VMS Software

VSI OpenVMS

VSI DECset for OpenVMS Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual

Document Number: DO-LSECLI-01A

Publication Date: April 2024

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: DECset Version 12.7

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI DECset for OpenVMS Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual

Legal Notice

Preface

This reference provides information on how to use the VSI Language-Sensitive Editor (LSE) command-line interface and callable routines on OpenVMS systems.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This reference is intended for experienced programmers and technical managers.

3. Document Structure

This reference contains the following information:

- Chapter 1 provides OpenVMS operating system-specific information for using LSE.
- Chapter 2 describes the LSE callable interface routines and how to use them.

4. Related Documents

The following documents are also helpful when using LSE:

- The *VSI DECset for OpenVMS Language-Sensitive Editor/Source Code Analyzer Reference Manual* provides a description of the LSE commands.
- The *VSI DECset for OpenVMS Guide to Language-Sensitive Editor* provides a tutorial description on the use of the LSE commands from the DECwindows interface, and contains other important user information.

LSE is a component of the VSI DECset toolkit. For more information on other VSI DECset components, see the reference manuals for the individual components.

5. References to Other Products

Some older products that VSI DECset components previously worked with might no longer be available or supported by VSI. Any reference in this manual to such products does not imply actual support, or that recent interoperability testing has been conducted with these products.

Note

These references serve only to provide examples to those who continue to use these products with VSI DECset.

Refer to the Software Product Description for a current list of the products that the VSI DECset components are warranted to interact with and support.

6. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

7. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

8. Conventions

VMScluster systems are now referred to as OpenVMS Cluster systems. Unless otherwise specified, references to OpenVMS Cluster systems or clusters in this document are synonymous with VMScluster systems.

The contents of the display examples for some utility commands described in this manual may differ slightly from the actual output provided by these commands on your system. However, when the behavior of a command differs significantly between OpenVMS Alpha and Integrity servers, that behavior is described in text and rendered, as appropriate, in separate examples.

In this manual, every use of DECwindows and DECwindows Motif refers to DECwindows Motif for OpenVMS software.

The following conventions are also used in this manual:

Convention	Meaning
Ctrl/ *x*	A sequence such as **Ctrl/** *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
…	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.

Convention	Meaning
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
[|]	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Chapter 1. Using LSE on OpenVMS Systems

1.1. Introduction

This chapter describes basic information for using VSI Language-Sensitive Editor (LSE) on OpenVMS systems.

LSE is a multilanguage text editor that speeds up writing and compiling source code. It is part of the DECset family of software development tools. Each of the DECset tools enable you to take advantage of the multilanguage software development capabilities on OpenVMS systems.

This chapter contains the following information:

- How logical names are used by LSE
- Format of the LSE command line and detailed descriptions of each command-line qualifier
- Using the command languages
- Templates for subroutine packages
- Diagnostic file support

The LSE commands referenced in this document are for the VMSLSE and Portable command languages. For information on the LSE callable interface, see Chapter 2.

1.2. LSE Logical Names

Table 1.1 lists the logical names and describes how they are used by LSE.

Table 1.1. LSE Logical Names

Logical Name	Description
LSE$READ_ONLY_DIRECTORY	Logical for read-only directories. Same as using the SET DIRECTORY READONLY command, except it can be executed from the DCL command line. Define this logical to be a list of directories for which LSE will create read-only buffers for input files from within them.
LSE$SOURCE	Logical for source directories. Same as using the SET DIRECTORY SOURCE command, except it can be executed from the DCL command line. Define this logical to be a list of directories for LSE to use when locating files.
LSE$INITIALIZATION	Logical for an initialization file. Same as invoking LSE with the /INITIALIZATION qualifier.
LSE$COMMAND	Logical for a command file. Same as invoking LSE with the /COMMAND qualifier.
LSE$ENVIRONMENT	Logical for an environment file. Same as invoking LSE with the /ENVIRONMENT qualifier.

Logical Name	Description
LSE$SECTION	Logical for a command file. Same as invoking LSE with the /SECTION qualifier.
LSE$SYSTEM_ENVIRONMENT	Logical for an environment file. Same as invoking LSE with the /SYSTEM_ENVIRONMENT qualifier.
LSE$CURRENT_FILE	Logical for the current file. Set to the last file edited by LSE and used when LSE is started, if no file is specified (unless the qualifier /NOCURRENT_FILE is specified).
LSE$START_LINE	Logical for the current file. Holds the line of the last editing position in the last file edited by LSE.
LSE$START_CHARACTER	Logical for the current file. Holds the character of the last editing position in the last file edited by LSE.
LSE$EXAMPLE	Logical that points to the LSE examples directory.

1.3. Using Command Languages

LSE has two command languages available: VMSLSE and Portable. The VMSLSE command language is the original LSE command language that has always been present in LSE, and has remained the most used. In addition, the VMSLSE command language is required for integration with DECwindows SCA. The Portable command language is a more recent command language devised for use in environments other than OpenVMS. The choice of default command language is made at DECset installation time, but can always be changed.

You can determine the current command language setting by issuing the following command:

```
LSE> PLSE SHOW ATTRIBUTES
```

For information on integrating LSE with VSI Source Code Analyzer (SCA) and VSI Code Management System (CMS), see Section 1.3.4 and Section 1.3.5.

This guide presents examples of both VMSLSE and Portable commands.

Note

The online LSE Help includes a command translation table from VMSLSE to Portable commands. At the command line, execute the following command:

```
LSE> PLSE HELP VMSLSE_Command_Translation_Table
```

For information on customizing the Portable command language, invoke the following Help command:

```
LSE> PLSE HELP Customizing_Command_Language
```

1.3.1. Setting the Default Command Language

If you often use commands that can only be invoked in one command language, you might want to change the default command language setting. To set the default command language, perform one of the following:

- In LSE—Enter SET COMMAND LANGUAGE VMSLSE or SET COMMAND LANGUAGE Portable.
- In an initialization file—Specify the desired SET COMMANDLANGUAGE command.
- LSE procedure calls in a command file—Enter LSE_SET_COMMAND_LANGUAGE ('VMSLSE') or LSE_SET_COMMAND_LANGUAGE ('PORTABLE').

1.3.2. Invoking LSE Command Languages

Only one of the command languages can be set as the default, and you will normally invoke the commands that apply to that command language. However, you might occasionally need to invoke a specific command from the nondefault language. The following examples show the syntax for LSE commands both within and outside the default command language setting:

- VMSLSE commands in a VMSLSE language setting (normal—no special command syntax needed)

```
LSE> CHECK LANGUAGE/HELP PASCAL
```

- Portable commands in a VMSLSE language setting

```
LSE> PLSE CHECK LANGUAGE HELP PASCAL
```

- VMSLSE commands in a Portable language setting

```
LSE> TPU LSE$DO_COMMAND ('SET SCREEN HEIGHT=18')
```

- Portable commands in a Portable language setting (normal—no special command syntax needed)

```
LSE> SET HEIGHT 18
```

You can also set your own defaults in private section files, such as saving all current settings into a binary file when you execute the SAVE_ENVIRONMENT command.

1.3.3. Using the SET PROMPT KEYPAD Command

By default, the VMSLSE command language uses the VMSLSE keypad, which you can change to the DEFAULT or USER keypad. Specify the following command at the LSE> prompt:

```
LSE>  PLSE SET PROMPT KEYPAD keypad_name
```

In this example, *keypad_name* represents one of the following:

- DEFAULT—Allows the use of any default key definition within a prompt (for example, REMOVE and INSERT)
- USER—Allows the use of any key definition within a prompt, including user-defined keys
- VMSLSE—Allows the use of the VMSLSE keypad (the default setting)

By default, the Portable command language uses the DEFAULT keypad.

1.3.4. Integrating LSE with SCA and CMS

Integration of LSE (character-cell or DECwindows) with CMS and the character-cell SCA is not enabled by default in the Portable command language setting. To enable this integration, specify the following command from the Portable command language:

```
LSE> ENABLE VMS INTEGRATION
```

This command sets the VMSSCA_ and VMSCMS_ grammar prefixes, enabling access to VMSLSE-compatible SCA and CMS commands. This command also redefines key bindings and menu labels related to cross referencing to SCA (for example, Ctrl/D is redefined to GOTODECLARATION).

The DISABLE VMS INTEGRATION command restores the default setting.

Note

The ENABLE VMS INTEGRATION command enables two additional command prefixes VMSCMS_ and VMSSCA_, which serve to “hide” the standard Help for CMS and SCA commands in this environment. You can access Help for these commands using either of the following methods:

- Execute Help with a SPAWN command:

  ```
  LSE> SPAWN CMS HELP cms-command
  LSE> SPAWN SCA HELP sca-command
  ```

- Interrupt operations with the DISABLE VMS INTEGRATION command, access the Help you need, then resume operations with the ENABLE VMS INTEGRATION command.

1.3.5. Integrating DECwindows LSE with DECwindows SCA

To integrate DECwindows LSE with DECwindows SCA requires that your default command language setting be VMSLSE. Once that has been ensured, you need to redefine some commands. (This is unlike the integration described in Section 1.3.4.)

If your default is the Portable command language, the first step is to change it, as follows:

```
LSE> SET COMMAND LANGUAGE VMSLSE
```

With the VMSLSE command language set, you must redefine the menu commands and optionally redefine the key bindings, as follows:

1. *Redefine the DECwindows LSE menu commands*—From the DECwindows LSE Options menu, select Menus... to display the "LSE: Menus" dialog box, then perform the following steps:

   a. From the right-hand list box, titled Labels, select the Find Occurrences label. In the TPU Code text field at the bottom, remove the "$KEY" string so the label definition appears as follows:

      ```
      LSE_FIND_OCCURRENCES
      ```

      After changing the label definition, click the Add Entry Arrow button to add the modified label.

   b. From the right-hand list box, titled Labels, select the Goto Declaration label. In the TPU Code text field at the bottom, remove the "$KEY" string so the label definition appears as follows:

      ```
      LSE_GOTO_DECLARATION
      ```

      After changing the label definition, click the Add Entry Arrow button to add the modified label.

2. *Optionally redefine the DECwindows LSE key bindings*—From the DECwindows LSE command line, redefine the following key bindings by executing each of these commands:

```
LSE> DEFINE KEY CTRL/D "TPU LSE_GOTO_DECLARATION"
LSE> DEFINE KEY GOLD-CTRL/D "TPU VMSSCA_GOTO_CONTEXT_DECLARATION('/
INDICATED')"
LSE> DEFINE KEY GOLD-CTRL/F "TPU LSE_FIND_OCCURRENCES"
```

Redefine these key bindings if you expect that you might, at any time, use the CTRL keys instead of the menu commands.

Note

If a DECwindows SCA session was not active when you executed the previous commands, attempts to use the changed "Find Occurrences" or "Goto Declaration" Source menu options or key bindings will result in the error message, "Cross reference utility not running". You simply need to launch DECwindows SCA; there is no need to redo the commands.

1.4. LSE Command Line

This section describes the format of the LSE command line and includes detailed descriptions of each command-line qualifier.

The LSEDIT command invokes LSE. This command has the following syntax:

```
LSEDIT [/qualifiers] [file-spec]
```

/qualifiers

Specifies the LSEDIT command qualifiers.

file-spec

Specifies the file to be edited. It must be an OpenVMS file specification. LSE uses the setting of the SET DIRECTORY SOURCE command or the corresponding LSE$SOURCE logical name to resolve the file specification.

LSE reads the file into a buffer if the file exists. The buffer name is taken from the name and type of the file specification in the command line. The file type determines the language for the buffer. For example, .FOR is the file type for Fortran, .PLI is the file type for PL/I, and .PAS is the file type for Pascal. If the file does not exist, it is created when you use the EXIT command to leave LSE.

If you do not specify a file name in your file specification, LSE uses the file name specified in your last LSEDIT command, provided you entered the EXIT command to end that editing session. If you do not specify a file type in your file specification, LSE adds .LSE if your default command language is VMSLSE, or .PLSE if your default command language is the Portable command language.

The cursor is positioned at the same place as when you last left LSE. The file name, type, and position are collectively called the **current file information**. The current file information is updated only when you use the EXIT command to leave LSE. If you use the /NOCURRENT_FILE qualifier, LSE does not use the file specification from the previous LSEDIT command as the input file specification. The QUIT command or Ctrl/Y does not change the current file information.

1.4.1. LSE Command-Line Qualifiers

When you invoke LSE, you can use several command-line qualifiers to provide more information to LSE on how to handle your files. Table 1.2 lists these command-line qualifiers. Detailed descriptions of the qualifiers and their defaults, indicated by (D), follow the table.

Table 1.2. LSE Command-Line Qualifiers

Qualifier	Default
/[NO]COMMAND=file-spec	See text
/[NO]CREATE	/CREATE
/[NO]CURRENT_FILE	/CURRENT_FILE
/[NO]DEBUG	/NODEBUG
/[NO]DISPLAY	/DISPLAY=CHARACTER_CELL
/[NO]ENVIRONMENT=file-spec-list	/NOENVIRONMENT
/[NO]INITIALIZATION=file-spec	See text
/[NO]INTERFACE	/INTERFACE=CHARACTER_CELL
/[NO]JOURNAL[=file-name]	/JOURNAL
/LANGUAGE=language	See text
/[NO]MODIFY	See text
/[NO]OUTPUT[=file-spec]	/OUTPUT[=file-spec]
/[NO]READ_ONLY	See text
/[NO]RECOVER	/NORECOVER
/[NO]SECTION=file-spec	/SECTION=LSE$SECTION
/START_POSITION=(line,character)	See text
/[NO]SYSTEM_ENVIRONMENT	/SYSTEM_ENVIRONMENT=LSE $SYSTEM_ENVIRONMENT
/[NO]WRITE	See text

/COMMAND=file-spec
/NOCOMMAND

Specifies a file containing DECTPU statements to be executed as part of the LSE initialization.

If you specify the /NOCOMMAND qualifier, LSE does not use a DECTPU initialization command file. (See the *DEC Text Processing Utility Reference Manual* for more information.)

You can define the logical name LSE$COMMAND to point to a file containing DECTPU statements. If neither the /COMMAND nor /NOCOMMAND qualifier appears on the command line, LSE tries to translate the logical name LSE$COMMAND. If it has a translation, that value is used in the same way as the /COMMAND qualifier value.

/CREATE (D)
/NOCREATE

Controls whether LSE creates a new file when the specified input file is not found. By default, LSE provides a buffer in which to create the file. When you exit from LSE or write out the contents of the buffer with the WRITE or COMPILE command, LSE creates a new file with the input file specification in the appropriate directory.

When you specify the /NOCREATE qualifier on the LSE command line and the name of a file to edit and the named file does not exist, LSE displays an error message and places you in a buffer called $MAIN.

/CURRENT_FILE (D)
/NOCURRENT_FILE

Specifies whether LSE uses the last file edited as the input file specification if no file is specified on the command line.

/DEBUG[=debug-filespec]
/NODEBUG (D)

Determines whether you will run a DECTPU debugger. This is useful in testing DECTPU procedures for an application you are creating. LSE reads, compiles, and executes the debug file before executing TPU$INIT_PROCEDURE.

