VMS Software

VSI OpenVMS

Utility Routines Manual

Document Number: DO-UTRMAN-01A

Publication Date: April 2024

Operating System and Version: VSI OpenVMS x86-64 Version 9.2-1 or higher;
VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

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

Utility Routines Manual

Copyright © 2024 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

HPE, HPE Integrity, HPE Alpha, and HPE Proliant are trademarks or registered trademarks of Hewlett Packard Enterprise.

Preface

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 manual is intended for programmers who want to invoke and use the functions provided by OpenVMS utilities.

3. Document Structure

Chapter 1 introduces the utility routines and lists the documentation format used to describe each set of utility routines, as well as the individual routines in each set. Each subsequent chapter contains an introduction to a set of utility routines, a programming example to illustrate the use of the routines in the set, and a detailed description of each routine.

This manual presents the following utility routine sets:

- Access Control List (ACL) editor routine
- Backup API routine
- Command Language Interface (CLI) routines
- Common File Qualifier routines
- Convert (CONVERT) routines
- Data Compression/Expansion (DCX) routines
- DEC Text Processing Utility (DECTPU) routines
- DIGITAL Distributed Time Service (DECdts) Portable Applications Programming Interface
- EDT routines
- Encryption (ENCRYPT) routines
- File Definition Language (FDL) routines
- Librarian (LBR) routines
- Lightweight Directory Access Protocol (LDAP) routines
- LOGINOUT (LGI) routines
- Mail utility (MAIL) routines
- National character set (NCS) utility routines

- Print Symbiont Modification (PSM) routines
- Symbiont/Job Controller Interface (SMB) routines
- Sort/Merge (SOR) routines
- Traceback facility (TBK) routines

4. 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.

5. OpenVMS Documentation

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

6. Typographical Conventions

The following conventions may be used in this manual:

Integrity servers	Abbreviation representing "VSI OpenVMS for Integrity servers".
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.
	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.) In the HTML version of this document, this convention appears as brackets, rather than a box.
…	Horizontal ellipsis points in examples indicate 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.
. . .	Vertical ellipsis points indicate 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 choices in parentheses if you specify more than one.

[ ]	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 optional 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 type	Bold type 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).
`Example`	This typeface indicates code examples, command examples, and interactive screen displays. In text, this type also identifies URLs, UNIX commands and pathnames, PC-based commands and folders, and certain elements of the C programming language.
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	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. Introduction to Utility Routines

A set of utility routines performs a particular task or set of tasks. For example, you can use the Print Symbiont Modification (PSM) routines to modify the print symbiont and the EDT routines to invoke the EDT editor from a program.

Some of the tasks performed by utility routines can also be performed at the Digital Command Language (DCL) level (for example, the DCL command EDIT invokes the EVE editor). While DCL commands invoke utilities that let you perform tasks at your terminal, you can perform some of these tasks at the programming level through the use of the utility routines.

When using a set of utility routines that performs the same tasks as the related utility, you should read the documentation for that utility; doing so will provide additional information about the tasks the routines can perform as a set. The following table lists the utilities and their corresponding routines:

Utility or Editor	**Utility Routines**
Access control list editor	ACL editor routine
Backup application programming interface	Backup API routine
Command Definition Utility	CLI routines
Common File Qualifier routines	UTIL$CQUAL routines
Convert and Convert/Reclaim utilities	CONVERT routines
Data Compression/Expansion (DCX) facility	DCX routines
DEC Text Processing Utility	DECTPU routines
Digital Distributed Time Service (DECdts) portable applications programming interface	DECdts API routines
EDT editor	EDT routines
Encryption routines	ENCRYPT routines
File Definition Language facility	FDL routines
Librarian utility	LBR routines
Lightweight Directory Access Protocol (LDAP) application programming interface	LDAP API routines
LOGINOUT callout routines	LGI routines
Mail utility	MAIL routines
National Character Set utility	NCS routines
Print Symbiont Modification (PSM) facility	PSM routines
Symbiont/Job Controller Interface facility	SMB routines
Sort/Merge utility	SOR routines
Traceback facility	TBK routines

When a set of utility routines performs functions that you cannot perform by invoking a utility, the functions provided by that set of routines is termed a **facility**. The following facilities have no other user interface except the programming interface provided by the utility routines described in this manual:

Facility	Utility Routines
Data Compression/Expansion facility	DCX routines
Print Symbiont Modification facility	PSM routines
Symbiont/Job Controller Interface facility	SMB routines
Traceback facility	TBK routines

Like all other system routines in the OpenVMS environment, the utility routines described in this manual conform to the *VSI OpenVMS Calling Standard*. Note that for stylistic purposes, the calling syntax illustrated for routines documented in this manual is consistent. However, you should consult your programming language documentation to determine the appropriate syntax for calling these routines.

Each chapter of this book documents one set of utility routines. Each chapter has the following major components, documented as a major heading:

- An introduction to the set of utility routines. This component discusses the utility routines as a group and explains how to use them.
- One or more programming examples that illustrate how the utility routines are used.
- A series of descriptions of each utility routine in the set.

Chapter 2. Access Control List (ACL) Editor Routine

This chapter describes the access control list editor (ACL editor) routine, ACLEDIT$EDIT. User-written applications can use this callable interface of the ACL editor to manipulate access control lists (ACLs).

2.1. Introduction to the ACL Editor Routine

The ACL editor is a utility that lets you create and maintain access control lists. Using ACLs, you can limit access to the following protected objects available to system users:

- Devices
- Files
- Group global sections
- Logical name tables
- System global sections
- Common event flag clusters
- Queues
- Resource domains
- Security classes
- Volumes

The ACL editor provides one callable interface that allows the application program to define an object for editing.

Note that the application program should declare referenced constants and return status symbols as external symbols; these symbols will be resolved upon linking with the utility shareable image.

See the *VSI OpenVMS Programming Concepts Manual* for fundamental conceptual information on the creation, translation, and maintenance of access control entries (ACEs).

2.2. Using the ACL Editor Routine: An Example

Example 2.1 shows a VAX BLISS program that calls the ACL editor routine.

Example 2.1. Calling the ACL Editor with a VAX BLISS Program

```
MODULE MAIN (LANGUAGE (BLISS32), MAIN = STARTUP) =
BEGIN
```

```
LIBRARY 'SYS$LIBRARY:LIB';
ROUTINE STARTUP =
BEGIN
LOCAL
  STATUS,     ! Routine return status
  ITMLST      : BLOCKVECTOR [6, ITM$S_ITEM, BYTE];
              ! ACL editor item list
EXTERNAL LITERAL
  ACLEDIT$V_JOURNAL,
  ACLEDIT$V_PROMPT_MODE,
  ACLEDIT$C_OBJNAM,
  ACLEDIT$C_OBJTYP,
  ACLEDIT$C_OPTIONS;
EXTERNAL ROUTINE
  ACLEDIT$EDIT
  : ADDRESSING_MODE (GENERAL),  ! Main routine
  CLI$GET_VALUE,    ! Get qualifier value
  CLI$PRESENT,      ! See if qualifier present
  LIB$PUT_OUTPUT,   ! General output routine
  STR$COPY_DX;      ! Copy string by descriptor
! Set up the item list to pass back to TPU so it can figure out what to do.
CH$FILL (0, 6*ITM$S_ITEM, ITMLST);
ITMLST[0, ITM$W_ITMCOD] = ACLEDIT$C_OBJNAM;
ITMLST[0, ITM$W_BUFSIZ] = %CHARCOUNT ('YOUR_OBJECT_NAME');
ITMLST[0, ITM$L_BUFADR] = $DESCRIPTOR ('YOUR_OBJECT_NAME');
ITMLST[1, ITM$W_ITMCOD] = ACLEDIT$C_OBJTYP;
ITMLST[1, ITM$W_BUFSIZ] = 4;
ITMLST[1, ITM$L_BUFADR] = UPLIT (ACL$C_FILE);
ITMLST[2, ITM$W_ITMCOD] = ACLEDIT$C_OPTIONS;
ITMLST[2, ITM$W_BUFSIZ] = 4;
ITMLST[2, ITM$L_BUFADR] = UPLIT (1 ^ ACLEDIT$V_PROMPT_MODE OR
      1 ^ ACLEDIT$V_JOURNAL);
RETURN ACLEDIT$EDIT (ITMLST);
END;      ! End of routine STARTUP
END
ELUDOM
```

2.3. ACL Editor Routine

This section describes the ACL editor routine.

ACLEDIT$EDIT—Edit Access Control List

ACLEDIT$EDIT—Edit Access Control List — The ACLEDIT$EDIT routine creates and modifies an access control list (ACL) associated with any protected object.

Format

```
ACLEDIT$EDIT item_list
```

Returns

OpenVMS 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 under Condition Values Returned.

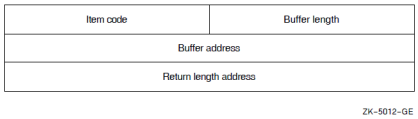
Argument

item_list

OpenVMS usage:	**item_list_3**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by descriptor**

Item list used by the callable ACL editor. The *item_list* argument is the address of one or more descriptors of arrays, routines, or longword bit masks that control various aspects of the editing session.

Each entry in an item list is in the standard format shown in the following figure:



The following table provides a detailed description of each item list entry:

Item Identifier	Description	
ACLEDIT$C_OBJNAM	Specifies the name of the object whose ACL is being edited.	
ACLEDIT$C_OBJTYP	A longword value that specifies the object type code for the type or class of the object whose ACL is being edited. These type codes are defined in $ACLDEF. The default object type is FILE (ACL$C_FILE).	
ACLEDIT$C_OPTIONS ACLEDIT$C_BIT_TABLE	Represents a longword bit mask of the various options available to control the editing session.	
	Flag	**Function**
	ACLEDIT$V_JOURNAL	Indicates that the editing session is to be journaled.
	ACLEDIT$V_RECOVER	Indicates that the editing session is to be recovered from an existing journal file.
	ACLEDIT$V_KEEP_RECOVER	Indicates that the journal file used to recover the editing session is not to be deleted when

Item Identifier	Description	
		the recovery is complete.
	ACLEDIT$V_KEEP_JOURNAL	Indicates that the journal file used for the editing session is not to be deleted when the session ends.
	ACLEDIT$V_PROMPT_MODE	Indicates that the session is to use automatic text insertion (prompting) to build new access control list entries (ACEs).
ACLEDIT$C_BIT_TABLE	Specifies a vector of 32 quadword string descriptors of strings that define the names of the bits present in the access mask. (The first descriptor defines the name of bit 0; the last descriptor defines the name of bit 31.) These descriptors are used in parsing or formatting an ACE. The buffer address field of the item descriptor contains the address of this vector.	
ACLEDIT$C_CLSNAM	A string descriptor that points to the class name of the object whose ACL is being modified. The following are valid class names: • COMMON_EVENT_FLAG_CLUSTER • DEVICE • FILE • GROUP_GLOBAL_SECTION • LOGICAL_NAME_TABLE • QUEUE • RESOURCE_DOMAIN • SECURITY_CLASS • SYSTEM_GLOBAL_SECTION • VOLUME If both OBJTYP and CLSNAM are omitted, the object is assumed to belong to the FILE class.	

Description

Use the ACLEDIT$EDIT routine to create and modify an ACL associated with any security object.

Under normal circumstances, the application calls the ACL editor to modify an object's ACL, and control is returned to the application when you finish or abort the editing session.

If you also want to use a customized version of the ACL editor section file, the logical name ACLEDT $SECTION should be defined. See the *VSI OpenVMS System Management Utilities Reference Manual* for more information.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

RMS$_xxx

See the *VSI OpenVMS Record Management Services Reference Manual* for a description of OpenVMS RMS status codes.

TPU$_xxx

See Chapter 8 for a description of the TPU-specific condition values that may be returned by ACLEDIT$EDIT.

Chapter 3. Backup (BACKUP) Routine

This chapter describes the Backup application programming interface (API). User-written applications can use the Backup API to perform BACKUP operations.

3.1. Introduction to the Backup API

The Backup API allows application programs to save individual files or the contents of entire disk volume sets. The Backup API also allows application programs to get information about files or disk and tape volumes.

In general, the Backup API gives application programs access to (relevant) BACKUP functions that are available to an interactive user via the DCL command BACKUP. The application program calls routine BACKUP$START with an argument that points to a variable-length array, which consists of option structures to specify the required BACKUP operation. The call to BACKUP$START in combination with the option structures in the variable-length array form the equivalent of a BACKUP command at DCL level.

Each relevant BACKUP qualifier is represented by an option structure or combination of option structures. Each option structure consists of a longword that contains the option structure identifier, followed by a value field of 1 to 7 longwords. Each option structure must be quadword-aligned within the variable-length array. There are six option structure types:

Option	**Definition**
bck_opt_struct_adr	32-bit address
bck_opt_struct_dsc	Static string descriptor
bck_opt_struct_dsc64	Reserved for use by VSI
bck_opt_struct_dt	Date/Time quadword (ADT)
bck_opt_struct_flag	Logical bit flags
bck_opt_struct_int	32-bit integer

The option structure types are defined in the language definition files. Table 3.1 lists the language definition files.

Table 3.1. Backup API Language Definition Files

Language	**API Definitions**	**Media Format (Save Set) Definitions**	**Backup Utility Data Structures**
BASIC	BAPIDEF.BAS	BACKDEF.BAS	BACKSTRUC.BAS
BLISS	BAPIDEF.R32	BACKDEF.R32	BACKSTRUC.R32
C	BAPIDEF.H	BACKDEF.H	BACKSTRUC.H
Fortran	BAPIDEF.FOR	BACKDEF.FOR	BACKSTRUC.FOR
MACRO	BAPIDEF.MAR	BACKDEF.MAR	BACKSTRUC.MAR

See the *VSI OpenVMS System Management Utilities Reference Manual* for detailed definitions of the DCL command BACKUP qualifiers. See the *VSI OpenVMS System Manager's Manual* for detailed information about using BACKUP. You can also use the Help facility for more information about the Backup command and its qualifiers.

3.2. Using the Backup API: An Example

Example 3.1 shows a VAX C program that calls the Backup API. This program produces the same result as the following DCL command:

```
$  BACKUP [.WRK]*.* A.BCK/SAVE
```

Example 3.1. Calling the Backup API with a VAX C Program

```
#include <stdio.h>
#include <stdlib.h>
#include <ssdef.h>
#include <descrip.h>
#include "sys$examples:bapidef.h"

typedef struct _buf_arg
               {
                bck_opt_struct_dsc arg1;
                bck_opt_struct_dsc arg2;
                bck_opt_struct_flag arg3;
                bck_opt_struct_flag arg4;
                bck_opt_struct_flag arg5;

               } buf_arg;
struct dsc$descriptor
               input_dsc,
               output_dsc,
               event_type_dsc;
buf_arg myarg_buff;
unsigned int status;

extern unsigned int backup$start(buf_arg  *myarg_buff);
unsigned int subtest(void *);

static char  input_str[]      = "[.wrk]";
static char  output_str[]      = "a.bck";

main()
{
        input_dsc.dsc$b_dtype =
        output_dsc.dsc$b_dtype = DSC$K_DTYPE_T;

        input_dsc.dsc$b_class =
        output_dsc.dsc$b_class = DSC$K_CLASS_S;

        input_dsc.dsc$w_length = sizeof(input_str);
        output_dsc.dsc$w_length = sizeof(output_str);

        input_dsc.dsc$a_pointer = input_str;
        output_dsc.dsc$a_pointer = output_str;


 myarg_buff.arg1.opt_dsc_type = BCK_OPT_K_INPUT;
 myarg_buff.arg1.opt_dsc = input_dsc;

 myarg_buff.arg2.opt_dsc_type = BCK_OPT_K_OUTPUT;
 myarg_buff.arg2.opt_dsc = output_dsc;
```

```
 myarg_buff.arg3.option_type = BCK_OPT_K_SAVE_SET_OUT;
 myarg_buff.arg3.opt_flag_value = TRUE;

 myarg_buff.arg4.option_type = BCK_OPT_K_OPERATION_TYPE;
 myarg_buff.arg4.opt_flag_value = BCK_OP_K_SAVE ;

 myarg_buff.arg5.option_type = BCK_OPT_K_END_OPT;
 myarg_buff.arg5.opt_flag_value = FALSE;

        status = backup$start(&myarg_buff);

        exit (status);
}
```

3.3. Backup API

This section describes the Backup API.

BACKUP$START

BACKUP$START — is the entry point through which applications invoke the OpenVMS Backup utility.

Format

BACKUP$START argument-buffer

Returns

OpenVMS usage:	**COND_VALUE**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this routine can return are listed under Condition Values Returned.

Argument

argument-buffer

OpenVMS usage:	**user-defined array**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Arguments that specify the BACKUP operation to be performed. The **argument-buffer** argument is the address of a variable-length array of one or more Backup API option structures that define the attributes of the requested BACKUP operation. The variable-length array is terminated by an option structure of 16 bytes that contains all zeros. Table 3.2 describes the option structures.

Note

The length of the terminating option structure is 2 longwords (16 bytes). The first longword identifies the option structure and has a value of 0. It is recommended that the second longword contain a value of 0.

Table 3.2. BACKUP Option Structure Types

Option Structure	Description
BCK_OPT_K_END_OPT	Flag that contains all zeros to denote the end of **argument-buffer**. This option structure consists of 2 longwords. The first longword, with a value of 0, identifies the BCK_OPT_K_END_OPT option structure. The second longword is ignored by BACKUP. However it is recommended that the second longword contain all zeros.
BCK_OPT_K_ALIAS	Flag that specifies whether to maintain the previous behavior of multiple processing of alias and primary file entries. Values are TRUE (default) or FALSE. (See the BACKUP qualifier /ALIAS.) **Note** Use of BCK_OPT_K_ALIAS and BCK_OPT_K_PHYSICAL in the same call results in a fatal error.
BCK_OPT_K_ASSIST	Flag that specifies whether to allow operator or user intervention if a request to mount a magnetic tape fails during a BACKUP operation. Values are TRUE (default) or FALSE. (See the BACKUP qualifier /ASSIST.)
BCK_OPT_K_BACKUP	Flag that specifies whether to select files according to the BACKUP date written in the file header record. Values are TRUE or FALSE. Use this flag to set the corresponding logical bit flag for BCK_OPT_K_BEFORE_TYPE and BCK_OPT_K_SINCE_TYPE. (See the BACKUP qualifiers /BEFORE, /SINCE, and /BACKUP.)
BCK_OPT_K_BEFORE_TYPE	Logical bit flags that qualify the date specified in the BCK_OPT_K_BEFORE_VALUE option structure. Type can be one of the following:

Option Structure	Description
	BCK_OPTYP_BEFORE_K_BACKUP Selects files last saved or copied by BACKUP before the date specified. Also selects files with no BACKUP date. **BCK_OPTYP_BEFORE_K_CREATED** Selects files created before the date specified. **BCK_OPTYP_BEFORE_K_EXPIRED** Selects files that have expired as of the date specified. **BCK_OPTYP_BEFORE_K_MODIFIED** (Default) Selects files last modified before the date specified. **BCK_OPTYP_BEFORE_K_SPECIFIED** Reserved for use by VSI. (See the BACKUP qualifiers /BEFORE, /BACKUP, /CREATED, /EXPIRED, and /MODIFIED.)
BCK_OPT_K_BEFORE_VALUE	Date-Time Quadword that specifies the date qualified by BCK_OPT_K_BEFORE_TYPE. You cannot use delta time. (See the BACKUP qualifier /BEFORE.)
BCK_OPT_K_BLOCK	Integer that specifies the block size in bytes for data records in the BACKUP save set. The default block size for magnetic tape is 8,192 bytes. The default block size for disk is 32,256 bytes. (See the BACKUP qualifier /BLOCK_SIZE.)
BCK_OPT_K_CARTRIDGE_MEDIA_IN[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_CARTRIDGE_MEDIA_IN and BCK_OPT_K_CARTRIDGE_NAME_IN or any of the BCK_OPT_K_SCRATCH_* option structures in the same call results in a fatal error.
BCK_OPT_K_CARTRIDGE_NAME_IN[1]	32-bit descriptor.

Option Structure	Description
	Note Use of BCK_OPT_K_CARTRIDGE_NAME_IN and BCK_OPT_K_CARTRIDGE_MEDIA_IN or any of the BCK_OPT_K_SCRATCH_* option structures in the same call results in a fatal error.
BCK_OPT_K_CARTRIDGE_SIDE_IN[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_CARTRIDGE_SIDE_IN without BCK_OPT_K_CARTRIDGE_NAME_IN in the same call results in a fatal error. **Note** Use of BCK_OPT_K_CARTRIDGE_SIDE_IN with any of the BCK_OPT_K_SCRATCH_* option structures in the same call results in a fatal error.
BCK_OPT_K_CARTRIDGE_MEDIA_OUT[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_CARTRIDGE_MEDIA_OUT and BCK_OPT_K_CARTRIDGE_NAME_OUT or any of the BCK_OPT_K_SCRATCH_* option structures in the same call results in a fatal error.
BCK_OPT_K_CARTRIDGE_NAME_OUT[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_CARTRIDGE_NAME_OUT and BCK_OPT_K_CARTRIDGE_MEDIA_OUT or any of the BCK_OPT_K_SCRATCH_* option structures in the same call results in a fatal error.
BCK_OPT_K_CARTRIDGE_SIDE_OUT[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_CARTRIDGE_SIDE_OUT without BCK_OPT_K_CARTRIDGE_NAME_OUT in the same call results in a fatal error.

Option Structure	Description
	Note Use of BCK_OPT_K_CARTRIDGE_SIDE_OUT with any of the BCK_OPT_K_SCRATCH_* option structures in the same call results in a fatal error.
BCK_OPT_K_COMMAND	Reserved for use by VSI.
BCK_OPT_K_COMMENT	32-bit descriptor that specifies a comment string to be placed in the output save set.(See the BACKUP qualifier /COMMENT.)
BCK_OPT_K_COMPARE	Flag that specifies whether to compare the entity specified by BCK_OPT_K_INPUT with the entity specified by BCK_OPT_K_OUTPUT. Values are TRUE and FALSE (default).(See the BACKUP qualifier /COMPARE.)
BCK_OPT_K_CONFIRM	Flag that specifies whether to prompt for confirmation before processing each file. Values are TRUE and FALSE (default). (See the BACKUP qualifier /CONFIRM.)
BCK_OPT_K_CRC	Flag that specifies whether the software cyclic redundancy check (CRC) is to be performed. Values are TRUE (default) and FALSE. (See the BACKUP qualifier /CRC.)
BCK_OPT_K_CREATED	Flag that specifies whether to select files according to the creation date written in the file header record. Values are TRUE or FALSE. Use this flag to set the corresponding logical bit flag for BCK_OPT_K_BEFORE_TYPE and BCK_OPT_K_SINCE_TYPE. (See the BACKUP qualifiers /BEFORE, /SINCE, and /CREATED.)
BCK_OPT_K_DATA_FORMAT_COMPRESS	Flag that specifies whether data compression or decompression to be performed. Values are TRUE or FALSE (default).
BCK_OPT_K_DCL_INTERFACE	Reserved for use by VSI.
BCK_OPT_K_DELETE	Flag that specifies whether a copy or backup operation is to delete the input files from the input volume when the operation is complete. Values are TRUE and FALSE (default).

Option Structure	Description
	(See the BACKUP qualifier /DELETE.)
BCK_OPT_K_DENSITY	Integer that specifies the recording density of the output magnetic tape in bits per inch (bits/in). The density specified must be supported by the magnetic tape hardware. The default density is the current density on the output tape drive. (See the BACKUP qualifier /DENSITY.) **Note:** Use of BCK_OPT_K_DENSITY and BCK_OPT_K_MEDIA_FORMAT in the same call results in a fatal error.
BCK_OPT_K_DISMOUNT	Reserved for use by VSI.
BCK_OPT_K_DISPOSITION[1]	Logical bit flags. Values are the following: *BCK_OPTYP_DISP_K_KEEP*, *BCK_OPTYP_DISP_K_RELEASE*.
BCK_OPT_K_DRIVE_CLASS_IN[1]	32-bit descriptor.
BCK_OPT_K_DRIVE_CLASS_OUT[1]	32-bit descriptor.
BCK_OPT_K_ENCRYPT[b]	Flag.
BCK_OPT_K_ENCRYPT_USERALG[b]	32-bit descriptor.
BCK_OPT_K_ENCRYPT_USERKEY[b]	32-bit descriptor. **Note:** Use of BCK_OPT_K_ENCRYPT_USERKEY and BCK_OPT_K_ENCRYPT_KEY_VALUE in the same call results in a fatal error.
BCK_OPT_K_ENCRYPT_KEY_VALUE[b]	32-bit descriptor. **Note:** Use of BCK_OPT_K_ENCRYPT_KEY_VALUE and BCK_OPT_K_ENCRYPT_USERKEY in the same call results in a fatal error.
BCK_OPT_K_EVENT_CALLBACK	Address of a routine in the calling application to be called to process BACKUP events. See the Description section for detailed information about event callbacks.
BCK_OPT_K_EXACT_ORDER	Flag that specifies whether a BACKUP operation is to accept an exact order of tape volume labels, preserve an existing volume label, and prevent previous volumes of a multivolume save operation from being overwritten. Values are TRUE (default) and FALSE. (See the BACKUP qualifier /EXACT_ORDER.)
BCK_OPT_K_EXCLUDE	32-bit descriptor that specifies the name of an input file to be excluded from the current BACKUP save or copy operation.

Option Structure	Description
	Wildcards are permitted. Each file specification, whether wildcarded or not, requires its own BCK_OPT_K_EXCLUDE option structure (lists are not supported).(See the BACKUP qualifier /EXCLUDE.)
BCK_OPT_K_EXPIRED	Flag that specifies whether to select files according to the expiration date written in the file header record. Values are TRUE or FALSE. Use this flag to set the corresponding logical bit flag for BCK_OPT_K_BEFORE_TYPE and BCK_OPT_K_SINCE_TYPE. (See the BACKUP qualifiers /BEFORE, /SINCE, and /EXPIRED.)
BCK_OPT_K_FAST	Flag that specifies whether to reduce processing time by performing a fast file scan of the input specifier. Values are TRUE and FALSE (default). (See the BACKUP qualifier /FAST.)
BCK_OPT_K_FILE_CALLBACK	Reserved for use by VSI.
BCK_OPT_K_FILEMERGE	Reserved for use by VSI.
BCK_OPT_K_FULL	Flag that specifies whether to display information produced by a BCK_OPT_K_LIST value of TRUE in a format similar to that produced by the DCL command DIRECTORY/FULL. Values are TRUE and FALSE (default). (See the BACKUP qualifiers /LIST and /FULL.)
BCK_OPT_K_GROUP	Integer that specifies the number of backup blocks or backup buffers BACKUP places in each redundancy group. The default is 10 blocks. (See the BACKUP qualifier /GROUP_SIZE.)
BCK_OPT_K_HANDLE	Reserved for use by VSI.
BCK_OPT_K_IGNORE_TYPES	Logical bit flags that override tape labeling checks or restrictions placed on files. Values are one of the following: **BCK_OPTYP_IGNORE_K_ACCESS** Processes files on a tape that is protected by a volume accessibility character, or a tape

Option Structure	Description
	created by HSC Backup. Applies to all tapes in the save set. **BCK_OPTYP_IGNORE_K_INTERLOCK** Processes files otherwise inaccessible because of file access conflicts. **BCK_OPTYP_IGNORE_K_LABELS** Ignores the contents of the volume header record. You cannot use this flag if the BCK_OPTYP_K_EXACT_ORDER option structure flag value is TRUE. **BCK_OPTYP_IGNORE_K_NOBACKUP** Processes both the file header and the contents of files marked with the NOBACKUP option. (See the BACKUP qualifier /IGNORE.)
BCK_OPT_K_IMAGE	Flag that directs that an entire volume or volume set be processed. Values are TRUE and FALSE (default). (See the BACKUP qualifier /IMAGE.)
BCK_OPT_K_INCREMENTAL	Flag that specifies whether to restore an incremental save set. Values are TRUE and FALSE (default). (See the BACKUP qualifier /INCREMENTAL.)
BCK_OPT_K_INITIALIZE	Flag that specifies whether to initialize an entire output volume, thereby making its previous contents inaccessible. Values are TRUE and FALSE (default, except for image restore and copy operations). (See the BACKUP qualifier /INITIALIZE.)
BCK_OPT_K_INPUT	32-bit descriptor that specifies a single input-specifier. You can use wildcards. You must use a separate BCK_OPT_K_INPUT option structure for each specification.(See the BACKUP Format description.)
BCK_OPT_K_INTERCHANGE	Flag that specifies whether to process files in a manner suitable for data interchange. Values are TRUE and FALSE (default). (See the BACKUP qualifier /INTERCHANGE.)

Option Structure	Description
BCK_OPT_K_JOURNAL	Flag that specifies whether a BACKUP journal file is to be processed. You can specify a journal file name other than BACKUP.BJL (the default) with the BCK_OPT_K_JOURNAL_FILE option structure. Values are TRUE and FALSE (default). (See the BACKUP qualifier /JOURNAL.)
BCK_OPT_K_JOURNAL_FILE	32-bit descriptor that specifies the name of a BACKUP journal file to be processed.(See the BACKUP qualifier /JOURNAL.)
BCK_OPT_K_LABEL	32-bit descriptor that specifies the volume label to be written. To specify more than one label, use additional BCK_OPT_K_LABEL option structures. (See the BACKUP qualifier /LABEL.) **Note** Use of BCK_OPT_K_LABEL with any BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.
BCK_OPT_K_LIST	Flag that specifies whether to process a BACKUP list file. You can specify a list output destination other than TTY: (the default) with the BCK_OPT_K_LIST_FILE option structure. Values are TRUE and FALSE (default). (See the BACKUP qualifier /LIST.)
BCK_OPT_K_LIST_FILE	32-bit descriptor that specifies the name of a file of a BACKUP journal file to be processed.(See the BACKUP qualifier /LIST.)
BCK_OPT_K_LOG	Flag that specifies whether to display the file specification of each file processed. The display is to SYS$OUTPUT. Values are TRUE and FALSE (default). (See the BACKUP qualifier /LOG.)
BCK_OPT_K_MEDIA_FORMAT	Logical bit flags that specify whether data records are automatically compacted and blocked together. The tape drive must support compaction. Values are one of the following: *BCK_OPTYP_MEDIA_K_COMPACTION*, *BCK_OPTYP_MEDIA_K_NO_COMPACTION (default)*.

Option Structure	Description
	(See the BACKUP qualifier /MEDIA_FORMAT.) **Note** Use of BCK_OPT_K_MEDIA_FORMAT and BCK_OPT_K_DENSITY in the same call results in a fatal error.
BCK_OPT_K_MODIFIED	Flag that specifies whether to select files according to the modification date written in the file header record. Values are TRUE and FALSE. Use this flag to set the corresponding logical bit flag for BCK_OPT_K_BEFORE_TYPE and BCK_OPT_K_SINCE_TYPE. (See the BACKUP qualifiers /BEFORE, /SINCE, and /MODIFIED.)
BCK_OPT_K_NEW_VERSION	Flag that specifies whether to create a new version of a file if a file with an identical file specification already exists at the location to which the file is being copied or restored. Values are TRUE and FALSE (default). Because this qualifier causes version numbers to change, using it with the BCK_OPT_K_VERIFY flag set to TRUE can cause unpredictable results. VSI recommends that you not use these two options in combination. (See the BACKUP qualifier /NEW_VERSION.)
BCK_OPT_K_OPERATION_TYPE	Logical bit flags that specify the type of BACKUP operation to be performed. Values are one of the following: *BCK_OP_K_SAVE (default)*, *BCK_OP_K_RESTORE BCK_OP_K_COPY*, *BCK_OPT_K_LIST*, *BCK_OPT_K_COMPARE*
BCK_OPT_K_OUTPUT	32-bit descriptor that specifies the name of a single output-specifier. You can use wildcards. Each file specification requires a separate BCK_OPT_K_OUTPUT option structure. Lists are not supported.(See BACKUP Format description.)
BCK_OPT_K_OVERLAY	Flag that specifies whether to overlay (at the same physical location) an existing file with a file specification identical to that of the file that is being copied or restored.

Option Structure	Description
	Values are TRUE and FALSE (default). (See the BACKUP qualifier /OVERLAY.)
BCK_OPT_K_OWNER_IN_VALUE	Integer that specifies the user identification code (UIC) of the files to be processed by a BACKUP input operation. The default is the UIC of the current process. If you do not include this option structure, BACKUP processes all files specified by BCK_OPT_K_INPUT.(See the BACKUP qualifier /BY_OWNER.)
BCK_OPT_K_OWNER_OUT_TYPE	Logical bit flags to specify the user identification code (UIC) of restored files. Values are one of the following: **BCK_OPTYP_OWN_OUT_K_DEFAULT** Sets the owner UIC to the UIC of the current process (default unless BCK_OPT_K_IMAGE or BCK_OPT_K_INCREMENTAL is TRUE). **BCK_OPTYP_OWN_OUT_K_ORIGINAL** Retains the owner UIC of the file being restored (default if BCK_OPT_K_IMAGE or BCK_OPT_K_INCREMENTAL is TRUE). **BCK_OPTYP_OWN_OUT_K_PARENT** Sets the owner UIC to the owner UIC of the directory to which the file is being written. The current process must have the SYSPRV user privilege, or be the owner of the output volume, or must have the parent UIC. (See the BACKUP qualifier /BY_OWNER.)
BCK_OPT_K_OWNER_OUT_VALUE	Integer that redefines the UIC of the files written by a BACKUP restore or copy operation, or specifies the UIC of an output save set. If BCK_OPT_K_OUTPUT specifies a save set, the default is the UIC of the current process. To specify the UIC of a Files-11 save set, the current process must have the SYSPRV user privilege, or must have the UIC specified. If BCK_OPT_K_OUTPUT specifies files, the UIC of the output files is set to the UIC specified. To specify the UIC, the UIC must be that of the current process, or must have the SYSPRV user privilege, or the current process must be the owner of the output device.

Option Structure	Description
	(See the BACKUP qualifier /BY_OWNER.)
BCK_OPT_K_PHYSICAL	Flag that specifies that a BACKUP operation is to ignore any file structure on the input volume and instead process the volume in terms of logical blocks. Values are TRUE and FALSE (default). Note that output operations on a save set must be performed with the same physical option as that used to create the save set. (See the BACKUP qualifier /PHYSICAL.) **Note** Use of BCK_OPT_K_PHYSICAL and BCK_OPT_K_UNSHELVE or BCK_OPT_K_ALIAS in the same call results in a fatal error.
BCK_OPT_K_PROTECTION	Logical bit flags that specify file protection. Bits 0 to 15 of the option structure value field are in the format of the RMS field XAB$W_PRO. See the *VSI OpenVMS Record Management Services Reference Manual* for information about the format of this field. (Also see BACKUP utility qualifier /PROTECTION.)
BCK_OPT_K_RECORD	Flag that specifies whether to record the current date and time in the BACKUP date field in each file header once a file is successfully saved or copied. Values are TRUE and FALSE (default). (See the BACKUP qualifier /RECORD.)
BCK_OPT_K_RELEASE_TAPE	Flag that specifies whether to dismount and unload a tape after a BACKUP save operation has either reached the end of the tape or has written and verified the save set. Values are TRUE and FALSE (default). (See the BACKUP qualifier /RELEASE_TAPE.)
BCK_OPT_K_REPLACE	Flag that specifies whether to replace (at a different physical location), with an identical version number, an existing file with a file specification identical to that of the file that is being copied or restored. Values are TRUE and FALSE (default).

