit. A logical unit available primarily for operating system use.

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*.

This character is named <tab> in the portable character set.

task. (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1*. *ANSI/ISO*. (2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. A family of classes or functions with variable types.

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}—which is defined in limits.h—bytes in length, including the newline character. The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). *X/Open*.

thread. The smallest unit of operation to be performed within a process. *IBM*.

tilde. The character ~. This character is named <tilde> in the portable character set.

TLBL. Tape Label Information job control statement.

token. The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM*.

tokenization. The process of parsing input into tokens.

traceback. A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and the status of the routines on the call-chain at the time the traceback was produced.

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. *ISO-JTC1*.

tree. A hierarchical collection of nodes that can have an arbitrary number of references to other nodes. A unique path connects every two nodes.

truncate. To shorten a value to a specified length.

type. The description of the data and the operations that can be performed on or by the data. See also *data type*.

type conversion. Synonym for *boundary alignment*.

type specifier. Used to indicate the data type of an object or function being declared.

U

undefined behavior. Referring to a program or function that may produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

underflow. A condition that occurs when the result of an operation is less than the smallest possible nonzero number.

union. (1) In the C language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

unspecified behavior. Referring to a program or function that may produce erroneous results without warning because of erroneous program constructs or erroneous data.

user name. A string that is used to identify a user. *ISO.1*.

V

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

visible. Visibility of identifiers is based on scoping rules and is independent of *access*.

VSAM (Virtual Storage Access Method). An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability. *IBM*.

W

while statement. A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

white space. (1) Space characters, tab characters, form-feed characters, and newline characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), newline characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

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.

wide-character code. An integral value corresponding to a single graphic symbol or control code. *X/Open*.

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

wide-oriented stream. See *orientation of a stream*.

wrapping of output. The automatic disposition of a line of output onto two or more lines, necessitated by the limitation of the width of the device or file to which output is directed.

write. (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1*. *ANSI/ISO*.

Index

Readers' Comments — We'd Like to Hear from You

IBM Language Environment for VSE/ESA
C Run-Time Programming Guide
Version 1 Release 4 Modification Level 4

Publication No. SC33-6688-05

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
SC33-6688-05

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

Fold and Tape **Please do not staple** Fold and Tape

SC33-6688-05

Cut or Fold
Along Line

Program Number: 5686-CF7

Printed in USA

SC33-6688-05

Spine information:

IBM LE/VSE C Run-Time Programming Guide Version 1 Release 4 Modification Level 4