The default debug file specification is SYS$SHARE:LSE$DEBUG.TPU. You can override this default on the command line to specify a debug file of your own. For example, the following command invokes LSE, using a debug file called SYS$SHARE:MYDEBUG.TPU:

```
$ LSEDIT/DEBUG=mydebug
```

You can define the logical name LSE$DEBUG to specify a debug file of your own. This is useful if you want to keep the debug file in a directory other than SYS$SHARE. You cannot use wildcards in the debug file specification. The TPU debugger provides commands to manipulate variables and control program execution. To start editing the code in the file you are debugging, use the debugger command GO. For more information about the DECTPU debugger, read the comments in the source file in SYS$SHARE:LSE$DEBUG.TPU, or see the *DEC Text Processing Utility Reference Manual*.

/DISPLAY=CHARACTER_CELL (D)
/DISPLAY=DECWINDOWS
/DISPLAY=screen_manager_filespec
/NODISPLAY

Specifies which screen manager you want to run.

The /DISPLAY command qualifier is optional. By default, LSE uses the character-cell screen manager. As an alternative to the /DISPLAY qualifier, you can define the logical name LSE $DISPLAY_MANAGER as DECWINDOWS, CHARACTER_CELL, or as a screen-manager file specification.

Note that this qualifier is synonymous to the /INTERFACE qualifier. In addition, it allows you to specify the negative form, /NODISPLAY.

If you specify /DISPLAY=CHARACTER_CELL, LSE uses the character-cell screen manager, which runs in a DECterm terminal emulator or on a physical terminal.

If you specify /DISPLAY=DECWINDOWS, LSE uses the DECwindows screen manager, which creates a DECwindows window in which to run LSE.

You cannot use the /NODISPLAY qualifier if the logical name LSE$DISPLAY_MANAGER is pointing to the DECwindows window manager.

/ENVIRONMENT=file-spec-list
/NOENVIRONMENT (D)

Specifies the name of one or more binary environment files containing LSE language, token, placeholder, alias, or package definitions. LSE reads in these definitions as part of the LSE startup

procedure. If you specify more than one file, you must enclose the files in parentheses and separate them with commas.

If definitions or deletions of items appear in more than one file, the definition that appears in the file listed first takes precedence.

SYS$LIBRARY: is the default device. The default file type is .ENV.

The logical name LSE$ENVIRONMENT is an alternative to the /ENVIRONMENT command qualifier. If the /ENVIRONMENT or /NOENVIRONMENT qualifier does not appear on the command line, LSE tries to translate the logical name LSE$ENVIRONMENT. If it has a translation, the value is used in the same way as the /ENVIRONMENT qualifier value. LSE translates the first ten indexes of the logical name LSE$ENVIRONMENT.

See the SAVE ENVIRONMENT command in the LSE command dictionary for information on using environment files.

/INITIALIZATION=file-spec
/NOINITIALIZATION

Specifies the name of a file containing a sequence of LSE commands to be executed as part of the LSE startup procedure. This file usually contains the occurrences of the NEW KEY command.

The logical name LSE$INITIALIZATION is an alternative to the /INITIALIZATION qualifier. If /INITIALIZATION or /NOINITIALIZATION does not appear on the command line, LSE tries to translate the logical name LSE$INITIALIZATION. If it has a translation, the value is used in the same way as the /INITIALIZATION qualifier value.

/INTERFACE=CHARACTER_CELL (D)
/INTERFACE=DECWINDOWS
/INTERFACE=screen_manager_filespec

Specifies which screen manager you want to run.

The /INTERFACE qualifier is optional. By default, LSE uses the character-cell screen manager. As an alternative to the /INTERFACE qualifier, you can define the logical name LSE $DISPLAY_MANAGER as DECWINDOWS, CHARACTER_CELL, or as a screen-manager file specification.

Note that this qualifier is synonymous to the /DISPLAY qualifier, but unlike /DISPLAY, you cannot specify negation.

If you specify /INTERFACE=CHARACTER_CELL, LSE uses the character-cell screen manager, which runs in a DECterm terminal emulator, or on a physical terminal.

If you specify /INTERFACE=DECWINDOWS, LSE uses the DECwindows screen manager, which creates a DECwindows window in which to run LSE.

/JOURNAL (D)
/JOURNAL[=file-name]
/NOJOURNAL

Enables journaling for the editing session.

The /JOURNAL qualifier without any value enables buffer-change journaling only. One buffer-change journal file is created for each editing buffer. The name of each buffer-change journal file

corresponds to the name of the buffer it is journaling. The default file type for buffer-change journal files is .TPU$JOURNAL.

If you supply a file name as the value to the /JOURNAL qualifier, keystroke journaling is also performed. The name of the keystroke journal is taken from the value supplied to the /JOURNAL qualifier. There is one keystroke journal file for the editing session. The default file type for keystroke journal files is .TJL.

To perform a recovery using a buffer-change journal file, use the RECOVER BUFFER command after starting the editor. Use the /RECOVER command-line qualifier only when attempting to recover using a keystroke journal file. If you perform a recovery using a keystroke journal file, be sure to restore the editing session that you began.

If you do not want to create a journal file of either type, use the /NOJOURNAL qualifier.

/LANGUAGE=language

Sets the language for the current input file by overriding the language indicated by the file type of the input file.

/MODIFY
/NOMODIFY

Specifies whether the buffer you create is modifiable or unmodifiable. If you specify the /MODIFY qualifier, the LSEDIT command creates a modifiable buffer. If you specify the /NOMODIFY qualifier, the LSEDIT command creates an unmodifiable buffer. If you do not specify either qualifier, LSE determines the buffer's modifiable status from the read-only or write setting. By default, a read-only buffer is unmodifiable and a write buffer is modifiable.

/OUTPUT[=file-spec] (D)
/NOOUTPUT

Specifies the name of the file that LSE creates from the input file when you exit from the editing session. Specifying a file specification on the /OUTPUT qualifier causes LSE to ignore the current file information. By default, LSE creates a new version of the input file.

Missing components of the file specification in the /OUTPUT qualifier take their values from the corresponding fields of the input file specification.

When you exit from the editing session, LSE writes other buffers to their associated files if the buffer contents have been modified during the session. If you specify the /NOOUTPUT qualifier, LSE prevents writing back the main buffer when you exit.

/READ_ONLY
/NOREAD_ONLY

Specifies that LSE create a read-only buffer for the input file. LSE does not create a new output file. Any changes to the file are lost when you exit from the editing session. This qualifier does not affect writing back other buffers to their associated files if they were modified during the editing session.

If the /[NO]READ_ONLY qualifier is not specified explicitly, the read or write status of the buffer for the input file is determined by the default settings of the SET DIRECTORY command, or LSE $READ_ONLY_DIRECTORY logical name.

/RECOVER
/NORECOVER (D)

Directs LSE to use the latest version of the file specified as the value to the /JOURNAL qualifier to recover changes that might have been lost due to a previous abnormal LSE termination.

Note

The qualifier /RECOVER should be used only when attempting to recover using a keystroke journal file. If you want to recover using a buffer-change journal file, use the RECOVER BUFFER command in LSE after the editor has been started.

When you recover a session, all files must be in the same state as they were at the start of the editing session being recovered. You must enter the LSEDIT/RECOVER command with the same qualifiers, initialization file, section file, and environment file as you did for the session being recovered.

All terminal characteristics must be in the same state as they were at the start of the editing session being recovered. If you changed the width or page length of the terminal, you must change it back to the value it had at the start of the editing session you want to recover. Check the following values by using the DCL command SHOW TERMINAL:

- Device type
- Edit mode
- Eight bit
- Page
- Width

/SECTION=file-spec
/SECTION=LSE$SECTION (D)
/NOSECTION

Specifies whether LSE is to map a section file containing DECTPU procedures, key definitions, and variables. By default, LSE maps section file LSE$SECTION. If you supply another file specification, LSE applies the default SYS$LIBRARY:.TPU$SECTION when it opens the file.

If you specify the /NOSECTION qualifier, LSE does not use a section file, and many LSE commands will not work. Therefore, when using the /NOSECTION qualifier, you should specify the /COMMAND qualifier. The command file should use only standard DECTPU built-ins.

/START_POSITION=(line,character)

Specifies the starting line and character in the file (top-of-file is /START_POSITION=(1,1)). If you do not specify /START_POSITION, LSE starts either at the top of the file, or at the position of the cursor when you last edited the file.

/SYSTEM_ENVIRONMENT=file-spec
/SYSTEM_ENVIRONMENT=LSE$SYSTEM_ENVIRONMENT (D)
/NOSYSTEM_ENVIRONMENT

Specifies the name of a system environment file. The difference between the file specified by this qualifier and the file specified by the /ENVIRONMENT qualifier is that definitions from the system environment file are not saved by a SAVE ENVIRONMENT command.

The default device is SYS$LIBRARY: and the default file type is .ENV.

/WRITE
/NOWRITE

Specifies that LSE create a new output file when you exit from the editing session. Any changes you made to the file are saved.

If the /[NO]WRITE qualifier is not specified explicitly, the read or write status of the buffer for the input file is determined by the default settings of the SET DIRECTORY command, or the LSE $READ_ONLY_DIRECTORY logical name.

1.5. Packages

LSE provides templates for subroutine packages. These packages define OpenVMS System Services, Run-Time Library (LIB$, STR$, SMG$), and Record Management System (RMS) routines. In addition, LSE provides a mechanism for defining packages for your own subroutine libraries.

The System Services and RMS packages consist of routine definitions and parameter definitions that are available automatically when you use LSE with any of the following languages:

- DEC Ada
- DEC BASIC
- VAX BLISS-32
- VSI C
- VSI C++
- VSI COBOL
- VAX COBOL
- VSI Fortran
- VAX MACRO
- VSI Pascal
- VAX PL/I

Routines are useful for describing subroutine libraries. Not only are they language-independent, but they need to be defined only once. Routine names are used in the same way tokens are used. For example, if you type the routine name SYS$FILESCAN and expand it, the following results:

```
sys$filescan    ( {srcstr},
                  {valuelst},
                  [fldflags] )
```

Most languages access OpenVMS System Services and RMS routines with the prefix SYS$. These languages must use the SYSTEM_SERVICES package. Other languages use different prefixes. For example, DEC Ada prohibits the prefix dollar sign ($) and must use the STARLET

package. VAX BLISS and DEC Pascal require the prefix dollar sign ($) and must use the KEYWORD_SYSTEM_SERVICES package.

For example, to call the $SNDOPR system service from a VAX PL/I program, enter the following line:

```
status := sys$sndopr
```

Then, you press the EXPAND key with the cursor just after sys$sndopr. This expands to the following lines:

```
status := sys$sndopr (
              {msgbuf},
              [chan])
```

This indicates that the $SNDOPR system service has two parameters: MSGBUF, which is required and CHAN, which is optional. Because CHAN is optional, LSE expands it with an optional placeholder that you can either delete or expand. Languages other than DEC Ada and VAX BLISS have similar features.

In DEC Ada, the dollar sign is not used as part of the system service name. Thus, you can enter the following line:

```
starlet.sndopr
```

Next, when you press the EXPAND key, it expands to the following lines:

```
STARLET.SNDOPR (
    STATUS => {status},
    MSGBUF => {msgbuf},
    [CHAN   => {chan}]);
```

In VAX BLISS, the system services start with a dollar sign without the leading SYS. Thus, you can enter the following line:

```
status = $sndopr
```

When you press the EXPAND key it expands to the following lines:

```
status = $sndopr (
             msgbuf = {~msgbuf~},
             [~chan   = {~chan~}~])
```

You can access OpenVMS online help for any of the system services in any language. If you want help on any routine, place the cursor on the routine name and press the HELP INDICATED key (PF1-PF2). You cannot use HELP INDICATED on the parameter names; however, the HELP entry for the system service will contain information on the parameters.

If you want to see the contents of a given package, parameter, or routine, use the SHOW command. If you want to modify the definitions of a package, use the EXTRACT command.

1.6. Diagnostic File Support

Diagnostic files communicate diagnostic messages to LSE from various tools. A tool, such as a compiler, generates a diagnostic file that LSE uses to display the diagnostics. After you display a diagnostic file in LSE, you can navigate through the file from one diagnostic to the next. You can use the GOTO SOURCE command to display the source that corresponds to a diagnostic in another window.

There are two formats for diagnostic files:

- User-file format—Provides a simple format for customer tools to communicate diagnostic information to LSE. You can list this format without a special dump utility.
- VSI internal-file format—Binary format that is used by VSI products to communicate diagnostic messages to LSE.

You can concatenate user-file and VSI internal-file diagnostic modules into one file and review them together.

Typically, a tool generates a module of zero or more diagnostics each time it processes a source file. For example, a compiler generates a diagnostic module for each compilation. Diagnostics typically are errors. Each diagnostic consists of the following:

- Regions — Define the location of the source associated with the diagnostic. There can be more than one region.
- Messages — Textual descriptions that explain the diagnostic. There can be more than one message.

The rules that apply to DCL apply to the user-file format. For example, nonquoted strings are converted to uppercase.

Section 1.6.1 describes the use of the /DIAGNOSTICS qualifier with the COMPILE command. Section 1.6.2 shows an example diagnostic module in the user-file format and explains how the module is used. Section 1.6.3 describes each of the commands used in the user-file format.

1.6.1. The /DIAGNOSTICS Qualifier

The /DIAGNOSTICS qualifier is used with the COMPILE command to specify that diagnostic files are generated by the compiler.

This command-line qualifier is required for Portable LSE and must be added to the language's compile command. For VMSLSE, the qualifier is added by default.

In VMSLSE, when you specify the /CAPABILITIES=DIAGNOSTICS qualifier for the DEFINE LANGUAGE and MODIFY LANGUAGE commands, a /DIAGNOSTICS qualifier is automatically appended to the COMPILE command. In Portable LSE, the /DIAGNOSTICS qualifier must be added to the COMPILE command.

1.6.2. User-File Format

Example 1.1 shows a diagnostic module in the user-file format. Comments are introduced by an exclamation mark (!).