Option Structure	Description
	(See the BACKUP qualifier /REPLACE.)
BCK_OPT_K_REWIND	Flag. Reserved for use by VSI.
BCK_OPT_K_REWIND_IN	Flag that specifies whether the input device is a tape drive, and that it is to be rewound to the beginning-of-tape marker before beginning the BACKUP operation. Values are TRUE and FALSE (default). (See the BACKUP qualifier /REWIND.)
BCK_OPT_K_REWIND_OUT	Flag that specifies whether the output device is a tape drive, and that it is to be rewound to the beginning-of-tape marker and initialized before beginning the BACKUP operation. Values are TRUE and FALSE (default). (See the BACKUP qualifier /REWIND.) **Note** Use of BCK_OPT_K_REWIND_OUT with any BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.
BCK_OPT_K_SAVE_SET_IN	Flag that indicates whether the input specifier is a BACKUP save-set file. Values are TRUE and FALSE (default; indicates that the input specifier refers to a Files-11 file). (See the BACKUP qualifier /SAVE_SET.)
BCK_OPT_K_SAVE_SET_OUT	Flag that indicates whether the output specifier specifies a BACKUP save-set file. Values are TRUE and FALSE (default; indicates that the output specifier refers to a Files-11 file). (See the BACKUP qualifier /SAVE_SET.)
BCK_OPT_K_SCRATCH_ASGN_TYPE[1]	Logical bit flags. **Note** Use of BCK_OPT_K_SCRATCH_ASGN_TYPE with BCK_OPT_K_LABEL, BCK_OPT_K_REWIND_OUT, any of the BCK_OPT_K_CARTRIDGE_* option structures, or any other BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.
BCK_OPT_K_SCRATCH_COLLECTION[1]	32-bit descriptor.

Option Structure	Description
	Note Use of BCK_OPT_K_SCRATCH_COLLECTION with BCK_OPT_K_LABEL, BCK_OPT_K_REWIND_OUT, any of the BCK_OPT_K_CARTRIDGE_* option structures, or any other BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.
BCK_OPT_K_SCRATCH_LOCATION[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_SCRATCH_LOCATION with BCK_OPT_K_LABEL, BCK_OPT_K_REWIND_OUT, any of the BCK_OPT_K_CARTRIDGE_* option structures, or any other BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.
BCK_OPT_K_SCRATCH_MEDIA_NAME[1]	32-bit descriptor. **Note** Use of BCK_OPT_K_SCRATCH_MEDIA_NAME with BCK_OPT_K_LABEL, BCK_OPT_K_REWIND_OUT, any of the BCK_OPT_K_CARTRIDGE_* option structures, or any other BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.
BCK_OPT_K_SELECT	32-bit descriptor that references the file specification of a file or files from the input save set to be processed by the current BACKUP save or copy operation. Wildcards are permitted. Each file specification, whether wildcards are used or not, requires its own BCK_OPT_K_SELECT option structure (lists are not supported).(See the BACKUP qualifier /SELECT.)
BCK_OPT_K_SINCE_TYPE	Logical bit flags that qualify the date specified in the BCK_OPT_K_SINCE_VALUE option structure. Type can be one of the following: **BCK_OPTYP_SINCE_K_BACKUP** Selects files last saved or copied by BACKUP on or after the date specified. Also selects files with no BACKUP date.

Option Structure	Description
	BCK_OPTYP_SINCE_K_CREATED Selects files created on or after the date specified. **BCK_OPTYP_SINCE_K_EXPIRED** Selects files that have expired since the date specified. **BCK_OPTYP_SINCE_K_MODIFIED** Selects files last modified on or after the date specified (default). **BCK_OPTYP_SINCE_K_SPECIFIED** Reserved for use by VSI. (See the BACKUP qualifiers /SINCE, /BACKUP, /CREATED, /EXPIRED, and /MODIFIED.)
BCK_OPT_K_SINCE_VALUE	Date-Time Quadword that specifies the date qualified by BCK_OPTYP_K_SINCE_TYPE. You cannot use delta time.(See the BACKUP qualifier /SINCE.)
BCK_OPT_K_STORAGE_MANAGEMENT[1]	32-bit descriptor.
BCK_OPT_K_TAPE_EXPIRATION	ADT (Date-Time) that specifies when the tape expires.(See the BACKUP qualifier /TAPE_EXPIRATION.)
BCK_OPT_K_TRUNCATE	Flag that specifies whether a copy or restore operation truncates a sequential output file at the end-of-file (EOF) when creating it. Values are TRUE and FALSE (default; the size of the output file is determined by the allocation of the input file). (See the BACKUP qualifier /TRUNCATE.)
BCK_OPT_K_UNSHELVE	Flag that is reserved for use with file-shelving layered products. Values are TRUE and FALSE. **Note** Use of BCK_OPT_K_UNSHELVE and BCK_OPT_K_PHYSICAL in the same call results in a fatal error.
BCK_OPT_K_VALIDATE_PARAMETERS	Reserved for use by VSI.

Option Structure	Description
BCK_OPT_K_VERIFY	Flag that specifies whether the contents of the output specifier be compared with the contents of the input specifier after a save, restore, or copy operation has been completed. Values are TRUE and FALSE (default). (See the BACKUP qualifier /VERIFY.)
BCK_OPT_K_VOLUME	Integer that specifies the specific disk volume in a disk volume set to be processed (valid only when BCK_OPT_K_IMAGE is TRUE). (See the BACKUP qualifier /VOLUME.)

[1]Reserved for use by Media Management Extension (MME) layered products.

[b]Reserved for future use by a security utility or layered product.

Description

Application programs call the Backup API to invoke the OpenVMS Backup utility via a call to the BACKUP$START routine. There is only one parameter, the address of an argument buffer that contains a number of option structures that together define the operation requested of the Backup utility. Most of these option structures are equivalent, singly or in combination, to the qualifiers available when invoking the BACKUP utility with the DCL command BACKUP; the call to the API is analogous to a user entering an interactive command to the Backup utility.

The call to BACKUP$START is synchronous; that is, it does not return until the operation is complete or is terminated by a fatal error. In the case of a fatal error, the call is aborted.

BACKUP Event Callbacks

An application can request that the BACKUP API notify the application whenever specific events occur. The application can specify different callback routines to handle different types of BACKUP events, or one routine to handle all events. To do so, the application registers the callback routine by including option structure BCK_OPTYP_K_EVENT_CALLBACK in the call to BACKUP$START. This option structure specifies an event type (or all events) and the address of a routine to be called when the event occurs. The application must include one such option structure for each requested event type. To specify all events, use BCK_EVENT_K_ALL. Table 3.4 lists the specific event types and identifiers.

A callback routine:

- Is called with one argument; a pointer to a bckEvent data structure that contains information to enable the application to process the event
- Returns an unsigned integer status value (of any valid OpenVMS message) in R0 to enable the API to perform proper logging of the event

Note

The API does not currently process the return status of the callback routine. However, VSI strongly recommends that the callback routine provide the appropriate status in R0 when returning control to the API.

The bckEvent structure contains information about the type of event, and also contains a descriptor of a data structure that contains information to be used to process the event. The bckEvent structure

may point to a bckControl structure that specifies control aspects of an event that may require user or operator action.

Table 3.3 describes the format of the bckEvent data structure. Table 3.6 describes the format of the bckControl data structure.

Table 3.3. bckEvent Format

Data Type	Element Name	Description
struct dsc$descriptor	bckevt_r_event_buffer	Pointer to event data
unsigned int	bckevt_l_event_type	Event type
unsigned int	bckevt_l_event_subtype	Event subtype (if any)
unsigned int	bckevt_q_event_ctx [2]	Reserved for use by VSI
unsigned int	bckevt_l_event_handle	Reserved for use by VSI

Table 3.4 describes the values returned in the bckEvent data structure.

Table 3.4. Event Callback Buffer Formats

Type/Subtype		Format	Value Returned
BCK_EVENT_K_CONTROL		bckControl	See Table 3.5.
BCK_EVENT_K_ERROR_MSG			
	(no subtype)	bckMsgVect	Message vector (use $PUTMSG to output message to user).
BCK_EVENT_K_JOURNAL_OPEN			
	(no subtype)	dsc$descriptor	String descriptor (name of file to create).
BCK_EVENT_K_JOURNAL_CLOSE			
	(no subtype)	dsc$descriptor	String descriptor (name of file to close).
BCK_EVENT_K_JOURNAL_WRITE			
	(no subtype)	512-byte block	File descriptor of journal buffer (condensed journal records, refer to the BJLDEF structure definition in the BAPIDEF files).
Type/Subtype		**Format**	**Value Returned**
BCK_EVENT_K_LIST_CLOSE			
	(no subtype)	Array of 2 longwords	LIST_TOTFILE: Total files listed. LIST_TOTSIZE: Total blocks listed. **Note:** The application should close the list file.
BCK_EVENT_K_LIST_OPEN			
	TRUE	dsc$descriptor	File specification of list file to open (TRUE = 1, indicates / FULL listing).

Type/Subtype		Format	Value Returned
	FALSE	dsc$descriptor	(FALSE = 0).
BCK_EVENT_K_LIST_WRITE			
	BRH$K_SUMMARY	BSRBLK	List BACKUP save set - save set summary record.
	BRH$K_VOLUME	BSRBLK	List BACKUP save set - volume summary record.
	BRH$K_PHYSVOL	PVABLK	List BACKUP save set - physical volume record.
	BRH$K_FILE	FARBLK	List BACKUP save set - file record.
BCK_EVENT_K_LISTJOUR_WRITE			Subtype is a condition value that indicates the type of action that occurred for the specified file/item. Obtain message text with the $GETMSG system service.
	TRUE	bckLisJourblk	Journal file listing information (TRUE = 1, indicates a change of volume or save set).
	FALSE	dsc$descriptor	Journal file listing of file/item specification string (descriptor) (FALSE = 0).
BCK_EVENT_K_LOG			
	BACKUP$_AECREATED	dsc$descriptor	String descriptor (file logging).
	BACKUP$_COMPARED	dsc$descriptor	String descriptor (file logging).
	BACKUP$_COPIED	dsc$descriptor	String descriptor (file logging).
	BACKUP$_CREATED	dsc$descriptor	String descriptor (file logging).
	BACKUP$_CREDIR	dsc$descriptor	String descriptor (file logging).
	BACKUP$_HEADCOPIED	dsc$descriptor	String descriptor (file logging).
	BACKUP$_INCDELETE	dsc$descriptor	String descriptor (file logging).
	BACKUP$_NEWSAVSET	dsc$descriptor	String descriptor (file logging).
BCK_EVENT_K_OP_PHASE			
	BACKUP$_STARTVERIFY	Condition Value	Start of verify operation (obtain message text with $GETMSG).
	BACKUP$_STARTDELETE	Condition Value	Start of delete operation (obtain message text with $GETMSG).
	BACKUP$_STARTRECORD	Condition Value	Start of record operation (obtain message text with $GETMSG).
BCK_EVENT_K_SAVESET_CLOSE			
	(no subtype)	RMS FOB	A BACKUP save set must be closed.
BCK_EVENT_K_SAVESET_OPEN			

Type/Subtype		Format	Value Returned
	(no subtype)	RMS FOB	A BACKUP save set must be opened or created.
BCK_EVENT_K_SAVESET_READ			
	(no subtype)	BACKUP Buffer Control Block (BCBBLK)	A BACKUP save set block/ buffer has been read from the input save set.
BCK_EVENT_K_SAVESET_WRITE			
	(no subtype)	BACKUP Buffer Control Block (BCBBLK)	A BACKUP save set block/ buffer is ready to be written to the output save set.
BCK_EVENT_K_STATISTICS			
	(no subtype)	bckMsgVect	Statistics message; one of the following message condition values (use $PUTMSG to output message to user): *BACKUP$_STAT_PHYSICAL*, *BACKUP $_STAT_SAVCOP_ACT*, *BACKUP$_STAT_INACTIVE*, *BACKUP$_STAT_COMPARE*, *BACKUP$_STAT_RESTORE*.
BCK_EVENT_K_USER_MSG			
	(no subtype)	bckMsgVect	Message vector (use $PUTMSG to output message to user).

Table 3.5 describes the control event subtypes of the BCK_EVENT_K_CONTROL event callback. Table 3.6 describes the format of the bckControl data structure.

Table 3.5. Control Event Subtypes

	Format	
Subtype	**Field**	**Description**
BCKEVTST_K_CONFIRM_EVENT		**Confirmation is required for compare or copy operation.**
	bckCntrl_l_event	BCKCNTRL_K_CONFIRM_EVENT
	bckCntrl_l_function	Backup operation type (integer value)
	bckCntrl_a_outmsgvect	Confirmation message (bckMsgVect, BACKUP$_CNTRL_CONFCOMP or BACKUP$_CNTRL_CONFCOPY)
	bckCntrl_v_response_required	TRUE (response is required)
	bckCntrl_r_response_buffer	dsc$descriptor ("Yes/No" string descriptor)
BCKEVTST_K_ASSIST_EVENT		**Operator or user assistance is required to determine continuation/actions.**

	Format	
Subtype	**Field**	**Description**
	bckCntrl_l_event	BCKCNTRL_K_USER_ASSIST_EVENT or BCKCNTRL_K_OPER_ASSIST_EVENT
	bckCntrl_l_function	Backup operation type (integer value)
	bckCntrl_a_outmsgvect	bckMsgVect (assist and other messages)
	bckCntrl_v_response_required	TRUE or FALSE (TRUE = 1, if response is required)
	bckCntrl_r_response_buffer	dsc$descriptor (response string descriptor)
BCKCNTRL_K_RESTART_EVENT		**BACKUP operation restart is initiated.**
	bckCntrl_l_event	BCKCNTRL_K_RESTART_EVENT
	bckCntrl_l_function	Backup operation type (integer value)
	bckCntrl_a_outmsgvect	bckMsgVect (operation restart message vector)
	bckCntrl_v_response_required	FALSE (= 0, no response is required)
	bckCntrl_r_response_buffer	dsc$descriptor ("Yes/No" string descriptor)

Control events are described by the Control event subtype, via the bckevt_l_event_subtype field in the bckEvent structure. Table 3.6 describes the format of the bckControl data structure.

Table 3.6. bckControl Format

Data Type	Element Name	Description
unsigned int	bckCntrl_l_event	Control event type.
unsigned int	bckCntrl_l_function	Backup operation type.
bckMsgVect	*bckCntrl_a_outmsgvect	Output messages and parameters.
union {		
unsigned int	bckCntrl_l_ctlflags	Flags.
struct {		
unsigned	bckCntrl_v_response_required : 1	Response required = 1.
unsigned	bckCntrl_v_fill_5 : 7	Filler.
}		
}		
struct dsc$descriptor	bckCntrl_r_response_buffer	Descriptor for buffer to which response text is to be written.
unsigned int	bckCntrl_l_response_status	Reserved for use by VSI.
unsigned int	bckCntrl_l_control_options	Reserved for use by VSI.

Error Messages

Where possible, the Backup API emulates the behavior of the interactive BACKUP utility if you pass a call that contains conflicting qualifiers by:

1. Making a best guess as to your intentions
2. Ignoring the least likely of the conflicting qualifiers
3. Issuing a message that warns of the conflicting qualifiers
4. Processing the BACKUP request

See the *VSI OpenVMS System Management Utilities Reference Manual* for a table of valid combinations of BACKUP qualifiers.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

BACKUP$_BADOPTDSC

Invalid callable interface option descriptor.

BACKUP$_BADOPTTYP

Invalid callable interface option type.

BACKUP$_BADOPTVAL

Invalid callable interface option value.

BACKUP$_BADOPTVALQ

Invalid callable interface option value.

BACKUP$_DUPOPT

Previously specified callable interface option type invalid.

BACKUP$_NOAPIARGS

Callable interface required parameter not specified or invalid.

Any condition value returned by the OpenVMS Backup utility.

Example

The following C example program demonstrates calling the Backup API to perform the following DCL commands:

```
$ BACKUP/LOG/VERIFY/CRC/ALIAS APITEST1_IN:*.*;* -
_$ APITEST1_OUT:A.BCK/SAVE_SET

$ BACKUP/LOG/VERIFY/CRC/ALIAS APITEST1_OUT:A.BCK/SAVE_SET -
_$ APITEST2_OUT:*.*;*
```

```
#include <stdio.h>
#include <stdlib.h>
#include <ssdef.h>
#include <descrip.h>
#include "sys$examples:bapidef.h"

/*
```

```
** Define a fixed size (simple) structure for specifying the
** BACKUP operation.
*/
typedef struct _buf_arg
               {
                bck_opt_struct_flag arg1;
                bck_opt_struct_flag arg2;
                bck_opt_struct_flag arg3;
                bck_opt_struct_flag arg4;
                bck_opt_struct_dsc arg5;
                bck_opt_struct_dsc arg6;
                bck_opt_struct_flag arg7;
                bck_opt_struct_flag arg8;
                bck_opt_struct_adr arg9;
                bck_opt_struct_adr arg10;
                bck_opt_struct_adr arg11;
                bck_opt_struct_flag arg12;
                bck_opt_struct_flag arg13;
               } buf_arg;


struct dsc$descriptor
               input_dsc,
               output_dsc,
               event_type_dsc;
buf_arg myarg_buff;
unsigned int status;

extern unsigned int backup$start(buf_arg  *myarg_buff);
unsigned int subtest(bckEvent *param);

static char  input_str[]      = "APITEST1_IN:";
static char  output_str1[]      = "APITEST1_OUT:a.bck";
static char  output_str2[]      = "APITEST2_OUT:";

main()
{
 myarg_buff.arg1.option_type = BCK_OPT_K_ALIAS;
 myarg_buff.arg1.opt_flag_value = TRUE;

 myarg_buff.arg2.option_type = BCK_OPT_K_VERIFY;
 myarg_buff.arg2.opt_flag_value = TRUE;

 myarg_buff.arg3.option_type = BCK_OPT_K_CRC;
 myarg_buff.arg3.opt_flag_value = TRUE;

 myarg_buff.arg4.option_type = BCK_OPT_K_LOG;
 myarg_buff.arg4.opt_flag_value = TRUE;

 myarg_buff.arg5.opt_dsc_type = BCK_OPT_K_INPUT;
 myarg_buff.arg5.opt_dsc.dsc$b_dtype = DSC$K_DTYPE_T;
 myarg_buff.arg5.opt_dsc.dsc$b_class = DSC$K_CLASS_S;
 myarg_buff.arg5.opt_dsc.dsc$w_length = sizeof(input_str) - 1;
 myarg_buff.arg5.opt_dsc.dsc$a_pointer = input_str;


 myarg_buff.arg6.opt_dsc_type = BCK_OPT_K_OUTPUT;
 myarg_buff.arg6.opt_dsc.dsc$b_dtype = DSC$K_DTYPE_T;
```

```
myarg_buff.arg6.opt_dsc.dsc$b_class = DSC$K_CLASS_S;
myarg_buff.arg6.opt_dsc.dsc$w_length = sizeof(output_str1) - 1;
myarg_buff.arg6.opt_dsc.dsc$a_pointer = output_str1;

myarg_buff.arg7.option_type = BCK_OPT_K_SAVE_SET_OUT;
myarg_buff.arg7.opt_flag_value = TRUE;

myarg_buff.arg8.option_type = BCK_OPT_K_OPERATION_TYPE;
myarg_buff.arg8.opt_flag_value = BCK_OP_K_SAVE ;

myarg_buff.arg9.opt_adr_type = BCK_OPT_K_EVENT_CALLBACK;
myarg_buff.arg9.opt_adr_attributes = BCK_EVENT_K_LOG;
myarg_buff.arg9.opt_adr_value[0] = (int *)subtest;
myarg_buff.arg9.opt_adr_value[1] = 0;


/*
** Specify that this application will handle user-visible messages.
** (The operation phase, and user/file-logging messages.)
*/
myarg_buff.arg10.opt_adr_type = BCK_OPT_K_EVENT_CALLBACK;
myarg_buff.arg10.opt_adr_attributes = BCK_EVENT_K_OP_PHASE;
myarg_buff.arg10.opt_adr_value[0] = (int *)subtest;
myarg_buff.arg10.opt_adr_value[1] = 0;

myarg_buff.arg11.opt_adr_type = BCK_OPT_K_EVENT_CALLBACK;
myarg_buff.arg11.opt_adr_attributes = BCK_EVENT_K_USER_MSG;
myarg_buff.arg11.opt_adr_value[0] = (int *)subtest;
myarg_buff.arg11.opt_adr_value[1] = 0;


/*
** Indicate the end of options that specify the BACKUP operation
** to be performed.
*/
myarg_buff.arg12.option_type = BCK_OPT_K_END_OPT;
myarg_buff.arg12.opt_flag_value = FALSE;

/*
** Notes:
**  An extra option structure (# 13) was allocated for testing.
**
**  The DCL command analogous to the following BACKUP API call
** is illustrated below.
**
** "$ BACKUP/LOG/VERIFY/CRC/ALIAS APITEST1_IN:*.*;* -"
** "_$  APITEST1_OUT:a.bck/SAVE_SET "
*/


       status = backup$start(&myarg_buff);

if (! (status & 1))
{
    exit (status);  /* EXIT if the first part of the test failed. */
}

/*
```

```
 **  Now use the resultant saveset to perform a restore operation.
 */

 /*
 **  Change the input string to specify the saveset, ("output_str1").
 */
 myarg_buff.arg5.opt_dsc.dsc$w_length = sizeof(output_str1) - 1;
 myarg_buff.arg5.opt_dsc.dsc$a_pointer = output_str1;


 /*
 **  Change the output string to specify the output device/directory).
 */
 myarg_buff.arg6.opt_dsc.dsc$w_length = sizeof(output_str2) - 1;
 myarg_buff.arg6.opt_dsc.dsc$a_pointer = output_str2;

 /*
 **  Change the option to denote it is now an input saveset,
 **  (not an output saveset).
 */
 myarg_buff.arg7.option_type = BCK_OPT_K_SAVE_SET_IN;


 /*
 **  Change the option to specify a restore operation,
 **  (not a save operation).
 */
 myarg_buff.arg8.opt_flag_value = BCK_OP_K_RESTORE;

 /*
 **  The DCL command analogous to the following BACKUP API call
 ** is illustrated below.
 **
 ** "$ BACKUP/LOG/VERIFY/CRC/ALIAS APITEST1_OUT:a.bck/SAVE_SET -"
 ** "_$  APITEST2_OUT:*.*;*"
 */

 status = backup$start(&myarg_buff);

        exit (status);
}


unsigned int subtest(bckEvent *param)
{

 printf("\n BACKUP API Event Type = %d,\n",param->bckevt_l_event_type);
 printf("               Subtype = %d\n",param->bckevt_l_event_subtype);

 if (param->bckevt_l_event_type == BCK_EVENT_K_LOG)
 {
     printf(" BACKUP API LOG Event item:\n %.*s\n",
      param->bckevt_r_event_buffer.dsc$w_length,
      param->bckevt_r_event_buffer.dsc$a_pointer);
 }


 if (param->bckevt_l_event_type == BCK_EVENT_K_OP_PHASE)
```

```
 {
     printf(" BACKUP API Operation Phase Event\n %.*s\n",
      param->bckevt_r_event_buffer.dsc$w_length,
      param->bckevt_r_event_buffer.dsc$a_pointer);
 }

 fflush(stdout);

        return (1);
}
```

Chapter 4. Command Language Interface (CLI) Routines

The command language interface (CLI) routines process command strings using information from a command table. A command table contains command definitions that describe the allowable formats for commands. To create or modify a command table, you must write a command definition file and then process this file with the Command Definition Utility (the SET COMMAND command). For information about how to use the Command Definition Utility, see the *VSI OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

4.1. Introduction to CLI Routines

The CLI routines include the following:

- CLI$DCL_PARSE
- CLI$DISPATCH
- CLI$GET_VALUE
- CLI$PRESENT

When you use the Command Definition Utility to add a new command to your process command table or to the DCL command table, use the CLI$PRESENT and CLI$GET_VALUE routines in the program invoked by the new command. These routines retrieve information about the command string that invokes the program.

When you use the Command Definition Utility to create an object module containing a command table and you link this module with a program, you must use all four CLI routines. First, use CLI $DCL_PARSE and CLI$DISPATCH to parse command strings and invoke routines. Then, use CLI $PRESENT and CLI$GET_VALUE within the routines that execute each command.

Note that the application program should declare referenced constants and return status symbols as external symbols; these symbols are resolved upon linking with a utility shareable image.

A CLI must be present in order to use the CLI routines. If your application can be run from a detached process, the application should first verify that a CLI exists. For information about how to verify that a CLI exists for a process, see the description of the $GETJPI system service in the *VSI OpenVMS System Services Reference Manual*.

Note

Do not use the CLI routines to obtain values from foreign commands. Using a foreign command to activate an image (instead of the SET COMMAND command) disrupts the building of the DCL parse tables.

4.2. Using the CLI Routines: An Example

Example 4.1 contains a command definition file (SUBCOMMANDS.CLD) and a Fortran program (INCOME.FOR). INCOME.FOR uses the command definitions in SUBCOMMANDS.CLD to process commands. To execute the example, enter the following commands:

```
$ SET COMMAND SUBCOMMANDS/OBJECT=SUBCOMMANDS
```

```
$ FORTRAN INCOME
$ LINK INCOME,SUBCOMMANDS
$ RUN INCOME
```

INCOME.FOR accepts a command string and parses it using CLI$DCL_PARSE. If the command string is valid, the program uses CLI$DISPATCH to execute the command. Each routine uses CLI$PRESENT and CLI$GET_VALUE to obtain information about the command string.

Example 4.1. Using the CLI Routines to Retrieve Information About Command Lines in a Fortran Program

```
!************************************************
!                   SUBCOMMANDS.CLD
!************************************************

MODULE INCOME_SUBCOMMANDS

DEFINE VERB ENTER
ROUTINE ENTER

DEFINE VERB FIX
ROUTINE FIX
QUALIFIER HOUSE_NUMBERS, VALUE (LIST)

DEFINE VERB REPORT
ROUTINE REPORT
QUALIFIER OUTPUT, VALUE (TYPE = $FILE,
                         DEFAULT = "INCOME.RPT")
                         DEFAULT


!************************************************
!                   INCOME.FOR
!************************************************
PROGRAM INCOME
INTEGER STATUS,
2       CLI$DCL_PARSE,
2       CLI$DISPATCH
INCLUDE '($RMSDEF)'
INCLUDE '($STSDEF)'
EXTERNAL INCOME_SUBCOMMANDS,
2        LIB$GET_INPUT

! Write explanatory text
STATUS = LIB$PUT_OUTPUT
2 ('Subcommands: ENTER - FIX - REPORT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LIB$PUT_OUTPUT
2 ('Press Ctrl/Z to exit')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get first subcommand
STATUS = CLI$DCL_PARSE (%VAL (0),
2                       INCOME_SUBCOMMANDS, ! CLD module
2                       LIB$GET_INPUT,      ! Parameter routine
2                       LIB$GET_INPUT,      ! Command routine
2                       'INCOME> ')         ! Command prompt
 ! Do it until user presses Ctrl/Z
DO WHILE (STATUS .NE. RMS$_EOF)
```

```
! If no error on dcl_parse
IF (STATUS) THEN
! Dispatch depending on subcommand
STATUS = CLI$DISPATCH ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Do not signal warning again
ELSE IF (IBITS (STATUS, 0, 3) .NE. STS$K_WARNING) THEN
CALL LIB$SIGNAL (%VAL (STATUS))
END IF
! Get another subcommand
STATUS = CLI$DCL_PARSE (%VAL (0),
2                         INCOME_SUBCOMMANDS, ! CLD module
2                         LIB$GET_INPUT,      ! Parameter routine
2                         LIB$GET_INPUT,      ! Command routine
2                         'INCOME> ')         ! Command prompt
END DO
END

INTEGER FUNCTION ENTER ()
INCLUDE '($SSDEF)'
TYPE *, 'ENTER invoked'
ENTER = SS$_NORMAL
END

INTEGER FUNCTION FIX ()
INTEGER STATUS,
2       CLI$PRESENT,
2       CLI$GET_VALUE
CHARACTER*15 HOUSE_NUMBER
INTEGER*2    HN_SIZE
INCLUDE '($SSDEF)'
EXTERNAL CLI$_ABSENT
TYPE *, 'FIX invoked'
! If user types /house_numbers=(n,...)
IF (CLI$PRESENT ('HOUSE_NUMBERS')) THEN
! Get first value for /house_numbers
STATUS = CLI$GET_VALUE ('HOUSE_NUMBERS',
2                         HOUSE_NUMBER,
2                         HN_SIZE)
! Do it until the list is depleted
DO WHILE (STATUS)
TYPE *, 'House number = ', HOUSE_NUMBER (1:HN_SIZE)
STATUS = CLI$GET_VALUE ('HOUSE_NUMBERS',
2                           HOUSE_NUMBER,
2                           HN_SIZE)
END DO
! Make sure termination status was correct
IF (STATUS .NE. %LOC (CLI$_ABSENT)) THEN
CALL LIB$SIGNAL (%VAL (STATUS))
END IF
END IF
FIX = SS$_NORMAL
END

INTEGER FUNCTION REPORT ()
INTEGER STATUS,
2       CLI$GET_VALUE
CHARACTER*255 FILENAME
```

```
INTEGER*2    FN_SIZE
INCLUDE '($SSDEF)'
TYPE *, 'REPORT entered'
! Get value for /output
STATUS = CLI$GET_VALUE ('OUTPUT',
2                        FILENAME,
2                        FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
TYPE *, 'Output file: ', FILENAME (1:FN_SIZE)
REPORT = SS$_NORMAL
END
```

4.3. CLI Routines

This section describes the individual CLI routines.

CLI$DCL_PARSE

Parse DCL Command String — The CLI$DCL_PARSE routine supplies a command string to DCL for parsing. DCL separates the command string into its individual elements according to the syntax specified in the command table.

Format

```
CLI$DCL_PARSE [command_string] ,table [,param_routine]  [,prompt_routine]
 [,prompt_string]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

command_string

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor—fixed length

Character string containing the command to be parsed. The *command_string* argument is the address of a descriptor specifying the command string to be parsed. If the command string includes a comment (delimited by an exclamation mark), DCL ignores the comment.

If the command string contains a hyphen to indicate that the string is being continued, DCL uses the routine specified in the *prompt_routine* argument to obtain the rest of the string. The command

string is limited to 256 characters. However, if the string is continued with a hyphen, CLI$DCL_PARSE can prompt for additional input until the total number of characters is 1024.

If you specify the *command_string* argument as zero and specify a prompt routine, then DCL prompts for the entire command string. However, if you specify the *command_string* argument as zero and also specify the *prompt_routine* argument as zero, DCL restores the parse state of the command string that originally invoked the image.

CLI$DCL_PARSE does not perform DCL-style symbol substitution on the command string.

table

OpenVMS usage:	address
type:	address
access:	read only
mechanism:	by value

Address of the compiled command tables to be used for command parsing. The command tables are compiled separately by the Command Definition Utility using the DCL command SET COMMAND/OBJECT and are then linked with your program. A global symbol is defined by the Command Definition Utility that provides the address of the tables. The global symbol's name is taken from the module name given on the MODULE statement in the command definition file, or from the file name if no MODULE statement is present.

param_routine

OpenVMS usage:	procedure
type:	procedure value
access:	read only
mechanism:	by reference

Name of a routine to obtain a required parameter not supplied in the command text. The *param_routine* argument is the address of a routine containing a required parameter that was not specified in the *command_string* argument.

To specify the parameter routine, use the address of LIB$GET_INPUT or the address of a routine of your own that has the same three-argument calling format as LIB$GET_INPUT. See the description of LIB$GET_INPUT in the *VSI OpenVMS RTL Library (LIB$) Manual* for information about the calling format.

If LIB$GET_INPUT returns error status, CLI$DCL_PARSE propagates the error status outward or signals RMS$_EOF in the cases listed in the Description section.

You can obtain the prompt string for a required parameter from the command table specified in the *table* argument.

prompt_routine

OpenVMS usage:	procedure
type:	procedure value
access:	read only

mechanism:	by reference

Name of a routine to obtain all or part of the text of a command. The *prompt_routine* argument is the address of a routine to obtain the text or the remaining text of the command depending on the *command_string* argument. If you specify a zero in the *command_string* argument, DCL uses this routine to obtain an entire command line. DCL uses this routine to obtain a continued command line if the command string (obtained from the *command_string* argument) contains a hyphen to indicate that the string is being continued.

To specify the prompt routine, use the address of LIB$GET_INPUT or the address of a routine of your own that has the same three-argument calling format as LIB$GET_INPUT. See the description of LIB $GET_INPUT in the *VSI OpenVMS RTL Library (LIB$) Manual* for information about the calling format.

If LIB$GET_INPUT returns error status, CLI$DCL_PARSE propagates the error status outward or signals RMS$_EOF in the cases listed in the Description section.

prompt_string

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Character string containing a prompt. The *prompt_string* argument is the address of a string descriptor pointing to the prompt string to be passed as the second argument to the *prompt_routine* argument.

If DCL is using the prompt routine to obtain a continuation line, DCL inserts an underscore character before the first character of the prompt string to create the continuation prompt. If DCL is using the prompt routine to obtain an entire command line (that is, a zero was specified as the *command_string* argument), DCL uses the prompt string exactly as specified.

The prompt string is limited to 32 characters. The string COMMAND> is the default prompt string.

Description

The CLI$DCL_PARSE routine supplies a command string to DCL for parsing. DCL parses the command string according to the syntax in the command table specified in the *table* argument.

The CLI$DCL_PARSE routine can prompt for required parameters if you specify a parameter routine in the routine call. In addition, the CLI$DCL_PARSE routine can prompt for entire or continued command lines if you supply the address of a prompt routine.

If you do not specify a command string to parse and the user enters a null string in response to the DCL prompt for a command string, CLI$DCL_PARSE immediately terminates and returns the status CLI $_NOCOMD.

If DCL prompts for a required parameter and the user presses Ctrl/Z, CLI$DCL_PARSE immediately terminates and returns the status CLI$_NOCOMD, regardless of whether you specify or do not specify a command string to parse. If DCL prompts for a parameter that is not required and the user presses Ctrl/ Z, CLI$DCL_PARSE returns the status CLI$_NORMAL.

Whenever CLI$DCL_PARSE encounters an error, it both signals and returns the error.

Condition Values Returned

CLI$_INVREQTYP

Calling process did not have a CLI to perform this function, or the CLI did not support the request.

CLI$_IVKEYW

Invalid keyword.

CLI$_IVQUAL

Unrecognized qualifier.

CLI$_IVVERB

Invalid or missing verb.

CLI$_NOCOMD

Routine terminated. You entered a null string in response to a prompt from the *prompt_routine* argument, causing the CLI$DCL_PARSE routine to terminate.

CLI$_NORMAL

Normal successful completion.

CLI$_ONEVAL

List of values not allowed; enter one only.

RMS$_EOF

Routine terminated. You pressed Ctrl/Z in response to a prompt, causing the CLI$DCL_PARSE routine to terminate.

CLI$DISPATCH

Dispatch to Action Routine — The CLI$DISPATCH routine invokes the subroutine associated with the verb most recently parsed by a CLI$DCL_PARSE routine call.

Format

```
CLI$DISPATCH [userarg]
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

userarg

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Data to be passed to the action routine. The *userarg* argument is a longword that contains the data to be passed to the action routine. This data can be used in any way you want.

Description

The CLI$DISPATCH routine invokes the subroutine associated with the verb most recently parsed by a CLI$DCL_PARSE routine call. If the routine is successfully invoked, the return status is the status returned by the action routine. Otherwise, a status of CLI$_INVROUT is returned.

Condition Values Returned

CLI$_INVREQTYP

Calling process did not have a CLI to perform this function or the CLI did not support the request.

CLI$_INVROUT

CLI$DISPATCH unable to invoke the routine. An invalid routine is specified in the command table, or no routine is specified.

CLI$GET_VALUE

Get Value of Entity in Command String — The CLI$GET_VALUE routine retrieves a value associated with a specified qualifier, parameter, keyword, or keyword path from the parsed command string.

Format

```
CLI$GET_VALUE entity_desc ,retdesc [,retlength]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

entity_desc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Character string containing the label (or name if no label is defined) of the entity. The *entity_desc* argument is the address of a string descriptor that points to an entity that may appear on a command line. The *entity_desc* argument can be expressed as one of the following:

- A parameter, qualifier, keyword name, or label
- A keyword path

The *entity_desc* argument can contain qualifiers, parameters, keyword names, or labels that were assigned with the LABEL clause in the command definition file. If you used the LABEL clause to assign a label to an entity, you must specify the label in the *entity_desc* argument. Otherwise, use the name of the entity.

Use a keyword path to reference keywords used as values of parameters, qualifiers, or other keywords. A keyword path contains a list of entity names or labels separated by periods. If the LABEL clause was used to assign a label to an entity, you must specify the label in the keyword path. Otherwise, you must use the name of the entity.

The following command string illustrates a situation where keyword paths are needed to uniquely identify keywords. In this command string, you can use the same keywords with more than one qualifier. (This is defined in the command definition file by having two qualifiers refer to the same DEFINE TYPE statement.)