Example 1.1. User-File Format Diagnostic

```
 start module ! This command signals the start of a module.
 start diagnostic    ! This region marks line 1 in the file, and  ❶
 ! it is not a primary region.
 region/file DEV$:[user.ex1]test.ada;1/line=1/column_range=(1,65535)  ❷
 region/nested/column_range=(18)! Marks the 18th column in the above
region.  ❸
 ! 2nd region
 region/file DEV$:[user.ex1]test.ada;1/line=3/column_range=(1,65535)
```

```
  ! The following nested region marks column 4 of line 3 for the file
 specified above.
  region/nested/column_range=(4)! Marks the 4th column in the above region.
  ! This is the primary region that LSE will highlight when positioned on
 this
  ! diagnostic.
  region/file DEV$:[user.ex1]test.ada;1/line=10/column_range=(1,65535)-
    /primary    !  This region marks all of line 10 in the file.
  region/nested/column_range=(4,4) ! Specifies a subregion at the above
 region.
  ! Messages  message/text=quoted "%ADAC-E-ASSIGNNERESTYP, Result type
 BOOLEAN in pre ..."
  message/text=quoted "        b at line 3 is not the same as type
 INTEGER ..."
  message/text=quoted "        subprogram 'in' formal a at line 1 [LSM
 5.2(1)]"
  end diagnostic
  ! The next example is taken from a C diagnostic.  The file region refers
 to a line
  ! in the text that contains a macro call and the text supplied by the
 text
  ! region is the macro expansion.   start diagnostic  ❹
  region/file DEV$:[user.c]macro.c;2/line=11/column_range=(5,25)-
  /primary
  region/text "    if (i>0) j=k else l=m;"-          /line=1/
column_range=(1,26)
  message/text=quoted "%CC-W-INSBEFORE, Insert "";"" before reserved
 word ..."
  end diagnostic
  end module
```

Key to Example 1.1

❶ The first diagnostic shows how regions and messages work together.
❷ The file regions refer to lines in the source that cause the error described in the text message.
❸ The nested regions in each of the file regions refer to the location in each line that contributes to the error.
❹ The second diagnostic shows how a text region can be used to display macro text for error messages.

1.6.3. User-File Format Command Descriptions

The following section describes the commands that define the user-file format.

END DIAGNOSTIC

END DIAGNOSTIC — Ends a diagnostic that begins with a START DIAGNOSTIC command.

Format

END DIAGNOSTIC

Description

This command ends a sequence of commands that make up a diagnostic.

Example

See Example 1.1 for a sample of the END DIAGNOSTIC command.

END MODULE

END MODULE — Ends a module in the user-file format that begins with the START MODULE command.

Format

```
END MODULE
```

Description

This command ends a sequence of commands that make up a user-file format diagnostic module.

Example

See Example 1.1 for a sample of the END MODULE command.

MESSAGE/FILE

MESSAGE/FILE — Defines a message in a file for a diagnostic that appears in the REVIEW buffer during a review session.

Format

```
MESSAGE/FILE file-spec
```

Command Parameter

file-spec

Specifies the file containing the message

Description

This command specifies a file that contains the message to be displayed in the REVIEW buffer for a diagnostic. The entire file is displayed in the REVIEW buffer. The message is usually an error message.

Example

```
MESSAGE/FILE DEV$:[USER]MESSAGE.TXT
```

The contents of the file specified are displayed as the message in the REVIEW buffer.

MESSAGE/TEXT

MESSAGE/TEXT — Defines a quoted or unquoted message for a diagnostic that appears in the REVIEW buffer during a review session.

Format

MESSAGE/TEXT=[UN]QUOTED message-definition

Command Parameter

message-definition

Specifies the message.

Description

The MESSAGE/TEXT=QUOTED command specifies that the message for the diagnostic is a quoted string. A quoted message is enclosed in double quotes (“”) with embedded double quotes (“ ‘’ ”) used to place quotes in the string.

The MESSAGE/TEXT=UNQUOTED command specifies that the message is the remaining text in the line. It does not have to be quoted. Nonquoted text is converted to uppercase and leading and trailing white space is removed.

The message is usually an error message. If no qualifier is specified for the MESSAGE command, /TEXT_QUOTED is the default.

Examples

1. `MESSAGE/TEXT=UNQUOTED  Here is another message.`

   This message is displayed in the REVIEW buffer. Leading and trailing white space is truncated and the lowercase letters are converted to uppercase, as follows:

   ```
   HERE IS ANOTHER MESSAGE.
   ```

2. `MESSAGE "Inserted "";"" at end of line"`

   If no qualifier is specified, or /TEXT alone is specified, the default becomes /TEXT=QUOTED. This message is included in the REVIEW buffer without the beginning and trailing quotes, as follows:

   ```
   Inserted “;” at end of line.
   ```

REGION/FILE

REGION/FILE — Specifies that the source location associated with a diagnostic is in a file.

Format

REGION/FILE file-spec

Command Qualifiers	Defaults
/LINE=number	/LINE=1
/COLUMN_RANGE= (number,number)	/COLUMN_RANGE= (1,1)
/LABEL=string	See text
/PRIMARY	

Qualifiers

/LINE=number
/LINE=1 (D)

Specifies the line number in the file for the region. The first line in a file is 1. The valid range for the /LINE qualifier is −1 and higher. The −1 indicates a line after the last line and 0 indicates a line before the first line. If the line value is 0 or −1, any column range values specified are ignored.

/COLUMN_RANGE= (number,number)
/COLUMN_RANGE= (1,1) (D)

Specifies a range of columns in the file that defines the region. If only the first number is specified, the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for a column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

/LABEL=string

Specifies a short message that is appended to the beginning of the region in the REVIEW buffer. The default is line *n*, where *n* is the line number of the source specified with the /LINE qualifier and the default for text regions is supplied text. The string must contain 14 or fewer characters.

/PRIMARY

Specifies the primary region among a group of regions. LSE positions the cursor on the primary region for a diagnostic when reviewing that diagnostic. If no region is specified as primary, the first sequential region (any region but a nested region) is assumed to be primary. If more than one region in a diagnostic is marked primary, the first one is used.

Command Parameter

file-spec

Specifies the file that contains the region. The full file specification for the file region, which includes the device, directory, and version, should be used to help ensure that LSE accesses the correct file when the GOTO SOURCE command is executed.

Description

This command defines an area in a file that is associated with a diagnostic. This area cannot span more than one line. If /FILE, /LIBRARY, /NESTED, or /TEXT is not specified with the REGION command, /FILE is the default and need not be entered.

Example

```
REGION/FILE DEV$:[user]program.src;23 -
             /LINE=10 -
             /COLUMN_RANGE=1 -
             /Label="Src Line 10:" -
             /PRIMARY
```

This region points to the first column of the tenth line in file DEV$:[user]program.src;23. The region has a user-specified label and is a primary region.

REGION/LIBRARY

REGION/LIBRARY — Specifies that the source location associated with a diagnostic is in a module within a text library.

Format

`REGION/LIBRARY file-spec`

Command Qualifiers	Defaults
/MODULE=module-name	
/LINE=number	/LINE=1
/COLUMN_RANGE= (number,number)	/COLUMN_RANGE= (1,1)
/LABEL=string	See text
/PRIMARY	

Qualifiers

/MODULE=module-name

Specifies the module in the library that contains the region.

/LINE=number
/LINE=1 (D)

Specifies the line number in the library module for the region. The first line in the module is 1. The valid range for the /LINE qualifier is −1 and higher. The −1 indicates a line after the last line and 0 indicates a line before the first line. If the line value is 0 or −1, any column range values specified are ignored.

/COLUMN_RANGE= (number,number)
/COLUMN_RANGE= (1,1) (D)

Specifies a range of columns in a module that defines the region. If only the first number is specified, then the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for a column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

/LABEL=string

Specifies a short message that is appended to the beginning of the region in the REVIEW buffer. The default is line *n*, where *n* is the line number of the source specified with the /LINE qualifier and the default for text regions is supplied text. The string must contain 14 or fewer characters.

/PRIMARY

Specifies the primary region among a group of regions. LSE positions the cursor on the primary region for a diagnostic when reviewing that diagnostic. If no region is specified as primary, the first sequential region (any region but a nested region) is assumed to be primary. If more than one region in a diagnostic is marked primary, the first one is used.

Command Parameter

file-spec

Specifies the library that contains the region. The full file specification for the library region, which includes the device, directory, and version, should be used to help ensure that LSE accesses the correct file when the GOTO SOURCE command is executed.

Description

This command defines an area in a library module for a diagnostic. This area cannot span more than one line.

EXAMPLE

```
REGION/LIBRARY  DEV$:[user]textlib.tlb;3 -
              /MODULE=textmod -
              /LINE=1 -
              /COLUMN_RANGE=(1,65535)
```

This region defines the entire first line in module textmod of library DEV$:[user]textlib.tlb;3. No label is specified, so the default of line 1 is used. This is not a primary region.

REGION/NESTED

REGION/NESTED — Specifies that the source location associated with a diagnostic is a subregion of the previous region.

Format

REGION/NESTED

Command Qualifier	Default
/COLUMN_RANGE= (number,number)	/COLUMN_RANGE= (1,1)

Command Qualifier

/COLUMN_RANGE= (number,number)
/COLUMN_RANGE= (1,1) (D)

Specifies a range of columns that define a subregion of the previous region. If only the first number is specified, the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for a column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

Description

This command defines an area that is a subregion of the /FILE, /TEXT, /LIBRARY, or /NESTED region. This area cannot span more than one line.

Each type of sequential region (file, text, or library) can have a nested region inside it. Nested regions can have nested regions. However, each subsequent nested region must fit inside the previous region. Regions

of the same size are considered to fit inside each other. A nested region cannot appear by itself; it must be a subregion of a sequential region. If more than four nested regions follow a sequential region, the rest are ignored.

If the GOTO SOURCE command is executed when reviewing a diagnostic file, LSE moves to the beginning of the innermost region of the region it is positioned on in the REVIEW buffer.

Example

```
REGION/FILE DEV$:[user]program.src;1 -
             /LINE=10 -
             /COLUMN_RANGE=(1,65535)

REGION/NESTED/COLUMN_RANGE=(2,10)

REGION/NESTED/COLUMN_RANGE=10
```

The nested regions define subregions of the file region. The first nested region defines the area from column 2 to column 10, inclusive, on line 10 in file DEV$:[user]program.src;1. The second nested region defines the last column in that region.

REGION/TEXT

REGION/TEXT — Specifies that the source location associated with a diagnostic is in the text that is included in this command as arguments.

Format

REGION/TEXT string [,string...]

Command Qualifiers	Defaults
/LINE=number	/LINE=1
/COLUMN_RANGE= (number,number)	/COLUMN_RANGE= (1,1)
/LABEL=string	See text
/PRIMARY	

Qualifiers

/LINE=number
/LINE=1 (D)

Specifies the line number of the strings included in the region. The first string included is line 1. The valid range for the /LINE qualifier is −1 and higher. The −1 indicates a line after the last line and 0 indicates a line before the first line. If the line value is 0 or −1, any column range values specified are ignored.

/COLUMN_RANGE= (number,number)
/COLUMN_RANGE= (1,1) (D)

Specifies a range of columns in the specified string that defines the region. If only the first number is specified, the second number defaults to the value of the first number; that is, /COLUMN_RANGE=5 is equivalent to /COLUMN_RANGE=(5,5). The valid range of numbers for a column range is 1 to 65535, inclusive, with 65535 indicating the end-of-line and 1 indicating the

first column on a line. Therefore, /COLUMN_RANGE=(12,65535) defines a region that starts in column 12 and runs to the end-of-line.

/LABEL=string

Specifies a short message that is appended to the beginning of the region in the REVIEW buffer. The default is line *n*, where *n* is the line number of the source specified with the /LINE qualifier and the default for text regions is supplied text. The string must contain 14 or fewer characters.

/PRIMARY

Specifies the primary region among a group of regions. LSE positions the cursor on the primary region for a diagnostic when reviewing that diagnostic. If no region is specified as primary, the first sequential region (any region but a nested region) is assumed to be primary. If more than one region in a diagnostic is marked primary, the first one is used.

Command Parameter

string [,string...]

A quoted string (or strings separated by commas) that is the supplied text for this command. This text appears in the REVIEW buffer for the region.

Description

This command defines an area in the text supplied with the command for a diagnostic. This area cannot span more than one line.

Example

```
REGION/TEXT "A := B;", -
             "C := D," -
             /LINE=1 -
             /COLUMN_RANGE=(7,7)
             /PRIMARY
```

This region points to the last column of the first supplied string. No label is specified, so the default of supplied text is used. This is a primary region. See Example 1.1 for more samples.

START DIAGNOSTIC

START DIAGNOSTIC — Specifies the start of a diagnostic.

Format

START DIAGNOSTIC

Diagnostic Body

END DIAGNOSTIC

Command Parameter

Diagnostic Body

A diagnostic consists of a START DIAGNOSTIC command, one or more regions, one or more messages, and an END DIAGNOSTIC command.

Description

This command marks the start of a diagnostic module in the user format.

Example

See Example 1.1 for samples of the START DIAGNOSTIC command.

START MODULE

START MODULE — Specifies the start of a diagnostic module in the user format.

Format

```
START MODULE

Module Body

END MODULE
```

Command Parameter

Module Body

A module consists of a START MODULE command, zero or more diagnostics, and an END MODULE command.

Description

This command marks the start of a diagnostic module in the user format.

Example

See Example 1.1 for a sample of the START MODULE command.

Chapter 2. Using LSE Callable Routines

This chapter describes the LSE callable routines. It describes the purpose of the LSE callable routines, the parameters for a routine call, and the primary status returns. The parameter in the call syntax represents the object that you pass to an LSE routine. Each parameter description lists the data type and the passing mechanism for the object. The data types are standard OpenVMS data types. The passing mechanism indicates how the parameter list is interpreted.

2.1. LSE Callable Routines

Callable LSE routines make LSE accessible from within other languages and applications. You can call LSE from a program written in any language that generates calls using the OpenVMS Procedure Calling and Condition Handling Standard. You can also call LSE from OpenVMS utilities, for example, MAIL. With callable LSE, you can perform text-processing functions within your program.

Callable LSE consists of a set of callable routines that reside in the LSE shareable image, LSESHR.EXE. You access callable LSE by linking to this shareable image, which includes the callable interface routine names and constants. As with the DCL-level LSE interface, you can use files for input to and output from callable LSE. You can also write your own routines for processing file input, output, and messages.

You should be familiar with the following items:

- The OpenVMS Procedure Calling and Condition Handling Standard
- The OpenVMS Run-Time Library (RTL)
- The precise manner in which data types are represented on an OpenVMS system
- The method for calling routines written in a language other than the one you are using for the main program

The calling program must ensure that parameters passed to a called procedure, in this case LSE, are of the type and form that the LSE procedure accepts.

The LSE routines described in this reference return condition values indicating the routine's completion status. When comparing a returned condition value with a test value, use the LIB$MATCH routine from the RTL. Do not test the condition value as if it were a simple integer.

2.1.1. Two Interfaces to Callable LSE

There are two interfaces that you can use to access callable LSE: the simplified callable interface and the full callable interface.

Simplified Callable Interface

The easiest way to use callable LSE is to use the simplified callable interface. LSE provides two alternative routines in its simplified callable interface. These routines in turn call additional routines that do the following:

- Initialize LSE
- Provide the editor with the parameters necessary for its operation

- Control the editing session
- Perform error handling

When using the simplified callable interface, you can use the LSE$LSE routine to specify an OpenVMS command line for LSE, or you can call the LSE$EDIT routine to specify an input file and an output file. LSE$EDIT builds a command string that is then passed to the LSE$LSE routine. These two routines are described in detail in Section 2.2.

If your application parses information that is not related to the operation of LSE, make sure the application obtains and uses all non-LSE parse information before the application calls the simplified callable interface. The reason is that the simplified callable interface destroys all parse information obtained and stored before the simplified callable interface was called.

If your application calls the DECwindows version of LSE, the application may call LSE$EDIT or LSE $LSE a single time only. Also, the application may not call XtInitialize before calling LSE.

Full Callable Interface

The full callable interface consists of the main callable LSE routines and the LSE Utility routines.

To use the full callable interface, you have your program access the main callable LSE routines directly. These routines do the following:

- Initialize LSE (LSE$INITIALIZE)
- Execute LSE procedures (LSE$EXECUTE_INIFILE and LSE$EXECUTE_COMMAND)
- Give control to the editor (LSE$CONTROL)
- Terminate the editing session (LSE$CLEANUP)

When using the full callable interface, you must provide values for certain parameters. In some cases, the values you supply are actually addresses for additional routines. For example, when you call LSE $INITIALIZE, you must include the address of a routine that specifies initialization options. Depending on your particular application, you may also have to write additional routines. For example, you may need to write routines for performing file operations, handling errors, and otherwise controlling the editing session. Callable LSE provides utility routines that can perform some of these tasks for you. These utility routines do the following:

- Parse the OpenVMS command line and build the item list used for initializing LSE
- Handle file operations
- Output error messages
- Handle conditions

If your application calls the DECwindows version of LSE, the application may call LSE$INITIALIZE a single time only. Also, the application may not call XtInitialize before calling LSE.

Various topics relating to the full callable interface are described in the following sections:

- Section 2.3 briefly describes the interface.
- Section 2.3.1 describes the main callable LSE routines (LSE$INITIALIZE, LSE $EXECUTE_INIFILE, LSE$CONTROL, LSE$EXECUTE_COMMAND, and LSE$CLEANUP).

- Section 2.3.2 discusses additional routines that LSE provides for use with the full callable interface.
- Section 2.3.3 defines the requirements for routines that you can write for use with the full callable interface.

2.1.2. Shareable Image

Whether you use the simplified callable interface or the full callable interface, you access callable LSE by linking to the LSE shareable image, LSESHR.EXE. This image contains the routine names and constants available for use by an application. In addition, LSESHR.EXE provides the following symbols:

- TPU$GL_VERSION, the version of the shareable image
- TPU$GL_UPDATE, the update number of the shareable image
- TPU$_FACILITY, the DECTPU facility code

2.1.3. Passing Parameters to Callable LSE Routines

Parameters are passed to callable LSE by reference or descriptor, using standard OpenVMS mechanisms. When the parameter is a routine, the parameter is passed by descriptor as a bound procedure value (BPV) data type.

A bound procedure value is a two-longword entity in which the first longword contains the address of a procedure entry mask, and the second longword is the environment value. The environment value is determined in a language-specific manner when the original bound procedure value is generated. When the bound procedure is called, the calling program loads the second longword into R1.

Figure 2.1 shows the structure of a bound procedure value.

Figure 2.1. Bound Procedure Value

Name of Your Routine
Environment

2.1.4. Error Handling

When you use the simplified callable interface, LSE establishes its own condition handler, LSE $HANDLER, to handle all errors. When you use the full callable interface, there are two ways to handle errors:

- You can use LSE's default condition handler.
- You can write your own condition handler to process some of the errors, and you can call LSE $HANDLER to process the rest.

2.1.5. Return Values

All LSE condition codes are declared as universal symbols. Therefore, you automatically have access to these symbols when you link your program to the shareable image. The condition code values are returned in R0.

Additional information about condition codes is provided in the descriptions of callable LSE routines found in subsequent sections. This information is provided under the section heading Condition Values Returned and indicates the values that are returned when the default condition handler is established.

2.2. Simplified Callable Interface

The LSE simplified callable interface consists of two routines: LSE$LSE and LSE$EDIT. These entry points to DECTPU are useful for the following kinds of applications:

- Those able to specify all the editing parameters on a single command line
- Those that need to specify only an input file and an output file

If your application parses information that is not related to the operation of LSE, make sure the application gets, and uses, all non-LSE parse information before the application calls the simplified callable interface. The simplified callable interface destroys all parse information obtained and stored before the simplified callable interface was called.

The following example calls LSE$EDIT to edit text in the file INFILE.DAT and writes the result to OUTFILE.DAT. Note that the parameters to LSE$EDIT must be passed by descriptor.

```
/*
  Sample C program that calls LSE.  This program uses LSE$EDIT to
  provide the names of the input and output files.
*/
#include descrip

int return_status;

static $DESCRIPTOR (input_file, "infile.dat");
static $DESCRIPTOR (output_file, "outfile.dat");

main (argc, argv)
    int argc;
    char *argv[];
    {
    /*
      Call LSE to edit text in "infile.dat" and write the result
      to "outfile.dat".  Return the condition code from LSE as the
      status of this program.
    */

    return_status = LSE$EDIT (&input_file, &output_file);
    exit (return_status);
    }
```

The next example performs the same task as the previous example. This time, the LSE$LSE entry point is used. LSE$LSE accepts a single argument, which is a command string starting with the verb LSEDIT. The command string can contain all the qualifiers that are accepted by the LSEDIT command.

```
/*
  Sample C program that calls LSE.  This program uses LSE$LSE and
  specifies a command string
*/

#include descrip
int return_status;

static $DESCRIPTOR (command_prefix, "LSE/NOJOURNAL/NOCOMMAND/OUTPUT=");
static $DESCRIPTOR (input_file, "infile.dat");
```

```
static $DESCRIPTOR (output_file, "outfile.dat");
static $DESCRIPTOR (space_desc, " ");

char command_line [100];
static $DESCRIPTOR (command_desc, command_line);

main (argc, argv)
    int argc;
    char *argv[];
    {
    /*
      Build the command line for LSE.  Note that the command verb
      is LSEDIT.  The string we construct in the buffer command_line
      will be
        LSEDIT/NOJOURNAL/NOCOMMAND/OUTPUT=outfile.dat infile.dat
    */

    return_status = STR$CONCAT (&command_desc,
                                &command_prefix,
                                &output_file,
                                &space_desc,
                                &input_file);

    if (! return_status)        exit (return_status);
    /*
      Now call LSE to edit the file
    */

    return_status = LSE$LSE (&command_desc);
    exit (return_status);
    }
```

2.3. Full Callable Interface

The LSE full callable interface consists of a set of routines that you can use to perform the following tasks:

- Specify initialization parameters
- Control file input/output (I/O)
- Specify commands to be executed by LSE
- Control how conditions are handled

You can call the individual LSE routines that perform these functions from a user-written program.

This interface has two sets of routines: the main LSE callable routines and the LSE Utility routines. These LSE routines, and your own routines that pass parameters to the LSE routines, are the mechanism that your application uses to control LSE.

The following sections describe the main callable routines, how parameters are passed to these routines, the LSE Utility routines, and the requirements of user-written routines.

2.3.1. Main Callable LSE Utility Routines

This section describes the following callable LSE routines:

- LSE$INITIALIZE
- LSE$EXECUTE_INIFILE
- LSE$CONTROL
- LSE$EXECUTE_COMMAND
- LSE$CLEANUP

Note

Before calling any of these routines, you must establish LSE$HANDLER or provide your own condition handler. See the routine description of LSE$HANDLER in Section 2.5 for information about establishing a condition handler.

2.3.2. Other LSE Utility Routines

The full callable interface includes several utility routines for which you can provide parameters. Depending on your application, you may be able to use these routines rather than write your own routines. These LSE Utility routines and their descriptions follow:

- LSE$CLIPARSE—Parses a command line and builds the item list for LSE$INITIALIZE.
- LSE$PARSEINFO—Parses a command and builds an item list for LSE$INITIALIZE.
- LSE$FILEIO—Is the default file I/O routine.
- LSE$MESSAGE—Writes error messages and strings by using the built-in procedure MESSAGE.
- LSE$HANDLER—Is the default condition handler.
- LSE$CLOSE_TERMINAL—Closes DECTPU's channel to the terminal (and its associated mailbox) for the duration of a CALL_USER routine.

Note that LSE$CLIPARSE and LSE$PARSEINFO destroy the context maintained by the CLI$ routines for parsing commands.

2.3.3. User-Written Routines

This section defines the requirements for user-written routines. When these routines are passed to LSE, they must be passed as bound procedure values. (See Section 2.1.3 for a description of bound procedure values.) Depending on your application, you may have to write one or all of the following routines:

- Routine for initialization callback This is a routine that LSE$INITIALIZE calls to obtain values for initialization parameters. The initialization parameters are returned as an item list.
- Routine for file I/O This is a routine that handles file operations. Instead of writing your own file I/O routine, you can use the LSE$FILEIO utility routine. LSE does not use this routine for journal file operations or for operations performed by the built-in procedure SAVE.
- Routine for condition handling This is a routine that handles error conditions. Instead of writing your own condition handler, you can use the default condition handler, LSE$HANDLER.
- Routine for the built-in procedure CALL_USER This is a routine that is called by the built-in procedure CALL_USER. You can use this mechanism to cause your program to get control during an editing session.

2.4. Examples of Using LSE Routines

Example 2.1, Example 2.2, Example 2.3, and Example 2.4 use callable LSE. The examples are included here for illustrative purposes only; VSI does not assume responsibility for supporting these examples.

Example 2.1. Sample VAX BLISS Template for Callable DECTPU

```
MODULE file_io_example (MAIN = top_level,
                        ADDRESSING_MODE (EXTERNAL = GENERAL)) =

BEGIN

FORWARD ROUTINE
    top_level,                  ! Main routine of this example
    lse_init,                   ! Initialize LSE
    lse_io;                     ! File I/O routine for LSE

!
! Declare the stream data structure passed to the file I/O routine
!

MACRO
    stream_file_id =  0, 0, 32, 0 % ,    ! File ID
    stream_rat =      6, 0,  8, 0 % ,    ! Record attributes
    stream_rfm =      7, 0,  8, 0 % ,    ! Record format
    stream_file_nm =  8, 0,  0, 0 % ;    ! File name descriptor

!
! Declare the routines that would actually do the I/O.  These must be
 supplied
! in another module
!

EXTERNAL ROUTINE
    my_io_open,                 ! Routine to open a file
    my_io_close,                ! Routine to close a file
    my_io_get_record,           ! Routine to read a record
    my_io_put_record;           ! Routine to write a record
!
! Declare the LSE routines
!

EXTERNAL ROUTINE
    lse$fileio,                 ! LSE's internal file I/O routine
    lse$handler,                ! LSE's condition handler
    lse$initialize,             ! Initialize LSE
    lse$execute_inifile,        ! Execute the initial procedures
    lse$execute_command,        ! Execute an LSE statement
    lse$control,                ! Let user interact with LSE
    lse$cleanup;                ! Have LSE clean up after itself
!
! Declare the LSE literals
!

EXTERNAL LITERAL
    lse$k_close,                ! File I/O operation codes
    lse$k_close_delete,
```

```
    lse$k_open,
    lse$k_get,
    lse$k_put,
    lse$_access,                ! File access codes
    lse$k_io,
    lse$k_input,
    lse$k_output,
    lse$_calluser,             ! Item list entry codes
    lse$_fileio,
    lse$_outputfile,
    lse$_sectionfile,
    lse$_commandfile,
    lse$_filename,
    lse$_journalfile,
    lse$_options,
    lse$m_recover,              ! Mask for values in options bit vector
    lse$m_journal,
    lse$m_read,
    lse$m_command,
    lse$m_create,
    lse$m_section,
    lse$m_display,
    lse$m_output,
    lse$m_reset_terminal,       ! Masks for cleanup bit vector
    lse$m_kill_processes,
    lse$m_delete_exith,
    lse$m_last_time,
    tpu$_nofileaccess,          ! DECTPU status codes
    tpu$_openin,
    tpu$_inviocode,
    tpu$_failure,
    tpu$_closein,
    tpu$_closeout,
    tpu$_readerr,
    tpu$_writeerr,
    tpu$_success;

ROUTINE top_level =    BEGIN
!++
! Main entry point of your program
!-
! Your_initialization_routine must be declared as a BPV
    LOCAL
        initialize_bpv: VECTOR [2],
        status,
        cleanup_flags;
    !
    ! First establish the condition handler
    !    ENABLE
        lse$handler ();
    !
    ! Initialize the editing session by passing LSE$INITIALIZE the address
 of
    ! the bound procedure value that defines the routine that LSE is
    ! to call to return the initialization item list.
    !
    initialize_bpv [0] = lse_init;
    initialize_bpv [1] = 0;
```

```
    lse$initialize (initialize_bpv);
    !
    ! Call LSE to execute the contents of the command file, the debug file,
    ! or the LSE$INIT_PROCEDURE from the section file.
    !
    lse$execute_inifile();
    !
    ! Let LSE take over.
    !
    lse$control();
    !
    ! Have LSE clean up after itself.
    !
    cleanup_flags = lse$m_reset_terminal OR     ! Reset the terminal
                    lse$m_kill_processes OR     ! Delete subprocesses
                    lse$m_delete_exith OR       ! Delete the exit handler
                    lse$m_last_time;            ! Last time calling the
 editor
    lse$cleanup (cleanup_flags);
    RETURN tpu$_success;
    END;

ROUTINE lse_init =
    BEGIN
    !
    ! Allocate the storage block needed to pass the file I/O routine as a
    ! bound procedure variable as well as the bit vector for the
 initialization
    ! options.
    !
    OWN
        file_io_bpv: VECTOR [2, LONG]
                     INITIAL (LSE_IO, 0),
        options;
    !
    ! These macros define the file names passed to LSE.
    !
    MACRO
        out_file = 'OUTPUT.TPU' % ,
        com_file = 'LSE$COMMAND' % ,
        sec_file = 'LSE$SECTION' % ,
        inp_file = 'FILE.TPU' % ;
    !
    ! Create the item list to pass to LSE.  Each item list entry consists
 of
    ! two words that specify the size of the item and its code, the address
 of
    ! the buffer containing the data, and a longword to receive a result
 (always
    ! zero, since LSE does not return any result values in the item list).
    !
    !               +--------------------------------+
    !               | Item Code      | Item Length   |
    !               +----------------+---------------+
    !               |        Buffer Address          |
    !               +--------------------------------+
    !               |    Return Address (always 0)   |
    !               +--------------------------------+
```

```
    !
    ! Remember that the item list is always terminated with a longword
 containing
    ! a zero.
    !
    BIND
        item_list = UPLIT BYTE (
            WORD (4),                       ! Options bit vector
            WORD (lse$_options),
            LONG (options),
            LONG (0),

            WORD (4),                       ! File I/O routine
            WORD (lse$_fileio),
            LONG (file_io_bpv),
            LONG (0),

            WORD (%CHARCOUNT (out_file)),    ! Output file
            WORD (lse$_outputfile),
            LONG (UPLIT (%ASCII out_file)),
            LONG (0),

            WORD (%CHARCOUNT (com_file)),    ! Command file
            WORD (lse$_commandfile),
            LONG (UPLIT (%ASCII com_file)),
            LONG (0),

            WORD (%CHARCOUNT (sec_file)),   ! Section file
            WORD (lse$_sectionfile),
            LONG (UPLIT (%ASCII sec_file)),
            LONG (0),

            WORD (%CHARCOUNT (inp_file)),    ! Input file
            WORD (lse$_filename),
            LONG (UPLIT (%ASCII inp_file)),
            LONG (0),
            LONG (0));                      ! Terminating longword of 0
    !
    ! Initialize the options bitvector
    !
    options = lse$m_display OR              ! We have a display
              lse$m_section OR              ! We have a section file
              lse$m_create OR               ! Create a new file if one does
 not
                                            !   exist
              lse$m_command OR              ! We have a section file
              lse$m_output;                 ! We supplied an output file
 spec
    !
    ! Return the item list as the value of this routine for LSE to
 interpret.
    !
    RETURN item_list;
    END;                                    ! End of routine lse_init

ROUTINE lse_io (p_opcode, stream: REF BLOCK [ ,byte], data) =
!
! This routine determines how to process a TPU I/O request.
```

```
!
    BEGIN
    LOCAL
        status;
!
! Is this one of ours, or do we pass it to LSE's file I/O routines?
!
    IF (..p_opcode NEQ lse$k_open) AND (.stream [stream_file_id] GTR 511)
    THEN
        RETURN lse$fileio (.p_opcode, .stream, .data);
!
! Either we are opening the file, or we know it is one of ours.
! Call the appropriate routine (not shown in this example).
!
    SELECTONE ..p_opcode OF
        SET
        [lse$k_open]:
            status = my_io_open (.stream, .data);
        [lse$k_close, lse$k_close_delete]:
            status = my_io_close (.stream, .data);
        [lse$k_get]:
            status = my_io_get_record (.stream, .data);
        [lse$k_put]:
            status = my_io_put_record (.stream, .data);
        [OTHERWISE]:
            status = tpu$_failure;
        TES;
    RETURN .status;
    END;                                        ! End of routine LSE_IO
END                                             ! End Module
 file_io_example
```