```
$ NEWCOMMAND/QUAL1=(START=5,END=10)/QUAL2=(START=2,END=5)
```

The keyword path QUAL1.START identifies the START keyword when it is used with QUAL1; the keyword path QUAL2.START identifies the keyword START when it is used with QUAL2. Because the name START is an ambiguous reference if used alone, the keywords QUAL1 and QUAL2 are needed to resolve the ambiguity.

You can omit keywords from the beginning of a keyword path if they are not needed to unambiguously resolve a keyword reference. A keyword path can be no more than eight names long.

If you use an ambiguous keyword reference, DCL resolves the reference by checking, in the following order:

1. The parameters in your command definition file, in the order they are listed
2. The qualifiers in your command definition file, in the order they are listed
3. The keyword paths for each parameter, in the order the parameters are listed
4. The keyword paths for each qualifier, in the order the qualifiers are listed

DCL uses the first occurrence of the entity as the keyword path. Note that DCL does not issue an error message if you provide an ambiguous keyword. However, because the keyword search order may change in future releases of OpenVMS, you should never use ambiguous keyword references.

If the *entity_desc* argument does not exist in the command table, CLI$GET_VALUE signals a syntax error (by means of the signaling mechanism described in the *VSI OpenVMS Programming Concepts Manual*).

retdesc

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Character string containing the value retrieved by CLI$GET_VALUE. The *retdesc* argument is the address of a string descriptor pointing to the buffer to receive the string value retrieved by CLI $GET_VALUE. The string is returned using the STR$COPY_DX Run-Time Library routine.

If there are errors in the specification of the return descriptor or in copying the results using that descriptor, the STR$COPY_DX routine will signal the errors. For a list of these errors, see the *VSI OpenVMS RTL String Manipulation (STR$) Manual*.

retlength

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	write only
mechanism:	by reference

Word containing the number of characters DCL returns to *retdesc*. The *retlength* argument is the address of the word containing the length of the retrieved value.

Description

The CLI$GET_VALUE routine retrieves a value associated with a specified qualifier, parameter, keyword, or keyword path from the parsed command string.

Note

Only use the CLI$GET_VALUE routine to retrieve values from parsed command strings (through DCL or CLI$DCL_PARSE). When you use a foreign command to activate an image, the DCL parsing process is interrupted. As a result, CLI$GET_VALUE returns either values from the previously parsed command string or a status of CLI$_ABSENT if it is the first command string parsed.

You can use the following label names with CLI$GET_VALUE to retrieve special strings:

$VERB	Describes the verb in the command string (the first four letters of the spelling as defined in the command table, instead of the string that was actually typed).
$LINE	Describes the entire command string as stored internally by DCL. In the internal representation of the command string, multiple spaces and tabs are removed,

	alphabetic characters are converted to uppercase, and comments are stripped. Integers are converted to decimal. If dates and times are specified in the command string, DCL fills in any defaulted fields. Also, if date-time strings (such as YESTERDAY) are used, DCL substitutes the corresponding absolute time value.

To obtain the values for a list of entities, call CLI$GET_VALUE repeatedly until all values have been returned. After each CLI$GET_VALUE call, the returned condition value indicates whether there are more values to be obtained. Call CLI$GET_VALUE until you receive a condition value of CLI $_ABSENT.

When you are using CLI$GET_VALUE to obtain a list of qualifier or keyword values, get all values in the list before starting to parse the next entity.

Condition Values Returned

SS$_NORMAL

Returned value terminated by a blank or an end-of-line. This shows that the value is the last, or only, value in the list.

CLI$_ABSENT

No value returned. The value is not present, or the last value in the list was already returned.

CLI$_COMMA

Returned value terminated by a comma. This shows there are additional values in the list.

CLI$_CONCAT

Returned value concatenated to the next value with a plus sign. This shows there are additional values in the list.

CLI$_INVREQTYP

Calling process did not have a CLI to perform this function or the CLI did not support the request.

CLI$PRESENT

Determine Presence of Entity in Command String — The CLI$PRESENT routine examines the parsed command string to determine whether the entity referred to by the *entity_desc* argument is present.

Format

```
CLI$PRESENT entity_desc
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

entity_desc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Character string containing the label (or name if no label is defined) of the entity. The *entity_desc* argument is the address of a string descriptor that points to an entity that may appear on a command line. An entity can be expressed as one of the following:

- A parameter, qualifier, or keyword name or label
- A keyword path

A keyword path is used to reference keywords that are accepted by parameters, qualifiers, or other keywords. A keyword path contains a list of entity names separated by periods. See the description of the *entity_desc* argument in the CLI$GET_VALUE routine for more information about specifying keyword paths as arguments for CLI routines.

The *entity_desc* argument can contain parameter, qualifier, or keyword names, or can contain labels that were assigned with the LABEL clause in the command definition file. If the LABEL clause was used to assign a label to a qualifier, parameter, or keyword, you must specify the label in the *entity_desc* argument. Otherwise, you must use the actual name of the qualifier, parameter, or keyword.

If the *entity_desc* argument does not exist in the command table, CLI$PRESENT signals a syntax error (by means of the signaling mechanism described in the *VSI OpenVMS Programming Concepts Manual*).

Description

The CLI$PRESENT routine examines the parsed command string to determine whether the entity referred to by the *entity_desc* argument is present.

When CLI$PRESENT tests whether a qualifier is present, the condition value indicates whether the qualifier is used globally or locally. You can use a global qualifier anywhere in the command line; you use a local qualifier only after a parameter. A global qualifier is defined in the command definition file with PLACEMENT=GLOBAL; a local qualifier is defined with PLACEMENT=LOCAL.

When you test for the presence of a global qualifier, CLI$PRESENT determines if the qualifier is present anywhere in the command string. If the qualifier is present in its positive form, CLI$PRESENT returns CLI$_PRESENT; if the qualifier is present in its negative form, CLI$PRESENT returns CLI $_NEGATED.

You can test for the presence of a local qualifier when you are parsing parameters that can be followed by qualifiers. After you call CLI$GET_VALUE to fetch the parameter value, call CLI$PRESENT to determine whether the local qualifier is present. If the local qualifier is present in its positive form, CLI$PRESENT returns CLI$_LOCPRES; if the local qualifier is present in its negative form, CLI $PRESENT returns CLI$_LOCNEG.

A positional qualifier affects the entire command line if it appears after the verb but before the first parameter. A positional qualifier affects a single parameter if it appears after a parameter. A positional qualifier is defined in the command definition file with the PLACEMENT=POSITIONAL clause.

To determine whether a positional qualifier is used globally, call CLI$PRESENT to test for the qualifier before you call CLI$GET_VALUE to fetch any parameter values. If the positional qualifier is used globally, CLI$PRESENT returns either CLI$_PRESENT or CLI$_NEGATED.

To determine whether a positional qualifier is used locally, call CLI$PRESENT immediately after a parameter value has been fetched by CLI$GET_VALUE. The most recent CLI$GET_VALUE call to fetch a parameter defines the context for a qualifier search. Therefore, CLI$PRESENT tests whether a positional qualifier was specified after the parameter that was fetched by the most recent CLI$GET_VALUE call. If the positional qualifier is used locally, CLI$PRESENT returns either CLI $_LOCPRES or CLI$_LOCNEG.

Condition Values Returned

CLI$_ABSENT

Specified entity not present, and it is not present by default.

CLI$_DEFAULTED

Specified entity not present, but it is present by default.

CLI$_INVREQTYP

Calling process did not have a CLI to perform this function, or the CLI did not support the request.

CLI$_LOCNEG

Specified qualifier present in negated form (with /NO) and used as a local qualifier.

CLI$_LOCPRES

Specified qualifier present and used as a local qualifier.

CLI$_NEGATED

Specified qualifier present in negated form (with /NO) and used as a global qualifier.

CLI$_PRESENT

Specified entity present in the command string. This status is returned for all entities except local qualifiers and positional qualifiers that are used locally.

Chapter 5. Common File Qualifier Routines

This chapter describes the common file qualifier (UTIL$CQUAL) routines. The UTIL$CQUAL routines allow you to parse the command line for qualifiers related to certain file attributes, and to match files you are processing against the selected criteria retrieved from the command line.

5.1. Introduction to the Common File Qualifier Routines

The common file qualifier routines begin with the characters UTIL$CQUAL. Your program calls these routines using the OpenVMS Calling Standard. When you call a UTIL$CQUAL routine, you must provide all the required arguments. Upon completion, the routine returns its completion status as a condition value. Section 5.3 provides detailed descriptions of the routines.

The following table lists the common file qualifier routines.

Table 5.1. UTIL$CQUAL Routines

Routine Name	Description
UTIL$CQUAL_FILE_PARSE	Parses the command line for the file qualifiers listed in Table 5.2, and obtains associated values. Returns a context value that is used when calling the matching and ending routines.
UTIL$CQUAL_FILE_MATCH	Compares the routine file input to the command line data obtained from the parse routine call.
UTIL$CQUAL_FILE_END	Deletes all virtual memory allocated during the command line parse routine call.
UTIL$CQUAL_CONFIRM_ACT	Prompts a user for a response from SYS $COMMAND.

5.2. Using the Common File Qualifier Routines

Follow these steps to use the common file qualifier routines:

1. Call UTIL$CQUAL_FILE_PARSE to parse the command line for the common file qualifiers. (See Table 5.2 for a list of the qualifiers.)

2. Call UTIL$CQUAL_FILE_MATCH for each checked file. UTIL$CQUAL_FILE_MATCH returns an indication that the file is, or is not, to be processed.

3. Call UTIL$CQUAL_FILE_END to release the virtual memory held by the common file qualifier package.

You may optionally call UTIL$CQUAL_CONFIRM_ACT to ask for user confirmation without calling the other common qualifier routines.

5.2.1. Calling UTIL$CQUAL_FILE_PARSE

When you call UTIL$CQUAL_FILE_PARSE, specify the qualifiers listed in Table 5.2 that you want to parse by setting bits in a flags longword and passing the longword address as the first parameter.

Table 5.2. UTIL$CQUAL_FILE_PARSE Command Line Qualifiers

Qualifier	Description
BEFORE=	Selects a file before the specified time.
CONFIRM	Prompts the user for confirmation.
SINCE=	Selects a file on or after the specified time.
MODIFIED	Specifies that the file's revision time (time of last modification) is used for comparison with the time specified in either the /BEFORE or /SINCE qualifier.
CREATED (default)	Specifies that the file's creation time is used for comparison with the time specified in either the /BEFORE or /SINCE qualifier.
BACKUP	Specifies that the file's most recent backup time is used for comparison with the time specified in either the /BEFORE or /SINCE qualifier.
EXPIRED	Specifies that the file's expiration date is used for comparison with the time specified in either the /BEFORE or /SINCE qualifier.
BY_OWNER=	Selects a file based on the file owner's user identification code. The default is the UIC of the current process.
EXCLUDE=	Selects a file only if it does not match the specification or list of specifications given with this qualifier.

The following segment from a sample C program shows the flags longword set to search for the common file qualifiers supported by this package:

```
input_flags = UTIL$M_CQF_CONFIRM | UTIL$M_CQF_EXCLUDE  |
              UTIL$M_CQF_BEFORE  | UTIL$M_CQF_SINCE    |
              UTIL$M_CQF_CREATED | UTIL$M_CQF_MODIFIED |
              UTIL$M_CQF_EXPIRED | UTIL$M_CQF_BACKUP   |
              UTIL$M_CQF_BYOWNER;
```

Optionally, you can provide the flags longword address for UTIL$CQUAL_FILE_PARSE to return an indication of what common file qualifiers were present on the command line. For example, if /CONFIRM is enabled and was found on the command line, the application can determine if confirmation prompts need to be built. The following is an example call in C:

```
status = UTIL$CQUAL_FILE_PARSE (&input_flags,
                                &context,
                                &output_flags);
```

The context variable contains the address of the common file qualifier value which is used in other common file qualifier routine calls.

5.2.1.1. Specifying Times

The times specified with the /SINCE= and /BEFORE= qualifiers must be in either absolute or combination time format. When DCL gathers these times from the command line, it converts truncated time values, combination time values, and keywords (such as BOOT, LOGIN, TODAY, TOMORROW,

or YESTERDAY) into absolute time format. Files are selected based on the times entered on the command line, and are compared to the time of the file's backup date, creation date (default), expiration date, or last modification date as indicated by the modifier qualifiers /BACKUP, /CREATED, /EXPIRED, and /MODIFIED respectively.

For complete information on specifying time values, see the *OpenVMS User's Manual* or the topic DCL_TIPS Date_Time in online help.

5.2.1.2. Specifying Exclude Pattern Strings

Pattern strings are used to exclude specific files from being processed. The pattern strings may contain a combination of a directory specification, file name, file type, and version number. Node names and device names are not permitted. Relative directory specifications are allowed (such as [.subdirectory] or [-]), but relative version numbers have no meaning as a pattern string component. UTIL$CQUAL_FILE_PARSE assumes relative version numbers are a wildcard, and matches all versions. An FID or DID specification is also not allowed.

To exclude more than one specification, use a comma-separated list enclosed within parentheses.

5.2.2. Calling UTIL$CQUAL_FILE_MATCH

When calling UTIL$CQUAL_FILE_MATCH, specify a file that you want checked against criteria in the common file qualifier context. The context address was returned as the first parameter in a prior call to UTIL$CQUAL_FILE_PARSE, and is the first parameter for UTIL$CQUAL_FILE_MATCH.

To specify a file, provide either a string descriptor containing the specification or an RMS FAB. The FAB must contain an NAM block that has been filled in by RMS, so that comparisons with excluded file specifications can occur. If the FAB indicates that the file is open, and any of the /BEFORE, /SINCE or /BY_OWNER qualifiers are to be evaluated, then the appropriate XAB blocks must be in the XAB chain (XABDAT and XABPRO). The XAB blocks must be filled in by RMS during the file open.

Note

The files passed in with a DID or an FID specification may cause the common qualifier package to stop processing if that portion of the file specification needs to be matched against a pattern string from the /EXCLUDE qualifier.

5.2.2.1. Specifying Prompts

You can provide one or two prompts when specifying prompts as confirmation messages. If confirmation is active, at least one prompt string must be specified. When providing two prompts, use the shorter prompt as the **prompt_string_1** parameter. Table 5.5 lists the valid confirmation prompt responses. CONDENSED and EXPANDED are used when switching between prompts.

The user responding CONDENSED (or just C) displays the **prompt_string_1** string. For a more descriptive or detailed prompt, use **prompt_string_2** in your call. For example, the OpenVMS utilities construct prompts from the short and long fields of an RMS NAML block. The prompt from the short field is passed through **prompt_string_1**, and the prompt from the long field is passed through **prompt_string_2**.

You have the option of specifying a prompt routine. The first parameter for the prompt routine will contain a string descriptor of the prompt to be displayed. The second parameter will contain the address of a buffer for the user's response. You must modify the response buffer to reflect the length of the

user's response. Table 5.5 lists the valid prompt routine responses. All other responses display an invalid response warning, and call the prompt routine again.

When two prompts are supplied to UTIL$CQUAL_FILE_MATCH, the optional parameter **current_form** can be used to determine which prompt string is displayed first. Table 5.4 lists the valid **current_form** values.

If the value stored in **current_form** is not in the values listed, then UTIL$K_CQF_SHORT is assumed. If the value is UTIL$K_CQF_UNSPECIFIED, or this parameter is absent from the call, then the form stored in the common file qualifier database is used. The value currently stored in the common file qualifier database is the final form active when UTIL$CQUAL_FILE_MATCH returned from the previous call with the current database context. If there was no previous call, UTIL$K_CQF_SHORT is stored in the database.

If the **current_form** parameter can be written to, the final active form is stored before UTIL $CQUAL_FILE_MATCH returns.

Note

If only one prompt string is provided to UTIL$CQUAL_FILE_MATCH, the final form will be the form corresponding to that prompt string even if the user requests the alternate form. For example, if only the short prompt string is provided and the user requests the long prompt, the user receives the short prompt. UTIL$K_CQF_SHORT is returned through the **current_form** parameter if that parameter is writable.

5.2.2.2. Ignoring Qualifiers

The final parameter, which is also optional, is a flags longword used to ignore certain qualifier processing when calling UTIL$CQUAL_FILE_MATCH. The modifier qualifiers for date comparisons (/CREATED, /MODIFIED, /BACKUP, and /EXPIRED) cannot be ignored. If either the /SINCE or /BEFORE modifier qualifiers are active, then the date comparison modifier qualifiers must be active to determine which dates to compare. For example, to operate on the top two versions of a file set when confirmation is active, an application can keep track of the first two instances and prompt the user. Once the application reaches that number, it sets the UTIL$M_CQF_CONFIRM bit in the **disable** parameter flags longword, and the user is not prompted for confirmation during that call. The following is an example call in C:

```
status = UTIL$CQUAL_FILE_MATCH (&context,
                                0,
                                &result_desc,
                                &short_prompt,
                                &long_prompt,
                                0,
                                &prompt_form,
                                &ignore_flags);
```

5.2.3. Calling UTIL$CQUAL_FILE_END

When calling UTIL$CQUAL_FILE_END, specify the context variable that contains the common file qualifier database context to be terminated. The database location was returned in a prior call to UTIL $CQUAL_FILE_PARSE. The UTIL$CQUAL_FILE_END call deallocates all virtual memory held by the common file qualifier value in the **context** parameter. The context variable is zeroed before this routine returns. The following is an example call in C:

```
status = UTIL$CQUAL_FILE_END (&context);
```

5.2.4. Calling UTIL$CQUAL_CONFIRM_ACT

Similar to UTIL$CQUAL_FILE_MATCH, the parameter list used when calling UTIL $CQUAL_CONFIRM_ACT is a subset of the UTIL$CQUAL_FILE_MATCH parameter list.

When specifying prompts as confirmation messages, you can provide one or two prompts. At least one prompt string must be specified. When providing two prompts, use the shorter of the two prompts as the **prompt_string_1** parameter. Table 5.5 lists valid responses to a confirmation prompt, and lists CONDENSED and EXPANDED to switch between prompts.

The user responding CONDENSED (or just C) causes the **prompt_string_1** string to be displayed. To give the user a more descriptive or detailed prompt, use **prompt_string_2** in your call. For example, the OpenVMS utilities construct prompts from the short and long fields of an RMS NAML block. The prompt from the short field is passed through **prompt_string_1**, and the prompt from the long field is passed through **prompt_string_2**.

You have the option of specifying a prompt routine. The first parameter for the prompt routine is a string descriptor of the prompt to be displayed. The second parameter contains the address of a buffer for the user's response. You must modify the response buffer to reflect the length of the user's response. Table 5.5 lists valid prompt routine responses. All other responses display an invalid response warning, and call the prompt routine again.

When two prompts are supplied to UTIL$CQUAL_CONFIRM_ACT, the optional parameter **current_form** can be used to determine which prompt string is displayed first. The valid values are listed in Table 5.4. If the value stored is other than the values listed, UTIL$K_CQF_SHORT is assumed. If the value is UTIL$K_CQF_UNSPECIFIED or this parameter is absent from the call, then UTIL $K_CQF_SHORT is used.

If the **current_form** parameter can be written to, the final active form is stored before UTIL $CQUAL_CONFIRM_ACT returns.

Note

If only one prompt string is passed into the UTIL$CQUAL_CONFIRM_ACT call, the final form will be the form corresponding to that prompt string even if the user requests the alternate form. For example, if only the short prompt string is provided and the user requests the long prompt, the user receives the short prompt again. UTIL$K_CQF_SHORT is returned through the **current_form** parameter if that parameter is writable.

The following is an example call in C:

```
        status = UTIL$CQUAL_CONFIRM_ACT (&short_prompt,
                                         &long_prompt,
                                         0,
                                         &prompt_form);
```

5.2.5. Creating a Command Language Definition File

For UTIL$CQUAL_FILE_PARSE to function properly, you need the following Command Language Definition (CLD) file template in the command tables being examined:

```
        define verb foo
            image foo
            parameter p1,prompt="File",value(list,impcat,required,type=
$infile)
            qualifier confirm
```

```
            qualifier exclude,value(required,list)
            qualifier before,value(default=today,type=$datetime)
            qualifier since,value(default=today,type=$datetime)
            qualifier created
            qualifier modified
            qualifier expired
            qualifier backup
            qualifier by_owner,value(type=$uic)
```

For example, if the line qualifier expired was omitted, a call to UTIL$CQUAL_FILE_PARSE would result in:

```
      $ foo *.c
        %CLI-F-SYNTAX, error parsing 'EXPIRED'
        -CLI-E-ENTNF, specified entity not found in command tables
        %TRACE-F-TRACEBACK, symbolic stack dump follows
          image    module    routine             line      rel PC
 abs
        ...
```

Note

A default value for the /SINCE= and /BEFORE= qualifiers is provided in the CLD file. If you do not require a value, specify a default or you may not get the desired result.

The following example shows a C program that retrieves files from the command line, and lists which ones will be processed, if processing is required.

Example 5.1. Using UTIL$CQUAL Routines to Process Files

```
$ create foo.c
#include <stdio.h>
#include <string.h>

#include <rms.h>
#include <starlet.h>
#include <descrip.h>
#include <lib$routines.h>
#include <libfildef.h>
#include <cli$routines.h>

#include <cqualdef.h>
#include <util$routines.h>

#ifdef NAML$C_BID               /* determine if HFS support is here */
#define HFS_Support 1
#else
#define HFS_Support 0
#endif

#if !HFS_Support                /* compensate for lack of HFS support */
#define naml$l_rsa nam$l_rsa
#define naml$b_rsl nam$b_rsl
#define naml$l_long_result nam$l_rsa
#define naml$l_long_result_size nam$b_rsl
#define NAML$C_MAXRSS NAM$C_MAXRSS
#define LIB$M_FIL_LONG_NAMES 0
#endif
```

```
unsigned int input_flags;
unsigned int output_flags;
unsigned int ignore_flags = 0;
unsigned int *context;
char get_value[NAM$C_MAXRSS];
char *prompt_string = "Confirmation for ";
char *prompt_end = "  [N] ? ";
char *process = "    Will process ";
char *noprocess = "    Will not process ";
char short_string[NAM$C_MAXRSS+80];
unsigned int prompt_form = 0;
unsigned int status;
struct fabdef *find_file_context;
unsigned int find_file_flags;
unsigned short ret_length;
$DESCRIPTOR(parm_1, "P1");
$DESCRIPTOR(get_val_desc, get_value);
$DESCRIPTOR(short_prompt, short_string);
$DESCRIPTOR(result_desc, "");
char long_string[NAML$C_MAXRSS+80];
char outstring[NAML$C_MAXRSS+80];
$DESCRIPTOR(long_prompt, long_string);

#if HFS_Support
struct namldef *nam_block;
#else
struct namdef *nam_block;
#endif

extern UTIL$_QUICONACT;         /* external literal */
extern UTIL$_QUIPRO;            /* external literal */

int main(void) {

input_flags =   UTIL$M_CQF_CONFIRM | UTIL$M_CQF_EXCLUDE  |
                UTIL$M_CQF_BEFORE  | UTIL$M_CQF_SINCE    |
                UTIL$M_CQF_CREATED | UTIL$M_CQF_MODIFIED |
                UTIL$M_CQF_EXPIRED | UTIL$M_CQF_BACKUP   |
                UTIL$M_CQF_BYOWNER;

if (!(status = UTIL$CQUAL_FILE_PARSE (  &input_flags,
                                        &context,
                                        &output_flags) & 1)) {
    return status;
    };

find_file_flags = LIB$M_FIL_MULTIPLE | LIB$M_FIL_LONG_NAMES;

get_val_desc.dsc$w_length = sizeof(get_value);
status = cli$get_value(&parm_1, &get_val_desc, &ret_length);

result_desc.dsc$b_class = DSC$K_CLASS_D;
result_desc.dsc$a_pointer = 0;

while (status & 1) {
    get_val_desc.dsc$w_length = ret_length;
```

```
    while ((status != (int)&UTIL$_QUIPRO) &&    /* treat as external
 literal*/

          (LIB$FIND_FILE(&get_val_desc, &result_desc,
                    &find_file_context, 0, 0, 0,
                    &find_file_flags) & 1)) {
#if HFS_Support
        nam_block = find_file_context->fab$l_naml;
#else
        nam_block = find_file_context->fab$l_nam;
#endif
        if ((output_flags && UTIL$M_CQF_CONFIRM) != 0) {
            strcpy(short_string, prompt_string);
            strncat(short_string, nam_block->naml$l_rsa,
                        (int)nam_block->naml$b_rsl);
            strcat(short_string, prompt_end);
            short_prompt.dsc$w_length = strlen(short_string);
            strcpy(long_string, prompt_string);
            strncat(long_string, nam_block->naml$l_long_result,
                        (int)nam_block->naml$l_long_result_size);
            strcat(long_string, prompt_end);
            long_prompt.dsc$w_length = strlen(long_string);
            }
        else {
            short_prompt.dsc$w_length = 0;
           long_prompt.dsc$w_length = 0;
            };
        if ((status = UTIL$CQUAL_FILE_MATCH(&context,
                                            0,
                                            &result_desc,
                                            &short_prompt,
                                            &long_prompt,
                                            0,
                                            &prompt_form,
                                            &ignore_flags)) & 1) {
            strcpy(outstring, process);
            }
        else {
            strcpy(outstring, noprocess);
            };
        if (prompt_form == UTIL$K_CQF_SHORT) {
            strncat(outstring, nam_block->naml$l_rsa,
                        (int)nam_block->naml$b_rsl);
            }
        else {
            strncat(outstring, nam_block->naml$l_long_result,
                        (int)nam_block->naml$l_long_result_size);

            };
        printf("%s\n", outstring);
        if (status == (int)&UTIL$_QUICONACT) {  /* treat as external
 literal*/
            output_flags &= ~UTIL$M_CQF_CONFIRM;
            };
        };
    if (status != (int)&UTIL$_QUIPRO) {
        get_val_desc.dsc$w_length = sizeof(get_value);
        status = cli$get_value(&parm_1, &get_val_desc, &ret_length);
```

```
        };
    };
status = UTIL$CQUAL_FILE_END (&context);
return status;
}
$  cc/list foo.c
$  link foo.c
$  set command foo.cld
$  define foo sys$disk:[]foo.exe
$  directory/noexclude

Directory MDA2000:[main]

EDTINI.EDT;1        FOO.BAR;1           FOO.C;2
FOO.C;1             FOO.CLD;2           FOO.CLD;1
FOO.EXE;3           FOO.EXE;2           FOO.EXE;1
FOO.LIS;1           FOO.OBJ;1           LAST.COM;1
LOGIN.COM;1         MAIL.MAI;1          MDA0.DAT;1
NOTE.DAT;1          QUEUE.COM;1         TPUINI.TPU;1

Total of 18 files.
$  foo/exclude=*.c *.*;*
    Will process MDA2000:[main]EDTINI.EDT;1
    Will process MDA2000:[main]FOO.BAR;1
    Will not process MDA2000:[main]FOO.C;2
    Will not process MDA2000:[main]FOO.C;1
    Will process MDA2000:[main]FOO.CLD;2
    Will process MDA2000:[main]FOO.CLD;1
    Will process MDA2000:[main]FOO.EXE;3
    Will process MDA2000:[main]FOO.EXE;2
    Will process MDA2000:[main]FOO.EXE;1
    Will process MDA2000:[main]FOO.LIS;1
    Will process MDA2000:[main]FOO.OBJ;1
    Will process MDA2000:[main]LAST.COM;1
    Will process MDA2000:[main]LOGIN.COM;1
    Will process MDA2000:[main]MAIL.MAI;1
    Will process MDA2000:[main]MDA0.DAT;1
    Will process MDA2000:[main]NOTE.DAT;1
    Will process MDA2000:[main]QUEUE.COM;1
    Will process MDA2000:[main]subdir.DIR;1
    Will process MDA2000:[main]TPUINI.TPU;1
$  foo/confirm *.*
Confirmation for MDA2000:[main]EDTINI.EDT;1  [N] ? n
    Will not process MDA2000:[main]EDTINI.EDT;1
Confirmation for MDA2000:[main]FOO.BAR;1  [N] ? n
    Will not process MDA2000:[main]FOO.BAR;1
Confirmation for MDA2000:[main]FOO.C;2  [N] ? y
    Will process MDA2000:[main]FOO.C;2
Confirmation for MDA2000:[main]FOO.CLD;2  [N] ? q
    Will not process MDA2000:[main]FOO.CLD;2
$  foo/since=yesterday/modified/exclude=(*.*;2,1*) foo.*;*,*.com;*
    Will process MDA2000:[main]FOO.BAR;1
    Will not process MDA2000:[main]FOO.C;2
    Will process MDA2000:[main]FOO.C;1
    Will not process MDA2000:[main]FOO.CLD;2
    Will process MDA2000:[main]FOO.CLD;1
    Will process MDA2000:[main]FOO.EXE;3
    Will not process MDA2000:[main]FOO.EXE;2
```

```
    Will process MDA2000:[main]FOO.EXE;1
    Will process MDA2000:[main]FOO.LIS;1
    Will process MDA2000:[main]FOO.OBJ;1
    Will not process MDA2000:[main]LAST.COM;1
    Will not process MDA2000:[main]LOGIN.COM;1
    Will process MDA2000:[main]QUEUE.COM;1
$ _
```

5.3. UTIL$CQUAL Routines

This section describes the UTIL$CQUAL routines.

UTIL$CQUAL_FILE_PARSE

UTIL$CQUAL_FILE_PARSE — The UTIL$CQUAL_FILE_PARSE routine parses the command line for the common file qualifiers.

Format

UTIL$CQUAL_FILE_PARSE flags ,context [,found_flags]

Returns

OpenVMS 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 Returned lists condition values that this routine returns.

Argument

flags

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Longword of bit flags. UTIL$CQUAL_FILE_PARSE scans the command line for the qualifiers whose associated bit is set in the flags longword. The following table lists the allowed mask and field specifier values.

Table 5.3. UTIL$CQUAL_FILE_PARSE Flags and Masks

Qualifier	Mask Value	Field Specifier
/CONFIRM	UTIL$M_CQF_CONFIRM	UTIL$V_CQF_CONFIRM
/EXCLUDE	UTIL$M_CQF_EXCLUDE	UTIL$V_CQF_EXCLUDE

Qualifier	Mask Value	Field Specifier
/BEFORE	UTIL$M_CQF_BEFORE	UTIL$V_CQF_BEFORE
/SINCE	UTIL$M_CQF_SINCE	UTIL$V_CQF_SINCE
/CREATED	UTIL$M_CQF_CREATED	UTIL$V_CQF_CREATED
/MODIFIED	UTIL$M_CQF_MODIFIED	UTIL$V_CQF_MODIFIED
/EXPIRED	UTIL$M_CQF_EXPIRED	UTIL$V_CQF_EXPIRED
/BACKUP	UTIL$M_CQF_BACKUP	UTIL$V_CQF_BACKUP
/BY_OWNER	UTIL$M_CQF_BYOWNER	UTIL$V_CQF_BYOWNER

context

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

The address of a longword that receives the common file qualifier database address. The address of the **context** variable must be passed to the UTIL$CQUAL_FILE_MATCH and UTIL$CQUAL_FILE_END routines when they are called.

found_flags

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Longword of bit flags. This optional parameter is the longword address of the value that indicates which common file qualifiers were present on the command line. The mask and field specifier values are the same values as the **flags** parameter, and are listed in Table 5.3.

Description

Using the CLI$PRESENT and CLI$GET_VALUE routines, the UTIL$CQUAL_FILE_PARSE routine searches the command line for the qualifiers specified in the flags longword. When command line parsing finishes, UTIL$CQUAL_FILE_PARSE returns a pointer to the common file qualifier value in the **context** parameter.

The **context** parameter must be used when calling either the UTIL$CQUAL_FILE_MATCH or UTIL $CQUAL_FILE_END routines. If a third parameter is specified, UTIL$CQUAL_FILE_PARSE returns a longword of flags indicating which qualifiers were found during the command line parse. The mask and field specifiers are listed in Table 5.3.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

LIB$_INVARG

Invalid argument. A bit in the flags parameter was set without an associated qualifier.

CLI$_INVQUAVAL

An unusable value was given on the command line for any of the following qualifiers: /EXCLUDE, /BEFORE, /SINCE, or /BY_OWNER (for example, /BEFORE=mintchip).

SS$_CONFQUAL

More than one of the following appeared on the command line at the same time: /CREATED, /MODIFIED, /EXPIRED, /BACKUP.

Any unsuccessful return from LIB$GET_VM.

UTIL$CQUAL_FILE_MATCH

UTIL$CQUAL_FILE_MATCH — The UTIL$CQUAL_FILE_MATCH routine matches a file with the selection criteria.

Format

UTIL$CQUAL_FILE_MATCH context [,user_fab] [,file_name] [,prompt_string_1] [,prompt_string_2] [,prompt_rtn] [,current_form] [,disable]

Returns

OpenVMS 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 Returned lists condition values that this routine returns.

Argument

context

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

The longword address that received the common file qualifier database address from a prior call to UTIL $CQUAL_FILE_PARSE.

user_fab

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**

access:	**read only**
mechanism:	**by reference**

The FAB address of the file to be evaluated. This FAB must point to a valid NAM or NAML block. If the file is open and the file header criteria are to be evaluated, the appropriate XABs (XABPRO or XABDAT) must be chained to the FAB and properly filled in by RMS. If the file is not open when this routine is called, then the XAB chain is not necessary, but may be present. This argument is optional. If it is not present, the **file_name** parameter must be present. Both arguments may not be present at the same time.

file_name

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

The file name descriptor address of the file to be processed. This parameter can be used instead of the **user_fab** argument. Both arguments may not be present at the same time.

prompt_string_1

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Longword address of a prompt string descriptor. This prompt is used when prompting to a terminal device and the current prompt form is UTIL$K_CQF_SHORT.

prompt_string_2

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by descriptor**

Longword address of a prompt string descriptor. This prompt is used when prompting to a terminal device and the current prompt form is UTIL$K_CQF_LONG.

prompt_rtn

OpenVMS usage:	**procedure**
type:	**longword (unsigned)**
access:	**function call**
mechanism:	**by value**

User-supplied longword routine address used for prompting and accepting input from the user. The user routine is responsible for end-of-file processing and must return RMS$_EOF when appropriate.

current_form

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read write**
mechanism:	**by reference**

This optional parameter supplies the initial prompt form displayed to the user. If it contains the value UTIL$K_CQF_UNSPECIFIED, then the form last requested by the user is used if that form is available. If there was no previous call to UTIL$CQUAL_FILE_MATCH, and the **current_form** is unspecified, UTIL$K_CQF_SHORT is assumed.

When exiting UTIL$CQUAL_FILE_MATCH, the **current_form** parameter contains the last user requested prompt form. If a previous call to UTIL$CQUAL_FILE_MATCH requested quit processing or quit confirmation prompting, then this parameter is not modified.

disable

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Longword of bit flags. This optional parameter specifies which common file qualifiers are ignored in the current call to UTIL$CQUAL_FILE_MATCH. Qualifiers that cannot be ignored are /CREATED, /MODIFIED, /EXPIRED, and /BACKUP).

Description

UTIL$CQUAL_FILE_MATCH compares the file named in either the **user_fab** or **file_name** parameter (only one can be specified) against criteria specified by the common file qualifier database pointed to by the **context** and the **disable** parameter flags. UTIL$CQUAL_FILE_MATCH returns a status as to whether the file does or does not match the criteria.

If a failure occurs during processing, such as those listed in the Abnormal Completion Codes, the routine quits processing files for the context under which the failure occurred. A processing failure is the same as receiving a quit processing response from a user prompt. Any additional calls to this routine with the context that incurred the processing failure will return UTIL$_QIOPRO. This applies even if the user responded ALL to a previous confirmation prompt.