Example 2.2. Normal LSE Setup in VSI Fortran

```
C       A sample FORTRAN program that calls LSE to act
C       normally, using the programmable interface.
C
C       IMPLICIT NONE
        INTEGER*4       CLEAN_OPT       !Options for cleanup routine.
        INTEGER*4       STATUS          !Return status from LSE routines.
        INTEGER*4       BPV_PARSE(2)    !Set up a Bound Procedure Value.
        INTEGER*4       LOC_PARSE       !A local function call.
C
C       Declare the LSE functions.
C
        INTEGER*4       LSE$CONTROL
        INTEGER*4       LSE$CLEANUP
        INTEGER*4       LSE$EXECUTE_INIFILE
        INTEGER*4       LSE$INITIALIZE
        INTEGER*4       LSE$CLIPARSE
C
C       Declare a local copy to hold the values of LSE cleanup variables.
C
        INTEGER*4       RESET_TERMINAL
        INTEGER*4       DELETE_JOURNAL
        INTEGER*4       DELETE_BUFFERS,DELETE_WINDOWS
        INTEGER*4       DELETE_EXITH,EXECUTE_PROC
        INTEGER*4       PRUNE_CACHE,KILL_PROCESSES
```

```
        INTEGER*4       CLOSE_SECTION
C
C       Declare the LSE functions used as external.
C
        EXTERNAL        LSE$HANDLER
        EXTERNAL        LSE$CLIPARSE
        EXTERNAL        TPU$_SUCCESS    !External error message.
        EXTERNAL        LOC_PARSE       !User-supplied routine to call LSE
$CLIPARSE.
C
C       Declare the LSE cleanup variables as external.
C       These are the external literals that hold the
C       value of the options.
C
        EXTERNAL        LSE$M_RESET_TERMINAL
        EXTERNAL        LSE$M_DELETE_JOURNAL
        EXTERNAL        LSE$M_DELETE_BUFFERS,LSE$M_DELETE_WINDOWS
        EXTERNAL        LSE$M_DELETE_EXITH,LSE$M_EXECUTE_PROC
        EXTERNAL        LSE$M_PRUNE_CACHE,LSE$M_KILL_PROCESSES
100     CALL LIB$ESTABLISH ( LSE$HANDLER )       !Establish the condition
 handler.
C
C       Set up the Bound Procedure Value for the call to LSE$INITIALIZE.
C
        BPV_PARSE( 1 ) = %LOC( LOC_PARSE )
        BPV_PARSE( 2 ) = 0
C
C       Call the LSE initialization routine to do some setup work.
C
        STATUS = LSE$INITIALIZE ( BPV_PARSE )
C
C       Check the status.  If it is not a success, then signal the error.
C
        IF ( STATUS .NE. %LOC ( TPU$_SUCCESS ) ) THEN
                CALL LIB$SIGNAL( %VAL( STATUS ) )
                GOTO 9999
        ENDIF
C
C       Execute the LSE$_ init files and also a command file if it
C       was specified in the command line call to LSE.
C
        STATUS = LSE$EXECUTE_INIFILE ( )
        IF ( STATUS .NE. %LOC ( TPU$_SUCCESS ) ) THEN !Make sure everything
 is ok.
                CALL LIB$SIGNAL( %VAL( STATUS ) )
                GOTO 9999
        ENDIF
C
C       Invoke LSE as it normally would appear.
C
        STATUS = LSE$CONTROL ( )        !Call LSE.
        IF ( STATUS .NE. %LOC ( TPU$_SUCCESS ) ) THEN !Make sure everything
 is ok.
                CALL LIB$SIGNAL( %VAL( STATUS ) )
C               GOTO 9999
        ENDIF
C
```

```
C       Get the value of the option from the external literals.  In
 FORTRAN, you
C       cannot use external literals directly so you must first get the
 value
C       of the literal from its external location.  Here we are getting the
C       values of the options that we want to use in the call to LSE
$CLEANUP.
C
        DELETE_JOURNAL  = %LOC ( LSE$M_DELETE_JOURNAL )
        DELETE_EXITH    = %LOC ( LSE$M_DELETE_EXITH )
        DELETE_BUFFERS  = %LOC ( LSE$M_DELETE_BUFFERS )
        DELETE_WINDOWS  = %LOC ( LSE$M_DELETE_WINDOWS )
        EXECUTE_PROC    = %LOC ( LSE$M_EXECUTE_PROC )
        RESET_TERMINAL  = %LOC ( LSE$M_RESET_TERMINAL )
        KILL_PROCESSES  = %LOC ( LSE$M_KILL_PROCESSES )
        CLOSE_SECTION   = %LOC ( LSE$M_CLOSE_SECTION )
C
C       Now that we have the local copies of the variables we can do the
C       logical OR to set the multiple options that we need.
C
        CLEAN_OPT = DELETE_JOURNAL .OR. DELETE_EXITH .OR.
        1       DELETE_BUFFERS .OR. DELETE_WINDOWS .OR. EXECUTE_PROC
        1       .OR. RESET_TERMINAL .OR. KILL_PROCESSES .OR. CLOSE_SECTION
C
C       Do the necessary cleanup.
C       LSE$CLEANUP wants the address of the flags as the parameter so
C       pass the %LOC of CLEAN_OPT, which is the address of the variable.
        STATUS = LSE$CLEANUP ( %LOC ( CLEAN_OPT ) )
        IF ( STATUS .NE. %LOC (TPU$_SUCCESS) ) THEN
                CALL LIB$SIGNAL( %VAL(STATUS) )
        ENDIF
9999    CALL LIB$REVERT         !Go back to normal processing - handlers.
        STOP
        END
C
C
        INTEGER*4  FUNCTION LOC_PARSE
        INTEGER*4       BPV(2)          !A local Bound Procedure Value
        CHARACTER*12    EDIT_COMM       !A command line to send to LSE
$CLIPARSE
C
C       Declare the LSE functions used.
C
        INTEGER*4       LSE$FILEIO
        INTEGER*4       LSE$CLIPARSE
C
C       Declare this routine as external because it is never called
 directly and
C       we need to tell FORTRAN that it is a function and not a variable.
C
        EXTERNAL        LSE$FILEIO        BPV(1) = %LOC(LSE$FILEIO)       !
Set up the bound procedure value.
        BPV(2) = 0
        EDIT_COMM(1:12) = 'LSE TEST.TXT'
C
C       Parse the command line and build the item list for LSE$INITIALIZE.
C
9999     LOC_PARSE = LSE$CLIPARSE (EDIT_COMM, BPV , 0)
```

```
        RETURN
        END
```

Example 2.3. Building a Callback Item List with VSI Fortran

```
       PROGRAM  TEST_LSEC
       IMPLICIT NONECC
       Define the expected LSE return statuses.
C
       EXTERNAL        TPU$_SUCCESS
       EXTERNAL        TPU$_QUITTING
       EXTERNAL        TPU$_EXITING
C
C
       Declare the LSE routines and symbols used.
C
       EXTERNAL        LSE$M_DELETE_CONTEXT
       EXTERNAL        LSE$HANDLER
       INTEGER*4       LSE$M_DELETE_CONTEXT
       INTEGER*4       LSE$INITIALIZE
       INTEGER*4       LSE$EXECUTE_INIFILE
       INTEGER*4       LSE$CONTROL
       INTEGER*4       LSE$CLEANUP
C
C      Use LIB$MATCH_COND to compare condition codes.
C
       INTEGER*4
       LIB$MATCH_COND
C
C       Declare the external callback routine.
C
       EXTERNAL        LSE_STARTUP       ! The LSE setup function.
       INTEGER*4       LSE_STARTUP
       INTEGER*4       BPV(2)            ! Set up a Bound Procedure Value.
C
C      Declare the functions used for working with the condition handler.
C
       INTEGER*4       LIB$ESTABLISH
       INTEGER*4       LIB$REVERT
C
C      Local flags and indices
C
       INTEGER*4       CLEANUP_FLAG       ! Flag(s) for LSE cleanup.
       INTEGER*4       RET_STATUS
       INTEGER*4       MATCH_STATUS
C
C      Initializations
C
       RET_STATUS       = 0
       CLEANUP_FLAG     = %LOC(LSE$M_DELETE_CONTEXT)
C
C      Establish the default LSE condition handler.
C
       CALL LIB$ESTABLISH(%REF(LSE$HANDLER))
C
C      Set up the Bound Procedure Value for the initialization callback.
C
       BPV(1)  =  %LOC (LSE_STARTUP)       BPV(2)  =  0
```

```
C
C       Call the LSE procedure for initialization.
C
        RET_STATUS = LSE$INITIALIZE(BPV)
        IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
        CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIFCC      Execute the LSE initialization file.
C
        RET_STATUS = LSE$EXECUTE_INIFILE()
        IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
        CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIF
C
C       Pass control to LSE.
C
        RET_STATUS = LSE$CONTROL()
C
C       Test for valid exit condition codes.  You must use LIB$MATCH_COND
C       because the severity of TPU$_QUITTING can be set by the LSE
C       application.
C
        MATCH_STATUS = LIB$MATCH_COND (RET_STATUS, %LOC (TPU$_QUITTING),
        1                                          %LOC (TPU$_EXITING))
        IF (MATCH_STATUS .EQ. 0) THEN       CALL LIB$SIGNAL
 (%VAL(RET_STATUS))
        ENDIF
C
C       Clean up after processing.
C
        RET_STATUS = LSE$CLEANUP(%REF(CLEANUP_FLAG))
        IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
        CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIF
C
C       Set the condition handler back to the default.
C
        RET_STATUS = LIB$REVERT()
        END
        INTEGER*4 FUNCTION LSE_STARTUP
        IMPLICIT NONE
        INTEGER*4
        OPTION_MASK
        ! Temporary variable for LSE
        CHARACTER*44
        SECTION_NAME   ! Temporary variable for LSE
C
C       External LSE routines and symbols.
C
        EXTERNAL        LSE$K_OPTIONS
        EXTERNAL        LSE$M_READ
        EXTERNAL        LSE$M_SECTION
        EXTERNAL        LSE$M_DISPLAY
        EXTERNAL        LSE$K_SECTIONFILE
        EXTERNAL        LSE$K_FILEIO
        EXTERNAL        LSE$FILEIO
        INTEGER*4       LSE$FILEIO
C
C       The bound procedure value used for setting up the file I/O routine.
```

```
C
      INTEGER*4       BPV(2)
C
C
      Define the structure of the item list defined for the callback.
C
      STRUCTURE /CALLBACK/
      INTEGER*2       BUFFER_LENGTH
      INTEGER*2       ITEM_CODE
      INTEGER*4       BUFFER_ADDRESS
      INTEGER*4       RETURN_ADDRESS
      END STRUCTURE
C
C     There are a total of four items in the item list.
C
      RECORD /CALLBACK/ CALLBACK (4)
C
C     Make sure it is not optimized!
C
      VOLATILE /CALLBACK/
C
C
      Define the options we want to use in the LSE session.
C
      OPTION_MASK = %LOC(LSE$M_SECTION) .OR. %LOC(LSE$M_READ)
      1
      .OR. %LOC(LSE$M_DISPLAY)
C
C     Define the name of the initialization section file.
C
      SECTION_NAME = 'LSE$SECTION'
C
C     Set up the required I/O routine.  Use the LSE default.
C
      BPV(1) = %LOC(LSE$FILEIO)
      BPV(2) = 0
C
C     Build the callback item list.
C
C     Set up the edit session options.
C
      CALLBACK(1).ITEM_CODE = %LOC(LSE$K_OPTIONS)
      CALLBACK(1).BUFFER_ADDRESS = %LOC(OPTION_MASK)
      CALLBACK(1).BUFFER_LENGTH = 4
      CALLBACK(1).RETURN_ADDRESS = 0
C
C     Identify the section file to be used.
C
      CALLBACK(2).ITEM_CODE = %LOC(LSE$K_SECTIONFILE)
      CALLBACK(2).BUFFER_ADDRESS = %LOC(SECTION_NAME)
      CALLBACK(2).BUFFER_LENGTH = LEN(SECTION_NAME)
      CALLBACK(2).RETURN_ADDRESS = 0
C
C     Set up the I/O handler.
C
      CALLBACK(3).ITEM_CODE = %LOC(LSE$K_FILEIO)
      CALLBACK(3).BUFFER_ADDRESS = %LOC(BPV)
      CALLBACK(3).BUFFER_LENGTH = 4
```

```
        CALLBACK(3).RETURN_ADDRESS = 0
C
C       End the item list with zeros to indicate we are finished.
C
        CALLBACK(4).ITEM_CODE = 0
        CALLBACK(4).BUFFER_ADDRESS = 0
        CALLBACK(4).BUFFER_LENGTH = 0
        CALLBACK(4).RETURN_ADDRESS = 0
C
C       Return the address of the item list.
C
        LSE_STARTUP = %LOC(CALLBACK)
        RETURN
        END
```