For a description of the /CONFIRM prompting, see UTIL$CQUAL_CONFIRM_ACT.

Note

The UTIL$CQUAL_FILE_MATCH **current_form** parameter is different from the same parameter in UTIL$CQUAL_CONFIRM_ACT. UTIL$CQUAL_FILE_MATCH retains the user's last requested form between calls.

Condition Values Returned

Normal Completion Codes:

Abnormal Completion Codes:

SS$_NORMAL

File matches the criteria and can be processed.

UTIL$_QUICONACT

User requests that confirmation prompting cease, but that other common file qualifier criteria be applied on subsequent file specifications.

UTIL$_FILFAIMAT

File failed the evaluation, and should not be processed.

UTIL$QUIPRO

User requests that processing stops.

LIB$INVARG

Incorrect parameter list.

SS$_ACCVIO

Unable to access one or more of the parameters (such as the common file database or **user_fab**).

UTIL$_FILFID

File specification contains an FID. Due to file specification aliases, converting an FID to a file specification is inappropriate for /EXCLUDE processing.

UTIL$_FILDID

File specification contains a DID. Due to directory specification aliases, converting a DID to a directory patch is inappropriate for /EXCLUDE processing when the directory patch needs to be compared.

LIB$_INVXAB

Invalid XAB chain. A necessary XAB (XABPRO or XABDAT) is missing from the opened file's XAB chain.

Any unsuccessful code from RMS, LIB$GET_VM, or any unsuccessful return status from the user-supplied routine (other than RMS$_EOF).

UTIL$CQUAL_FILE_END

UTIL$CQUAL_FILE_END — The UTIL$CQUAL_FILE_END routine returns all allocated virtual memory from the call to UTIL$CQUAL_FILE_PARSE.

Format

`UTIL$CQUAL_FILE_END` context

Returns

OpenVMS 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 Returned lists condition values that this routine returns.

Argument

context

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read write**
mechanism:	**by reference**

The longword address that received the common file qualifier database address from a prior call to UTIL $CQUAL_FILE_PARSE.

Description

UTIL$CQUAL_FILE_END deallocates the virtual memory obtained by the common file qualifier package during the call to UTIL$CQUAL_FILE_PARSE. The virtual memory held information for calls to UTIL$CQUAL_FILE_MATCH.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

Any unsuccessful code from LIB$FREE_VM.

UTIL$CQUAL_CONFIRM_ACT

UTIL$CQUAL_CONFIRM_ACT — The UTIL$CQUAL_CONFIRM_ACT routine prompts the user for confirmation, using the optional prompt routine if present, and returns an indication of the user's response.

Format

UTIL$CQUAL_CONFIRM_ACT [prompt_string_1] [,prompt_string_2] [,prompt_rtn] [,current_form]

Returns

OpenVMS 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 Returned lists condition values that this routine returns.

Argument

prompt_string_1

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by descriptor**

Longword address of a prompt string descriptor. The prompt is used when prompting to a terminal device, and the current prompt form is UTIL$K_CQF_SHORT.

prompt_string_2

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by descriptor**

Longword address of a prompt string descriptor. The prompt is used when prompting to a terminal device, and the current prompt form is UTIL$K_CQF_LONG.

prompt_rtn

OpenVMS usage:	**procedure**
type:	**longword (unsigned)**
access:	**function call**
mechanism:	**by value**

Longword address of a user-supplied routine for prompting and accepting user input. The user routine is responsible for end-of-file processing and must return RMS$_EOF when appropriate.

current_form

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read write**
mechanism:	**by reference**

This optional parameter supplies the initial prompt form to be displayed to the user. If present, this parameter receives the form of the last prompt displayed. The following table shows the valid prompting form values:

Table 5.4. Prompting Form Values

Value	Description
UTIL$K_CQF_SHORT	Use **prompt_string_1**.
UTIL$K_CQF_LONG	Use **prompt_string_2**.
UTIL$K_CQF_UNSPECIFIED	None specified; use default.

Description

UTIL$CQUAL_CONFIRM_ACT prompts the user for confirmation. You must supply at least one prompt string to this routine. If you supply both strings, you should have an expanded and condensed form of the prompt. The condensed form should be supplied through the **prompt_string_1** parameter; the expanded form through **prompt_string_2**. The prompt string supplied by **prompt_string_1** is initially used if the prompt_string_1 is present, does not have a length of zero, and either:

- The **current_form** parameter is not specified
- The **current_form** parameter is specified and contains:
  - UTIL$K_CQF_SHORT
  - UTIL$K_CQF_UNSPECIFIED
  - A value greater than UTIL$K_CQF_MAX_FORM

The prompt string supplied by **prompt_string_2** is used initially if **prompt_string_2** is present, does not have a length of zero, and either:

- **prompt_string_1** is not present or has a length of zero
- The **current_form** parameter is specified and contains the value UTIL$K_CQF_LONG

Once the initial form is displayed, the user can switch between the two forms by responding to the prompt with either CONDENSED or EXPANDED. The user can only switch to another form if there was a prompt string provided for that form. Responding with either CONDENSED or EXPANDED causes a reprompt to occur, even if the current display form was not switched.

If a prompt routine is provided, the routine is called with the address of the prompt string descriptor in the first parameter, and the string descriptor address to receive the user's response in the second parameter. The routine returns a success status or RMS$_EOF.

If an unsuccessful status other than RMS$_EOF is received, then UTIL$CQUAL_CONFIRM_ACT exits without processing any response in the response buffer (the second parameter that was passed to the prompt routine). UTIL$CQUAL_CONFIRM_ACT returns the status received from the user prompt routine. The prompt routine is responsible for end-of-file processing, and must return RMS$_EOF when appropriate. If an optional prompt routine is provided, it should be provided for all calls to UTIL $CQUAL_CONFIRM_ACT. Not doing so can cause unpredictable end-of-file processing.

When the user is prompted, they may respond with the following:

Table 5.5. Prompt Responses

PositiveResponse	NegativeResponse	StopProcessing	StopPrompting	SwitchPrompts
YES	NO	QUIT	ALL	CONDENSED

PositiveResponse	NegativeResponse	StopProcessing	StopPrompting	SwitchPrompts
TRUE	FALSE	Ctrl/Z		EXPANDED
1	0			
	<Return>			

Note

Entering ALL assumes that subsequent files are a positive response from the user, and no further prompting occurs. The routine UTIL$CQUAL_FILE_MATCH properly handles this response. Since UTIL$CQUAL_CONFIRM_ACT does not contain context from a previous call, callers of this routine should not call UTIL$CQUAL_CONFIRM_ACT if the user has previously responded ALL unless the application needs explicit confirmation on certain items.

The user can use any combination of uppercase and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, T, TR, or TRU for TRUE), but these abbreviations must be unique.

After a valid response is received from the user, the procedure returns the **current_form** parameter. The **current_form** parameter contains the last form presented to the user if it was specified and write access is permitted.

Condition Values Returned

SS$_NORMAL

Positive answer.

LIB$_NEGANS

Negative answer.

UTIL$_QUIPRO

Quit processing.

UTIL$_QUICONACT

Continue processing, but cease prompting.

LIB$_INVARG

Invalid argument list (no prompt strings).

SS$_ACCVIO

Access violation (on user routine address).

Any unsuccessful return from RMS, SYS$ASSIGN, $QIOW, or from the user-supplied routine (other than RMS$_EOF).

Chapter 6. Convert (CONVERT) Routines

This chapter describes the CONVERT routines. These routines perform the functions of both the Convert and Convert/Reclaim utilities.

6.1. Introduction to CONVERT Routines

The Convert utility copies records from one or more files to an output file, changing the record format and file organization to that of the output file. You can invoke the functions of the Convert utility from within a program by calling the following series of three routines, in this order:

1. CONV$PASS_FILES
2. CONV$PASS_OPTIONS
3. CONV$CONVERT

Note that the application program should declare referenced constants and return status symbols as external symbols; these symbols are resolved upon linking with the utility shareable image. Also note that File Definition Language (FDL) errors may be returned to the calling program where applicable.

The Convert/Reclaim utility reclaims empty buckets in Prolog 3 indexed files so new records can be written in them. You can invoke the functions of the Convert/Reclaim utility from within a program by calling the CONV$RECLAIM routine.

While these routines can be invoked within a single thread of a threaded process, the callable Convert utility is not a reentrant, thread safe utility. Multiple concurrent invocations of the callable Convert utility interface are not supported. These routines are not reentrant and cannot be called from the asynchronous system trap (AST) level. In addition, these routines require ASTs to remain enabled in order to function properly.

6.2. Using the CONVERT Routines: Examples

Example 6.1 shows how to use the CONVERT routines in a Fortran program.

Example 6.1. Using the CONVERT Routines in a Fortran Program

```
*            This program calls the routines that perform the
*            functions of the Convert Utility.  It creates an
*            indexed output file named CUSTDATA.DAT from the
*            specifications in an FDL file named INDEXED.FDL.
*            The program then loads CUSTDATA.DAT with records
*            from the sequential file SEQ.DAT.  No exception
*            file is created.  This program also returns the
*            "BRIEF" CONVERT statistics.

*            Program declarations

        IMPLICIT        INTEGER*4 (A - Z)

*            Set up parameter list: number of options, CREATE,
*            NOSHARE, FAST_LOAD, MERGE, APPEND, SORT, WORK_FILES,
*            KEY=0, NOPAD, PAD CHARACTER, NOTRUNCATE,
```

```
*            NOEXIT, NOFIXED_CONTROL, FILL_BUCKETS, NOREAD_CHECK,
*            NOWRITE_CHECK, FDL, and NOEXCEPTION.
*
        INTEGER*4       OPTIONS(19)
        1  /18,1,0,1,0,0,1,2,0,0,0,0,0,0,0,0,1,0/

*            Set up statistics list.  Pass an array with the
*            number of statistics that you want.  There are four
*            --- number of files, number of records, exception
*            records, and good records, in that order.

        INTEGER*4       STATSBLK(5) /4,0,0,0,0/

*            Declare the file names.

        CHARACTER       IN_FILE*7 /'SEQ.DAT'/,
        1               OUT_FILE*12 /'CUSTDATA.DAT'/,
        1               FDL_FILE*11 /'INDEXED.FDL'/

*            Call the routines in their required order.

        STATUS = CONV$PASS_FILES (IN_FILE, OUT_FILE, FDL_FILE)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        STATUS = CONV$PASS_OPTIONS (OPTIONS)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        STATUS = CONV$CONVERT (STATSBLK)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

*            Display the statistics information.

        WRITE (6,1000) (STATSBLK(I),I=2,5)
1000    FORMAT (1X,'Number of files processed: ',I5/,
        1       1X,'Number of records: ',I5/,
        1       1X,'Number of exception records: ',I5/,
        1       1X,'Number of valid records: ',I5)

        END
```

Example 6.2 shows how to use the advanced features of the CONVERT routines in a C program.

Example 6.2. Using the CONVERT Routines in a C Program

```
/*
**  This module calls the routines that perform the functions
**  of the Convert utility.  It creates an indexed output file
**  named CUSTDATA.DAT from the specifications in an FDL file
**  named INDEXED.FDL, and loads CUSTDATA.DAT with records from
**  the sequential file SEQ.DAT.  No exception file is created.
**  This module also returns the CONVERT and SORT statistics
**  for each key that is loaded by utilizing the new callback
**  feature that is available through the CONV$CONVERT call.
*/

#include <stdio>
#include <descrip>
#include <lib$routines>
```

```
#include <conv$routines>
#include <convdef>
#include <starlet>
/*

**  Allocate a statistics block structure using the template provided by
**  <convdef.h>. This structure will be passed to the CONV$CONVERT routine
**  to receive both the basic and extended statistics from CONVERT. The
**  fields returned to the structure from CONVERT are listed in table 5-1.
**
**  The number of statistics to be returned is passed as the first element
**  in the array. The value CONV$K_MAX_STATISTICS will return the set of
**  basic statistics, while the value CONV$K_EXT_STATISTICS will return all
**  statistics.
*/
struct conv$statistics stats;

/*
**  Main program (CONVSTAT) starts here
*/
int CONVSTAT (void)

{
$DESCRIPTOR (input_file, "SEQ.DAT");
$DESCRIPTOR (output_file, "CUSTDATA.DAT");
$DESCRIPTOR (fdl_file, "INDEXED.FDL");

void callback();

int stat;

/*
**  Allocate an options block structure using the template provided by
**  <convdef.h>. This structure will be passed to the CONV$PASS_OPTIO
NS
**  routine to indicate what options are to be used for the file convert.
**  The fields passed to the structure are listed in table 5-2.
*/
struct conv$options param_list;

param_list.conv$l_options_count   = CONV$K_MAX_OPTIONS;
param_list.conv$l_create          = 1;
param_list.conv$l_share           = 0;
param_list.conv$l_fast            = 1;
param_list.conv$l_merge           = 0;
param_list.conv$l_append          = 0;
param_list.conv$l_sort            = 1;
param_list.conv$l_work_files      = 2;
param_list.conv$l_key             = 0;
param_list.conv$l_pad             = 0;
param_list.conv$l_pad_character   = 0;
param_list.conv$l_truncate        = 0;
param_list.conv$l_exit            = 0;
param_list.conv$l_fixed_control   = 0;
param_list.conv$l_fill_buckets    = 0;
param_list.conv$l_read_check      = 0;
param_list.conv$l_write_check     = 0;
param_list.conv$l_fdl             = 1;
```

```
param_list.conv$l_exception       = 0;
param_list.conv$l_prologue        = 0;
param_list.conv$l_ignore_prologue = 1;
param_list.conv$l_secondary       = 1;

/*
** Init the number of statistics to be returned
*/
stats.conv$l_statistics_count = CONV$K_EXT_STATISTICS;

LIB$INIT_TIMER();   /* Start a timer */

/*
** First call to pass all the file names
*/
stat = CONV$PASS_FILES ( &input_file, &output_file, &fdl_file);
if (!(stat & 1)) return stat;

/*
** Second call to pass particular options chosen as indicated in array.
*/
stat = CONV$PASS_OPTIONS ( &param_list );
if (!(stat & 1)) return stat;

/*
** Final call to perform actual convert, passing statistics block and
** callback routine address.
*/
stat = CONV$CONVERT ( &stats, 0, &callback );
if (stat & 1)
{
/*
** Successful Convert! Print out counters from statistics.
*/
 printf ("Number of files processed   : %d\n", stats.conv$l_file_count);
 printf ("Number of records           : %d\n", stats.conv$l_record_count);
 printf ("Number of exception records : %d\n", stats.conv$l_except_count);
 printf ("Number of valid records     : %d\n", stats.conv$l_valid_count);
 LIB$SHOW_TIMER();
 }
 return stat;       /* success or failure */
}

void callback ()
{
   int status, SYS$ASCTIM();
   int cvtflg = 1;
   static char date[15];
   $DESCRIPTOR(out_date, date);

printf ("Statistics for Key     : %d\n", stats.conv$l_key_number);
printf (" Records Sorted        : %d\n", stats.conv$l_rec_out);
printf (" Sort Nodes            : %d\n", stats.conv$l_nodes);
printf (" Work file allocation  : %d\n", stats.conv$l_wrk_alq);
printf (" Initial Sort Runs     : %d\n", stats.conv$l_ini_runs);
printf (" Merge Order           : %d\n", stats.conv$l_mrg_order);
printf (" Merge Passes          : %d\n", stats.conv$l_mrg_passes);
printf (" Sort Direct IO        : %d\n", stats.conv$l_sort_dio_count);
```

```
printf (" Sort Buffered IO      : %d\n", stats.conv$l_sort_bio_count);
status = SYS$ASCTIM (0, &out_date, &stats.conv$q_sort_elapsed_time,
 cvtflg);
if (!(status & 1)) LIB$STOP (status);
printf (" Sort Elapsed Time     : %s\n", date);
status = SYS$ASCTIM (0, &out_date, &stats.conv$q_sort_cpu_time, cvtflg);
if (!(status & 1)) LIB$STOP (status);
printf (" Sort Cpu Time         : %s\n", date);
printf (" Sort Page Faults      : %d\n\n", stats.conv$l_sort_pf_count);

printf (" Load Direct IO        : %d\n", stats.conv$l_load_dio_count);
printf (" Load Buffered IO      : %d\n", stats.conv$l_load_bio_count);
status = SYS$ASCTIM (0, &out_date, &stats.conv$q_load_elapsed_time,
 cvtflg);
if (!(status & 1)) LIB$STOP (status);
printf (" Load Elapsed Time     : %s\n", date);
status = SYS$ASCTIM (0, &out_date, &stats.conv$q_load_cpu_time, cvtflg);
if (!(status & 1)) LIB$STOP (status);
printf (" Load Cpu Time         : %s\n", date);
printf (" Load Page Faults      : %d\n\n", stats.conv$l_load_pf_count);

return;
}
```

Example 6.3 shows how to use the CONV$RECLAIM routine in a Fortran program.

Example 6.3. Using the CONV$RECLAIM Routine in a Fortran Program

```
*            This program calls the routine that performs the
*            function of the Convert/Reclaim utility.  It
*            reclaims empty buckets from an indexed file named
*            PROL3.DAT.  It also returns all the CONVERT/RECLAIM
*            statistics.
*            Program declarations

        IMPLICIT        INTEGER*4 (A - Z)

*            Set up a statistics block.  There are four -- data
*            buckets scanned, data buckets reclaimed, index
*            buckets reclaimed, total buckets reclaimed.

        INTEGER*4       OUTSTATS(5) /4,0,0,0,0/

*            Declare the input file.

        CHARACTER       IN_FILE*9 /'PROL3.DAT'/

*            Call the routine.

        STATUS = CONV$RECLAIM (IN_FILE, OUTSTATS)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

*            Display the statistics.

        WRITE (6,1000) (OUTSTATS(I),I=2,5)
1000    FORMAT (1X,'Number of data buckets scanned: ',I5/,
        1       1X,'Number of data buckets reclaimed: ',I5/,
        1       1X,'Number of index buckets reclaimed: ',I5/,
```

```
        1       1X,'Total buckets reclaimed: ',I5)

        END
```

Example 6.4 shows how to use the CONV$RECLAIM routine in a C program.

Example 6.4. Using the CONV$RECLAIM Routine in a C Program

```
/*
**  This module calls the routine that performs the
**  function of the CONVERT/RECLAIM utility.  It reclaims
**  empty buckets from an indexed file named PROL3.DAT.
**
**  This module also returns and prints all of the
**  CONVERT/RECLAIM statistics.
*/

#include <stdio>
#include <descrip>

CONVREC ()
{
$DESCRIPTOR (filename, "PROL3.DAT");/* Provide your file name */
struct { int statistics_count,      /* must precede actual statistics */
             scanned_buckets,
             data_buckets_reclaimed,
             index_buckets_reclaimed,
             total_buckets_reclaimed; } stats = 4 /* 4 statistic arguments
 */;
int     stat;
/*
**  Perform actual operation.
*/
stat = CONV$RECLAIM ( &filename, &stats );
if (stat & 1)
    {
    /*
    **  Successful RECLAIM. Now format and print the counts.
    */
    printf ("Data buckets scanned    : %d\n", stats.scanned_buckets);
    printf ("Data buckets reclaimed  : %d\n",
 stats.data_buckets_reclaimed);
    printf ("Index buckets reclaimed : %d\n",
 stats.index_buckets_reclaimed);
    printf ("Total buckets reclaimed : %d\n",
 stats.total_buckets_reclaimed);
    }
return stat /* succes or failure */;
}
```

6.3. CONVERT Routines

This section describes the individual CONVERT routines.

CONV$CONVERT

Initiate Conversion — The CONV$CONVERT routine uses the Convert utility to perform the actual conversion begun with CONV$PASS_FILES and CONV$PASS_OPTIONS. Optionally, the routine can return statistics about the conversion. Note that the CONV$CONVERT routine may return appropriate File Definition Language (FDL) error messages to the calling program, where applicable.

Format

```
CONV$CONVERT [status_block_address] [,flags] [,callback_routine]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

status_block_address

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

The conversion statistics. The *status_block_address* argument is the address of a variable-length array of longwords that receives statistics about the conversion.

You can request conversion statistics using zero-based, symbolic offsets (CONV$K_) into the variable-length array of longwords that contains the statistics. The array is defined as a structure (CONV $STATISTICS) of named longwords (CONV$L_) to support access by high-level programming languages.

Table 6.1 lists the array elements by number and by symbol. The first element specifies the number of statistics to return by array order. For example, if you assign the symbol CONV $L_STATISTICS_COUNT the value 2, the routine returns the statistics from the first two statistics elements:

- Number of files converted
- Number of records converted

Table 6.1. Conversion Statistics Array

Array Element	Field Name	Description
#0	CONV$L_STATISTICS_COUNT	Number of statistics specified
#1	CONV$L_FILE_COUNT	Number of files

Array Element	Field Name	Description
#2	CONV$L_RECORD_COUNT	Number of records
#3	CONV$L_EXCEPT_COUNT	Number of exception record
#4	CONV$L_VALID_COUNT	Number of valid records
#5	CONV$L_KEY_NUMBER	Most recent key processed
#6	CONV$L_REC_OUT	Number of records sorted
#7	CONV$L_NODES	Nodes in sort tree
#8	CONV$L_WRK_ALQ	Work file allocation
#9	CONV$L_INI_RUNS	Initial dispersion runs
#10	CONV$L_MRG_ORDER	Maximum merge order
#11	CONV$L_MRG_PASSES	Number of merge passes
#12	CONV$L_SORT_DIO_COUNT	Sort direct IO
#13	CONV$L_SORT_BIO_COUNT	Sort buffered IO
#14	CONV$Q_SORT_ELAPSED_TIME	Sort elapsed time
#15	CONV$Q_SORT_CPU_TIME	Sort CPU time
#16	CONV$L_SORT_PF_COUNT	Number of page faults for sort
#17	CONV$L_LOAD_DIO_COUNT	Load direct IO
#18	CONV$L_LOAD_BIO_COUNT	Load buffered IO
#19	CONV$Q_LOAD_ELAPSED_TIME	Load elapsed time
#20	CONV$Q_LOAD_CPU_TIME	Load CPU time
#21	CONV$L_LOAD_PF_COUNT	Number of page faults for load

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the CONV$PASS_FILES *fdl_filespec* argument is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing control flags (or a mask). If you omit the *flags* argument or specify it as zero, no flags are set. The flags and their meanings are described in the following table:

Flag	Function
CONV$V_FDL_STRING	Interprets the *fdl_filespec* argument supplied in the call to CONV$PASS_FILES as an FDL specification in string form. By default, this argument is interpreted as the file name of an FDL file.
CONV$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

callback_routine

OpenVMS usage:	procedure
type:	procedure value
access:	read only
mechanism:	by reference

Name of a user-supplied routine to process the statistics information. The *callback_routine* argument is the address of the procedure value of a user-supplied routine to call at the completion of each key load.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

CONV$_BADBLK

Invalid option block.

CONV$_BADLOGIC

Internal logic error detected.

CONV$_BADSORT

Error trying to sort input file.

CONV$_CLOSEIN

Error closing file specification as input.

CONV$_CLOSEOUT

Error closing file specification as output.

CONV$_CONFQUAL

Conflicting qualifiers.

CONV$_CREA_ERR

Error creating output file.

CONV$_CREATEDSTM

File specification has been created in stream format.

CONV$_DELPRI

Cannot delete primary key.

CONV$_DUP

Duplicate key encountered.

CONV$_EXTN_ERR

Unable to extend output file.

CONV$_FATALEXC

Fatal exception encountered.

CONV$_FILLIM

Exceeded open file limit.

CONV$_IDX_LIM

Exceeded maximum index level.

CONV$_ILL_KEY

Illegal key or value out of range.

CONV$_ILL_VALUE

Illegal parameter value.

CONV$_INP_FILES

Too many input files.

CONV$_INSVIRMEM

Insufficient virtual memory.

CONV$_KEY

Invalid record key.

CONV$_LOADIDX

Error loading secondary index `n`.

CONV$_NARG

Wrong number of arguments.

CONV$_NOKEY

No such key.

CONV$_NOTIDX

File is not an indexed file.

CONV$_NOTSEQ

Output file is not a sequential file.

CONV$_NOWILD

No wildcard permitted.

CONV$_OPENEXC

Error opening exception file specification.

CONV$_OPENIN

Error opening file specification as input.

CONV$_OPENOUT

Error opening file specification as output.

CONV$_ORDER

Routine called out of order.

CONV$_PAD

Packet Assembly/Disassembly (PAD) option ignored; output record format not fixed.

CONV$_PLV

Unsupported prolog version.

CONV$_PROERR

Error reading prolog.

CONV$_PROL_WRT

Prolog write error.

CONV$_READERR

Error reading file specification.

CONV$_REX

Record already exists.

CONV$_RMS

Record caused RMS severe error.

CONV$_RSK

Record shorter than primary key.

CONV$_RSZ

Record does not fit in block/bucket.

CONV$_RTL

Record longer than maximum record length.

CONV$_RTS

Record too short for fixed record format file.

CONV$_SEQ

Record not in order.

CONV$_UDF_BKS

Cannot convert UDF records into spanned file.

CONV$_UDF_BLK

Cannot fit UDF records into single block bucket.

CONV$_VALERR

Specified value is out of legal range.

CONV$_VFC

Record too short to fill fixed part of VFC record.

CONV$_WRITEERR

Error writing file specification.

CONV$PASS_FILES

Specify Conversion Files — The CONV$PASS_FILES routine specifies a file to be converted using the CONV$CONVERT routine.

Format

```
CONV$PASS_FILES input_filespec ,output_filespec [,fdl_filespec]
 [,exception_filespec] [,flags]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

input_filespec

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

The name of the file to be converted. The *input_filespec* argument is the address of a string descriptor pointing to the name of the file to be converted.

output_filespec

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

The name of the file that receives the records from the input file. The *output_filespec* argument is the address of a string descriptor pointing to the name of the file that receives the records from the input file.

fdl_filespec

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

The name of the FDL file that defines the output file. The *fdl_filespec* argument is the address of a string descriptor pointing to the name of the FDL file.

exception_filespec

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

The name of the file that receives copies of records that cannot be written to the output file. The *exception_filespec* argument is the address of a string descriptor pointing to this name.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the *fdl_filespec* argument is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing the control flags (or mask). If you omit this argument or specify it as zero, no flags are set. If you specify a flag, it remains in effect until you explicitly reset it in a subsequent call to a CONVERT routine.

The flags and their meanings are described in the following table:

Flag	Function
CONV$V_FDL_STRING	Interprets the *fdl_filespec* argument as an FDL specification in string form. By default, this

Flag	Function
	argument is interpreted as a file name of an FDL file.
CONV$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

Description

The CONV$PASS_FILES routine specifies a file to be converted using the CONV$CONVERT routine. A single call to CONV$PASS_FILES allows you to specify an input file, an output file, an FDL file, and an exception file. If you have multiple input files, you must call CONV$PASS_FILES once for each file. You need to specify only the *input_filespec* argument for the additional files, as follows:

```
status = CONV$PASS_FILES (input_filespec)
```

The additional calls must immediately follow the original call that specified the output file specification.

Wildcard characters are not allowed in the file specifications passed to the CONVERT routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

CONV$_INP_FILES

Too many input files.

CONV$_INSVIRMEM

Insufficient virtual memory.

CONV$_NARG

Wrong number of arguments.

CONV$_ORDER

Routine called out of order.

CONV$PASS_OPTIONS

Specify Processing Options — The CONV$PASS_OPTIONS routine specifies which qualifiers are to be used by the Convert utility (CONVERT).

Format

```
CONV$PASS_OPTIONS [parameter_list_address] [,flags]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

parameter_list_address

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of a variable-length array of longwords used to specify the CONVERT qualifiers. The array is symbolically defined as a structure (CONV$OPTIONS) that you can access in one of the following ways:

- As an array of named longwords using zero-based symbols (CONV$L_ …)
- As an array using zero-based offsets (CONV$K_ …)

The first longword in the array (CONV$L_OPTIONS_COUNT) specifies the number of elements in the array, and each remaining element is associated with a CONVERT qualifier, as shown in Table 6.2. You can use the first element to assign values to the first `n` CONVERT qualifiers—where `n` is the value of CONV$L_OPTIONS_COUNT—and take default values for the remaining qualifiers. For example, to assign values to only the first three qualifiers and to take the default value for the remaining qualifiers, specify CONV$L_OPTIONS_COUNT=3. This effectively changes the size of the array to include only the first three elements, as follows, which have values you specify:

- /CREATE
- /SHARE
- /FAST_LOAD

The remaining qualifiers take the default values depicted in Table 6.2.

To assign individual values to the CONVERT qualifiers, access the array and specify the desired value (`1` or `0`). See the *VSI OpenVMS Record Management Utilities Reference Manual* for detailed descriptions of the CONVERT qualifiers.

If you do not specify *`parameter_list_address`*, your program effectively sends the routine all of the default values listed in Table 6.2.

Table 6.2. CONVERT Qualifiers

Element Number	Symbolic Value	Longword Default Value	Qualifier Default Value
#0	CONV $L_OPTIONS_COUNT	None	Not applicable

Element Number	Symbolic Value	Longword Default Value	Qualifier Default Value
#1	CONV$L_CREATE	1	/CREATE
#2	CONV$L_SHARE	0	/NOSHARE
#3	CONV$L_FAST	1	/FAST_LOAD
#4	CONV$L_MERGE	0	/NOMERGE
#5	CONV$L_APPEND	0	/NOAPPEND
#6	CONV$L_SORT	1	/SORT
#7	CONV$L_WORK_FILES	2	/WORK_FILES=2
#8	CONV$L_KEY	0	/KEY=0
#9	CONV$L_PAD	0	/NOPAD
10	CONV $L_PAD_CHARACTER	0 [1]	Pad character=0
11	CONV$L_TRUNCATE	0	/NOTRUNCATE
12	CONV$L_EXIT	0	/NOEXIT
13	CONV $L_FIXED_CONTROL	0	/NOFIXED_CONTROL
14	CONV$L_FILL_BUCKETS	0	/NOFILL_BUCKETS
15	CONV$L_READ_CHECK	0	/NOREAD_CHECK
16	CONV$L_WRITE_CHECK	0	/NOWRITE_CHECK
17	CONV$L_FDL	0	/NOFDL
18	CONV$L_EXCEPTION	0	/NOEXCEPTION
19	CONV$L_PROLOGUE	None	/PROLOGUE= `n` [2]
20	CONV $L_IGNORE_PROLOGUE	0	Not applicable
21	CONV$L_SECONDARY	1	SECONDARY=1

[1]Null character. To specify non-null pad character, insert ASCII value of desired pad character.

[2]System or process default setting.

If you specify /EXIT and the utility encounters an exception record, CONVERT returns with a fatal exception status.

If you specify an FDL file specification in the CONV$PASS_FILES routine, you must place a *1* in the FDL longword. If you also specify an exceptions file specification in the CONV$PASS_FILES routine, you must place a *1* in the EXCEPTION longword. You may specify either, both, or neither of these files, but the values in the CONV$PASS_FILES call must match the values in the parameter list. If they do not, the routine returns an error.

The PROLOG longword overrides the KEY PROLOG attribute supplied by the FDL file. If you use the PROLOG longword, enter one of the following values:

- The value *0* (default) specifies the system or process prolog type.
- The value *2* specifies a Prolog 1 or 2 file in all instances, even when circumstances would allow you to create a Prolog 3 file.
- The value *3* specifies a Prolog 3 file. If a Prolog 3 file is not allowed, you want the conversion to fail.

If the size of the options block that you pass to CONV$PASS_OPTIONS includes the SECONDARY longword value, then you must specify a value for the IGNORE_PROLOGUE field.

This field is used in conjunction with the PROLOGUE offset to determine if the prologue version of the output file is to be taken from a passed FDL, the input file, the process default or system default, or from the options block itself.

A value of 0 (zero) for the IGNORE_PROLOGUE field indicates that the prologue version of the output file is to be taken from the PROLOGUE value specified in the options block.

If the PROLOGUE value in the options block contains a 0 (zero), the process default or system default prologue version will be used. This will override the prologue version specified in an FDL file or in the input file's characteristics.

A value of 1 (one) for the IGNORE_PROLOGUE field implies that the prologue version of the output file will come from the FDL file (if specified) or from the input file's characteristics.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the *fdl_filespec* argument, used in calls to the CONV $PASS_FILES routine, is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing the control flags (or a mask). If you omit this argument or specify it as zero, no flags are set. If you specify a flag, it remains in effect until you explicitly reset it in a subsequent call to a CONVERT routine.

The flags and their meanings are described in the following table:

Flag	Function
CONV$V_FDL_STRING	Interprets the *fdl_filespec* argument supplied in the call to CONV$PASS_FILES as an FDL specification in string form. By default, this argument is interpreted as the file name of an FDL file.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

Description

You can use an options array to generate programmatic CONVERT commands. For example, you can generate the following programmatic CONVERT command by configuring the options array described by the pseudocode that follows the example command line:

```
$ CONVERT/FAST_LOAD/SORT/WORK_FILES=6/EXIT

OPTIONS ARRAY [12]          {Allocate a 13-cell array}
OPTIONS[0]  =  12           {Number of options]
```

```
OPTIONS[1]  =   1          {Specifies the /CREATE option}
OPTIONS[2]  =   0          {Specifies the /NOSHARE option}
OPTIONS[3]  =   1          {Specifies the /FAST_LOAD option}
OPTIONS[4]  =   0          {Specifies the /NOMERGE option}
OPTIONS[5]  =   0          {Specifies the /NOAPPEND option}
OPTIONS[6]  =   1          {Specifies the /SORT option}
OPTIONS[7]  =   6          {Specifies the /WORK_FILES=6 option}
OPTIONS[8]  =   0          {Specifies the /KEY=0 option}
OPTIONS[9]  =   0          {Specifies the /NOPAD option}
OPTIONS[10] =   0          {Specifies the null pad character}
OPTIONS[11] =   0          {Specifies the /NOTRUNCATE option}
OPTIONS[12] =   1          {Specifies the /EXIT option}
```

Condition Values Returned

SS$_NORMAL

Normal successful completion.

CONV$_BADBLK

Invalid option block.

CONV$_CONFQUAL

Conflicting qualifiers.

CONV$_INSVIRMEM

Insufficient virtual memory.

CONV$_NARG

Wrong number of arguments.

CONV$_OPENEXC

Error opening exception file *file specification*.

CONV$_ORDER

Routine called out of order.

CONV$RECLAIM

Invoke Convert/Reclaim Utility — The CONV$RECLAIM routine invokes the functions of the Convert/Reclaim utility.

Format

```
CONV$RECLAIM input_filespec [,statistics_blk] [,flags] [key_number]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

input_filespec

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

Name of the Prolog 3 indexed file to be reclaimed. The *input_filespec* argument is the address of a string descriptor pointing to the name of the Prolog 3 indexed file.

statistics_blk

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Bucket reclamation statistics. The *statistics_blk* argument is the address of a variable-length array of longwords that receives statistics on the bucket reclamation. You can choose which statistics you want returned by specifying a number in the first element of the array. This number determines how many of the four possible statistics the routine returns.

You can request bucket reclamation statistics using symbolic names or numeric offsets into the variable-length array of longwords that contains the statistics. The array is defined as a structure of named longwords (RECL$STATISTICS) to support access by high-level programming languages.

Table 6.3 lists the array elements by number and by symbol. The first element specifies one or more statistics by array order. For example, if you assign the symbol RECL$L_STATISTICS_COUNT the value 3, the routine returns the statistics from the first three statistics elements:

- Data buckets scanned
- Data buckets reclaimed
- Index buckets reclaimed

Table 6.3. Bucket Reclamation Statistics Array

Array Element	Field Name	Description
#0	RECL$L_STATISTICS_COUNT	Number of statistics specified
#1	RECL$L_SCAN_COUNT	Data buckets scanned

Array Element	Field Name	Description
#2	RECL$L_DATA_COUNT	Data buckets reclaimed
#3	RECL$L_INDEX_COUNT	Index buckets reclaimed
#4	RECL$L_TOTAL_COUNT	Total buckets reclaimed

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the *fdl_filespec* argument, used in calls to the CONV $PASS_FILES routine, is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing control flags (or a mask). If you omit the *flags* argument or specify it as zero, no flags are set. The flag is defined as follows:

CONV$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

key_number

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The optional *key_number* argument permits the calling program to selectively reclaim buckets by key number. If the calling program omits this argument or passes a NULL value in the argument, all buckets are reclaimed, without regard to key designation. If the calling program passes a valid key number as the value for this argument, the routine reclaims only the buckets for the specified key.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

CONV$_BADLOGIC

Internal logic error detected.