Example 2.4. Specifying a User-Written File I/O Routine in VSI C

```
/*
Simple example of a C program to invoke LSE.  This program provides its
own FILEIO routine instead of using the one provided by LSE.
*/

#include descrip
#include stdio

/* Data structures needed */

struct bpv_arg                  /* Bound procedure value */
    {
    int *routine_add ;          /* Pointer to routine */
    int env ;                   /* Environment pointer */
    } ;
struct item_list_entry          /* Item list data structure */
    {
    short int buffer_length;
    /* Buffer length */
    short int item_code;        /* Item code */
    int *buffer_add;            /* Buffer address */
    int *return_len_add;        /* Return address */
    } ;
struct stream_type
    {
    int ident;                  /* Stream id */
    short int alloc;            /* File size */
    short int flags;            /* File record attributes/format */
    short int length;           /* Resultant file name length */
    short int stuff;            /* File name descriptor class & type */
    int nam_add;                /* File name descriptor text pointer */
    } ;
globalvalue tpu$_success;       /* TPU Success code */
globalvalue tpu$_quitting;      /* Exit code defined by TPU */
globalvalue                     /* Cleanup codes defined by LSE */
    lse$m_delete_journal, lse$m_delete_exith,
    lse$m_delete_buffers, lse$m_delete_windows, lse$m_delete_cache,
    lse$m_prune_cache, lse$m_execute_file, lse$m_execute_proc,
    lse$m_delete_context, lse$m_reset_terminal, lse$m_kill_processes,
    lse$m_close_section, lse$m_delete_others, lse$m_last_time;
globalvalue                     /* Item codes for item list entries */
```

```
    lse$k_fileio, lse$k_options, lse$k_sectionfile,
    lse$k_commandfile ;
globalvalue                     /* Option codes for option item */
    lse$m_display, lse$m_section, lse$m_command, lse$m_create ;
globalvalue                     /* Possible item codes in item list */
    lse$_access, lse$_filename, lse$_defaultfile,
    lse$_relatedfile, lse$_record_attr, lse$_maximize_ver,
    lse$_flush, lse$_filesize;
globalvalue                     /* Possible access types for lse$_access */
    lse$k_io, lse$k_input, lse$k_output;
globalvalue                     /* RMS File Not Found message code */
    rms$_fnf;
globalvalue                     /* FILEIO routine functions */
    lse$k_open, lse$k_close, lse$k_close_delete,
    lse$k_get, lse$k_put;
int lib$establish ();           /* RTL routine to establish an event
 handler */
int lse$cleanup ();             /* LSE routine to free resources used */
int lse$control ();             /* LSE routine to invoke the editor */
int lse$execute_inifile ();     /* LSE routine to execute initialization
 code */
int lse$handler ();             /* LSE signal handling routine */
int lse$initialize ();          /* LSE routine to initialize the editor */

/*
   This function opens a file for either read or write access, based on
   the item list passed as the data parameter.  Note that a full
 implementation
   of the file open routine would have to handle the default file, related
   file, record attribute, maximize version, flush and file size item code
   properly.
 */
open_file (data, stream)

int *data;
struct stream_type *stream;
{
    struct item_list_entry *item;
    char *access;               /* File access type */
    char filename[256];         /* Max file specification size */
    FILE *fopen();
    /* Process the item list */
    item = data;
    while (item->item_code != 0 && item->buffer_length != 0)
        {
        if (item->item_code == lse$_access)
            {
            if (item->buffer_add == lse$k_io) access = "r+";
            else if (item->buffer_add == lse$k_input) access = "r";
            else if (item->buffer_add == lse$k_output) access = "w";
            }
        else if (item->item_code == lse$_filename)
            {
            strncpy (filename, item->buffer_add, item->buffer_length);
            filename [item->buffer_length] = 0;
            lib$scopy_r_dx (&item->buffer_length, item->buffer_add,
                                                    &stream->length);
            }
```

```
        else if (item->item_code == lse$_defaultfile)
            {
                        /* Add code to handle default file  */
            }
                        /* spec here                        */
        else if (item->item_code == lse$_relatedfile)
            {
                        /* Add code to handle related       */
            }
                        /* file spec here                   */
        else if (item->item_code == lse$_record_attr)
            {
                        /* Add code to handle record        */
            }
                        /* attributes for creating files    */
        else if (item->item_code == lse$_maximize_ver)
            {
                        /* Add code to maximize version     */
            }
                        /* number with existing file here   */
        else if (item->item_code == lse$_flush)
            {
                        /* Add code to cause each record    */
            }
                        /* to be flushed to disk as written */
        else if (item->item_code == lse$_filesize)
            {
                        /* Add code to handle specification */
            }
                        /* of initial file allocation here  */
        ++item;
         /* get next item */
        }
    stream->ident = fopen(filename,access);
    if (stream->ident != 0)
        return tpu$_success;
    else
        return rms$_fnf;}
/*
  This procedure closes a file.
 */
close_file (data,stream)
struct stream_type *stream;

{
    close(stream->ident);
    return tpu$_success;
}
/*
  This procedure reads a line from a file.
 */
read_line(data,stream)
struct dsc$descriptor *data;
struct stream_type *stream;
{
    char textline[984];                 /* Max line size for TPU records */
    int len;
    globalvalue rms$_eof;               /* RMS End-Of-File code */
```

```
    if (fgets(textline,984,stream->ident) == NULL)
        return rms$_eof;
    else
        {
        len = strlen(textline);
        if (len > 0)            len = len - 1;
        return lib$scopy_r_dx (&len, textline, data);
        }
}
/*
  This procedure writes a line to a file.
 */
write_line(data,stream)
struct dsc$descriptor *data;
struct stream_type *stream;
{
    char textline[984];                 /* Max line size for TPU records */
    strncpy (textline, data->dsc$a_pointer, data->dsc$w_length);
    textline [data->dsc$w_length] = 0;
    fputs(textline,stream->ident);
    fputs("\n",stream->ident);
    return tpu$_success;
}
/*
   This procedure will handle I/O for LSE.
 */
fileio(code,stream,data)
int *code;
int *stream;
int *data;
{
    int status;
/* Dispatch based on code type.  Note that a full implementation of the
  */
/* file I/O routines would have to handle the close and delete code
 properly */
/* instead of simply closing the file.
   */
    if (*code == lse$k_open)                    /* Initial access to file
 */
        status = open_file (data,stream);
    else if (*code == lse$k_close)              /* End access to file */
        status = close_file (data,stream);
    else if (*code == lse$k_close_delete)       /* Treat same as close */
        status = close_file (data,stream);
    else if (*code == lse$k_get)                /* Read a record from a
 file */
        status = read_line (data,stream);
    else if (*code == lse$k_put)                /* Write a record to a file
 */
        status = write_line (data,stream);
    else
        {                                       /* Who knows what we got?
 */
        status = tpu$_success;
        printf ("Bad FILEIO I/O function requested");
        }
    return status;
```

```
}
/*
   This procedure formats the initialization item list and returns it as
   a return value.
 */
callrout()
{
    static struct bpv_arg add_block =
        { fileio, 0 } ;         /* BPV for fileio routine */
    int options ;    char *section_name = "LSE$SECTION";
    static struct item_list_entry arg[] =
        {/* length code              buffer add return add */
               { 4,lse$k_fileio,     0,         0 },
               { 4,lse$k_options,    0,         0 },
               { 0,lse$k_sectionfile,0,         0 },
               { 0,0,                0,         0 }
        };
    /* Setup file I/O routine item entry */
    arg[0].buffer_add = &add_block;

    /* Setup options item entry.  Leave journaling off. */
    options = lse$m_display | lse$m_section;
    arg[1].buffer_add = &options;

    /* Setup section file name */
    arg[2].buffer_length = strlen(section_name);
    arg[2].buffer_add = section_name;
    return arg;
}

/*
   Main program.  Initializes LSE, then passes control to it.
 */main()

{
    int return_status ;
    int cleanup_options;
    struct bpv_arg add_block;
/* Establish as condition handler the normal LSE handler */
    lib$establish(lse$handler);
/* Setup a BPV to point to the callback routine */
    add_block.routine_add = callrout ;
    add_block.env = 0;
/* Do the initialize of LSE */
    return_status = lse$initialize(&add_block);
    if (!return_status)
        exit(return_status);
/* Have LSE execute the procedure LSE$INIT_PROCEDURE from the section file
 */
/* and then compile and execute the code from the command file */
    return_status = lse$execute_inifile();
    if (!return_status)
        exit (return_status);
/*
 Turn control over to LSE
 */
    return_status = lse$control ();
    if (!return_status)
```

```
        exit(return_status);
/*
 Now clean up.
 */
    cleanup_options = lse$m_last_time | lse$m_delete_context;
    return_status = lse$cleanup (&cleanup_options);
    exit (return_status);
    printf("Experiment complete");
}
```

2.5. LSE Routines

The following pages describe the individual LSE routines.

In this section, VMS Usage refers to OpenVMS Alpha and OpenVMS I64 usage.

LSE$CLEANUP

LSE$CLEANUP — Cleans up internal data structures, frees memory, and restores terminals to their initial state. This is the final routine called in each interaction with LSE.

Format

LSE$CLEANUP flags

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed in the Condition Value Returned section.

Argument

flags

VMS Usage:	mask_longword
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

Flags (or mask) defining the cleanup options. The *flags* argument is the address of a longword bit mask defining the cleanup options or the address of a 32-bit mask defining the cleanup options. This mask is the logical OR of the flag bits you want to set. LSE$V . . . indicates a bit item and LSE$M . . . indicates a mask. Table 2.1 describes the various cleanup options.

Table 2.1. LSE$CLEANUP Options

Symbol [1]	Function
LSE$M_DELETE_JOURNAL	Closes and deletes the journal file if it is open.
LSE$M_DELETE_EXITH	Deletes LSE's exit handler.
LSE$M_DELETE_BUFFERS	Deletes all text buffers. If this is not the last time you are calling LSE, then all variables referring to these data structures are reset as if by the built-in procedure DELETE. If a buffer is deleted, then all ranges and markers within that buffer, and any subprocesses using that buffer, are also deleted.
LSE$M_DELETE_WINDOWS	Deletes all windows. If this is not the last time you are calling LSE, then all variables referring to these data structures are reset as if by the built-in procedure DELETE.
LSE$M_DELETE_CACHE	Deletes the virtual file manager's data structures and caches. If this deletion is requested, then all buffers are also deleted. If the cache is deleted, the initialization routine has to reinitialize the virtual file manager the next time it is called.
LSE$M_PRUNE_CACHE	Frees up any virtual file manager caches that have no pages allocated to buffers. This frees up any caches that may have been created during the session but that are no longer needed.
LSE$M_EXECUTE_FILE	Reexecutes the command file if LSE $EXECUTE_INIFILE is called again. You must set this bit if you plan to specify a new file name for the command file. This option is used in conjunction with the option bit passed to LSE $INITIALIZE indicating the presence of the /COMMAND qualifier.
LSE$M_EXECUTE_PROC	Looks up LSE$INIT_PROCEDURE and executes it the next time LSE$EXECUTE_INIFILE is called.
LSE$M_DELETE_CONTEXT	Deletes the entire context of LSE. If this option is specified, then all other options are implied, except for executing the initialization file and initialization procedure.
LSE$M_RESET_TERMINAL	Resets the terminal to the state it was in upon entry to LSE. The terminal mailbox and all windows are deleted. If the terminal is reset, then it is reinitialized the next time LSE$INITIALIZE is called.
LSE$M_KILL_PROCESSES	Deletes all subprocesses created during the session.
LSE$M_CLOSE_SECTION [2]	Closes the section file and releases the associated memory. All buffers, windows, and processes are deleted. The cache is purged and the flags are set for reexecution of the initialization file and initialization procedure. If the section is closed

Symbol [1]	Function
	and if the option bit indicates the presence of the /SECTION qualifier, then the next call to LSE $INITIALIZE attempts a new restore operation.
LSE$M_DELETE_OTHERS	Deletes all miscellaneous preallocated data structures, whose memory is reallocated the next time LSE$INITIALIZE is called.
LSE$M_LAST_TIME	This bit should be set only when you are calling LSE for the last time. Note that if you set this bit and then recall LSE, the results are unpredictable.

[1]The prefix can be LSE$M_ or LSE$V_. LSE$M_ denotes a mask corresponding to the specific field in which the bit is set. LSE$V_ is a bit number.

[2]Using the simplified callable interface does not set LSE$_CLOSE_SECTION. Therefore, you can make multiple calls to LSE$LSE without having to open and close the section file on each call.

Condition Value Returned

TPU$_SUCCESS	Normal successful completion.

Description

This routine is the final routine called in each interaction with LSE. It tells LSE to clean up its internal data structures and to prepare for additional invocations. You can control what this routine resets by setting or clearing the flags described previously.

When you finish with LSE, call this routine to free the memory and restore the characteristics of the terminal to their original settings.

If you intend to exit after calling LSE$CLEANUP, do not delete the data structures; the OpenVMS system, does this automatically. Allowing your OpenVMS system to delete the structures improves the performance of your program.

Notes

1. When you use the simplified interface, LSE automatically sets the following flags:

   - LSE$V_RESET_TERMINAL
   - LSE$V_DELETE_BUFFERS
   - LSE$V_DELETE_JOURNAL
   - LSE$V_DELETE_WINDOWS
   - LSE$V_DELETE_EXITH
   - LSE$V_EXECUTE_PROC
   - LSE$V_EXECUTE_FILE
   - LSE$V_PRUNE_CACHE
   - LSE$V_KILL_PROCESSES

2. If this routine does not return a success status, no other calls to the editor should be made.

LSE$CLIPARSE

LSE$CLIPARSE — Parses a command line and builds the item list for LSE$INITIALIZE. It calls CLI $DCL_PARSE to establish a command table and a command to parse. It then calls LSE$PARSEINFO to build an item list for LSE$INITIALIZE. If your application parses information that is not related to the operation of LSE, make sure the application gets, and uses, all non-LSE parse information before the application calls LSE$CLIPARSE. LSE$CLIPARSE destroys all parse information obtained and stored before LSE$CLIPARSE was called.

Format

```
LSE$CLIPARSE string, fileio, call_user
```

Returns

VMS Usage:	item_list
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

Arguments

string

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by descriptor

Command line. The **string** argument is the address of a descriptor of an LSE command.

fileio

VMS Usage:	vector_longword_unsigned
type:	bound procedure value
access:	read-only
mechanism:	by descriptor

File I/O routine. The **fileio** argument is the address of a descriptor of a file I/O routine.

call_user

VMS Usage:	vector_longword_unsigned
type:	bound procedure value
access:	read-only
mechanism:	by descriptor

Call-user routine. The **call_user** argument is the address of a descriptor of a call-user routine.

LSE$CLOSE_TERMINAL

LSE$CLOSE_TERMINAL — Closes the LSE channel to the terminal.

Format

LSE$CLOSE_TERMINAL

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed in the Condition Value Returned section.

Condition Value Returned

TPU$_SUCCESS	Normal successful completion.

Description

This routine is used with the built-in procedure CALL_USER and its associated call-user routine to control LSE access to the terminal. When a call-user routine invokes LSE$CLOSE_TERMINAL, LSE closes its channel to the terminal and the channel of LSE's associated mailbox.