CONV$_INSVIRMEM

Insufficient virtual memory.

CONV$_INVBKT

Invalid bucket at VBN n.

CONV$_NOTIDX

File is not an indexed file.

CONV$_NOWILD

No wildcard permitted.

CONV$_OPENIN

Error opening *file specification* as input.

CONV$_PLV

Unsupported prolog version.

CONV$_PROERR

Error reading prolog.

CONV$_PROL_WRT

Prolog write error.

CONV$_READERR

Error reading *file specification*.

CONV$_WRITEERR

Error writing output file.

Chapter 7. Data Compression/ Expansion (DCX) Routines

The set of routines described in this chapter comprises the Data Compression/Expansion (DCX) facility. There is no DCL-level interface to this facility, nor is there a DCX utility.

7.1. Introduction to DCX Routines

Using the DCX routines described in this chapter, you can decrease the size of text, binary data, images, and any other type of data. Compressed data uses less space, but there is a trade-off in terms of access time to the data. Compressed data must first be expanded to its original state before it is usable. Thus, infrequently accessed data makes a good candidate for data compression.

The DCX facility provides routines that analyze and compress data records and expand the compressed records to their original state. In this process, no information is lost. A data record that has been compressed and then expanded is in the same state as it was before it was compressed.

Most collections of data can be reduced in size by DCX. However, there is no guarantee that the size of an individual data record will always be smaller after compression; in fact, some may grow larger.

The DCX facility allows for the independent analysis, compression, and expansion of more than one stream of data records at the same time. This capability is provided by means of a "context variable," which is an argument in each DCX routine. Most applications have no need for this capability; for these applications, there is a single context variable.

Some of the DCX routines make calls to various Run-Time Library (RTL) routines, for example, LIB $GET_VM. If any of these RTL routines fails, a return status code indicating the cause of the failure is returned. In such a case, you must refer to the documentation of the appropriate RTL routine to determine the cause of the failure. The status codes documented in this chapter are primarily DCX status codes.

Note also that the application program should declare referenced constants and return status symbols as external symbols; these symbols are resolved upon linking with the utility shareable image.

7.1.1. Compression Routines

Compressing a file with the DCX routines involves the following steps:

1. Initialize an analysis work area—Use the DCX$ANALYZE_INIT routine to initialize a work area for analyzing the records. The first (and, typically, the only) argument passed to DCX $ANALYZE_INIT is an integer variable for storing the context value. The DCX facility assigns a value to the context variable and associates the value with the created work area. Each time you want to analyze a record in that area, specify the associated context variable. You can analyze two or more files at once by creating a different work area for each file, giving each area a different context variable, and analyzing the records of each file in the appropriate work area.

2. Analyze the records in the file—Use the DCX$ANALYZE_DATA routine to pass each record in the file to an analysis work area. During analysis, the DCX facility gathers information that DCX $MAKE_MAP uses to create the compression/expansion function for the file. To ensure that the first byte of each record is passed to the DCX facility rather than being interpreted as a carriage control, specify CARRIAGECONTROL = NONE when you open the file to be compressed.

3. Create the compression/expansion function—Use the DCX$MAKE_MAP routine to create the compression/expansion function. You pass DCX$MAKE_MAP a context variable, and DCX $MAKE_MAP uses the information stored in the associated work area to compute a compression/expansion function for the records being compressed. If DCX$MAKE_MAP returns a status value of DCX$_AGAIN, repeat Steps 2 and 3 until DCX$MAKE_MAP returns a status of DCX $_NORMAL, indicating that a compression/expansion function has been created.

   In Example 7.1, the integer function GET_MAP analyzes each record in the file to be compressed and invokes DCX$MAKE_MAP to create the compression/expansion function. The function value of GET_MAP is the return status of DCX$MAKE_MAP, and the address and length of the compression/expansion function are returned in the GET_MAP argument list. The main program, COMPRESS_FILES, invokes the GET_MAP function, examines its function value, and, if necessary, invokes the GET_MAP function again (see the ANALYZE DATA program section).

4. Clean up the analysis work area—Use the DCX$ANALYZE_DONE routine to delete a work area. Identify the work area to be deleted by passing DCX$ANALYZE_DONE routine a context variable.

5. Save the compression/expansion function—You cannot expand compressed records without the compression/expansion function. Therefore, before compressing the records, write the compression/expansion function to the file that will contain the compressed records.

   If your programming language cannot use an address directly, pass the address of the compression/expansion function to a subprogram (WRITE_MAP in Example 7.1). Pass the subprogram the length of the compression/expansion function as well.

   In the subprogram, declare the dummy argument corresponding to the function address as a one-dimensional, adjustable, byte array. Declare the dummy argument corresponding to the function length as an integer, and use it to dimension the adjustable array. Write the function length and the array containing the function to the file that is to contain the compressed records. (The length must be stored so that you can read the function from the file using unformatted I/O; see Section 7.1.2.)

6. Compress each record—Use the DCX$COMPRESS_INIT routine to initialize a compression work area. Specify a context variable for the compression area just as for the analysis area.

   Use the DCX$COMPRESS_DATA routine to compress each record. As you compress each record, use unformatted I/O to write the compressed record to the file containing the compression/expansion function. For each record, write the length of the record and the substring containing the record. See the COMPRESSDATA section in Example 7.1. (The length is stored with the substring so that you can read the compressed record from the file using unformatted I/O; see Section 7.1.2.)

7. Use DCX$COMPRESS_DONE to delete the work area created by DCX$COMPRESS_INIT. Identify the work area to be deleted by passing DCX$COMPRESS_DATA a context variable. Use LIB$FREE_VM to free the virtual memory that DCX$MAKE_MAP used for the compression/expansion function.

7.1.2. Expansion Routines

Expanding a file with the DCX routines involves the following steps:

1. Read the compression/expansion function—When reading the compression/expansion function from the compressed file, do not make any assumptions about the function's size. The best practice is to read the length of the function from the compressed file and then invoke the LIB$GET_VM routine to get the necessary amount of storage for the function. The LIB$GET_VM routine returns the address of the first byte of the storage area.

If your programming language cannot use an address directly, pass the address of the storage area to a subprogram. Pass the subprogram the length of the compression/expansion function as well.

In the subprogram, declare the dummy argument corresponding to the storage address as a one-dimensional, adjustable, byte array. Declare the dummy argument corresponding to the function length as an integer and use it to dimension the adjustable array. Read the compression/expansion function from the compressed file into the dummy array. Because the compression/expansion function is stored in the subprogram, do not return to the main program until you have expanded all of the compressed records.

2. Initialize an expansion work area—Use the DCX$EXPAND_INIT routine to initialize a work area for expanding the records. The first argument passed to DCX$EXPAND_INIT is an integer variable to contain a context value (see step 1 in Section 7.1.1). The second argument is the address of the compression/expansion function.
3. Expand the records—Use the DCX$EXPAND_DATA routine to expand each record.
4. Clean up the work area—Use the DCX$EXPAND_DONE routine to delete an expansion work area. Identify the work area to be deleted by passing DCX$EXPAND_DONE a context variable.

7.2. Using the DCX Routines: Examples

Example 7.1 shows how to use the callable DCX routines to compress a file in a VSI Fortran program.

Example 7.2 expands a compressed file. The first record of the compressed file is an integer containing the number of bytes in the compression/expansion function. The second record is the compression/expansion function. The remainder of the file contains the compressed records. Each compressed record is stored as two records: an integer containing the length of the record and a substring containing the record.

Example 7.1. Compressing a File in a VSI Fortran Program

```
PROGRAM COMPRESS_FILES
! COMPRESSION OF FILES

! status variable
INTEGER STATUS,
2       IOSTAT,
2       IO_OK,
2       STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE '($FORDEF)'
EXTERNAL DCX$_AGAIN

! context variable
INTEGER CONTEXT
! compression/expansion function
INTEGER MAP,
2       MAP_LEN

! normal file name, length, and logical unit number
CHARACTER*256 NORM_NAME
INTEGER*2 NORM_LEN
INTEGER NORM_LUN
! compressed file name, length, and logical unit number
```

```
CHARACTER*256 COMP_NAME
INTEGER*2 COMP_LEN
INTEGER COMP_LUN

! Logical end-of-file
LOGICAL EOF
! record buffers;  32764 is maximum record size
CHARACTER*32764 RECORD,
2               RECORD2
INTEGER RECORD_LEN,
2       RECORD2_LEN

! user routine
INTEGER GET_MAP,
2       WRITE_MAP

! Library procedures
INTEGER DCX$ANALYZE_INIT,
2       DCX$ANALYZE_DONE,
2       DCX$COMPRESS_INIT,
2       DCX$COMPRESS_DATA,
2       DCX$COMPRESS_DONE,
2       LIB$GET_INPUT,
2       LIB$GET_LUN,
2       LIB$FREE_VM

! get name of file to be compressed and open it
STATUS = LIB$GET_INPUT (NORM_NAME,
2                       'File to compress: ',
2                       NORM_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_LUN (NORM_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = NORM_LUN,
2     FILE = NORM_NAME(1:NORM_LEN),
2     CARRIAGECONTROL = 'NONE',
2     STATUS = 'OLD')




! ************
! ANALYZE DATA
! ************
! initialize work area
STATUS = DCX$ANALYZE_INIT (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! get compression/expansion function (map)
STATUS = GET_MAP (NORM_LUN,
2                 CONTEXT,
2                 MAP,
2                 MAP_LEN)
DO WHILE (STATUS .EQ. %LOC(DCX$_AGAIN))
  ! go back to beginning of file
  REWIND (UNIT = NORM_LUN)
  ! try map again
  STATUS = GET_MAP (NORM_LUN,
2                   CONTEXT,
```

```
2                   MAP,
2                   MAP_LEN)
  END DO
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! clean up work area
STATUS = DCX$ANALYZE_DONE (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))


! *************
! COMPRESS DATA
! *************
! go back to beginning of file to be compressed
REWIND (UNIT = NORM_LUN)
! open file to hold compressed records
STATUS = LIB$GET_LUN (COMP_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (COMP_NAME,
2                       'File for compressed records: ',
2                       COMP_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = COMP_LUN,
2     FILE = COMP_NAME(1:COMP_LEN),
2     STATUS = 'NEW',
2     FORM = 'UNFORMATTED')

! initialize work area
STATUS = DCX$COMPRESS_INIT (CONTEXT,
2                           MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! write compression/expansion function to new file
CALL WRITE_MAP (COMP_LUN,
2               %VAL(MAP),
2               MAP_LEN)

! read record from file to be compressed
EOF = .FALSE.
READ (UNIT = NORM_LUN,
2     FMT = '(Q,A)',
2     IOSTAT = IOSTAT) RECORD_LEN,
2                      RECORD(1:RECORD_LEN)
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
     CALL LIB$SIGNAL (%VAL(STATUS))
     ELSE
     EOF = .TRUE.
     STATUS = STATUS_OK
     END IF
  END IF


DO WHILE (.NOT. EOF)
  ! compress the record
  STATUS = DCX$COMPRESS_DATA (CONTEXT,
2                             RECORD(1:RECORD_LEN),
2                             RECORD2,
2                             RECORD2_LEN)
```

```
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  ! write compressed record to new file
  WRITE (UNIT = COMP_LUN) RECORD2_LEN
  WRITE (UNIT = COMP_LUN) RECORD2 (1:RECORD2_LEN)
  ! read from file to be compressed
  READ (UNIT = NORM_LUN,
2       FMT = '(Q,A)',
2       IOSTAT = IOSTAT) RECORD_LEN,
2                        RECORD (1:RECORD_LEN)
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
       CALL LIB$SIGNAL (%VAL(STATUS))
       ELSE
       EOF = .TRUE.
       STATUS = STATUS_OK
       END IF
    END IF
  END DO

! close files and clean up work area
CLOSE (NORM_LUN)
CLOSE (COMP_LUN)
STATUS = LIB$FREE_VM (MAP_LEN,
2                     MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = DCX$COMPRESS_DONE (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END

INTEGER FUNCTION GET_MAP (LUN,      ! passed
2                         CONTEXT,  ! passed
2                         MAP,      ! returned
2                         MAP_LEN)  ! returned
! Analyzes records in file opened on logical
! unit LUN and then attempts to create a
! compression/expansion function using
! DCX$MAKE_MAP.


! dummy arguments
! context variable
INTEGER CONTEXT
! logical unit number
INTEGER LUN
! compression/expansion function
INTEGER MAP,
2       MAP_LEN

! status variable
INTEGER STATUS,
2       IOSTAT,
2       IO_OK,
2       STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE '($FORDEF)'
```

```
! Logical end-of-file
LOGICAL EOF
! record buffer;  32764 is the maximum record size
CHARACTER*32764 RECORD
INTEGER RECORD_LEN

! library procedures
INTEGER DCX$ANALYZE_DATA,
2       DCX$MAKE_MAP

! analyze records
EOF = .FALSE.
READ (UNIT = LUN,
2     FMT = '(Q,A)',
2     IOSTAT = IOSTAT) RECORD_LEN,RECORD
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
     CALL LIB$SIGNAL (%VAL(STATUS))
     ELSE
     EOF = .TRUE.
     STATUS = STATUS_OK
     END IF
  END IF


DO WHILE (.NOT. EOF)
  STATUS = DCX$ANALYZE_DATA (CONTEXT,
2                            RECORD(1:RECORD_LEN))
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  READ (UNIT = LUN,
2       FMT = '(Q,A)',
2       IOSTAT = IOSTAT) RECORD_LEN,RECORD
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
       CALL LIB$SIGNAL (%VAL(STATUS))
       ELSE
       EOF = .TRUE.
       STATUS = STATUS_OK
       END IF
    END IF
  END DO

STATUS = DCX$MAKE_MAP (CONTEXT,
2                      MAP,
2                      MAP_LEN)
GET_MAP = STATUS

END

SUBROUTINE WRITE_MAP (LUN,     ! passed
2                     MAP,     ! passed
2                     MAP_LEN) ! passed
IMPLICIT INTEGER(A-Z)
! write compression/expansion function
! to file of compressed data
```

```
! dummy arguments
INTEGER LUN,        ! logical unit of file
2       MAP_LEN     ! length of function
BYTE MAP (MAP_LEN)  ! compression/expansion function

! write map length
WRITE (UNIT = LUN) MAP_LEN
! write map
WRITE (UNIT = LUN) MAP

END
```

Example 7.2 shows how to expand a compressed file in a VSI Fortran program.

Example 7.2. Expanding a Compressed File in a VSI Fortran Program

```
PROGRAM EXPAND_FILES
IMPLICIT INTEGER(A-Z)
! EXPANSION OF COMPRESSED FILES

! file names, lengths, and logical unit numbers
CHARACTER*256 OLD_FILE,
2             NEW_FILE
INTEGER*2 OLD_LEN,
2         NEW_LEN
INTEGER OLD_LUN,
2       NEW_LUN

! length of compression/expansion function
INTEGER MAP,
2       MAP_LEN

! user routine
EXTERNAL EXPAND_DATA

! library procedures
INTEGER LIB$GET_LUN,
2       LIB$GET_INPUT,
2       LIB$GET_VM,
2       LIB$FREE_VM

! open file to expand
STATUS = LIB$GET_LUN (OLD_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (OLD_FILE,
2                       'File to expand: ',
2                       OLD_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = OLD_LUN,
2     STATUS = 'OLD',
2     FILE = OLD_FILE(1:OLD_LEN),
2     FORM = 'UNFORMATTED')
! open file to hold expanded data
STATUS = LIB$GET_LUN (NEW_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (NEW_FILE,
2                       'File to hold expanded data: ',
```

```
2                       NEW_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = NEW_LUN,
2     STATUS = 'NEW',
2     CARRIAGECONTROL = 'LIST',
2     FILE = NEW_FILE(1:NEW_LEN))


! expand file
! get length of compression/expansion function
READ (UNIT = OLD_LUN) MAP_LEN
STATUS = LIB$GET_VM (MAP_LEN,
2                    MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! expand records
CALL EXPAND_DATA (%VAL(MAP),
2                 MAP_LEN,     ! length of function
2                 OLD_LUN,     ! compressed data file
2                 NEW_LUN)     ! expanded data file
! delete virtual memory used for function
STATUS = LIB$FREE_VM (MAP_LEN,
2                     MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END

SUBROUTINE EXPAND_DATA (MAP,     ! passed
2                       MAP_LEN, ! passed
2                       OLD_LUN, ! passed
2                       NEW_LUN) ! passed
! expand data program

! dummy arguments
INTEGER MAP_LEN,   ! length of expansion function
2       OLD_LUN,   ! logical unit of compressed file
2       NEW_LUN    ! logical unit of expanded file
BYTE MAP(MAP_LEN)  ! array containing the function

! status variables
INTEGER STATUS,
2       IOSTAT,
2       IO_OK,
2       STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE '($FORDEF)'

! context variable
INTEGER CONTEXT

! logical end_of_file
LOGICAL EOF
! record buffers
CHARACTER*32764 RECORD,
2               RECORD2
INTEGER RECORD_LEN,
2       RECORD2_LEN
```

```
! library procedures
INTEGER DCX$EXPAND_INIT,
2       DCX$EXPAND_DATA,
2       DCX$EXPAND_DONE

! read data compression/expansion function
READ (UNIT = OLD_LUN) MAP
! initialize work area
STATUS = DCX$EXPAND_INIT (CONTEXT,
2                         %LOC(MAP(1)))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! expand records
EOF = .FALSE.
! read length of compressed record
READ (UNIT = OLD_LUN,
2     IOSTAT = IOSTAT) RECORD_LEN
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
    CALL LIB$SIGNAL (%VAL(STATUS))
    ELSE
    EOF = .TRUE.
    STATUS = STATUS_OK
    END IF
  END IF
DO WHILE (.NOT. EOF)
  ! read compressed record
  READ (UNIT = OLD_LUN) RECORD (1:RECORD_LEN)
  ! expand record
  STATUS = DCX$EXPAND_DATA (CONTEXT,
2                           RECORD(1:RECORD_LEN),
2                           RECORD2,
2                           RECORD2_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  ! write expanded record to new file
  WRITE (UNIT = NEW_LUN,
2        FMT = '(A)') RECORD2(1:RECORD2_LEN)
  ! read length of compressed record
  READ (UNIT = OLD_LUN,
2       IOSTAT = IOSTAT) RECORD_LEN
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL(STATUS))
      ELSE
      EOF = .TRUE.
      STATUS = STATUS_OK
      END IF
    END IF
  END DO
! clean up work area
STATUS = DCX$EXPAND_DONE (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END
```

7.3. DCX Routines

This section describes the individual DCX routines.

DCX$ANALYZE_DATA

Perform Statistical Analysis on a Data Record — The DCX$ANALYZE_DATA routine performs statistical analysis on a data record. The results of the analysis are accumulated internally in the context area and are used by the DCX$MAKE_MAP routine to compute the mapping function.

Format

```
DCX$ANALYZE_DATA context ,record
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Value identifying the data stream that DCX$ANALYZE_DATA analyzes. The *context* argument is the address of a longword containing this value. DCX$ANALYZE_INIT initializes this value; you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

record

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Record to be analyzed. DCX$ANALYZE_DATA reads the *record* argument, which is the address of a descriptor for the record string. The maximum length of the record string is 65,535 characters.

Description

The DCX$ANALYZE_DATA routine performs statistical analysis on a single data record. This routine is called once for each data record to be analyzed.

During analysis, the DCX facility gathers information that DCX$MAKE_MAP uses to create the compression/expansion function for the file. After the data records have been analyzed, call the DCX $MAKE_MAP routine. Upon receiving the DCX$_AGAIN status code from DCX$MAKE_MAP, you must again analyze the same data records (in the same order) using DCX$ANALYZE_DATA and then call DCX$MAKE_MAP again. On the second iteration, DCX$MAKE_MAP returns the DCX $_NORMAL status code, and the data analysis is complete.

Condition Values Returned

DCX$_INVCTX

Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$ANALYZE_SDESC_R2.

DCX$ANALYZE_DONE

Specify Analysis Completed — The DCX$ANALYZE_DONE routine deletes the context area and sets the context variable to zero, undoing the work of the DCX$ANALYZE_INIT routine. Call DCX $ANALYZE_DONE after data records have been analyzed and the DCX$MAKE_MAP routine has created the map.

Format

```
DCX$ANALYZE_DONE context
```

Returns

OpenVMS usage:	cond_value
type:	longword
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 under Condition Values Returned.

Argument

context

OpenVMS usage:	context
type:	longword
access:	modify
mechanism:	by reference

Value identifying the data stream that DCX$ANALYZE_DONE deletes. The *context* argument is the address of a longword containing this value. DCX$ANALYZE_INIT initializes this value; you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

Condition Values Returned

DCX$_INVCTX

Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$FREE_VM.

DCX$ANALYZE_INIT

Initialize Analysis Context — The DCX$ANALYZE_INIT routine initializes the context area for a statistical analysis of the data records to be compressed.

Format

```
DCX$ANALYZE_INIT context [,item_code ,item_value]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value identifying the data stream that DCX$ANALYZE_INIT initializes. The *context* argument is the address of a longword containing this value. DCX$ANALYZE_INIT writes this context into the *context* argument; you should not modify its value. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

item_code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item code specifying information that you want DCX$ANALYZE_INIT to use in its analysis of data records and in its computation of the mapping function. DCX$ANALYZE_INIT reads this *item_code* argument, which is the address of the longword contained in the item code.

For each *item_code* argument specified in the call, you must also specify a corresponding *item_value* argument. The *item_value* argument contains the interpretation of the *item_code* argument.

The following symbolic names are the five legal values of the *item_code* argument:

DCX$C_BOUNDED
DCX$C_EST_BYTES
DCX$C_EST_RECORDS
DCX$C_LIST
DCX$C_ONE_PASS

item_value

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Value of the corresponding *item_code* argument. DCX$ANALYZE_INIT reads the *item_value* argument, which is the address of a longword containing the item value.

The *item_code* and *item_value* arguments always occur as a pair, and together they specify one piece of "advice" for the DCX routines to use in computing the map function. Note that, unless stated otherwise in the list of item codes and item values, no piece of "advice" is binding on DCX; that is, DCX is free to follow or not to follow the "advice."

The following table shows, for each *item_code* argument, the possible values for the corresponding *item_value* argument:

Item Code	Corresponding Item Value
DCX$C_BOUNDED	A Boolean variable. If bit <0> is true (equals *1*), you are stating your intention to submit for analysis all data records that will be compressed; doing so often enables DCX to compute a better compression algorithm. If bit <0> is false (equals *0*) or if the DCX$C_BOUNDED item code is not specified, DCX computes a compression algorithm without regard for whether all records to be compressed will also be submitted for analysis.

Item Code	Corresponding Item Value
DCX$C_EST_BYTES	A longword value containing your estimate of the total number of data bytes that will be submitted for compression. This estimate is useful in those cases where fewer than the total number of bytes are presented for analysis. If you do not specify the DCX$C_EST_BYTES item code, DCX submits for compression the same number of bytes that was presented for analysis. Note that you may specify DCX$C_EST_RECORDS or DCX $C_EST_BYTES, or both.
DCX$C_EST_RECORDS	A longword value containing your estimate of the total number of data records that will be submitted for compression. This estimate is useful in those cases where fewer than the total number of records are presented for analysis. If you do not specify the DCX$C_EST_RECORDS item code, DCX submits for compression the same number of bytes that was presented for analysis.
DCX$C_LIST	Address of an array of 2* *n*+1 longwords. The first longword in the array contains the value 2* *n*+1. The remaining longwords are paired; there are *n* pairs. The first member of the pair is an item code, and the second member of the pair is the address of its corresponding item value. The DCX$C_LIST item code allows you to construct an array of item-code and item-value pairs and then to pass the entire array to DCX $ANALYZE_INIT. This is useful when your language has difficulty interpreting variable-length argument lists. Note that the DCX$C_LIST item code may be specified, in a single call, alone or together with any of the other item-code and item-value pairs.
DCX$C_ONE_PASS	A Boolean variable. If bit <0> is true (equals *1*), you make a binding request that DCX make only one pass over the data to be analyzed. If bit <0> is false (equals *0*) or if the DCX$C_ONE_PASS item code is not specified, DCX may make multiple passes over the data, as required. Typically, DCX makes one pass.

Description

The DCX$ANALYZE_INIT routine initializes the context area for a statistical analysis of the data records to be compressed. The first (and typically the only) argument passed to DCX$ANALYZE_INIT is an integer variable to contain the context value. The DCX facility assigns a value to the context variable and associates the value with the created work area. Each time you want a record analyzed in that area, specify the associated context variable. You can analyze two or more files at once by creating a different work area for each file, giving each area a different context variable, and analyzing the records of each file in the appropriate work area.

Condition Values Returned

DCX$_INVITEM

Error; invalid item code. The number of arguments specified in the call was incorrect (this number should be odd), or an unknown item code was specified.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$GET_VM.

DCX$COMPRESS_DATA

Compress a Data Record — The DCX$COMPRESS_DATA routine compresses a data record. Call this routine for each data record to be compressed.

Format

```
DCX$COMPRESS_DATA context ,in_rec ,out_rec [,out_length]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Value identifying the data stream that DCX$COMPRESS_DATA compresses. The *context* argument is the address of a longword containing this value. DCX$COMPRESS_INIT initializes the value; you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

in_rec

OpenVMS usage:	char_string
type:	character string

access:	read only
mechanism:	by descriptor

Data record to be compressed. The *in_rec* argument is the address of the descriptor of the data record string.

out_rec

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Data record that has been compressed. The *out_rec* argument is the address of the descriptor of the compressed record that DCX$COMPRESS_DATA returns.

out_length

OpenVMS usage:	word_signed
type:	word integer (signed)
access:	write only
mechanism:	by reference

Length (in bytes) of the compressed data record. The *out_length* argument is the address of a word into which DCX$COMPRESS_DATA returns the length of the compressed data record.

Description

The DCX$COMPRESS_DATA routine compresses a data record. Call this routine for each data record to be compressed. As you compress each record, write the compressed record to the file containing the compression/expansion map. For each record, write the length of the record and substring string containing the record to the same file. See the COMPRESS DATA section in Example 7.1.

Condition Values Returned

DCX$_INVCTX

Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$_INVDATA

Error. You specified the item value DCX$C_BOUNDED in the DCX$ANALYZE_INIT routine and attempted to compress a data record (using DCX$COMPRESS_DATA) that was not presented for analysis (using DCX$ANALYZE_DATA). Specifying the DCX$C_BOUNDED item value means that you must analyze all data records that are to be compressed.

DCX$_INVMAP

Error; invalid map. The *map* argument was not specified correctly in the DCX$ANALYZE_INIT routine or the context area is invalid.

DCX$_NORMAL

Normal successful completion.

DCX$_TRUNC

Error. The compressed data record has been truncated because the *out_rec* descriptor did not specify enough memory to accommodate the record.

This routine also returns any condition values returned by LIB$ANALYZE_SDESC_R2 and LIB $SCOPY_R_DX.

DCX$COMPRESS_DONE

Specify Compression Complete — The DCX$COMPRESS_DONE routine deletes the context area and sets the context variable to zero.

Format

```
DCX$COMPRESS_DONE context
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value identifying the data stream that DCX$COMPRESS_DONE deletes. The *context* argument is the address of a longword containing this value. DCX$COMPRESS_INIT writes the value into the *context* argument; you should not modify its value. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

Description

The DCX$COMPRESS_DONE routine deletes the context area and sets the context variable to zero, undoing the work of the DCX$COMPRESS_INIT routine. Call DCX$COMPRESS_DONE when all data records have been compressed (using DCX$COMPRESS_DATA). After calling DCX $COMPRESS_DONE, call LIB$FREE_VM to free the virtual memory that DCX$MAKE_MAP used for the compression/expansion function.

Condition Values Returned

DCX$_INVCTX

Error. The context variable is invalid or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$FREE_VM.

DCX$COMPRESS_INIT

Initialize Compression Context — The DCX$COMPRESS_INIT routine initializes the context area for the compression of data records.

Format

```
DCX$COMPRESS_INIT context ,map
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value identifying the data stream that DCX$COMPRESS_INIT initializes. The *context* argument is the address of a longword containing this value. You should not modify the *context* value after DCX $COMPRESS_INIT initializes it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

map

OpenVMS usage:	address
type:	longword (unsigned)

access:	read only
mechanism:	by reference

The function created by DCX$MAKE_MAP. The *map* argument is the address of the compression/expansion function's virtual address.

The *map* argument must remain at this address until data compression is completed and the context is deleted by means of a call to DCX$COMPRESS_DONE.

Description

The DCX$COMPRESS_INIT routine initializes the context area for the compression of data records.

Call the DCX$COMPRESS_INIT routine after calling the DCX$ANALYZE_DONE routine.

Condition Values Returned

DCX$_INVMAP

Error; invalid map. The *map* argument was not specified correctly, or the context area is invalid.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$GET_VM and LIB$FREE_VM.

DCX$EXPAND_DATA

Expand a Compressed Data Record — The DCX$EXPAND_DATA routine expands (or restores) a compressed data record to its original state.

Format

```
DCX$EXPAND_DATA context ,in_rec ,out_rec [,out_length]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)

access:	read only
mechanism:	by reference

Value identifying the data stream that DCX$EXPAND_DATA expands. The *context* argument is the address of a longword containing this value. DCX$EXPAND_INIT initializes this value; you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

in_rec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Data record to be expanded. The *in_rec* argument is the address of the descriptor of the data record string.

out_rec

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Data record that has been expanded. The *out_rec* argument is the address of the descriptor of the expanded record returned by DCX$EXPAND_DATA.

out_length

OpenVMS usage:	word_signed
type:	word integer (signed)
access:	write only
mechanism:	by reference

Length (in bytes) of the expanded data record. The *out_length* argument is the address of a word into which DCX$EXPAND_DATA returns the length of the expanded data record.

Description

The DCX$EXPAND_DATA routine expands (or restores) a compressed data record to its original state. Call this routine for each data record to be expanded.

Condition Values Returned

DCX$_INVCTX

Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$_INVDATA

Error. A compressed data record is invalid (probably truncated) and therefore cannot be expanded.

DCX$_INVMAP

Error; invalid map. The *map* argument was not specified correctly, or the context area is invalid.

DCX$_NORMAL

Normal successful completion.

DCX$_TRUNC

Warning. The expanded data record has been truncated because the *out_rec* descriptor did not specify enough memory to accommodate the record.

This routine also returns any condition values returned by LIB$ANALYZE_SDESC_R2 and LIB $SCOPY_R_DX.

DCX$EXPAND_DONE

Specify Expansion Complete — The DCX$EXPAND_DONE routine deletes the context area and sets the context variable to zero.

Format

```
DCX$EXPAND_DONE context
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value identifying the data stream that DCX$EXPAND_DONE deletes. The *context* argument is the address of a longword containing this value. DCX$EXPAND_INIT initializes this value; you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

Description

The DCX$EXPAND_DONE routine deletes the context area and sets the context variable to zero, thus undoing the work of the DCX$EXPAND_INIT routine. Call DCX$EXPAND_DONE when all data records have been expanded (using DCX$EXPAND_DATA).

Condition Values Returned

DCX$_INVCTX

Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$FREE_VM.

DCX$EXPAND_INIT

Initialize Expansion Context — The DCX$EXPAND_INIT routine initializes the context area for the expansion of data records.

Format

```
DCX$EXPAND_INIT context ,map
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value identifying the data stream that DCX$EXPAND_INIT initializes. The *context* argument is the address of a longword containing this value. After DCX$EXPAND_INIT initializes this *context* value, you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

map

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Compression/expansion function (created by DCX$MAKE_MAP). The *map* argument is the address of the compression/expansion function's virtual address.

The *map* argument must remain at this address until data expansion is completed and *context* is deleted by means of a call to DCX$EXPAND_DONE.

Description

The DCX$EXPAND_INIT routine initializes the context area for the expansion of data records.

Call the DCX$EXPAND_INIT routine as the first step in the expansion (or restoration) of compressed data records to their original state.

Before you call DCX$EXPAND_INIT, read the length of the compressed file from the compression/expansion function (the map). Invoke LIB$GET_VM to get the necessary amount of storage for the function. LIB$GET_VM returns the address of the first byte of the storage area.

Condition Values Returned

DCX$_INVMAP

Error; invalid map. The *map* argument was not specified correctly, or the context area is invalid.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$GET_VM.

DCX$MAKE_MAP

Compute the Compression/Expansion Function — The DCX$MAKE_MAP routine uses the statistical information gathered by DCX$ANALYZE_DATA to compute the compression/expansion function.

Format

```
DCX$MAKE_MAP context ,map_addr [,map_size]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value identifying the data stream that DCX$MAKE_MAP maps. The *context* argument is the address of a longword containing this value. DCX$ANALYZE_INIT initializes this value; you should not modify it. You can define multiple *context* arguments to identify multiple data streams that are processed simultaneously.

map_addr

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Starting address of the compression/expansion function. The *map_addr* argument is the address of a longword into which DCX$MAKE_MAP stores the virtual address of the compression/expansion function.

map_size

OpenVMS usage:	longword_signed
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Length of the compression/expansion function. The *map_size* argument is the address of the longword into which DCX$MAKE_MAP writes the length of the compression/expansion function.

Description

The DCX$MAKE_MAP routine uses the statistical information gathered by DCX$ANALYZE_DATA to compute the compression/expansion function. In essence, this map is the algorithm used to shorten (or compress) the original data records as well as to expand the compressed records to their original form.

The map must be available in memory when any data compression or expansion takes place; the address of the map is passed as an argument to the DCX$COMPRESS_INIT and DCX$EXPAND_INIT routines, which initialize the data compression and expansion procedures, respectively.

The map is stored with the compressed data records, because the compressed data records are indecipherable without the map. When compressed data records have been expanded to their original

state and no further compression is desired, you should delete the map using the LIB$FREE_VM routine.

DCX requires that you submit data records for analysis and then call the DCX$MAKE_MAP routine. Upon receiving the DCX$_AGAIN status code, you must again submit data records for analysis (in the same order) and call DCX$MAKE_MAP again; on the second iteration, DCX$MAKE_MAP returns the DCX$_NORMAL status code.

Condition Values Returned

DCX$_AGAIN

Informational. The map has not been created and the *map_addr* and *map_size* arguments have not been written because further analysis is required. The data records must be analyzed (using DCX$ANALYZE_DATA) again, and DCX$MAKE_MAP must be called again before DCX $MAKE_MAP will create the map and return the DCX$_NORMAL status code.

DCX$_INVCTX

Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.

DCX$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by LIB$GET_VM and LIB$FREE_VM.

Chapter 8. DEC Text Processing Utility (DECTPU) Routines

This chapter describes callable DEC Text Processing Utility (DECTPU) routines. It describes the purpose of the DECTPU callable routines, the parameters for the routine call, and the primary status returns. The parameter in the call syntax represents the object that you pass to a DECTPU 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.

This chapter is written for system programmers who are familiar with the:

- OpenVMS Calling Standard
- OpenVMS Run-Time Library
- Precise manner in which data types are represented on an Alpha processor
- Method for calling routines written in a language other than the one you are using for the main program

8.1. Introduction to DECTPU Routines

Callable DECTPU routines make DECTPU accessible from within other languages and applications supported by OpenVMS. DECTPU can be called from a program written in any language that generates calls using the OpenVMS Calling Standard. You can also call DECTPU from OpenVMS utilities, for example, the Mail utility. Callable DECTPU lets you perform text-processing functions within your program.

Callable DECTPU consists of a set of callable routines that resides in the DECTPU shareable images. You access callable DECTPU by linking against the shareable images, which include the callable interface routine names and constants. As with the DCL-level DECTPU interface, you can use files for input to and output from callable DECTPU. You can also write your own routines for processing file input, output, and messages.

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

The DECTPU routines described in this chapter return condition values indicating the routine's completion status. When comparing a returned condition value with a test value, you should use the LIB$MATCH routine from the Run-Time Library. Do not test the condition value as if it were a simple integer.

8.1.1. Interfaces to Callable DECTPU

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

8.1.1.1. Simplified Callable Interface

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

- Initialize the editor
- 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 TPU$TPU routine to specify a command line for DECTPU, or you can call the TPU$EDIT routine to specify an input file and an output file. TPU $EDIT builds a command string that is then passed to the TPU$TPU routine. These two routines are described in detail in Section 8.2.

If your application parses information that is not related to the operation of DECTPU, make sure the application obtains and uses all non-DECTPU parse information before the application calls the simplified callable interface. You must do this because the simplified callable interface destroys all parse information obtained and stored before the simplified callable interface was called.

8.1.1.2. Full Callable Interface

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

- Initialize the editor (TPU$INTIALIZE)
- Execute DECTPU procedures (TPU$EXECUTE_INIFILE and TPU$EXECUTE_COMMAND)
- Give control to the editor (TPU$CONTROL)
- Terminate the editing session (TPU$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 TPU $INITIALIZE, you must include the address of a routine that specifies initialization options. Depending on your particular application, you might also have to write additional routines. For example, you might need to write routines for performing file operations, handling errors, and otherwise controlling the editing session. Callable DECTPU provides utility routines that can perform some of these tasks for you. These utility routines can do the following:

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

If your application calls the DECwindows version of DECTPU, the application can call TPU $INITIALIZE only once.

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

- Section 8.3 begins by briefly describing the interface. However, most of this section describes the main callable DECTPU routines (TPU$INITIALIZE, TPU$EXECUTE_INIFILE, TPU $CONTROL, TPU$EXECUTE_COMMAND, and TPU$CLEANUP).
- Section 8.3.2 discusses additional routines that DECTPU provides for use with the full callable interface.

- Section 8.3.3 defines the requirements for routines that you can write for use with the full callable interface.

The full callable interface consists of the main callable DECTPU routines and the DECTPU utility routines.

8.1.2. The DECTPU Shareable Image

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

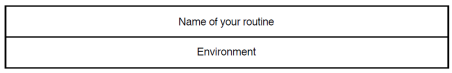
- TPU$GL_VERSION—The version of the shareable image
- TPU$GL_UPDATE—The update number of the shareable image
- TPU$_FACILITY—The DECTPU facility code

For more information about how to link to the shareable image TPUSHR.EXE, refer to the *OpenVMS Programming Environment Manual*.[1]

8.1.3. Passing Parameters to Callable DECTPU Routines

Parameters are passed to callable DECTPU routines by reference or by descriptor. 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 a procedure value and the second longword is the environment value (see the following figure). 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.



8.1.4. Error Handling

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

- You can use the DECTPU default condition handler, TPU$HANDLER.
- You can write your own condition handler to process some of the errors and call TPU$HANDLER to process the rest.

The default condition handler, TPU$HANDLER, is described in Section 8.7. Information about writing your own condition handler can be found in the *VSI OpenVMS Programming Concepts Manual*.

8.1.5. Return Values

All DECTPU 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

[1]This manual has been archived but is available on the *OpenVMS Documentation CD-ROM*.

are returned in R0. Return codes for DECTPU can be found in the *DEC Text Processing Utility Reference Manual*. DECTPU return codes and their messages are accessible from the Help/Message facility.

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

8.2. Simplified Callable Interface

The DECTPU simplified callable interface consists of two routines: TPU$TPU and TPU$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 DECTPU, make sure the application obtains and uses all non-DECTPU parse information before the application calls the simplified callable interface. You must do this because the simplified callable interface destroys all parse information obtained and stored before the simplified callable interface was called.

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

```
/*
  Sample C program that calls DECTPU.  This program uses TPU$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 DECTPU to edit text in "infile.dat" and write the result
      to "outfile.dat".  Return the condition code from DECTPU as the
      status of this program.
    */

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

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

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

```
*/

#include descrip

int return_status;

static $DESCRIPTOR (command_prefix, "TPU/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 DECTPU.  Note that the command verb
      is TPU instead of EDIT/TPU.  The string we construct in the
      buffer command_line will be
        "TPU/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 DECTPU to edit the file
    */
    return_status = TPU$TPU (&command_desc);
    exit (return_status);
    }
```

The following section contains detailed information about the routines in the full DECTPU callable interface. If you use the simplified interface, that interface calls these routines for you. If you use the full interface, your code calls these routines directly.

8.3. Full Callable Interface

The DECTPU 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
- Specify commands to be executed by the editor
- Control how conditions are handled

When you use the simplified callable interface, these operations are performed automatically. The individual DECTPU routines that perform these functions can be called from a user-written program and are known as the DECTPU full callable interface. This interface has two sets of routines: the main DECTPU callable routines and the DECTPU utility routines. These DECTPU routines, as well as your own routines that pass parameters to the DECTPU routines, are the mechanism that your application uses to control DECTPU.

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

8.3.1. Main Callable DECTPU Utility Routines

The following callable DECTPU routines are described in this chapter:

- TPU$INITIALIZE
- TPU$EXECUTE_INIFILE
- TPU$CONTROL
- TPU$EXECUTE_COMMAND
- TPU$CLEANUP

Note

Before calling any of these routines, you must establish TPU$HANDLER or provide your own condition handler. See the routine description of TPU$HANDLER in this chapter and the *VSI OpenVMS Calling Standard* for information about establishing a condition handler.

8.3.2. Other DECTPU Utility Routines

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

- TPU$CLIPARSE—Parses a command line and builds the item list for TPU$INITIALIZE
- TPU$PARSEINFO—Parses a command and builds an item list for TPU$INITIALIZE
- TPU$FILEIO—The default file I/O routine
- TPU$MESSAGE—Writes error messages and strings using the built-in procedure MESSAGE
- TPU$HANDLER—The default condition handler
- TPU$CLOSE_TERMINAL—Closes the DECTPU channel to the terminal (and its associated mailbox) for the duration of a CALL_USER routine
- TPU$SPECIFY_ASYNC_ACTION—Specifies an asynchronous event for interrupting the TPU $CONTROL routine
- TPU$TRIGGER_ASYNC_ACTION—Interrupts the TPU$CONTROL routine on a specified asynchronous event

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

8.3.3. User-Written Routines

This section defines the requirements for user-written routines. When these routines are passed to DECTPU, they must be passed as bound procedure values. (See Section 8.1.3 for a description of bound procedure values.) Depending on your application, you might have to write one or all of the following routines:

- Routine for initialization callback—This is a routine that TPU$INITIALIZE calls to obtain values for initialization parameters. The initialization parameters are returned as an item list.
- Routine for file I/O—This is a routine that handles file operations. Instead of writing your own file I/O routine, you can use the TPU$FILEIO utility routine. DECTPU 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, TPU$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.

8.4. Using the DECTPU Routines: Examples

Example 8.1, Example 8.2, Example 8.3, and Example 8.4 use callable DECTPU. These examples are included here for illustrative purposes only; VSI does not assume responsibility for supporting these examples.

Example 8.1. 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
    tpu_init,                   ! Initialize TPU
    tpu_io;                     ! File I/O routine for TPU
!
! 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 DECTPU routines
!
EXTERNAL ROUTINE
    tpu$fileio,                 ! DECTPU's internal file I/O routine
    tpu$handler,                ! DECTPU's condition handler
    tpu$initialize,             ! Initialize DECTPU
    tpu$execute_inifile,        ! Execute the initial procedures
    tpu$execute_command,        ! Execute a DECTPU statement
    tpu$control,                ! Let user interact with DECTPU
    tpu$cleanup;                ! Have DECTPU cleanup after itself
!
! Declare the DECTPU literals
!
EXTERNAL LITERAL
    tpu$k_close,                ! File I/O operation codes
    tpu$k_close_delete,
    tpu$k_open,
    tpu$k_get,
    tpu$k_put,

    tpu$k_access,               ! File access codes
    tpu$k_io,
    tpu$k_input,
    tpu$k_output,

    tpu$_calluser,             ! Item list entry codes
    tpu$_fileio,
    tpu$_outputfile,
    tpu$_sectionfile,
    tpu$_commandfile,
    tpu$_filename,
    tpu$_journalfile,
    tpu$_options,

    tpu$m_recover,              ! Mask for values in options bitmask
    tpu$m_journal,
    tpu$m_read,
    tpu$m_command,
    tpu$m_create,
    tpu$m_section,
    tpu$m_display,
    tpu$m_output,

    tpu$m_reset_terminal,       ! Masks for cleanup bitmask
    tpu$m_kill_processes,
    tpu$m_delete_exith,
    tpu$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
        tpu$handler ();
    !
    ! Initialize the editing session, passing TPU$INITIALIZE the address of
    ! the bound procedure value which defines the routine which DECTPU is
    ! to call to return the initialization item list
    !
    initialize_bpv [0] = tpu_init;
    initialize_bpv [1] = 0;
    tpu$initialize (initialize_bpv);
    !
    ! Call DECTPU to execute the contents of the command file, the debug
 file
    ! or the TPU$INIT_PROCEDURE from the section file.
    !
    tpu$execute_inifile();
    !
    ! Let DECTPU take over.
    !
    tpu$control();
    !
    ! Have DECTPU cleanup after itself
    !
    cleanup_flags = tpu$m_reset_terminal OR     ! Reset the terminal
                    tpu$m_kill_processes OR     ! Delete Subprocesses
                    tpu$m_delete_exith OR       ! Delete the exit handler
                    tpu$m_last_time;            ! Last time calling the
 editor

    tpu$cleanup (cleanup_flags);

    RETURN tpu$_success;

    END;
ROUTINE tpu_init =

    BEGIN

    !
    ! Allocate the storage block needed to pass the file I/O routine as a
```

```
   ! bound procedure variable as well as the bitmask for the
initialization
   ! options
   !
   OWN
       file_io_bpv: VECTOR [2, LONG]
                    INITIAL (TPU_IO, 0),
       options;
   !
   ! These macros define the file names passed to DECTPU
   !
   MACRO
       out_file = 'OUTPUT.TPU' % ,
       com_file = 'TPU$COMMAND' % ,
       sec_file = 'TPU$SECTION' % ,
       inp_file = 'FILE.TPU' % ;

   !
   ! Create the item list to pass to DECTPU.  Each item list entry
consists of
   ! two words which 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 DECTPU 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 bitmask
           WORD (tpu$_options),
           LONG (options),
           LONG (0),

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

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

           WORD (%CHARCOUNT (com_file)),    ! Command file
           WORD (tpu$_commandfile),
```

```
            LONG (UPLIT (%ASCII com_file)),
            LONG (0),

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

            WORD (%CHARCOUNT (inp_file)),    ! Input file
            WORD (tpu$_filename),
            LONG (UPLIT (%ASCII inp_file)),
            LONG (0),

            LONG (0));                      ! Terminating longword of 0
    !
    ! Initialize the options bitmask
    !
    options = tpu$m_display OR              ! We have a display
              tpu$m_section OR              ! We have a section file
              tpu$m_create OR               ! Create a new file if one does
 not
                                            !   exist
              tpu$m_command OR              ! We have a section file
              tpu$m_output;                 ! We supplied an output file
 spec

    !
    ! Return the item list as the value of this routine for DECTPU to
 interpret
    !
    RETURN item_list;

    END;                                    ! End of routine tpu_init
ROUTINE tpu_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 TPU's file I/O routines?
!
    IF (..p_opcode NEQ tpu$k_open) AND (.stream [stream_file_id] GTR 511)
    THEN
        RETURN tpu$fileio (.p_opcode, .stream, .data);

!
! Either we're opening the file, or we know it's one of ours
! Call the appropriate routine (not shown in this example)
!
    SELECTONE ..p_opcode OF
        SET

        [tpu$k_open]:
            status = my_io_open (.stream, .data);
```

```
        [tpu$k_close, tpu$k_close_delete]:
            status = my_io_close (.stream, .data);

        [tpu$k_get]:
            status = my_io_get_record (.stream, .data);

        [tpu$k_put]:
            status = my_io_put_record (.stream, .data);

        [OTHERWISE]:
            status = tpu$_failure;

        TES;

    RETURN .status;

    END;                                        ! End of routine TPU_IO

END                                             ! End Module
 file_io_example

ELUDOM
```

Example 8.2 shows normal DECTPU setup in VSI Fortran.

Example 8.2. Normal DECTPU Setup in VSI Fortran

```
C       A sample Fortran program that calls DECTPU to act
C       normally, using the programmable interface.
C
C       IMPLICIT NONE

        INTEGER*4       CLEAN_OPT       !options for clean up routine
        INTEGER*4       STATUS          !return status from DECTPU routines
        INTEGER*4       BPV_PARSE(2)    !set up a bound procedure value
        INTEGER*4       LOC_PARSE       !a local function call
C       declare the DECTPU functions

        INTEGER*4       TPU$CONTROL
        INTEGER*4       TPU$CLEANUP
        INTEGER*4       TPU$EXECUTE_INIFILE
        INTEGER*4       TPU$INITIALIZE
        INTEGER*4       TPU$CLIPARSE
C       declare a local copy to hold the values of DECTPU cleanup variables

        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       declare the DECTPU functions used as external

        EXTERNAL        TPU$HANDLER
        EXTERNAL        TPU$CLIPARSE

        EXTERNAL        TPU$_SUCCESS    !external error message
```

```
        EXTERNAL        LOC_PARSE       !user supplied routine to
C                                       call TPUCLIPARSE and setup
C       declare the DECTPU cleanup variables as external these are the
C       external literals that hold the value of the options

        EXTERNAL        TPU$M_RESET_TERMINAL
        EXTERNAL        TPU$M_DELETE_JOURNAL
        EXTERNAL        TPU$M_DELETE_BUFFERS,TPU$M_DELETE_WINDOWS
        EXTERNAL        TPU$M_DELETE_EXITH,TPU$M_EXECUTE_PROC
        EXTERNAL        TPU$M_PRUNE_CACHE,TPU$M_KILL_PROCESSES

100     CALL LIB$ESTABLISH ( TPU$HANDLER )       !establish the condition
 handler
C       set up the bound procedure value for the call to TPU$INITIALIZE

        BPV_PARSE( 1 ) = %LOC( LOC_PARSE )
        BPV_PARSE( 2 ) = 0
C       call the DECTPU initialization routine to do some set up work

        STATUS = TPU$INITIALIZE ( BPV_PARSE )

C       Check the status if it is not a success then signal the error

        IF ( STATUS .NE. %LOC ( TPU$_SUCCESS ) ) THEN

                CALL LIB$SIGNAL( %VAL( STATUS ) )
                GOTO 9999

        ENDIF
C       execute the TPU$_ init files and also a command file if it
C       was specified in the command line call to DECTPU

        STATUS = TPU$EXECUTE_INIFILE ( )

        IF ( STATUS .NE. %LOC ( TPU$_SUCCESS ) ) THEN !make sure everything
 is ok

                CALL LIB$SIGNAL( %VAL( STATUS ) )
                GOTO 9999

        ENDIF
C       invoke the editor as it normally would appear

        STATUS = TPU$CONTROL ( )        !call the DECTPU editor

        IF ( STATUS .NE. %LOC ( TPU$_SUCCESS ) ) THEN !make sure everything
 is ok

                CALL LIB$SIGNAL( %VAL( STATUS ) )
C               GOTO 9999
        ENDIF
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 TPU
$CLEANUP.
```

```
        DELETE_JOURNAL  = %LOC ( TPU$M_DELETE_JOURNAL )
        DELETE_EXITH    = %LOC ( TPU$M_DELETE_EXITH )
        DELETE_BUFFERS  = %LOC ( TPU$M_DELETE_BUFFERS )
        DELETE_WINDOWS  = %LOC ( TPU$M_DELETE_WINDOWS )
        EXECUTE_PROC    = %LOC ( TPU$M_EXECUTE_PROC )
        RESET_TERMINAL  = %LOC ( TPU$M_RESET_TERMINAL )
        KILL_PROCESSES  = %LOC ( TPU$M_KILL_PROCESSES )
        CLOSE_SECTION   = %LOC ( TPU$M_CLOSE_SECTION )
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.

        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       do the necessary clean up
C       TPU$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 = TPU$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 TPU
$CLIPARSE
C       Declare the DECTPU functions used

        INTEGER*4       TPU$FILEIO
        INTEGER*4       TPU$CLIPARSE
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

        EXTERNAL        TPU$FILEIO

        BPV(1) = %LOC(TPU$FILEIO)       !set up the bound procedure value
        BPV(2) = 0

        EDIT_COMM(1:12) = 'TPU TEST.TXT'
C       parse the command line and build the item list for TPU$INITIALIZE
9999     LOC_PARSE = TPU$CLIPARSE (EDIT_COMM, BPV , 0)

        RETURN
```

```
        END
```

Example 8.3 shows how to build a callback item list with VSI Fortran.

Example 8.3. Building a Callback Item List with VSI Fortran

```
       PROGRAM  TEST_TPU
C
       IMPLICIT NONE
C
C       Define the expected DECTPU return statuses
C
       EXTERNAL        TPU$_SUCCESS
       EXTERNAL        TPU$_QUITTING
       EXTERNAL        TPU$_EXITING
C
C       Declare the DECTPU routines and symbols used
C
       EXTERNAL        TPU$M_DELETE_CONTEXT
       EXTERNAL        TPU$HANDLER
       INTEGER*4       TPU$M_DELETE_CONTEXT
       INTEGER*4       TPU$INITIALIZE
       INTEGER*4       TPU$EXECUTE_INIFILE
       INTEGER*4       TPU$CONTROL
       INTEGER*4       TPU$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        TPU_STARTUP       ! the DECTPU set-up function
       INTEGER*4       TPU_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 DECTPU cleanup
       INTEGER*4       RET_STATUS
       INTEGER*4       MATCH_STATUS
C
C      Initializations
C
       RET_STATUS       = 0
       CLEANUP_FLAG     = %LOC(TPU$M_DELETE_CONTEXT)
C
C      Establish the default DECTPU condition handler
C
       CALL LIB$ESTABLISH(%REF(TPU$HANDLER))
C
C      Set up the bound procedure value for the initialization callback
```

```
C
       BPV(1)  =  %LOC (TPU_STARTUP)
       BPV(2)  =  0
C
C      Call the DECTPU procedure for initialization
C
       RET_STATUS = TPU$INITIALIZE(BPV)

       IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
       CALL LIB$SIGNAL (%VAL(RET_STATUS))
       ENDIF
C
C      Execute the DECTPU initialization file
C
       RET_STATUS = TPU$EXECUTE_INIFILE()

       IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
       CALL LIB$SIGNAL (%VAL(RET_STATUS))
       ENDIF
C
C      Pass control to DECTPU
C
       RET_STATUS = TPU$CONTROL()
C
C      Test for valid exit condition codes.  You must use LIB$MATCH_COND
C      because the severity of TPU$_QUITTING can be set by the TPU
C      application
C
       MATCH_STATUS = LIB$MATCH_COND (RET_STATUS, %LOC (TPU$_QUITTING),
       1                                          %LOC (TPU$_EXITING))
       IF (MATCH_STATUS .EQ. 0) THEN
       CALL LIB$SIGNAL (%VAL(RET_STATUS))
       ENDIF
C
C      Clean up after processing
C
       RET_STATUS = TPU$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 TPU_STARTUP

       IMPLICIT NONE

       INTEGER*4       OPTION_MASK       ! temporary variable for DECTPU
       CHARACTER*44       SECTION_NAME   ! temporary variable for DECTPU
C
C      External DECTPU routines and symbols
C
```

```
        EXTERNAL        TPU$K_OPTIONS
        EXTERNAL        TPU$M_READ
        EXTERNAL        TPU$M_SECTION
        EXTERNAL        TPU$M_DISPLAY
        EXTERNAL        TPU$K_SECTIONFILE
        EXTERNAL        TPU$K_FILEIO
        EXTERNAL        TPU$FILEIO
        INTEGER*4       TPU$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 DECTPU session
C
        OPTION_MASK = %LOC(TPU$M_SECTION) .OR. %LOC(TPU$M_READ)
        1       .OR. %LOC(TPU$M_DISPLAY)
C
C       Define the name of the initialization section file
C
        SECTION_NAME = 'TPU$SECTION'
C
C       Set up the required I/O routine.  Use the DECTPU default.
C
        BPV(1) = %LOC(TPU$FILEIO)
        BPV(2) = 0
C
C       Build the callback item list
C
C       Set up the edit session options
C
        CALLBACK(1).ITEM_CODE = %LOC(TPU$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(TPU$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(TPU$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
       TPU_STARTUP = %LOC(CALLBACK)

       RETURN
       END
```

Example 8.4 shows how to specify a user-written file I/O routine in VAX C.

Example 8.4. Specifying a User-Written File I/O Routine in VAX C

```
/*
Segment of a simple VAX C program to invoke DECTPU.  This program provides
 its
own FILEIO routine instead of using the one provided by DECTPU. This
 program
will run correctly if you write the routines it calls.
*/

/*
** To compile this example use the command:
$ CC <file-name>

** To link this example after a successful compilation:

$ LINK <file-name>,sys$input/
SYS$LIBRARY:VAXCRTL/SHARE
<PRESS-Ctrl/Z>

The TPUSHR shareable image is found by the linker in IMAGELIB.OLB.

*/
#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 TPU */
    tpu$m_delete_journal, tpu$m_delete_exith,
    tpu$m_delete_buffers, tpu$m_delete_windows, tpu$m_delete_cache,
    tpu$m_prune_cache, tpu$m_execute_file, tpu$m_execute_proc,
    tpu$m_delete_context, tpu$m_reset_terminal, tpu$m_kill_processes,
    tpu$m_close_section, tpu$m_delete_others, tpu$m_last_time;
globalvalue                     /* Item codes for item list entries */
    tpu$k_fileio, tpu$k_options, tpu$k_sectionfile,
    tpu$k_commandfile ;
globalvalue                     /* Option codes for option item */
    tpu$m_display, tpu$m_section, tpu$m_command, tpu$m_create ;

globalvalue                     /* Possible item codes in item list */
    tpu$k_access, tpu$k_filename, tpu$k_defaultfile,
    tpu$k_relatedfile, tpu$k_record_attr, tpu$k_maximize_ver,
    tpu$k_flush, tpu$k_filesize;

globalvalue                     /* Possible access types for tpu$k_access
 */
    tpu$k_io, tpu$k_input, tpu$k_output;

globalvalue                     /* OpenVMS RMS File Not Found message code
 */
    rms$_fnf;
globalvalue                     /* FILEIO routine functions */
    tpu$k_open, tpu$k_close, tpu$k_close_delete,
    tpu$k_get, tpu$k_put;
int lib$establish ();           /* RTL routine to establish an event
 handler */
int tpu$cleanup ();             /* TPU routine to free resources used */
int tpu$control ();             /* TPU routine to invoke the editor */
int tpu$execute_inifile ();     /* TPU routine to execute initialization
 code */
int tpu$handler ();             /* TPU signal handling routine */
int tpu$initialize ();          /* TPU routine to initialize the editor */
```

```
/*
   This function opens a file for either read or write access, based upon
   the itemlist 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 == tpu$k_access)
            {
            if (item->buffer_add == tpu$k_io) access = "r+";
            else if (item->buffer_add == tpu$k_input) access = "r";
            else if (item->buffer_add == tpu$k_output) access = "w";
            }
        else if (item->item_code == tpu$k_filename)
            {
            strncpy (filename, item->buffer_add, item->buffer_length);
            filename [item->buffer_length] = 0;
            lib$scopy_r_dx (&item->buffer_length, item->buffer_add,
                                                    &stream->length);
            }
        else if (item->item_code == tpu$k_defaultfile)
            {                       /* Add code to handle default file  */
            }                       /* spec here                        */
        else if (item->item_code == tpu$k_relatedfile)
            {                       /* Add code to handle related       */
            }                       /* file spec here                   */
        else if (item->item_code == tpu$k_record_attr)
            {                       /* Add code to handle record        */
            }                       /* attributes for creating files    */
        else if (item->item_code == tpu$k_maximize_ver)
            {                       /* Add code to maximize version     */
            }                       /* number with existing file here   */
        else if (item->item_code == tpu$k_flush)
            {                       /* Add code to cause each record    */
            }                       /* to be flushed to disk as written */
        else if (item->item_code == tpu$k_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 TPU
```

```
 */
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 == tpu$k_open)                /* Initial access to file */
        status = open_file (data,stream);
    else if (*code == tpu$k_close)          /* End access to file */
        status = close_file (data,stream);
    else if (*code == tpu$k_close_delete)   /* Treat same as close */
        status = close_file (data,stream);
    else if (*code == tpu$k_get)            /* Read a record from a file
 */
        status = read_line (data,stream);
    else if (*code == tpu$k_put)            /* Write a record to a file */
        status = write_line (data,stream);
    else
        {                                   /* Who knows what we have? */
        status = tpu$_success;
        printf ("Bad FILEIO I/O function requested");
        }
    return status;
}
/*
   This procedure formats the initialization item list and returns it as
   its return value.
 */
callrout()
{
    static struct bpv_arg add_block =
        { fileio, 0 } ;         /* BPV for fileio routine */
    int options ;
    char *section_name = "TPU$SECTION";
    static struct item_list_entry arg[] =
        {/* length code              buffer add return add */
              { 4,tpu$k_fileio,     0,         0 },
              { 4,tpu$k_options,    0,         0 },
              { 0,tpu$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 = tpu$m_display | tpu$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 TPU, then passes control to it.
 */
main()
{
    int return_status ;
    int cleanup_options;
    struct bpv_arg add_block;

/* Establish as condition handler the normal DECTPU handler */

    lib$establish(tpu$handler);

/* Setup a BPV to point to the callback routine */

    add_block.routine_add = callrout ;
    add_block.env = 0;

/* Do the initialize of DECTPU */

    return_status = tpu$initialize(&add_block);
    if (!return_status)
        exit(return_status);

/* Have TPU execute the procedure TPU$INIT_PROCEDURE from the section file
 */
/* and then compile and execute the code from the command file */

    return_status = tpu$execute_inifile();
    if (!return_status)
        exit (return_status);

/* Turn control over to DECTPU */

    return_status = tpu$control ();
    if (!return_status)
        exit(return_status);

/* Now clean up. */

    cleanup_options = tpu$m_last_time | tpu$m_delete_context;
    return_status = tpu$cleanup (&cleanup_options);
    exit (return_status);

    printf("Experiment complete");
}
```

8.5. Creating and Calling a USER Routine

This section describes the steps involved in creating an executable image for the USER routine and how to call the routine from a C program in the DECTPU environment. The following list describes the steps in creating the executable image:

1. Write a program in the appropriate high-level language; in the supporting example, the language is C. The program must contain a global routine named TPU$CALLUSER.
2. Compile the program.
3. Link the program with an options file to create a shareable image.
4. Define the logical name TPU$CALLUSER to point to the file containing the USER routine.
5. Invoke DECTPU.
6. From within a DECTPU session, call the high-level program to perform its function by specifying the built-in procedure CALL_USER with the appropriate parameters. The built-in procedure passes the specified parameters to the appropriate routine.

8.5.1. The CALL_USER Code

This is an example of a USER routine written in the VAX C programming language. The comments in the code explain the various routine functions.

```
/* call_user.c */
/*
A sample of a TPU CALL_USER routine written in VAX C.
The routine is compiled and linked as a shareable image and then the
DCL logical TPU$CALLUSER is defined to point at the image.

From within TPU, when the built-in CALL_USER is called, this image
will be activated and the tpu$call_user routine will be called.

This example is for VAX C but can be updated to work with DEC C with little
effort.

*/
#include <descrip.h>

extern int lib$sget1_dd(),
 vaxc$crtl_init();

globalvalue
    tpu$_success;

/*
   Because we know we are being called from a non-C based routine, call
   the CRTL initialization routine once
*/

static int
    rtl_inited = 0;


extern int tpu$calluser (
```

```
 int *int_param,
 struct dsc$descriptor *str_param,
 struct dsc$descriptor *result_param )
/*
  A sample TPU CALL_USER routine that checks access to the file specified
  in the str_param descriptor.

  Return (in result_param):
 ACCESS  - specified access is allowed
        NOACCESS - specified access is not allowed
 ERROR - Either invalid param or the file does not exist
 PARAM_ERROR - Invalid param passed
 MEMORY_ERROR - An error occured allocating memory

  An example from TPU code would be:

  file_access := CALL_USER (0, "SYS$LOGIN:LOGIN.COM");
  !
  ! Only look at the return value of ACCESS,
  !
  IF file_access = "ACCESS"
  THEN
        file_exists := 1;
  ELSE
        file_exists := 0;
  ENDIF;

  See the description of the CALL_USER built-in for more information on how
 to
  use the built-in.

*/
{
    static char
 *error_str = "ERROR",
 *param_error_str = "PARAM_ERROR",
 *memory_error_str = "MEMORY_ERROR",
 *access_str = "ACCESS",
 *noaccess_str = "NOACCESS";
    char
 *result_str_ptr;
    int
 result_str_length;
    /*
 If this is the first time in, call the VAXCRTL routine to init things
    */
    if (rtl_inited == 0) {
 vaxc$crtl_init();
 rtl_inited = 1;
    }
    /*
 The integer must be between 0 and 7 for the
 call to the C RTL routine ACCESS
    */
    if ((*int_param < 0) || (*int_param > 7)) {
 result_str_length = strlen (param_error_str);
 result_str_ptr = param_error_str;
    }
```

```
   else {
/*
    If we were passed a null string,
    set the param_error return value
*/
if (str_param->dsc$w_length == 0) {
    result_str_length = strlen (param_error_str);
    result_str_ptr = param_error_str;
}
else {
    /*
        Because there is NO way of knowing if the descriptor we have
 been passed ends with a \0, we need to create a valid string
 pass to the rtl routine "access"
    */
    char
 *str_ptr;
    /*
 Allocate memory enough for the string plus the null character
    */
    str_ptr = (char *) malloc (str_param->dsc$w_length + 1);
    /*
 Make sure the memory allocation worked...
    */
    if (str_ptr == 0) {
 result_str_length = strlen (memory_error_str);
 result_str_ptr = memory_error_str;
    }
    else {
 /*
     Move the bytes from the descriptor into the memory
     pointed to by str_ptr, and end it with a \0
     Then call the access routine, free the memory
 */
 sprintf (str_ptr, "%.*s\0", str_param->dsc$w_length,
  str_param->dsc$a_pointer);
 if (access (str_ptr, *int_param) == 0) {
     result_str_length = strlen (access_str);
     result_str_ptr = access_str;
 }
 else {
     result_str_length = strlen (noaccess_str);
     result_str_ptr = noaccess_str;
        }
 free (str_ptr);
    }
}
   }
   /* Setup the return descriptor */
   lib$sget1_dd (&result_str_length, result_param);
   /*
Copy the result bytes into the descriptor's dynamic
memory
   */
   memcpy (result_param->dsc$a_pointer, result_str_ptr,
 result_str_length);

   return tpu$_success;
```

```
}
```

Use the following command to compile the routine with the VAX C compiler:

```
$ CC/LIST call_user.c
```

8.5.2. Linking the CALL_USER Image

To link the CALL_USER image as a shareable image requires a linker option file similar to the one that follows:

```
! CALL_USER.OPT
call_user.obj
UNIVERSAL=TPU$CALLUSER
SYS$LIBRARY:VAXCRTL/SHARE
```

After you create the linker option file, use the following command to link the shareable image:

```
$ LINK CALL_USER/OPT/SHARE/MAP/FULL
```

This command produces a shareable image named CALL_USER.EXE.

The description of the DECTPU built-in CALL_USER states that you must define the logical name TPU $CALLUSER to point to the image that contains the USER procedure. Use the following command to define the logical name:

```
$ DEFINE TPU$CALLUSER SYS$DISK:[]CALL_USER.EXE
```

If you move the image to another device and directory, you must appropriately revise the pointer.

8.6. Accessing the USER Routine from DECTPU

To access the USER routine from DECTPU, your code must call the CALL_USER built-in procedure. The CALL_USER built-in procedure activates the shareable image pointed to by the logical name TPU $CALLUSER and calls the USER routine within that image. The following is an example of DECTPU code that can be used with the USER example routine in Section 8.5.1.