When the call-user routine returns control to it, LSE automatically reopens a channel to the terminal and redisplays the visible windows.

A call-user routine can use LSE$CLOSE_TERMINAL at any point in the program and as many times as necessary. If the terminal is already closed to LSE when LSE$CLOSE_TERMINAL is used, the call is ignored.

LSE$CONTROL

LSE$CONTROL — Is the main processing routine of LSE. It is responsible for reading the text and commands and executing them. When you call this routine (after calling LSE$INITIALIZE), control is turned over to LSE.

Format

LSE$CONTROL (last-line, last-char, out-file)

Returns

VMS Usage:	cond_value

type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value inR0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

last-line

VMS Usage:	integer
type:	longword (signed)
access:	write-only
mechanism:	by reference

A signed longword to receive the final position in buffer LSE$MAIN_BUFFER. The first line in the file is line 1.

last-char

VMS Usage:	integer
type:	longword (signed)
access:	write-only
mechanism:	by reference

A signed longword to receive the final column position in buffer LSE$MAIN_BUFFER. The first column on a line is column 1.

out-file

VMS Usage:	char-string
type:	character string
access:	write-only
mechanism:	by descriptor

A character string that receives the file specification of the file to which the buffer, pointed to by the DECTPU variable LSE$MAIN_BUFFER, was written on exit. If LSE$MAIN_BUFFER was not written to its designated output file, the **out-file** is the file specification of the file read into LSE $MAIN_BUFFER. If you enter the QUIT command, this specification is the null string. You can use this information to return to this file during a subsequent edit. LSE uses STR$COPY to fill in this string.

The **last-line** and **last-char** arguments together describe the last current position in the buffer pointed to by the DECTPU variable LSE$MAIN_BUFFER. You can use this information to return to this file position in a subsequent edit.

Condition Value Returned

TPU$_EXITING	A result of EXIT (when the default condition handler is enabled).

TPU$_QUITTING	A result of QUIT (when the default condition handler is enabled).
TPU$_RECOVERFAIL	A recovery operation was terminated abnormally.

Description

This routine controls the edit session. It is responsible for reading the text and commands and executing them. Windows on the screen are updated to reflect the edits that are performed.

LSE$EDIT

LSE$EDIT — Builds a command string from its parameters and passes it to the LSE$LSE routine. LSE $EDIT is another entry point to the LSE simplified callable interface.

Format

```
LSE$EDIT input, output
```

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

input

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by descriptor

Input file name. The **input** argument is the address of a descriptor of a file specification.

output

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by descriptor

Output file name. The **output** argument is the address of a descriptor of an output file specification. It is used with the /OUTPUT command qualifier.

Condition Value Returned

The LSE$EDIT routine returns any value returned by LSE$LSE.

Description

This routine builds a command string and passes it to LSE$LSE. If the length of the output string is greater than 0, you can include it in the command line by using the /OUTPUT qualifier, as follows:

```
LSEDIT [/OUTPUT= output] input
```

If your application parses information that is not related to the operation of LSE, make sure the application gets, and uses, all non-LSE parse information before the application calls LSE$EDIT. LSE $EDIT destroys all parse information obtained and stored before LSE$EDIT is called.

LSE$EXECUTE_COMMAND

LSE$EXECUTE_COMMAND — Allows your program to execute DECTPU statements.

Format

LSE$EXECUTE_COMMAND string

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

string

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by value

DECTPU statement. The *string* argument is the address of a descriptor of a character string denoting one or more DECTPU statements.

Condition Values Returned

TPU$_SUCCESS	Normal successful completion.
TPU$_EXITING	EXIT built-in procedure was invoked.

TPU$_QUITTING	QUIT built-in procedure was invoked.
TPU$_EXECUTEFAIL	Execution aborted. This could be because of execution errors or compilation errors.

Description

This routine performs the same function as the built-in procedure EXECUTE described in the *DEC Text Processing Utility Reference Manual*.

LSE$EXECUTE_INIFILE

LSE$EXECUTE_INIFILE — Allows you to execute a user-written initialization file. This routine must be executed after the editor is initialized, but before any other commands are processed.

Format

LSE$EXECUTE_INIFILE

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Condition Values Returned

TPU$_SUCCESS	Normal successful completion.
TPU$_EXITING	A result of EXIT. If the default condition handler is being used, the session is terminated.
TPU$_QUITTING	A result of QUIT. If the default condition handler is being used, the session is terminated.
TPU$_COMPILEFAIL	The compilation of the initialization file was unsuccessful.
TPU$_EXECUTEFAIL	The execution of the statements in the initialization file was unsuccessful.
TPU$_FAILURE	General code for all other errors.

Description

This routine causes DECTPU to perform the following steps:

1. The command file is read into a buffer. If you specified a file on the command line that cannot be found, an error message is displayed and the routine is aborted. The default is LSE $COMMAND.TPU.

2. If you specified the /DEBUG qualifier on the command line, the DEBUG file is read into a buffer. The default is SYS$SHARE:LSE$DEBUG.TPU.
3. The DEBUG file is compiled and executed (if available).
4. TPU$INIT_PROCEDURE is executed (if available).
5. The command buffer is compiled and executed (if available).
6. TPU$INIT_POSTPROCEDURE is executed (if available).

Note

If you call this routine after calling LSE$CLEANUP, you must have set the flags LSE $M_EXECUTE_PROC and LSE$M_EXECUTE_FILE beforehand. Otherwise, the initialization file will not execute.

LSE$FILEIO

LSE$FILEIO — Handles all LSE file operations. Your own file I/O routine can call this routine to perform some operations for it. However, the routine that opens the file must perform all operations for that file. For example, if LSE$FILEIO opens the file it must also close it.

Format

`LSE$FILEIO code, stream, data`

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

Item code specifying an LSE function. The *code* argument is the address of a longword containing an item code from LSE specifying a function to perform. You can specify the following item codes in the file I/O routine:

- LSE$K_OPEN This item code specifies that the data parameter is the address of an item list. This item list contains the information necessary to open the file. The stream parameter should be filled in

with a unique identifying value to be used for all future references to this file. The resultant file name should also be copied with a dynamic string descriptor.

- LSE$K_CLOSE The file specified by the *stream* argument is to be closed. All memory being used by its structures can be released.
- LSE$K_CLOSE_DELETE The file specified by the *stream* argument is to be closed and deleted. All memory being used by its structures can be released.
- LSE$K_GET The data parameter is the address of a dynamic string descriptor to be filled with the next record from the file specified by the *stream* argument. The routine should use the routines provided by the OpenVMS Run-Time Library to copy text into this descriptor. LSE frees the memory allocated for the data read when the file I/O routine indicates that the end of the file has been reached.
- LSE$K_PUT The *data* parameter is the address of a descriptor for the data to be written to the file specified by the *stream* argument.

stream

VMS Usage:	unspecified
type:	longword (unsigned)
access:	modify
mechanism:	by reference

File description. The *stream* argument is the address of a data structure consisting of four longwords. This data structure is used to describe the file to be manipulated.

This data structure is used to refer to all files. It is written to when an open file request is made. All other requests use information in this structure to determine which file is being referenced.

Figure 2.2 shows the stream data structure.

Figure 2.2. Stream Data Structure

File Identifier		
RFM		Allocation
Class	Type	Length
Address of Name		

The first longword is used to hold a unique identifier for each file. The user-written file I/O routine is restricted to values between 0 and 511. Thus, you can have up to 512 files open simultaneously.

The second longword is divided into three fields. The low word is used to store the allocation quantity, that is, the number of blocks allocated to this file from the FAB (FAB$L_ALQ). This value is used later to calculate the output file size for preallocation of disk space. The low-order byte of the second word is used to store the record attribute byte (FAB$B_RAT) when an existing file is opened. The high-order byte is used to store the record format byte (FAB$B_RFM) when an existing file is opened. The values in the low word and the low-order and high-order bytes of the second word are used for creating the output file in the same format as the input file. These three fields are to be filled in by the routine opening the file.

The last two longwords are used as a descriptor for the resultant or the expanded file name. This name is used later when LSE processes EXIT commands. This descriptor is to be filled in with the file name after an open operation. It should be allocated with either the routine LIB$SCOPY_R_DX or the routine LIB$SCOPY_DX from the RTL. This space is freed by LSE when it is no longer needed.

data

VMS Usage:	item_list_3
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Stream data. The *data* argument is either the address of an item list or the address of a descriptor.

Note

The meaning of this parameter depends on the item code specified in the code field.

When the LSE$K_OPEN item code is entered, the data parameter is the address of an item list containing information about the open request. The following LSE item codes are available for specifying information about the open request:

- LSE$_ACCESS allows you to specify one of three item codes in the buffer address field, as follows:
  - LSE$K_IO
  - LSE$K_INPUT
  - LSE$K_OUTPUT
- LSE$_FILENAME is used for specifying the address of a string to use as the name of the file you are opening. The length field contains the length of this string, and the address field contains the address.
- LSE$_DEFAULTFILE is used for assigning a default file name to the file being opened. The buffer-length field contains the length, and the buffer-address field contains the address of the default file name.
- LSE$_RELATEDFILE is used for specifying a related file name for the file being opened. The buffer-length field contains the length, and the buffer-address field contains the address of a string to use as the related file name.
- LSE$_RECORD_ATTR specifies that the buffer-address field contains the value for the record attribute byte in the FAB (FAB$B_RAT) used for file creation.
- LSE$_RECORD_FORM specifies that the buffer-address field contains the value for the record format byte in the FAB (FAB$B_RFM) used for file creation.
- LSE$_MAXIMIZE_VER specifies that the version number of the output file should be one higher than the highest existing version number.
- LSE$_FLUSH specifies that the file should have every record flushed after it is written.

- LSE$_FILESIZE specifies the value for the allocation quantity when creating the file. The value is specified in the buffer-address field.
- LSE$_EOF_BLOCK specifies the end-of-file block number of the file. The BUFADR field is the address of a longword into which the file I/O routine must write the file's end-of-file block number (from XAB$L_EBK in $XABFHC).
- LSE$_EOF_FFB specifies the file's first free byte offset into the end-of-file block. The BUFADR field is the address of a word into which the file I/O routine must write the file's first free byte offset into the end-of-file block (XAB$W_FFB in $XABFHC).

Condition Values Returned

The LSE$FILEIO routine returns an RMS status code to LSE. The file I/O routine is responsible for signaling all errors if you want any messages displayed.

Description

By default, LSE$FILEIO creates variable-length files with carriage-return record attributes (FAB $B_RFM = VAR, FAB$B_RAT = CR). If you pass to it the LSE$_RECORD_ATTR or LSE $_RECORD_FORM item, that item is used instead. The following combinations of formats and attributes are acceptable:

All other combinations are converted to VAR format with CR attributes.

This routine always puts values greater than 511 in the first longword of the stream data structure. Because a user-written file I/O routine is restricted to the values 0 through 511, you can distinguish the file-control blocks (FCB) this routine fills in from the ones you created.

Note

LSE uses LSE$FILEIO by default when you use the simplified callable interface. When you use the full callable interface, you must explicitly invoke LSE$FILEIO or provide your own file I/O routine.

LSE$HANDLER

LSE$HANDLER — Is the LSE condition handler. The LSE condition handler invokes the Put Message (SYS$PUTMSG) system service, passing it the address of LSE$MESSAGE.

Format

```
LSE$HANDLER signal_vector, mechanism_vector
```

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value.

Arguments

signal_vector

VMS Usage:	arg_list
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Signal vector. See the *VSI OpenVMS System Services Reference Manual* for information about the signal vector passed to a condition handler.

mechanism_vector

VMS Usage:	arg_list
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

Mechanism vector. See the *VSI OpenVMS System Services Reference Manual* for information about the mechanism vector passed to a condition handler.

Description

This routine performs the actual output of the message. The Put Message (SYS$PUTMSG) system service formats only the message. It gets the settings for the message flags and facility name from the variables described in Section 2.1. You must use the DECTPU built-in procedure SET to modify those values.

If the condition value received by the handler has a fatal status or does not have an LSE, DECTPU, CLI $, or SCA facility code, the condition is resignaled.

If the condition is TPU$_QUITTING, TPU$_EXITING, or TPU$_RECOVERFAIL, a request to unwind is made to the establisher of the condition handler.

After handling the message, the condition handler returns with a continue status. DECTPU error message requests are made by signaling a condition to indicate which message should be written out. The arguments in the signal array are a correctly formatted message argument vector. This vector sometimes contains multiple conditions and formatted ASCII output (FAO) arguments for the associated messages. For example, if the editor attempts to open a file that does not exist, the DECTPU message TPU$_NOFILEACCESS is signaled. The FAO argument to this message is a string for the name of the file. This condition has an error status, followed by the RMS status field (STS) and status-value field (STV). Because this condition does not have a fatal severity, LSE continues after handling the error.

The editor does not automatically return from LSE$CONTROL. If you call the LSE$CONTROL routine, you must explicitly establish a way to regain control (for example, using the built-in procedure CALL_USER). Also, if you establish your own condition handler but call the LSE handler for certain conditions, the default condition handler must be established at the point in your program where you want to return control.

See the *VSI OpenVMS Calling Standard* for information about the OpenVMS Condition Handling Standard.

LSE$INITIALIZE

LSE$INITIALIZE — Initializes LSE for editing. This routine allocates global data structures, initializes global variables, and calls the appropriate setup routines for each of the major components of the editor, including the Virtual File Manager, Screen Manager, and I/O subsystem.

Format

```
LSE$INITIALIZE callback [,user_arg]
```

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

callback

VMS Usage:	vector_longword_unsigned
type:	bound procedure value
access:	read-only
mechanism:	by descriptor

Callback routine. The *callback* argument is the address of a user-written routine that returns the address of an item list containing initialization parameters or a routine for handling file I/O operations. This callback routine must call a parsing routine, which can be LSE$CLIPARSE or a user-written parsing routine.

Callable LSE defines thirteen item codes that you can use for specifying initialization parameters. You do not have to arrange the item codes in any particular order in the list. Figure 2.3 shows the general format of an item descriptor. For information about how to build an item list, see the OpenVMS programmer's manual associated with the language you are using.

Figure 2.3. Format of an Item Descriptor

Item Code	Buffer Length
Buffer Address	
Return Address	

The return address in an item descriptor is usually 0.

Table 2.2 describes the available item codes.