```
! Module: CALL_USER.TPU - the access routine
!
! Constants used with the call to this procedure (or directly to the
 call_user
! routine).
!
CONSTANT
   ACCESS_FILE_EXISTS := 0,
   ACCESS_FILE_EXECUTE := 1,
   ACCESS_FILE_WRITE := 2,
   ACCESS_FILE_DELETE := 2,
   ACCESS_FILE_READ := 4,
   ACCESS_FILE_EXE_DEL := ACCESS_FILE_EXECUTE + ACCESS_FILE_DELETE,
   ACCESS_FILE_EXE_WRITE := ACCESS_FILE_EXE_DEL,
   ACCESS_FILE_DEL_READ := ACCESS_FILE_DELETE + ACCESS_FILE_READ,
   ACCESS_FILE_DEL_WRITE := ACCESS_FILE_DEL_READ,
   ACCESS_FILE_EXE_READ := ACCESS_FILE_EXECUTE + ACCESS_FILE_READ;
```

```
PROCEDURE access (val, the_file)
!
! Call the CRTL function ACCESS via the TPU CALL_USER built-in
!
! 0 = exists
! 1 = execute
! 2 = write (& delete)
! 4 = read
! (add them for combinations)
! Return Values:
!  1 = requested access is allowed
!  0 = requested access is NOT allowed
! -1 = an error occured with the built-in
! Side Effects:
!   A message may end up in the message buffer if there is an error
!
LOCAL
    ret_val;
! Handle the call_user errors
ON_ERROR
    [TPU$_BADUSERDESC] :
 MESSAGE (ERROR_TEXT);
 RETURN -1;
    [TPU$_NOCALLUSER] :
 MESSAGE ("Could not find access call_user routine - check logicals");
 RETURN -1;
    [TPU$_CALLUSERFAIL] :
 MESSAGE ("Something is wrong in the access call_user routine");
 MESSAGE (ERROR_TEXT);
 RETURN -1
    [OTHERWISE] :
 MESSAGE (ERROR_TEXT);
 RETURN -1;
ENDON_ERROR;

ret_val := CALL_USER (val, the_file);
CASE ret_val
    ["ACCESS"]     :
 RETURN 1;
    ["NOACCESS"]   :
 RETURN 0;
    [OUTRANGE]     :
 MESSAGE ("Error with call to access routine: " + ret_val);
ENDCASE;
RETURN -1;
ENDPROCEDURE;
```

You can extend the EVE editor using the DECTPU code described at the beginning of this section. Copy the code to a file named CALL_USER.TPU in the current working directory and then execute the following commands:

```
GET FILE CALL_USER.TPU
EXTEND ALL
```

To use the DECTPU routine ACCESS from EVE, write a DECTPU procedure EVE_EXISTS, coded as follows:

```
PROCEDURE eve_exists (the_file)
IF access (ACCESS_FILE_EXISTS, the_file) = 1
THEN
    MESSAGE ("File " + the_file + " exists");
ELSE
    MESSAGE ("No such file " + the_file );
ENDIF;
ENDPROCEDURE;
```

This enables calls from the command line such as:

```
Command: exists sys$login:login.com
```

This command directs that the message window indicate whether the file SYS$LOGIN:LOGIN.COM exists.

8.7. DECTPU Routines

This section describes the individual DECTPU routines.

TPU$CLEANUP

Free System Resources Used During DECTPU Session — The TPU$CLEANUP routine cleans up internal data structures, frees memory, and restores terminals to their initial state. This is the final routine called in each interaction with DECTPU.

Format

```
TPU$CLEANUP flags
```

Returns

OpenVMS 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 under Condition Value Returned.

Argument

flags

OpenVMS 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. Following are the various cleanup options:

Flag[1]	**Function**
TPU$M_DELETE_JOURNAL	Closes and deletes the journal file if it is open.
TPU$M_DELETE_EXITH	Deletes the DECTPU exit handler.
TPU$M_DELETE_BUFFERS	Deletes all text buffers. If this is not the last time you are calling DECTPU, then all variables referring to these data structures are reset, as if by the built-in procedure DELETE. If a buffer is deleted, then all ranges and markers within that buffer, and any subprocesses using that buffer, are also deleted.
TPU$M_DELETE_WINDOWS	Deletes all windows. If this is not the last time you are calling DECTPU, then all variables referring to these data structures are reset, as if by the built-in procedure DELETE.
TPU$M_DELETE_CACHE	Deletes the virtual file manager's data structures and caches. If this deletion is requested, then all buffers are also deleted. If the cache is deleted, the initialization routine has to reinitialize the virtual file manager the next time it is called.
TPU$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 are no longer needed.
TPU$M_EXECUTE_FILE	Reexecutes the command file if TPU $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 TPU $INITIALIZE indicating the presence of the /COMMAND qualifier.
TPU$M_EXECUTE_PROC	Looks up TPU$INIT_PROCEDURE and executes it the next time TPU$EXECUTE_INIFILE is called.
TPU$M_DELETE_CONTEXT	Deletes the entire context of DECTPU. If this option is specified, then all other options are implied, except for executing the initialization file and initialization procedure.
TPU$M_RESET_TERMINAL	Resets the terminal to the state it was in upon entry to DECTPU. The terminal mailbox and all windows are deleted. If the terminal is reset, then it is reinitialized the next time TPU$INITIALIZE is called.
TPU$M_KILL_PROCESSES	Deletes all subprocesses created during the session.
TPU$M_CLOSE_SECTION[2]	Closes the section file and releases the associated memory. All buffers, windows, and processes are

Flag[1]	Function
	deleted. The cache is purged and the flags are set for reexecution of the initialization file and initialization procedure. If the section is closed and if the option bit indicates the presence of the SECTION qualifier, then the next call to TPU $INITIALIZE attempts a new restore operation.
TPU$M_DELETE_OTHERS	Deletes all miscellaneous preallocated data structures. Memory for these data structures is reallocated the next time TPU$INITIALIZE is called.
TPU$M_LAST_TIME	This bit should be set only when you are calling DECTPU for the last time. Note that if you set this bit and then recall DECTPU, the results are unpredictable.

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

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

Description

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

When you finish with DECTPU, 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 TPU$CLEANUP, do not delete the data structures; the operating system does this automatically. Allowing the operating system to delete the structures improves the performance of your program.

Notes

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

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

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

Condition Value Returned

TPU$_SUCCESS

Normal successful completion.

TPU$CLIPARSE

Parse a Command Line — The TPU$CLIPARSE routine parses a command line and builds the item list for TPU$INITIALIZE.

Format

```
TPU$CLIPARSE string ,fileio ,call_user
```

Returns

OpenVMS usage:	item_list
type:	longword (unsigned)
access:	read only
mechanism:	by reference

This routine returns the address of an item list.

Arguments

string

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

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

fileio

OpenVMS 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

OpenVMS 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 calls CLI$DCL_PARSE to establish a command table and a command to parse. It then calls TPU$PARSEINFO to build an item list for TPU$INITIALIZE.

If your application parses information that is not related to the operation of DECTPU, make sure the application obtains and uses all non-DECTPU parse information before the application calls TPU $CLIPARSE. You must do this because TPU$CLIPARSE destroys all parse information obtained and stored before TPU$CLIPARSE was called.

TPU$CLOSE_TERMINAL

Close Channel to Terminal — The TPU$CLOSE_TERMINAL routine closes the DECTPU channel to the terminal.

Format

```
TPU$CLOSE_TERMINAL
```

Returns

OpenVMS 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 under Condition Value Returned.

Description

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

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

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

Condition Value Returned

TPU$_SUCCESS

Normal successful completion.

TPU$CONTROL

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

Format

```
TPU$CONTROL [integer]
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

integer

OpenVMS usage:	integer
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Prevents DECTPU from displaying the message “Editing session is not being journalled” when the calling program gives control to DECTPU. Specify a true (odd) integer to preserve compatibility in future releases. If you omit the parameter, DECTPU displays the message if journalling is not enabled.

Description

This routine controls the editing session. It is responsible for reading the text and commands and for executing them. Windows on the screen are updated to reflect the edits made. Your program can regain control by interrupting DECTPU using the TPU$SPECIFY_ASYNC_ACTION routine, together with the TPU$TRIGGER_ASYNC_ACTION routine.

Note

Control is also returned to your program if an error occurs or when you enter either the built-in procedure QUIT or the built-in procedure EXIT.

Condition Values Returned

TPU$_EXITING

A result of EXIT (when the default condition handler is established).

TPU$_NONANSICRT

A result of operation termination — results when you call DECTPU with TPU$DISPLAYFILE set to nodisplay and you attempt to execute screen-oriented commands.

TPU$_QUITTING

A result of QUIT (when the default condition handler is established).

TPU$_RECOVERFAIL

A recovery operation was terminated abnormally.

TPU$EDIT

Edit a File — The TPU$EDIT routine builds a command string from its parameters and passes it to the TPU$TPU routine. TPU$EDIT is another entry point to the DECTPU simplified callable interface.

Format

```
TPU$EDIT input ,output
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

input

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

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

output

OpenVMS usage:	char_string
type:	character string
access:	read only

mechanism:	by descriptor

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

Description

This routine builds a command string and passes it to TPU$TPU. If the length of the output descriptor is nonzero, then the /OUTPUT qualifier is added to the command string. The /OUTPUT qualifier causes a file to be written to the specified file even if no modifications are made to the input file. If the QUIT built-in procedure is called, it prompts the user as if changes had been made to the buffer. This allows applications to check for the existence of the output file to see if the editing session was terminated, which is consistent with other OpenVMS callable editors.

If your application parses information that is not related to the operation of DECTPU, make sure the application obtains and uses all non-DECTPU parse information before the application calls TPU$EDIT. Your application must do this because TPU$EDIT destroys all parse information obtained and stored before TPU$EDIT is called.

Condition Values Returned

This routine returns the same values as TPU$TPU.

TPU$EXECUTE_COMMAND

Execute One or More DECTPU Statements — The TPU$EXECUTE_COMMAND routine allows your program to execute DECTPU statements.

Format

```
TPU$EXECUTE_COMMAND string
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

string

OpenVMS 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.

Description

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

Condition Values Returned

TPU$_SUCCESS

Normal successful completion.

TPU$_EXECUTEFAIL

Execution aborted. This could be because of execution errors or compilation errors.

TPU$_EXITING

EXIT built-in procedure was invoked.

TPU$_QUITTING

QUIT built-in procedure was invoked.

TPU$EXECUTE_INIFILE

Execute Initialization Files — The TPU$EXECUTE_INIFILE routine allows you to execute a user-written initialization file. This routine must be executed after the editor is initialized and before any other commands are processed.

Format

```
TPU$EXECUTE_INIFILE
```

Returns

OpenVMS 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 under Condition Values Returned.

Description

Calling the TPU$EXECUTE_INIFILE routine causes DECTPU to perform the following steps:

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

2. If you specified the /DEBUG qualifier on the command line, the DEBUG file is read into a buffer. The default is SYS$SHARE:TPU$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 TPU$CLEANUP, you must set the flags TPU $_EXECUTEPROCEDURE and TPU$_EXECUTEFILE. Otherwise, the initialization file does not execute.

Condition Values Returned

TPU$_SUCCESS

Normal successful completion.

TPU$_COMPILEFAIL

The compilation of the initialization file was unsuccessful.

TPU$_EXECUTEFAIL

The execution of the statements in the initialization file was unsuccessful.

TPU$_EXITING

A result of EXIT. If the default condition handler is being used, the session is terminated.

TPU$_FAILURE

General code for all other errors.

TPU$_QUITTING

A result of QUIT. If the default condition handler is being used, the session is terminated.

TPU$FILEIO

Perform File Operations — The TPU$FILEIO routine handles all DECTPU file operations. Your own file I/O routine can call this routine to perform some operations for it. However, the routine that opens the file must perform *all* operations for that file. For example, if TPU$FILEIO opens the file, it must also close it.

Format

```
TPU$FILEIO code ,stream ,data
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item code specifying a DECTPU function. The *code* argument is the address of a longword containing an item code from DECTPU specifying a function to perform. Following are the item codes that you can specify in the file I/O routine:

- TPU$K_OPEN—This item code specifies that the data parameter is the address of an item list. This item list contains the information necessary to open the file. The stream parameter should be filled in with a unique identifying value to be used for all future references to this file. The resultant file name should also be copied with a dynamic string descriptor.
- TPU$K_CLOSE—The file specified by the *stream* argument is to be closed. All memory being used by its structures can be released.
- TPU$K_CLOSE_DELETE—The file specified by the *stream* argument is to be closed and deleted. All memory being used by its structures can be released.
- TPU$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 Run-Time Library to copy text into this descriptor. DECTPU frees the memory allocated for the data read when the file I/O routine indicates that the end of the file has been reached.
- TPU$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.

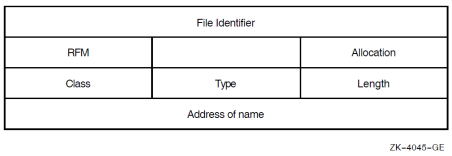
stream

OpenVMS 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 describes 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.

The following figure shows the stream data structure:



The first longword holds 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 DECTPU 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 Run-Time Library. This space is freed by DECTPU when it is no longer needed.

data

OpenVMS 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 TPU$K_OPEN item code is issued, the data parameter is the address of an item list containing information about the open request. The following DECTPU item codes are available for specifying information about the open request:

- TPU$K_ACCESS item code lets you specify one of three item codes in the buffer address field, as follows:
  - TPU$K_IO
  - TPU$K_INPUT
  - TPU$K_OUTPUT

- TPU$K_FILENAME item code is used for specifying the address of a string to use as the name of the file you are opening. The length field contains the length of this string, and the address field contains the address.
- TPU$K_DEFAULTFILE item code 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.
- TPU$K_RELATEDFILE item code is used for specifying a related file name for the file being opened. The buffer length field contains the length, and the buffer address field contains the address of a string to use as the related file name.
- TPU$K_RECORD_ATTR item code specifies that the buffer address field contains the value for the record attribute byte in the FAB (FAB$B_RAT) used for file creation.
- TPU$K_RECORD_FORM item code specifies that the buffer address field contains the value for the record format byte in the FAB (FAB$B_RFM) used for file creation.
- TPU$K_MAXIMIZE_VER item code specifies that the version number of the output file should be one higher than the highest existing version number.
- TPU$K_FLUSH item code specifies that the file should have every record flushed after it is written.
- TPU$K_FILESIZE item code is used for specifying a value to be used as the allocation quantity when creating the file. The value is specified in the buffer address field.

Description

By default, TPU$FILEIO creates variable-length files with carriage-return record attributes (FAB $B_RFM = VAR, FAB$B_RAT = CR). If you pass to it the TPU$K_RECORD_ATTR or TPU $K_RECORD_FORM item, that item is used instead.

The following combinations of formats and attributes are acceptable:

Format	**Attributes**
STM,STMLF,STMCR	0,BLK,CR,BLK+CR
VAR	0,BLK,FTN,CR,BLK+FTN,BLK+CR

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 easily distinguish the file control blocks (FCB) this routine fills in from the ones you created.

Note

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

Condition Values Returned

The TPU$FILEIO routine returns an OpenVMS RMS status code to DECTPU. The file I/O routine is responsible for signaling all errors if any messages are desired.

TPU$FILE_PARSE

Parse the Given File Specification — The TPU$FILE_PARSE routine provides a simplified interface to the $PARSE system service. DECTPU calls this routine when the built-in procedure FILE_PARSE is executed from TPU code.

Format

```
TPU$FILE_PARSE result-string ,flags ,filespec ,default-spec ,related-spec
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. See Condition Values Returned.

Arguments

result-string

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Includes the components of the file specification specified by the *flags* argument. The memory for the return string is allocated via the Run-Time Library routine LIB$SGET1_DD. Use the Run-Time Library routine LIB$SFREE1_DD to deallocate the memory for the return string.

flags

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Determine what file specification components should be returned. The following table shows the valid values for the *flags* argument:

Flag Bit[1]	**Description**
TPU$M_NODE	Returns the node component of the file specification.
TPU$M_DEV	Returns the device component of the file specification.

Flag Bit[1]	Description
TPU$M_DIR	Returns the directory component of the file specification.
TPU$M_NAME	Returns the name component of the file specification.
TPU$M_TYPE	Returns the type component of the file specification.
TPU$M_VER	Returns the version component of the file specification.
TPU$M_HEAD	Returns NODE, DEVICE and DIRECTORY components of the file specification. If the TPU M_NODE, TPUM_DEV or TPU$M_DIR bits are set while TPU$M_HEAD is set, the routine signals the error TPU$_INCKWDCOM and returns control to the caller.
TPU$M_TAIL	Returns NAME, TYPE and VERSION components of the file specification. If the TPU M_NAME, TPUM_TYPE or TPU$M_VER bits are set while TPU$M_TAIL is set, the routine signals the error TPU$_INCKWDCOM and returns control to the caller.

[1]TPU$M … indicates a mask. There is a corresponding value for each mask in the form TPU$V ….

filespec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

The object file specification.

default-spec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Contains the default file specification. The default file specification fields are used in the result string as substitutes for fields omitted in the *filespec* argument. You can also make substitutions in the result string using the *related-spec* argument.

Use the value 0 when no *default-spec* is to be applied to the file specification.

related-spec

OpenVMS usage:	char_string
type:	character string

access:	read only
mechanism:	by descriptor

Contains the related file specification. The fields in the related file specification are substituted in the *result-string* if a particular field is missing from both the *filespec* and *default-spec* arguments.

Use the value 0 when no *default-spec* is to be applied to the file specification.

Description

The TPU$FILE_PARSE routine returns a string containing the fields requested of the file specified. The file is not required to exist when the parse is done. The intention of the TPU$FILE_PARSE routine is to construct a valid file specification from the information passed in through the file specification, the default file specification, and the related file specification.

The routine uses the $PARSE system service to return the requested information.

The TPU$FILE_PARSE routine is also called by DECTPU when the TPU built-in procedure FILE_PARSE is executed from TPU code. The return value of the built-in procedure is the string returned in the *result-string* argument.

Condition Values Returned

TPU$_SUCCESS

Normal successful completion. If the return string contains a null-string, then the last match of the search operations has occurred.

TPU$_INCKWDCOM

The *flags* argument had an illegal combination of values.

TPU$_PARSEFAIL

The parse failed.

TPU$FILE_SEARCH

Search File System for Specified File — The TPU$FILE_SEARCH routine provides a simplified interface to the $SEARCH system service. DECTPU call this routine when TPU code executes the FILE_SEARCH built-in procedure.

Format

```
TPU$FILE_SEARCH result-string ,flags ,filespec ,default-spec ,related-spec
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. See Condition Values Returned.

Arguments

result-string

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Includes the components of the file specification passed by the *flags* argument. The memory for the return string is allocated via the Run-Time Library routine LIB$SGET1_DD. To deallocate memory for the string, use the Run-Time Library routine LIB$SFREE1_DD.

flags

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Determines what file specification components should be returned. The following table lists the valid flag values:

Flag[1]	**Function**
TPU$M_NODE	Returns the node component of the file specification.
TPU$M_DEV	Returns the device component of the file specification.
TPU$M_DIR	Returns the directory component of the file specification.
TPU$M_NAME	Returns the name component of the file specification.
TPU$M_TYPE	Returns the type component of the file specification.
TPU$M_VER	Returns the version component of the file specification.
TPU$M_REPARSE	Reparses the file specification before processing. This is intended to be used to reset the file search.
TPU$M_HEAD	Returns NODE, DEVICE, and DIRECTORY components of the file specification. If the TPU M_NODE, TPUM_DEV or TPU$M_DIR bits are set while TPU$M_HEAD is set, the routine will signal the error TPU$_INCKWDCOM and return.
TPU$M_TAIL	Returns NAME, TYPE and VERSION components of the file specification. If the TPU

Flag[1]	Function
	M_NAME, TPUM_TYPE or TPU$M_VER bits are set while TPU$M_TAIL is set, the routine will signal the error TPU$_INCKWDCOM and return.

[1]TPU$M … indicates a mask. There is a corresponding value for each mask in the form TPU$V ….

filespec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Object file specification.

default-spec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

The default file specification. The default file specification fields are used to fill in the *result-string* when fields are omitted in the *filespec* argument. Use the *related-spec* argument to specify other substitutions.

Use the value 0 when no *default-spec* is to be applied to the file specification.

related-spec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Contains the related file specification. The fields in the related file specification are used in the *result-string* for fields omitted in the *filespec* and *default-spec* arguments.

Use the value 0 when no *default-spec* is to be applied to the file specification.

Description

This routine allows an application to verify the existence of, and return components of, a file specification. Wildcard operations are permitted. The routine uses the $PARSE and $SEARCH system services to seek the file specification.

If no wildcards are included in the file specification string and the *result-string* returns a zero (0) length string, no file was found. If wildcard characters were present in the file specification and the *result-string* returns a zero (0) length string, there are no more files that match the wildcards.

To find all the files that match a wildcard specification, repeatedly call this routine, passing the same arguments, until the routine returns a zero-length result string.

The TPU$FILE_SEARCH routine is called by DECTPU when the TPU built-in procedure FILE_SEARCH is executed from TPU code. The return value of the built-in procedure is the string returned in the *result-string* argument.

Condition Values Returned

TPU$_SUCCESS

Normal successful completion. If the return string contains a null string, the final match operation was detected.

TPU$_INCKWDCOM

The *flags* argument had an illegal combination of values.

TPU$_PARSEFAIL

The requested repeat parse failed.

TPU$_SEARCHFAIL

An error occurred during the search operation.

TPU$HANDLER

TPU$HANDLER — The TPU$HANDLER routine is the DECTPU condition handler. The DECTPU condition handler invokes the $PUTMSG system service, passing it the address of TPU$MESSAGE.

Format

```
TPU$HANDLER signal_vector ,mechanism_vector
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value in R0. See Condition Values Returned.

Arguments

signal_vector

OpenVMS 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

OpenVMS 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

The TPU$MESSAGE routine performs the actual output of the message. The $PUTMSG system service only formats the message. It gets the settings for the message flags and facility name from the variables described in Section 8.1.2. Those values can be modified only by the DECTPU built-in procedure SET.

If the condition value received by the handler has a FATAL status or does not have the DECTPU 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 OpenVMS RMS status field (STS) and status value field (STV). Because this condition does not have a fatal severity, the editor continues after handling the error.

The editor does not automatically return from TPU$CONTROL. If you call the TPU$CONTROL routine, you must explicitly establish a way to regain control (for example, using the built-in procedure CALL_USER). If you establish your own condition handler but call the DECTPU handler for certain conditions, the default condition handler *must* be established at the point in your program where you want to return control. You can also interrupt TPU$CONTROL by having your program specify and then trigger an asynchronous routine via the TPU$SPECIFY_ASNYC_ACTION and TPU $TRIGGER_ASYNC_ACTION routines.

See the *VSI OpenVMS Calling Standard* for details on writing a condition handler.

TPU$INITIALIZE

Initialize DECTPU for Processing — The TPU$INITIALIZE routine initializes DECTPU for text processing. 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 Screen Manager and the I/O subsystem.

Format

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

Returns

OpenVMS 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 under Condition Values Returned.

Argument

callback

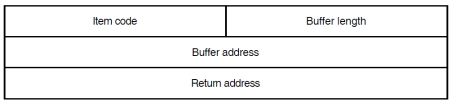
OpenVMS usage:	vector_longword_unsigned
type:	bound procedure value
access:	read only
mechanism:	by descriptor

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 command line parsing routine, which can be TPU$CLIPARSE or a user-written parsing routine.

Callable DECTPU defines item codes that you can use to specify initialization parameters. The following rules must be followed when building the item list:

- If you use the TPU$_OTHER_FILENAMES item code, it must follow the TPU$_FILENAME item code.
- If you use either the TPU$_CHAIN item code or the TPU$_ENDLIST code, it must be the last item code in the list.

The following figure shows the general format of an item descriptor. For information about how to build an item list, refer to the programmer's manual associated with the language you are using. Any reference to command line qualifiers refer to those command line qualifiers that you use with the EDIT/TPU command.



The return address in an item descriptor is usually 0.

The following item codes are available:

Item Code	Description
TPU$_OPTIONS	Enables the command qualifiers. The bits in the bit mask specified by the buffer address field correspond to the various DECTPU command qualifiers.
TPU$_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 can be a null string.
TPU$_SECTIONFILE	Passes the string that is the name of the binary initialization file (section file) to be mapped in. The buffer length field is the length of the string, and the buffer address field is the address of the string. If the TPU$V_SECTION bit is set, this item code must be specified.
TPU$_OUTPUTFILE	Passes the string specified with the /OUTPUT qualifier. The buffer length field is the length of the string, and the buffer address field specifies the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “OUTPUT_FILE”). The string can be a null string.
TPU$_DISPLAYFILE	Passes the string specified with the /DISPLAY qualifier. The buffer length field defines the length of the string, and the buffer address field defines the string address. The interface between the TPUSHR image and the display file image is not documented. Applications should only use this option with documented display files such as TPU $CCTSHR or TPU$MOTIFSHR.
TPU$_COMMANDFILE	Passes the string specified with the /COMMAND qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “COMMAND_FILE”). The string can be a null string.
TPU$_FILENAME	Passes the string that is the name of the first input file specified on 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, “FIRST_FILE_NAME”). This file name can be a null string.

Item Code	Description
TPU$_OTHER_FILENAMES	Passes a string that contains the name of an input file that follows the first input file on the command line. The buffer length field specifies the length of this string, and the buffer address field specifies its address. Each additional file specified on the command line requires its own TPU$_OTHER_FILENAMES item entry. These strings are returned by the GET_INFO (COMMAND_LINE, “NEXT_FILE_NAME”) built-in procedure in the order they appear in the item list. This item code must appear after the TPU $_FILENAME item in the item list.
TPU$_FILEIO	Passes the bound procedure value of a routine to be used for handling file operations. You can provide your own file I/O routine, or you can call TPU$FILEIO, the utility routine provided by DECTPU for handling file operations. The buffer address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that DECTPU loads into R1 before calling the routine.
TPU$_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 DECTPU loads into R1 before calling the routine.
TPU$_INIT_FILE	Passes the string specified with the /INITIALIZATION qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “INIT_FILE”).
TPU$_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 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.
TPU$_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

Item Code	Description
	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 to character 1.
TPU$_CHARACTERSET	Passes the string specified with the /CHARACTER_SET qualifier. The buffer length field specifies the string length and the buffer address field specifies the string address. Valid strings are “DEC_MCS ” (the default value), “ISO_LATIN1 ”, and “GENERAL ”. If the application tries to pass any other string, the routine signals an error and passes the default string (DEC_MCS).
TPU$_WORKFILE	Passes the string specified with the `/WORK` qualifier. The buffer length field specifies the string length and the buffer address specifies the string address. This string is available with GET_INFO (COMMAND_LINE, “WORK_FILE”).
TPU$_CHAIN	Passes the address of the next item list to the process specified by the buffer address field.
TPU$_ENDLIST	Signals the end of the item list.
TPU$_PARENT_WIDGET	Passes the appropriate parent widget when invoking the DECwindows version of the editor. This routine is not specified by the application; DECTPU invokes its own application shell. The widget address is passed in the buffer address field. This item code is only valid when using the DECwindows interface.
TPU$_APPLICATION_CONTEXT	Passes the application context to use with the TPU $_PARENT_WIDGET. DECTPU defaults to its own application context. The buffer address field specifies the application context address. This item code is only valid when using the DECwindows interface.
TPU$_DEFAULTSFILE	Specifies which file DECTPU uses to initialize the X defaults database. The buffer length field specifies the string length and the buffer address field specifies the string address. This item code is only valid when using the DECwindows interface.
TPU$_CTRL_C_ROUTINE	Passes the bound procedure value of a routine to be used for handling Ctrl/C asynchronous system traps (ASTs). DECTPU calls the routine when a Ctrl/C AST occurs. If the routine returns a FALSE value, DECTPU assumes that the Ctrl/C has been handled. If the routine returns a TRUE value, DECTPU aborts any currently executing DECTPU

Item Code	Description
	procedure. 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 DECTPU loads into R1 before calling the routine.
TPU$_DEBUGFILE	Passes the string specified with the /DEBUG command qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string.
TPU$_FILE_SEARCH	Passes the bound procedure value of a routine to be used to replace the TPU$FILE_SEARCH routine which is called when the built-in procedure FILE_SEARCH is called from TPU code. See the description of the TPU$FILE_SEARCH and the user routine FILE_SEARCH for more information.
TPU$_FILE_PARSE	Passes the bound procedure value of a routine to be used to replace the TPU$FILE_PARSE routine which is called when the built-in procedure FILE_PARSE is called from TPU code. See the description of the TPU$FILE_PARSE and the user routine FILE_PARSE for more information.

Table 8.1 lists the bits and corresponding masks enabled by the item code TPU$K_OPTIONS and shows how each bit affects TPU$INITIALIZE operation. Several bits in the TPU$_OPTIONS mask require additional item code entries in the item list. An example of this is TPU$M_COMMAND which requires a TPU$_COMMANDFILE entry in the item list.

Table 8.1. Valid Masks for the TPU$K_OPTIONS Item Code

Mask[1]	GET_INFO Request String [2]	Description
TPU$M_COMMAND	COMMAND	If DECTPU senses the presence of the TPU$_COMMANDFILE item, it tries to read, compile and execute the unbound TPU code.
TPU$M_COMMAND_DFLTED	Not applicable	Specifies that DECTPU should use the default command file name of TPU $COMMAND.TPU when reading in the command file. No error is reported if the default command file is not found. TPU$INITIALIZE fails when the TPU $M_COMMAND_DFLTED bit is set to 0 and no file is specified in the item list.
TPU$M_CREATE	CREATE	The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.

Mask[1]	**GET_INFO Request String** [2]	**Description**
TPU$M_DEBUG	Not applicable	If DECTPU senses the presence of the TPU$_DEBUGFILE item, it tries to read the file, and then proceeds to compile and execute its contents as TPU statements.
TPU$M_DEFAULTS	Not applicable	If DECTPU senses the presence of the TPU$_DEFAULTSFILE item, it uses the specified DECwindows X resource file to initialize the DECwindows X resource database.
TPU$M_DISPLAY	DISPLAY	If DECTPU senses the presence of the TPU$_DISPLAYFILE item, it tries to image activate the specified image as its screen manager. When the bit is 0, DECTPU uses SYS$OUTPUT for display and only the READ_LINE built-in procedure may be used for input.
TPU$M_INIT	INITIALIZATION	If DECTPU senses the presence of the TPU$_INIT_FILE item, it returns the specified string through the built-in procedure GET_INFO (COMMAND_LINE, “INITIALIZATION_FILE”). Processing of the initialization file is left to the application.
TPU$M_JOURNAL	JOURNAL	If DECTPU senses the presence of the TPU$_JOURNALFILE item, it outputs the keystrokes entered during the editing session to the specified file. **Note:** VSI recommends the use of buffer change journalling in new applications.
TPU$M_MODIFY	MODIFY	The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.
TPU$M_NODEFAULTS	Not applicable	DECTPU initializes the DECwindows X resource database only with resource files that the DECwindows toolkit routine *XtApplInitialize* loads into the database.
TPU$M_NOMODIFY	NOMODIFY	The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.
TPU$M_OUTPUT	OUTPUT	The behavior of DECTPU is not affected by this bit. Its interpretation

Mask[1]	GET_INFO Request String [2]	Description
		is left to the application layered on DECTPU.
TPU$M_READ	READ_ONLY	The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.
TPU$M_RECOVER	RECOVER	The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.
TPU$M_SECTION	SECTION	If DECTPU senses the presence of the TPU$_SECTIONFILE item, it tries to read the specified file as a binary initialization file. TPU$INITIALIZE fails if this bit is set to 1 and the TPU $_SECTIONFILE item is not present in the item list.
TPU$M_SEC_LNM_MODE	Not applicable	If DECTPU senses the presence of the TPU$M_SEC_LNM_MODE item, it looks only at executive mode logical names when attempting to read in a section file.
TPU$M_WORK	WORK	If DECTPU senses the presence of the TPU$_WORKFILE item, it uses the specified file for memory management. If no item list entry is present, and this bit is set to 1, a file is created in SYS $LOGIN:.TPU$WORK.
TPU$M_WRITE	WRITE	The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.

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

[2]Most bits in the mask have a corresponding GET_INFO (COMMAND_LINE) request string.

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

user_arg

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only
mechanism:	by value

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

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

Description

This 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.

Condition Values Returned

TPU$_SUCCESS

Initialization was completed successfully.

TPU$_FAILURE

General code for all other errors during initialization.

TPU$_INSVIRMEM

Insufficient virtual memory exists for the editor to initialize.

TPU$_NOFILEROUTINE

No routine has been established to perform file operations.

TPU$_NONANSICRT

The input device (SYS$INPUT) is not a supported terminal.

TPU$_RESTOREFAIL

An error occurred during the restore operation.

TPU$_SYSERROR

A system service did not work correctly.

TPU$MESSAGE

Write Message String — The TPU$MESSAGE routine writes error messages and strings using the built-in procedure, MESSAGE. Call this routine to have messages written and handled in a manner consistent with DECTPU. This routine should be used only after TPU$EXECUTE_INIFILE.

Format

```
TPU$MESSAGE string
```

Returns

OpenVMS 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 $PUTMSG system service.

Argument

string

OpenVMS 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.

TPU$PARSEINFO

Parse Command Line and Build Item List — The TPU$PARSEINFO routine parses a command and builds the item list for TPU$INITIALIZE.

Format

```
TPU$PARSEINFO fileio ,call_user
```

Returns

OpenVMS usage:	item_list
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The routine returns the address of an item list.

Arguments

fileio

OpenVMS usage:	vector_longword_unsigned
type:	bound procedure value
access:	read only
mechanism:	by descriptor

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

call_user

OpenVMS usage:	vector_longword_unsigned
type:	bound procedure value
access:	read only
mechanism:	by descriptor

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

Description

The TPU$PARSEINFO routine parses a command and builds the item list for TPU$INITIALIZE.

This routine uses the command language (CLI) routines to parse the current command. It makes queries about the command parameters and qualifiers that DECTPU 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 DECTPU, make sure the application obtains and uses all non-DECTPU parse information before the application calls the TPU $PARSEINFO interface. This is because TPU$PARSEINFO destroys all parse information obtained and stored before TPU$PARSEINFO was called.

TPU$SIGNAL

Signal a TPU Status — The TPU$SIGNAL routine allows applications and user-written TPU routines such as FILEIO to easily signal error messages in order for TPU error handlers to perform correctly.

Format

```
TPU$SIGNAL condition-code
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. In most cases, the routine returns either the same signal passed to it in the condition value argument, or the return value of LIB$SIGNAL. If the routine fails, it signals TPU $_FAILURE and returns control to the caller.

Argument

condition-code

OpenVMS usage:	cond_value
type:	longword (unsigned)

access:	read only
mechanism:	by value

The condition-code is an unsigned longword that contains the condition code to be signaled. In most cases, this argument is a TPU message code.

Description

TPU$SIGNAL performs the same function as the Run-Time Library routine LIB$SIGNAL, but it also processes TPU facility messages to allow TPU language ON_ERROR handlers to be called.

For example, assume that a user-written file input/output routine is designed to signal the error TPU $_OPENIN when it fails to open a file. Calling the TPU$SIGNAL routine and passing the value TPU $_OPENIN allows a case-style TPU ON_ERROR handler to receive the error, thus preserving the documented return values for TPU built-in procedures such as READ_FILE.

Note

You must call TPU$INITIALIZE before you call the TPU$SIGNAL routine.

If TPU$_QUITTING, TPU$_EXITING, or TPU$_RECOVERFAIL are passed to the routine, it calls the Run-Time Library routine LIB$SIGNAL.

If facility messages other than TPU messages are passed to the TPU$SIGNAL routine, it calls the LIB $SIGNAL routine and passes the appropriate condition value.

TPU$SPECIFY_ASYNC_ACTION

Register an Asynchronous Action — The TPU$SPECIFY_ASYNC_ACTION routine allows applications using the DECTPU full callable interface to register asynchronous actions with DECTPU.

Format

```
TPU$SPECIFY_ASYNC_ACTION facility_index [,tpu_statement]
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

facility_index

OpenVMS usage:	longword_unsigned
type:	longword (signed)

access:	read only
mechanism:	by reference

Represents an index of the asynchronous action. This index is used with the TPU $TRIGGER_ASYNC_ACTION routine to let DECTPU know what action to perform. It may also be used to delete an action routine (by omitting the *tpu_statement*). You may register several asynchronous actions depending on your application's needs. This facility index number may be any positive integer.

tpu_statement

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

The DECTPU statement you want executed when you call the TPU$TRIGGER_ASYNC_ACTION routine. The statement is compiled and then stored internally. If you omit the parameter, DECTPU removes the action from its list of asynchronous events.

Description

The TPU$SPECIFY_ASYNC_ACTION routine, along with TPU$TRIGGER_ASYNC_ACTION, allow applications to interrupt DECTPU after calling TPU$CONTROL. The specified DECTPU statement is compiled and saved.

This routine must be called after TPU$INITIALIZE. It will not complete successfully if keystroke journalling is enabled.

Condition Values Returned

TPU$_SUCCESS

Normal successful completion.

TPU$_COMPILEFAIL

The code specified in *tpu_statement* did not compile successfully.

TPU$_INVPARM

An invalid parameter was passed.

TPU$_JNLACTIVE

Keystroke journalling is active. This routine requires that either journalling be turned off or that buffer change journalling be used.

TPU$TPU

Invoke DECTPU — The TPU$TPU routine invokes DECTPU and is equivalent to the DCL command EDIT/TPU.

Format

```
TPU$TPU command
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

command

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Command string. Note that the verb is TPU instead of EDIT/TPU. The *command* argument is the address for a descriptor of a command line.

Description

This routine takes the command string specified and passes it to the editor. DECTPU 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 TPU$CLOSE_SECTION. This feature lets you make multiple calls to TPU$TPU 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 DECTPU, make sure the application obtains and uses all non-DECTPU parse information before the application calls TPU$TPU. This is because TPU$TPU destroys all parse information obtained and stored before TPU$TPU was called.

Condition Values Returned

This routine returns any condition value returned by TPU$INITIALIZE, TPU$EXECUTE_INIFILE, TPU$CONTROL, and TPU$CLEANUP.

TPU$TRIGGER_ASYNC_ACTION

Execute DECTPU Command at Asynchronous Level — The TPU$TRIGGER_ASYNC_ACTION routine allows applications using the DECTPU full callable interface to interrupt the DECTPU TPU $CONTROL loop at an asynchronous level.

Format

```
TPU$TRIGGER_ASYNC_ACTION facility_index
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

facility_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The *facility_index* argument represents the asynchronous action to be taken. This is the same index passed to the TPU$SPECIFY_ASYNC_ACTION routine registering what DECTPU statements to execute.

Description

The TPU$TRIGGER_ASYNC_ACTION routine, along with TPU$SPECIFY_ASYNC_ACTION routine allow applications to interrupt DECTPU after calling TPU$CONTROL. The command that was specified for this *facility_index* is put on the DECTPU queue of work items and is handled as soon as no other work items are present. This allows DECTPU to complete and stabilize its environment before executing the command. This routine must be called after control has been passed to DECTPU via the TPU$CONTROL routine.

Condition Values Returned

TPU$_SUCCESS

Normal successful completion.

TPU$_UNKFACILITY

The *facility_index* passed to this routine does not match any facility index passed to TPU $SPECIFY_ASYNC_ACTION.

FILEIO

User-Written Routine to Perform File Operations — The user-written FILEIO routine is used to handle DECTPU file operations. The name of this routine can be either your own file I/O routine or the name of the DECTPU file I/O routine (TPU$FILEIO).

Format

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

Returns

OpenVMS usage:	cond_value
type:	longword (usigned)
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 under Condition Values Returned.