Table 2.2. LSE$INITIALIZE Item Codes

Item Code	Description
LSE$_OPTIONS	Enables the command qualifiers. Ten bits in the buffer-address field correspond to the various LSE command qualifiers. The remaining 22 bits in the buffer-address field are reserved.
LSE$_JOURNALFILE	Passes the string specified with the /JOURNAL qualifier. The buffer-length field is the length of the string, and the buffer-address field is the address of the string. This string is available with GET_INFO (COMMAND_LINE, “JOURNAL_FILE”). This string may be a null string.
LSE$_SECTIONFILE	Passes the string that is the name of the binary initialization file (section file) to be mapped in. The buffer-length field is the length of the string and the buffer-address field is the address of the string. The LSE CLD file has a default value for this string. If the LSE$V_SECTION bit is set, this item code must be specified.
LSE$_OUTPUTFILE	Passes the string specified with the /OUTPUT qualifier. The buffer-length field is the length of the string, and the buffer-address field specifies the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “OUTPUT_FILE”). The string may be a null string.
LSE$_DISPLAYFILE	Passes the string specified with the /DISPLAY qualifier. The buffer-length field is the length of the string, and the buffer-address field specifies the address of the string.
LSE$_COMMANDFILE	Passes the string specified with the /COMMAND qualifier. The buffer-length field is the length of the string, and the buffer-address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “COMMAND_FILE”). The string may be a null string.
LSE$_FILENAME	Passes the string that is the name of the input file specified in the command line. The buffer-length field specifies the length of this string, and the buffer-address field specifies its address. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “FILE_NAME”). This file name may be a null string.
LSE$_FILEIO	Passes the bound procedure value of a routine to be used for handling file operations. You may provide your own file I/O routine, or you can call

Item Code	Description
	LSE$FILEIO, the utility routine provided by LSE for handling file operations. The buffer-address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that LSE loads into R1 before calling the routine.
LSE$_CALLUSER	Passes the bound procedure value of the user-written routine that the built-in procedure CALL_USER is to call. The buffer-address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that LSE loads into R1 before calling the routine.
LSE$_INIT_FILE	Passes the string specified with the /INITIALIZATION qualifier. The buffer-length field is the length of the string, and the buffer-address field is the address of the string. This string is returned by using the built-in procedure GET_INFO (COMMAND_LINE, “INIT_FILE”).
LSE$_START_LINE	Passes the starting line number for the edit. The buffer-address field contains the first of the two integer values you specified as part of the /START_POSITION command qualifier. The value is available by using the built-in procedure GET_INFO (COMMAND_LINE, “LINE”). Usually an initialization procedure uses this information to set the starting position in the main editing buffer. The first line in the buffer is line 1.
LSE$_START_CHAR	Passes the starting column position for the edit. The buffer-address field contains the second of the two integer values you specified as part of the /START_POSITION command qualifier. The value is available using the built-in procedure GET_INFO (COMMAND_LINE, “CHARACTER”). Usually an initialization procedure uses this information to set the starting position in the main editing buffer. The first column on a line corresponds to character 1.
LSE$_CTRL_C_ROUTINE	Passes the bound procedure value of a routine to be used for handling Ctrl/C ASTs. LSE calls the routine when a Ctrl/C AST occurs. If the routine returns a FALSE value, LSE assumes that the Ctrl/C has been handled. If the routine returns a TRUE value, LSE aborts any currently executing LSE procedure. The buffer-address field specifies the address of a two-longword vector. The first longword of the vector contains the address of

Item Code	Description
	the routine. The second longword specifies the environment value that LSE loads into R1 before calling the routine.
LSE$_DEBUGFILE	Passes the string specified with the /DEBUG command qualifier. The buffer-length field is the length of the string, and the buffer-address field is the address of the string.

Table 2.3 shows the bits and corresponding masks enabled by the item code LSE$_OPTIONS.

Table 2.3. LSE$_OPTIONS Masks and Bits

Mask [1]	Bit [2]	Function
LSE$M_RECOVER	LSE$V_RECOVER	Performs a recovery operation.
LSE$M_JOURNAL	LSE$V_JOURNAL	Journals the edit session.
LSE$M_READ	LSE$V_READ	Makes this a READ_ONLY edit session for the main buffer.
LSE$M_SECTION	LSE$V_SECTION	Maps in a binary initialization file (a DECTPU section file) during startup.
LSE$M_CREATE	LSE$V_CREATE	Creates an input file if the one specified does not exist.
LSE$M_OUTPUT	LSE$V_OUTPUT	Writes the modified input file upon exiting.
LSE$M_COMMAND	LSE$V_COMMAND	Executes a command file during startup.
LSE$M_DISPLAY	LSE$V_DISPLAY	Attempts to use the terminal for screen oriented editing and display purposes.
LSE$M_INIT	LSE$V_INIT	Indicates the presence of an initialization file.
LSE$M_COMMAND_DFLTED	LSE$V_COMMAND_DFLTED	Indicates whether the user defaulted the name of the command line. A setting of TRUE means the user did not specify a command file. If this bit is set to FALSE and the user did not specify a file, LSE $INITIALIZE fails.
LSE$M_WRITE	LSE$V_WRITE	Indicates whether the /WRITE qualifier was specified on the command line.
LSE$M_MODIFY	LSE$V_MODIFY	Indicates whether the /MODIFY qualifier was specified on the command line.

Mask [1]	Bit [2]	Function
LSE$M_NOMODIFY	LSE$V_NOMODIFY	Indicates whether the /NOMODIFY qualifier was specified on the command line.
LSE$M_DEBUG	LSE$V_DEBUG	Indicates whether the /DEBUG qualifier was specified.

[1] LSE$M … indicates a mask.

[2] LSE$V … indicates a bit item.

To create the bits, start with the value 0, then use the OR operator on the mask (LSE$M . . .) of each item you want to set. Another way to create the bits is to treat the 32 bits as a bit vector and set the bit (LSE$V . . .) corresponding to the item you want.

user_arg

VMS Usage:	user_arg
type:	bound procedure value
access:	read-only
mechanism:	by value

User argument. The *user_arg* argument is passed to the user-written initialization routine INITIALIZE.

The *user_arg* argument is provided to allow an application to pass information through LSE $INITIALIZE to the user-written initialization routine. LSE does not interpret this data in any way.

Condition Values Returned

TPU$_SUCCESS	Initialization was completed successfully.
TPU$_SYSERROR	A system service did not work correctly.
TPU$_NONANSICRT	The input device (SYS$INPUT) is not a supported terminal.
TPU$_RESTOREFAIL	An error occurred during the restore operation.
TPU$_NOFILEROUTINE	No routine has been established to perform file operations.
TPU$_INSVIRMEM	Insufficient virtual memory exists for the editor to initialize.
TPU$_FAILURE	General code for all other errors during initialization.

Description

This routine is the first routine that must be called after establishing a condition handler.

This routine initializes the editor according to the information received from the callback routine. The initialization routine defaults all file specifications to the null string and all options to off. However, it does not default the file I/O or call-user routine addresses.

If you do not specify a section file, the software features of the editor are limited.

LSE$LSE

LSE$LSE — Invokes LSE and is equivalent to the DCL command LSEDIT.

Format

LSE$LSE command

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

command

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by descriptor

Command string. The *command* argument is the address of a descriptor of a command line.

Condition Values Returned

The LSE$LSE routine returns any condition value returned by LSE$INITIALIZE, LSE $EXECUTE_INFILE, LSE$CONTROL, and LSE$CLEANUP.

Description

This routine takes the command string specified and passes it to the editor. LSE uses the information from this command string for initialization purposes, just as though you had entered the command at the DCL level.

Using the simplified callable interface does not set LSE$V_CLOSE_SECTION. This feature allows you to make multiple calls to LSE$LSE without requiring you to open and close the section file on each call.

If your application parses information that is not related to the operation of LSE, make sure the application gets, and uses, all non-LSE parse information before the application calls LSE$LSE. LSE $LSE destroys all parse information obtained and stored before LSE$LSE was called.

LSE$MESSAGE

LSE$MESSAGE — Writes error messages and strings by using the built-in procedure, MESSAGE. You can call this routine to have messages written and handled in a manner consistent with LSE. This routine should be used only after LSE$EXECUTE_INIFILE.

Format

LSE$MESSAGE string

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value.

Note

The return status should be ignored because it is intended for use by the Put Message (SYS$PUTMSG) system service.

Argument

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by descriptor

Formatted message. The *string* argument is the address of a descriptor of text to be written. It must be completely formatted. This routine does not append the message prefixes. However, the text is appended to the message buffer if one exists. In addition, if the buffer is mapped to a window, the window is updated.

LSE$PARSEINFO

LSE$PARSEINFO — Parses a command and builds the item list for LSE$INITIALIZE.

Format

LSE$PARSEINFO fileio, call_user

Returns

VMS Usage:	item_list
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

The routine returns the address of an item list.

Arguments

`fileio`

VMS Usage:	vector_longword_unsigned
type:	bound procedure value
access:	read-only
mechanism:	by descriptor

File I/O routine. The *fileio* argument is the address of a descriptor of a file I/O routine.

`call_user`

VMS Usage:	vector_longword_unsigned
type:	bound procedure value
access:	read-only
mechanism:	by descriptor

Call-user routine. The *call_user* argument is the address of a descriptor of a call-user routine.

Description

This routine parses a command and builds the item list for LSE$INITIALIZE.

This routine uses the Command Language Interpreter (CLI) routines to parse the current command. It makes queries about the command parameters and qualifiers that LSE expects. The results of these queries are used to set up the proper information in an item list. The addresses of the user routines are used for those items in the list. The address of this list is the return value of the routine.

If your application parses information that is not related to the operation of LSE, make sure the application gets, and uses, all non-LSE parse information before the application calls LSE$PARSEINFO interface. LSE$PARSEINFO destroys all parse information obtained and stored before LSE $PARSEINFO was called.

FILEIO

FILEIO — Handles LSE file operations. The name of this routine can be either your own file I/O routine or the name of the LSE file I/O routine (LSE$FILEIO).

Format

```
FILEIO code, stream, data
```

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only

mechanism:	by reference

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed in the Condition Values Returned section.

Arguments

code

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

Item code specifying an LSE function. The code argument is the address of a longword containing an item code from LSE that specifies a function to perform.

stream

VMS Usage:	unspecified
type:	longword (unsigned)
access:	modify
mechanism:	by reference

File description. The stream argument is the address of a data structure containing four longwords. This data structure is used to describe the file to be manipulated.

data

VMS Usage:	item_list_3
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Stream data. The *data* argument is either the address of an item list or the address of a descriptor.

Note

The value of this parameter depends on which item code you specify.

Condition Values Returned

The condition values returned are determined by the user and should indicate success or failure of the operation.

Description

The bound procedure value of the FILEIO routine is specified in the item list built by the callback routine. This routine is called to perform file operations. Instead of using your own file I/O routine, you

can call LSE$FILEIO and pass it the parameters for any file operation that you do not want to handle. Note, however, that LSE$FILEIO must handle all I/O requests for any file it opens. Also, if it does not open the file, it cannot handle any I/O requests for the file. In other words, you cannot intermix the file operations between your own file I/O routine and the one supplied by LSE.

HANDLER

HANDLER — Performs condition handling. It is a user-written routine.

Format

`HANDLER signal_vector, mechanism_vector`

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value.

Arguments

`signal_vector`

VMS Usage:	arg_list
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Signal vector. See the *VSI OpenVMS System Services Reference Manual* for information about the signal vector passed to a condition handler.

`mechanism_vector`

VMS Usage:	arg_list
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

Mechanism vector. See the *VSI OpenVMS System Services Reference Manual* for information about the mechanism vector passed to a condition handler.

Description

If you need more information about writing condition handlers and the OpenVMS Condition Handling Standard, refer to the Introduction to VMS System Routines.

Instead of writing your own condition handler, you can use the default condition handler, LSE $HANDLER. If you want to write your own routine, you must call LSE$HANDLER with the same parameters that your routine received to handle LSE internal signals.

INITIALIZE

INITIALIZE — Is passed to LSE$INITIALIZE as a bound procedure value and called to supply information needed to initialize LSE. It is a user-written routine.

Format

`INITIALIZE [user_arg]`

Returns

VMS Usage:	item_list
type:	longword (unsigned)
access:	read-only
mechanism:	by reference

This routine returns the address of an item list.

Arguments

VMS Usage:	user_arg
type:	longword (unsigned)
access:	read-only
mechanism:	by value

User argument.

Description

The user-written INITIALIZE routine is passed to LSE$INITIALIZE as a bound procedure value and called to supply information needed to initialize LSE.

If the *user_arg* parameter was specified in the call to LSE$INITIALIZE, the initialization callback routine is called with only that parameter. If `user_arg` was not specified in the call to LSE $INITIALIZE, the initialization callback routine is called with no parameters.

The *user_arg* parameter is provided to allow an application to pass information through LSE $INITIALIZE to the user-written initialization routine. LSE does not interpret this data in any way.

The user-written callback routine is expected to return the address of an item list containing initialization parameters. Because the item list is used outside the scope of the initialization callback routine, it should be allocated in static memory.

The item list entries are discussed in the section on LSE$INITIALIZE. Most of the initialization parameters have a default value: strings default to the null string and flags default to false. The only required initialization parameter is the address of a routine for file I/O. If an entry for the file I/O routine address is not present in the item list, LSE$INITIALIZE returns with a failure status.

USER

USER — Allows your program to get control during an LSE editing session (for example, to leave the editor temporarily and perform a calculation). This user-written routine is invoked by the DECTPU built-in procedure CALL_USER. The built-in procedure CALL_USER passes three parameters to this routine. These parameters are then passed to the appropriate part of your application to be used as specified. (For example, they may be used as operands in a calculation within a FORTRAN program.) Using the string routines provided by the OpenVMS Run-Time Library (RTL), your application fills in the *stringout* parameter in the call-user routine, which returns the *stringout* value to the built-in procedure CALL_USER.

Format

`USER integer, stringin, stringout`

Returns

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write-only
mechanism:	by value

Longword condition value.

Arguments

`integer`

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read-only
mechanism:	by descriptor

First parameter to the built-in procedure CALL_USER. This is an input-only parameter and must not be modified.

`stringin`

VMS Usage:	char_string
type:	character string
access:	read-only
mechanism:	by descriptor

Second parameter to the built-in procedure CALL_USER. This is an input-only parameter and must not be modified.

`stringout`

VMS Usage:	char_string

type:	character string
access:	read-only
mechanism:	by descriptor

Return value for the built-in procedure CALL_USER. Your program should fill in this descriptor with a dynamic string allocated by the string routines provided by the RTL. LSE frees this string when necessary.

Description

The description of the built-in procedure CALL_USER in the *DEC Text Processing Utility Reference Manual* shows an example of a BASIC program that is a call-user routine.

Example

```
INTEGER FUNCTION TPU$CALLUSER (x,y,z)
     IMPLICIT NONE
     INTEGER X
     CHARACTER*(*) Y
     STRUCTURE /dynamic/ Z
               INTEGER*2 length
               BYTE    dtype
               BYTE    class
               INTEGER ptr
     END STRUCTURE
     RECORD /dynamic/ Z
     CHARACTER*80 local_copy
     INTEGER rs,lclen
     INTEGER STR$COPY_DX
     local_copy = '<' // y // '>'
     lclen = LEN(Y) + 2
     RS = STR$COPY_DX(Z,local_copy(1:lclen))
     TPU$CALLUSER = RS
     END
```

You can call this FORTRAN program with a DECTPU procedure. The following is an example of one such procedure:

```
PROCEDURE MY_CALL
     local status;
     status := CALL_USER (0,'ABCD');
     MESSAGE('"' + '"');
     ENDPROCEDURE
```