Arguments

code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item code specifying a DECTPU function. The *code* argument is the address of a longword containing an item code from DECTPU, which specifies a function to perform.

stream

OpenVMS 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

OpenVMS 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.

Description

The bound procedure value of this 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 TPU $FILEIO and pass it the parameters for any file operation you do not want to handle. Note, however, that TPU$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 mix the file operations between your own file I/O routine and the one supplied by DECTPU.

Condition Values Returned

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

FILE_PARSE

User-Written Routine to Perform File Parse Operations — This is a user-written routine that can be used in place of the TPU$FILE_PARSE routine.

Format

```
FILE_PARSE result-string ,flags ,filespec ,default-spec ,related-spec
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. The return value is ignored by DECTPU. User-written FILE_PARSE routines should include calls to the TPU$SIGNAL routine to ensure proper error handling.

Arguments

result-string

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Return value for the built-in procedure FILE_PARSE. The calling program should fill in this descriptor with a dynamic string allocated by the string routines, such as the Run-Time Library routine LIB $SGET1_DD. DECTPU frees this string when necessary.

flags

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)

access:	read only
mechanism:	by reference

The following table lists the valid flag values used to request file specification components:

Flag[1]	**Function**
TPU$M_NODE	Requests for the node component of the file specification.
TPU$M_DEV	Requests for the device component of the file specification.
TPU$M_DIR	Requests for the directory component of the file specification.
TPU$M_NAME	Requests for the name component of the file specification.
TPU$M_TYPE	Requests for the type component of the file specification.
TPU$M_VER	Requests for the version component of the file specification.
TPU$M_HEAD	Requests for the NODE, DEVICE, and DIRECTORY components of the file specification.
TPU$M_TAIL	Requests for NAME, TYPE, and VERSION components of the file specification.

[1]TPU$M … indicates a mask. There is a corresponding value for each mask in the form TPU$V ….

filespec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

The object file specification.

default-spec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Contains the default file specification. The value 0 is passed if there is no *default-spec* argument.

related-spec

OpenVMS usage:	char_string
type:	character string
access:	read only

mechanism:	by descriptor

The *related-spec* argument contains the related file specification. The value 0 is passed if there is no related-spec.

Description

This routine allows an application to replace the TPU$FILE_PARSE routine with its own file-parsing routine. The calling program passes the address of the file-parsing routine to TPU$INITIALIZE using the TPU$_FILE_PARSE item code.

When the DECTPU built-in procedure FILE_PARSE is called from TPU code, DECTPU calls either the user-written routine (if one was passed to TPU$INITIALIZE) or the TPU$FILE_PARSE routine. The return value of the built-in procedure is the string returned in the *result-string* argument.

To ensure proper operation of the user's ON_ERROR error handlers, errors should be signaled using the TPU$SIGNAL routine.

FILE_SEARCH

User-Written Routine to Perform File Search Operations — This is a user-written routine that is used in place of the TPU$FILE_SEARCH routine.

Format

```
FILE_SEARCH result-string ,flags ,filespec ,default-spec ,related-spec
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. If an odd numeric value is returned, the next call to the built-in procedure FILE_SEARCH automatically sets the TPU$M_REPARSE bit in the flags longword. TPU $M_REPARSE is also set if the *result-string* has a length of 0.

Arguments

result-string

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Return value for the built-in procedure FILE_SEARCH. Your program should fill in this descriptor with a dynamic string allocated by the string routines such as the Run-Time Library routine LIB $SGET1_DD. DECTPU frees this string when necessary.

The TPU$M_REPARSE bit is set in the flags longword if the *result-string* has a length of zero. The bit is intended to reset the file search when wildcard searches are performed.

flags

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The following table shows the flags used for specifying the file components:

Flag[1]	**Function**
TPU$M_NODE	Requests for the node component of the file specification.
TPU$M_DEV	Requests for the device component of the file specification.
TPU$M_DIR	Requests for the directory component of the file specification.
TPU$M_NAME	Requests for the name component of the file specification.
TPU$M_TYPE	Requests for the type component of the file specification.
TPU$M_VER	Requests for the version component of the file specification.
TPU$M_REPARSE	Reparses the file specification before processing. This is intended as a way to restart the file search. This flag will automatically be set by DECTPU if on a previous call to the FILE_SEARCH user routine the *result-string* has a zero length or the routine returns a odd (noneven) status.
TPU$M_HEAD	Requests for the NODE, DEVICE, and DIRECTORY components of the file specification.
TPU$M_TAIL	Requests for the NAME, TYPE, and VERSION component of the file specification.

[1]TPU$M … indicates a mask. There is a corresponding value for each mask in the form TPU$V ….

filespec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

The object file specification.

default-spec

OpenVMS usage:	char_string

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

The *default-spec* argument contains the default file specification.

The value 0 is passed if there is no *default-spec*.

related-spec

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

The *related-spec* argument contains the related file specification.

The value 0 is passed if there is no *related-spec*.

Description

The FILE_SEARCH user routine allows an application to replace the TPU$FILE_SEARCH routine with its own file-searching routine. The calling program passes the address of the routine to the TPU $INITIALIZE routine using the TPU$_FILE_SEARCH item code.

When the DECTPU built-in procedure FILE_SEARCH is called from TPU code, DECTPU calls either the user-written FILE_SEARCH routine (if one was passed to TPU$INITIALIZE) or the TPU $FILE_SEARCH routine. The return value of the built-in procedure is the string returned in the *result-string* argument.

To ensure proper operation of the user's ON_ERROR handlers, errors in the user-written FILE_PARSE routine should be signaled using the TPU$SIGNAL routine.

HANDLER

User-Written Condition Handling Routine — The user-written HANDLER routine performs condition handling.

Format

```
HANDLER signal_vector ,mechanism_vector
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value.

Arguments

signal_vector

OpenVMS 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

OpenVMS 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 programming concepts, refer to *VSI OpenVMS Programming Concepts Manual*.

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

INITIALIZE

User-Written Initialization Routine — The user-written initialization callback routine is passed to TPU$INITIALIZE as a bound procedure value and called to supply information needed to initialize DECTPU.

Format

```
INITIALIZE [user_arg]
```

Returns

OpenVMS usage:	item_list
type:	longword (unsigned)
access:	read only
mechanism:	by reference

This routine returns the address of an item list.

Arguments

user_arg

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only
mechanism:	by value

User argument.

Description

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

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

The *user_arg* parameter is provided to allow an application to pass information through TPU $INITIALIZE to the user-written initialization routine. DECTPU 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 about TPU$INITIALIZE. . Most of the initialization parameters have a default value; strings default to the null string, and flags default to false. The only required initialization parameter is the address of a routine for file I/O. If an entry for the file I/O routine address is not present in the item list, TPU$INITIALIZE returns with a failure status.

USER

User-Written Routine Called from a DECTPU Editing Session — The user-written USER routine allows your program to take control during a DECTPU editing session (for example, to leave the editor temporarily and perform a calculation).

Format

```
USER integer ,stringin ,stringout
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value.

Arguments

integer

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

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

stringin

OpenVMS 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

OpenVMS usage:	char_string
type:	character string
access:	modify
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 (such as LIB$SGET1_DD) provided by the Run-Time Library. The DECTPU editor frees this string when necessary.

Description

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 can be used as operands in a calculation within a Fortran program.) Using the string routines provided by the Run-Time Library, your application fills in the `stringout` parameter in the call-user routine, which returns the `stringout` value to the built-in procedure CALL_USER.

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.

See Section 8.5 for a description of how to create an executable image for the USER routine and how to call the routine from a C program in the DECTPU environment.

Chapter 9. DECdts Portable Applications Programming Interface

You can use the Digital Distributed Time Service (DECdts) programming routines to obtain timestamps that are based on Coordinated Universal Time (UTC). You can also use the DECdts routines to translate among different timestamp formats and perform calculations on timestamps. Applications can use the timestamps that DECdts supplies to determine event sequencing, duration, and scheduling. Applications can call the DECdts routines from DECdts server or clerk systems.

The Digital Distributed Time Service routines are written in the C programming language. You should be familiar with the basic DECdts concepts before you attempt to use the applications programming interface (API).

The DECdts API routines can perform the following basic functions:

- Retrieve timestamp information
- Convert between binary timestamps that use different time structures
- Convert between binary timestamps and ASCII representations
- Convert between UTC time and local time
- Convert the binary time values in the OpenVMS (Smithsonian-based) format to or from UTC-based binary timestamps (OpenVMS systems only)
- Manipulate binary timestamps
- Compare two binary time values
- Calculate binary time values
- Obtain time zone information

DECdts can convert between several types of binary time structures that are based on different calendars and time unit measurements. DECdts uses UTC-based time structures and can convert other types of time structures to its own presentation of UTC-based time.

The following sections describe DECdts time representations, DECdts time structures, API header files, and API routines.

9.1. DECdts Time Representation

UTC is the international time standard that has largely replaced Greenwich Mean Time (GMT). The standard is administered by the International Time Bureau (BIH) and is widely used. DECdts uses opaque binary timestamps that represent UTC for all of its internal processes. You cannot read or disassemble a DECdts binary timestamp; the DECdts API allows applications to convert or manipulate timestamps, but they cannot be displayed. DECdts also translates the binary timestamps into ASCII text strings, which can be displayed.

9.1.1. Absolute Time Representation

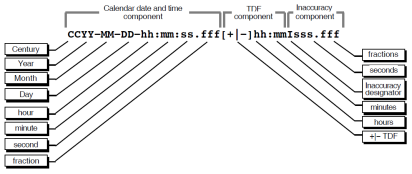
An **absolute time** is a point on a time scale. For DECdts, absolute times reference the UTC time scale; absolute time measurements are derived from system clocks or external time-providers. When DECdts

reads a system clock time, it records the time in an opaque binary timestamp that also includes the inaccuracy and other information. When you display an absolute time, DECdts converts the time to ASCII text, as shown in the following display:

```
1996-11-21-13:30:25.785-04:00I000.082
```

DECdts displays all times in a format that complies with the International Standards Organization (ISO) 8601 (1988) standard. Note that the inaccuracy portion of the time is not defined in the ISO standard (times that do not include an inaccuracy are accepted). Figure 9.1 explains the ISO format that generated the previous display.

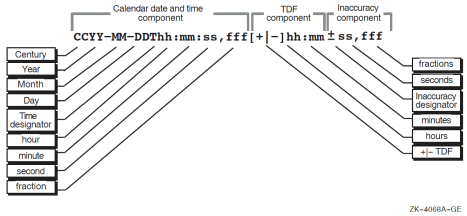
Figure 9.1. Time Display Format



In Figure 9.1, the relative time preceded by the plus (+) or minus (-) character indicates the hours and minutes that the calendar date and time are offset from UTC. The presence of this **time differential factor** (TDF) in the string also indicates that the calendar date and time are the local time of the system, not UTC. Local time is UTC minus the TDF. The Inaccuracy designator I indicates the beginning of the inaccuracy component associated with the time.

Although DECdts displays all times in the previous format, variations in the ISO format shown in Figure 9.2 are also accepted as input for the ASCII conversion routines.

Figure 9.2. Time Display Format Variants



In Figure 9.2, the Time designator T separates the calendar date from the time, a comma separates seconds from fractional seconds, and the plus or minus character indicates the beginning of the inaccuracy component.

The following examples show some valid time formats.

The following represents July 4, 1776 17:01 GMT and an infinite inaccuracy (default).

```
1776-7-4-17:01:00
```

The following represents a local time of 12:01 (17:01 GMT) on July 4, 1776 with a TDF of -5 hours and an inaccuracy of 100 seconds.

```
1776-7-4-12:01:00-05:00I100
```

Both of the following represent 12:00 GMT in the current day, month, and year with an infinite inaccuracy.

```
12:00 and T12
```

The following represents July 14, 1792 00:00 GMT with an infinite inaccuracy.

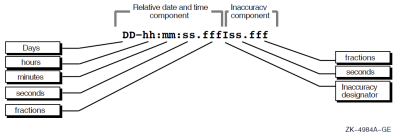
```
1792-7-14
```

9.1.2. Relative Time Representation

A **relative time** is a discrete time interval that is usually added to or subtracted from another time. A TDF associated with an absolute time is one example of a relative time. A relative time is normally used as input for commands or system routines.

Figure 9.3 shows the full syntax for a relative time.

Figure 9.3. Relative Time Syntax



Notice that a relative time does not use the calendar date fields, because these fields concern absolute time. A positive relative time is unsigned; a negative relative time is preceded by a minus (-) sign. A relative time is often subtracted from or added to another relative or absolute time. The relative times that DECdts uses internally are opaque binary timestamps. The DECdts API offers several routines that can be used to calculate new times using relative binary timestamps.

The following example shows a relative time of 21 days, 8 hours, and 30 minutes, 25 seconds with an inaccuracy of 0.300 second.

```
21-08:30:25.000I00.300
```

The following example shows a negative relative time of 20.2 seconds with an infinite inaccuracy (default).

```
-20.2
```

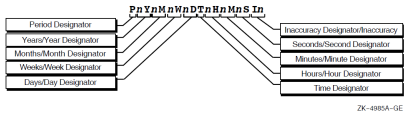
The following example shows a relative time of 10 minutes, 15.1 seconds with an inaccuracy of 4 seconds.

```
10:15.1I4
```

Representing Periods of Time

A given duration of a period of time can be represented by a data element of variable length that uses the syntax shown in Figure 9.4.

Figure 9.4. Time Period Syntax



The data element contains the following parts:

- The designator *P* precedes the part that includes the calendar components, including the following:
  - The number of years followed by the designator *Y*
  - The number of months followed by the designator *M*
  - The number of weeks followed by the designator *W*

    - The number of days followed by the designator `D`
- The designator `T` precedes the part that includes the time components, including the following:
    - The number of hours followed by the designator `H`
    - The number of minutes followed by the designator `M`
    - The number of seconds followed by the designator `S`
- The designator `I` precedes the number of seconds of inaccuracy.

The following example represents a period of 1 year, 6 months, 15 days, 11 hours, 30 minutes, and 30 seconds and an infinite inaccuracy.

```
P1Y6M15DT11H30M30S
```

The following example represents a period of 3 weeks and an inaccuracy of 4 seconds.

```
P3WI4
```

9.2. Time Structures

DECdts can convert between several types of binary time structures that are based on different base dates and time unit measurements. DECdts uses UTC-based time structures and can convert other types of time structures to its own presentation of UTC-based time. The DECdts API routines are used to perform these conversions for applications on your system.

Table 9.1 lists the absolute time structures that the DECdts API uses to modify binary times for applications.

Table 9.1. Absolute Time Structures

Structure	Time Units	Base Date	Approximate Range
utc	100-nanosecond	15 October 1582	A.D. 1 to A.D. 30,000
tm	second	1 January 1900	A.D. 1 to A.D. 30,000
timespec	nanosecond	1 January 1970	A.D. 1970 to A.D. 2106

Table 9.2 lists the relative time structures that the DECdts API uses to modify binary times for applications.

Table 9.2. Relative Time Structures

Structure	Time Units	Approximate Range
utc	100-nanosecond	± 30,000 years
tm	second	± 30,000 years
reltimespec	nanosecond	± 68 years

The remainder of this section explains the DECdts time structures in detail.

9.2.1. The utc Structure

Coordinated Universal Time (UTC) is useful for measuring time across local time zones and for avoiding the seasonal changes (summer time or daylight saving time) that can affect the local time. DECdts

uses 128-bit binary numbers to represent time values internally; throughout this manual, these binary numbers representing time values are referred to as **binary timestamps**. The DECdts utc structure determines the ordering of the bits in a binary timestamp; all binary timestamps that are based on the utc structure contain the following information:

- The count of 100-nanosecond units since 00:00:00.00, 15 October 1582 (the date of the Gregorian reform to the Christian calendar)
- The count of 100-nanosecond units of inaccuracy applied to the above
- The time differential factor (TDF), expressed as the signed quantity
- The timestamp version number

The binary timestamps that are derived from the DECdts utc structure have an opaque format. This format is a cryptic character sequence that DECdts uses and stores internally. The opaque binary timestamp is designed for use in programs, protocols, and databases.

Note

Applications use the opaque binary timestamps when storing time values or when passing them to DECdts.

The API provides the necessary routines for converting between opaque binary timestamps and character strings that can be displayed and read by users.

9.2.2. The tm Structure

The tm structure is based on the time in years, months, days, hours, minutes, and seconds since 00:00:00 GMT (Greenwich Mean Time), 1 January 1900. The tm structure is defined in the <time.h> header file.

The tm structure declaration follows:

```
struct tm {
                int tm_sec;     /* Seconds (0 - 59)                    */
                int tm_min;     /* Minutes (0 - 59)                    */
                int tm_hour;    /* Hours (0 - 23)                      */
                int tm_mday;    /* Day of Month (1 - 31)               */
                int tm_mon;     /* Month of Year (0 - 11)              */
                int tm_year;    /* Year - 1900                         */
                int tm_wday;    /* Day of Week (Sunday = 0)            */
                int tm_yday;    /* Day of Year (0 - 364)               */
                int tm_isdst;   /* Nonzero if Daylight Savings Time    */
                                /*  is in effect                       */
          };
```

Not all of the tm structure fields are used for each routine that converts between tm structures and utc structures. See the parameter descriptions that accompany the routines in this chapter for additional information about which fields are used for specific routines.

9.2.3. The timespec Structure

The timespec structure is normally used in combination with or in place of the tm structure to provide finer resolution for binary times. The timespec structure is similar to the tm structure, but the timespec

structure specifies the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970. You can find the structure in the <utc.h> header file.

The timespec structure declaration follows:

```
struct timespec {

             unsigned long tv_sec; /*  Seconds since 00:00:00 GMT,     */
                                   /*   1 January 1970                 */
             long tv_nsec;         /*  Additional nanoseconds since    */
                                   /*   tv_sec                         */

            }         timespec_t;
```

9.2.4. The reltimespec Structure

The reltimespec structure represents relative time. This structure is similar to the timespec structure, except that the first field is *signed* in the reltimespec structure. (The field is *unsigned* in the timespec structure.) You can find the reltimespec structure in the <utc.h> header file.

The reltimespec structure declaration follows:

```
struct reltimespec {

                long tv_sec;   /*  Seconds of relative time     */
                long tv_nsec;  /*  Additional nanoseconds of    */
                               /*   relative time               */

               }   reltimespec_t;
```

9.2.5. The OpenVMS Time Structure

The OpenVMS time structure is based on Smithsonian time, which has a base date of November 17, 1858. The binary OpenVMS structure is a signed, 64-bit integer that has a positive value for absolute times. You can use the DECdts API to translate an OpenVMS structure representing an absolute time to or from the DECdts UTC-based binary timestamp.

9.3. DECdts API Header Files

On OpenVMS systems, the header files are located in the SYS$LIBRARY directory. The <time.h> and <utc.h> header files contain the data structures, type definitions, and define statements that are referenced by the DECdts API routines. The <time.h> header file is present on all OpenVMS systems. The <utc.h> header file includes <time.h> and contains the timespec , reltimespec , and utc structures.

9.4. Linking Programs with the DECdts API

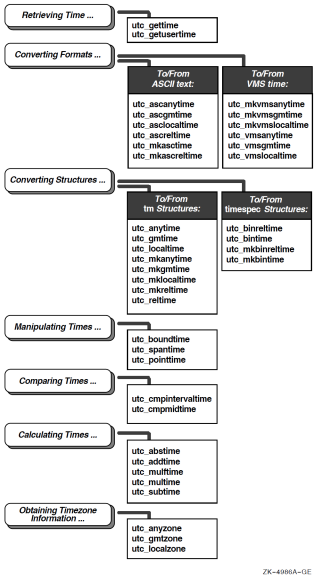
The DECdts API is implemented by a shared image. To use the API with your program, you must link the program with this shared image. On DECnet-Plus for OpenVMS systems, the DECdts API is implemented by the shared image SYS$LIBRARY:DTSS$SHR.EXE. The following example shows how to link a program with the DECdts shared image:

```
$ CC MYPROGRAM.C/OUTPUT=MYPROGRAM.OBJ
$ LINK MYPROGRAM.OBJ, SYS$INPUT:/OPTIONS
SYS$LIBRARY:DTSS$SHR.EXE/SHARE [Ctrl-z]
$
```

9.5. DECdts API Routine Functions

Figure 9.5 categorizes the DECdts portable interface routines by function.

Figure 9.5. DTS Portable Interface Categories



An alphabetical listing of the DECdts portable interface routines and a brief description of each one follows:

utc_abstime	Computes the absolute value of a binary relative time.
utc_addtime	Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time.
utc_anytime	Converts a binary timestamp into a tm structure, using the TDF information contained in the timestamp to determine the TDF returned with the tm structure.
utc_anyzone	Gets the time zone label and offset from GMT, using the TDF contained in the input utc.
utc_ascanytime	Converts a binary timestamp into an ASCII string that represents an arbitrary time zone.
utc_ascgmtime	Converts a binary timestamp into an ASCII string that expresses a GMT time.
utc_asclocaltime	Converts a binary timestamp to an ASCII string that represents a local time.
utc_ascreltime	Converts a binary timestamp that expresses a relative time to its ASCII representation.
utc_binreltime	Converts a relative binary timestamp into timespec structures that express relative time and inaccuracy.
utc_bintime	Converts a binary timestamp into a timespec structure.
utc_boundtime	Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event.
utc_cmpintervaltime	Compares two binary timestamps or two relative binary timestamps.

utc_cmpmidtime	Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies.
utc_gettime	Returns the current system time and inaccuracy as an opaque binary timestamp.
utc_getusertime	Returns the time and process-specific TDF, rather than the system-specific TDF.
utc_gmtime	Converts a binary timestamp into a tm structure that expresses GMT or the equivalent UTC.
utc_gmtzone	Gets the time zone label and zero offset from GMT, given utc.
utc_localtime	Converts a binary timestamp into a tm structure that expresses local time.
utc_localzone	Gets the time zone label and offset from GMT, given utc.
utc_mkanytime	Converts a tm structure and TDF (expressing the time in an arbitrary time zone) into a binary timestamp.
utc_mkascreltime	Converts a null-terminated character string, which represents a relative timestamp to a binary timestamp.
utc_mkasctime	Converts a null-terminated character string, which represents an absolute timestamp, to a binary timestamp.
utc_mkbinreltime	Converts a timespec structure expressing a relative time to a binary timestamp.
utc_mkbintime	Converts a timespec structure into a binary timestamp.
utc_mkgmtime	Converts a tm structure that expresses GMT or UTC to a binary timestamp.
utc_mklocaltime	Converts a tm structure that expresses local time to a binary timestamp.
utc_mkreltime	Converts a tm structure that expresses relative time to a binary timestamp.
utc_mkvmsanytime	Converts a binary OpenVMS format time and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.
utc_mkvmsgmtime	Converts a binary OpenVMS format time expressing GMT (or the equivalent UTC) into a binary timestamp.
utc_mkvmslocaltime	Converts a local binary OpenVMS format time to a binary timestamp, using the host system's TDF.
utc_mulftime	Multiplies a relative binary timestamp by a floating-point value.
utc_multime	Multiplies a relative binary timestamp by an integer factor.
utc_pointtime	Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time.
utc_reltime	Converts a binary timestamp that expresses a relative time into a tm structure.
utc_spantime	Given two (possibly unordered) UTC timestamps, returns a single UTC time interval whose inaccuracy spans the two input timestamps.
utc_subtime	Computes the difference between two binary timestamps that express two relative times (an absolute time and a relative time, two relative times, or two absolute times).

utc_vmsanytime	Converts a binary timestamp to a binary OpenVMS-format time, using the TDF contained in the binary timestamp.
utc_vmsgmtime	Converts a binary timestamp to a binary OpenVMS-format time expressing GMT or the equivalent UTC.
utc_vmslocaltime	Converts a binary timestamp to a local binary OpenVMS format time, using the host system's time differential factor.

Note

Absolute time is a point on a time scale; absolute time measurements are derived from system clocks or external time-providers. For DECdts, absolute times reference the UTC standard and include the inaccuracy and other information. When you display an absolute time, DECdts converts the time to ASCII text, as shown in the following display:

```
1996-11-21-13:30:25.785-04:00I000.082
```

Relative time is a discrete time interval that is usually added to or subtracted from an absolute time. A time differential factor (TDF) associated with an absolute time is one example of a relative time. Note that a relative time does not use the calendar date fields, because these fields concern absolute time.

Coordinated Universal Time (UTC) is the international time standard that DECdts uses. The zero hour of UTC is based on the zero hour of Greenwich Mean Time (GMT). The documentation consistently refers to the time zone of the Greenwich Meridian as GMT. However, this time zone is also sometimes referred to as UTC.

The **time differential factor** (TDF) is the difference between UTC and the time in a particular time zone.

OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys$timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

Unless otherwise specified, the default input and output parameters for the DECdts API routine commands are as follows:

- If *utc* is not specified as an input parameter, the current time is used.
- If *inacc* is not specified as an input parameter, infinity is used.
- If no output parameter is specified, no result (or an error) is returned.

The following command reference section includes all DECdts API routines.

utc_abstime

utc_abstime — Computes the absolute value of a relative binary timestamp.

Format

```
#include <utc.h>
int utc_abstime( result, *utc1)
```

```
utc_t result ;
const utc_t *utc1;
```

Parameter

Input

utc1

Relative binary timestamp.

Output

result

Absolute value of the input relative binary timestamp.

Description

The **Absolute Time** routine computes the absolute value of a relative binary timestamp. The input timestamp represents a relative (delta) time.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example scales a relative time, computes its absolute value, and prints the result.

```
utc_t       relutc, scaledutc;
char        timstr[UTC_MAX_STR_LEN];

/*
 *   Make sure relative timestamp represents a positive interval...
 */

utc_abstime(&relutc,            /* Out: Abs-value of rel time  */
            &relutc);           /* In:  Relative time to scale */

/*
 *   Scale it by a factor of 17...
 */

utc_multime(&scaledutc,         /* Out: Scaled relative time   */
            &relutc,            /* In:  Relative time to scale */
            17L);               /* In:  Scale factor           */

utc_ascreltime(timstr,          /* Out: ASCII relative time    */
               UTC_MAX_STR_LEN, /* In:  Length of input string */
               &scaledutc);     /* In:  Relative time to       */
                                /*      convert                */
```

```
printf("%s\n",timstr);

/*
 *   Scale it by a factor of 17.65...
 */

utc_mulftime(&scaledutc,        /* Out: Scaled relative time   */
             &relutc,           /* In:  Relative time to scale */
             17.65);            /* In:  Scale factor           */

utc_ascreltime(timstr,          /* Out: ASCII relative time    */
               UTC_MAX_STR_LEN, /* In:  Length of input string */
               &scaledutc);     /* In:  Relative time to       */
                                /*      convert                */

printf("%s\n",timstr);
```

utc_addtime

utc_addtime — Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time.

Format

```
#include <utc.h>
int utc_addtime( result, *utc1, *utc2)

utc_t result ;
const utc_t *utc1;
const utc_t *utc2;
```

Parameter

Input

utc1

Binary timestamp or relative binary timestamp.

utc2

Binary timestamp or relative binary timestamp.

Output

result

Resulting binary timestamp or relative binary timestamp, depending on the operation performed:

- *relative time* + *relative time* = **relative time**
- *absolute time* + *relative time* = **absolute time**
- *relative time* + *absolute time* = **absolute time**

- *absolute time* + *absolute time* is undefined. See **NOTES**.

Description

The **Add Time** routine adds two binary timestamps, producing a third binary timestamp whose inaccuracy is the sum of the two input inaccuracies. One or both of the input timestamps typically represent a relative (delta) time. The TDF in the first input timestamp is copied to the output.

Although no error is returned, do **not** use the combination *absolute time* + *absolute time*.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example shows how to compute a timestamp that represents a time at least 5 seconds in the future.

```
utc_t               now, future, fivesec;
reltimespec_t       tfivesec;
timespec_t          tzero;

/*
 *   Construct a timestamp that represents 5 seconds...
 */
tfivesec.tv_sec = 5;
tfivesec.tv_nsec = 0;
tzero.tv_sec = 0;
tzero.tv_nsec = 0;
utc_mkbinreltime(&fivesec, /* Out: 5 secs in binary timestamp */
                 &tfivesec, /* In:  5 secs in timespec         */
                 &tzero);   /* In:  0 secs inaccuracy in timespec */

/*
 *  Get the maximum possible current time...
 *  (NULL input parameter is used to specify the current time.)
 */
utc_pointtime((utc_t *)0, /* Out: Earliest possible current time */
              (utc_t *)0, /* Out: Midpoint of current time       */
              &now,       /* Out: Latest possible current time   */
              (utc_t *)0);/* In:  Use current time               */

/*
 *   Add 5 seconds to get future timestamp...
 */
utc_addtime(&future,     /* Out: Future binary timestamp        */
            &now,        /* In:  Latest possible time now       */
            &fivesec);   /* In:  5 secs                         */
```

Related Functions

utc_subtime

utc_anytime

utc_anytime — Converts a binary timestamp to a tm structure, using the time differential factor (TDF) information contained in the timestamp to determine the TDF returned with the tm structure.

Format

```
#include <utc.h>
int utc_anytime( timetm, *tns, *inacctm, *ins, *tdf, *utc)

struct tm timetm ;
long *tns;
struct tm *inacctm;
long *ins;
long *tdf;
const utc_t *utc;
```

Parameter

Input

utc

Binary timestamp.

Output

timetm

Time component of the binary timestamp expressed in the timestamp's local time.

tns

Nanoseconds since time component of the binary timestamp.

inacctm

Seconds of inaccuracy component of the binary timestamp. If the inaccuracy is finite, then tm_mday returns a value of --1 and tm_mon and tm_year return values of 0. The field tm_yday contains the inaccuracy in days. If the inaccuracy is infinite, all tm structure fields return values of --1.

ins

Nanoseconds of inaccuracy component of the binary timestamp.

tdf

TDF component of the binary timestamp in units of seconds east or west of GMT.

Description

The Any Time routine converts a binary timestamp to a tm structure. The TDF information contained in the timestamp is returned with the time and inaccuracy components; the TDF component determines the offset from GMT and the local time value of the tm structure. Additional returns include nanoseconds since Time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example converts a timestamp, using the TDF information in the timestamp, then prints the result.

```
utc_t               evnt;
struct tm           tmevnt;
timespec_t          tevnt, ievnt;
char                tznam[80];

/*
 *   Assume evnt contains the timestamp to convert...
 *
 *   Get time as a tm structure, using the time zone information in
 *   the timestamp...
 */
utc_anytime(&tmevnt,         /* Out: tm struct of time of evnt  */
            (long *)0,       /* Out: nanosec of time of evnt    */
            (struct tm *)0,  /* Out: tm struct of inacc of evnt */
            (long *)0,       /* Out: nanosec of inacc of evnt   */
            (int *)0,        /* Out: tdf of evnt                */
            &evnt);          /* In:  binary timestamp of evnt   */

/*
 *   Get the time and inaccuracy as timespec structures...
 */
utc_bintime(&tevnt,          /* Out: timespec of time of evnt   */
            &ievnt,          /* Out: timespec of inacc of evnt  */
            (int *)0,        /* Out: tdf of evnt                */
            &evnt);          /* In:  Binary timestamp of evnt   */

/*
 *   Construct the time zone name from time zone information in the
 *   timestamp...
 */
utc_anyzone(tznam,           /* Out: Time zone name             */
            80,              /* In:  Size of time zone name     */
            (long *)0,       /* Out: tdf of event               */
            (long *)0,       /* Out: Daylight saving flag       */
            &evnt);          /* In:  Binary timestamp of evnt   */

/*
 *   Print timestamp in the format:
 *
 *           1991-03-05-21:27:50.023I0.140 (GMT-5:00)
 *           1992-04-02-12:37:24.003Iinf (GMT+7:00)
 *
 */

printf("%d-%02d-%02d-%02d:%02d:%02d.%03d",
        tmevnt.tm_year+1900, tmevnt.tm_mon+1, tmevnt.tm_mday,
```

```
        tmevnt.tm_hour, tmevnt.tm_min, tmevnt.tm_sec,
        (tevnt.tv_nsec/1000000));

if ((long)ievnt.tv_sec == -1)
    printf("Iinf");
else
    printf("I%d.%03d", ievnt.tv_sec, (ievnt.tv_nsec/1000000));

printf(" (%s)\n", tznam);
```

Related Functions

utc_mkanytime, *utc_anyzone*, *utc_gettime*, *utc_getusertime*, *utc_gmtime*, *utc_localtime*

utc_anyzone

utc_anyzone — Gets the time zone label and offset from GMT, using the TDF contained in the input utc.

Format

```
#include <utc.h>
int utc_anyzone( tzname, tzlen, *tdf, isdst, *utc)

char tzname ;
size_t tzlen ;
long *tdf;
int *isdst;
const utc_t *utc;
```

Parameter

Input

tzlen

Length of the tzname buffer.

utc

Binary time.

Output

tzname

Character string that is long enough to hold the time zone label.

tdf

Longword with differential in seconds east or west of GMT.

isdst

Integer with a value of --1, indicating that no information is supplied as to whether it is standard time or daylight saving time. A value of --1 is always returned.

Description

The **Any Zone** routine gets the time zone label and offset from GMT, using the TDF contained in the input *utc*. The label returned is always of the form GMT + *n* or GMT - *n*, where *n* is the TDF expressed in hours:minutes. (The label associated with an arbitrary time zone is not known; only the offset is known.)

All of the output parameters are optional. No value is returned and no error occurs if the pointer is null.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or an insufficient buffer.

Example

See the sample program for the *utc_anytime* routine.

Related Functions

utc_anytime, *utc_gmtzone*, *utc_localzone*

utc_ascanytime

utc_ascanytime — Converts a binary timestamp to an ASCII string that represents an arbitrary time zone.

Format

```
#include <utc.h>
int utc_ascanytime( *cp, stringlen, *utc)

char *cp;
size_t stringlen ;
const utc_t *utc;
```

Parameter

Input

stringlen

The length of the cp buffer.

utc

Binary timestamp.

Output

cp

ASCII string that represents the time.

Description

The **ASCII Any Time** routine converts a binary timestamp to an ASCII string that expresses a time. The TDF component in the timestamp determines the local time used in the conversion.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example converts a time to an ASCII string that expresses the time in the time zone where the timestamp was generated.

```
utc_t      evnt;
char       localTime[UTC_MAX_STR_LEN];

/*
 *   Assuming that evnt contains the timestamp to convert, convert
 *   the time to ASCII in the following format:
 *
 *          1991-04-01-12:27:38.37-8:00I2.00
 */

utc_ascanytime(localtime,         /* Out: Converted time      */
               UTC_MAX_STR_LEN,   /* In:  Length of string    */
               &evnt);            /* In:  Time to convert     */
```

Related Functions

utc_ascgmtime, *utc_asclocaltime*

utc_ascgmtime

utc_ascgmtime — Converts a binary timestamp to an ASCII string that expresses a GMT time.

Format

```
#include <utc.h>
int utc_ascgmtime( *cp, stringlen, *utc)

char *cp;
size_t stringlen ;
const utc_t *utc;
```

Parameter

Input

stringlen

Length of the cp buffer.

utc

Binary timestamp.

Output

cp

ASCII string that represents the time.

Description

The **ASCII GMT Time** routine converts a binary timestamp to an ASCII string that expresses a time in GMT.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example converts the current time to GMT format.

```
char   gmTime[UTC_MAX_STR_LEN];

/*
 *   Convert the current time to ASCII in the following format:
 *
 *           1991-04-01-12:27:38.37I2.00
 */

utc_ascgmtime(gmTime,                /* Out: Converted time     */
              UTC_MAX_STR_LEN,       /* In:  Length of string   */
              (utc_t*) NULL);        /* In:  Time to convert    */
                                     /* Default is current time */
```

Related Functions

utc_ascanytime, *utc_asclocaltime*

utc_asclocaltime

utc_asclocaltime — Converts a binary timestamp to an ASCII string that represents a local time.

Format

```
#include <utc.h>
int utc_asclocaltime( *cp, stringlen, *utc)

char *cp;
size_t stringlen ;
const utc_t *utc;
```

Parameter

Input

stringlen

Length of the cp buffer.

utc

Binary timestamp.

Output

cp

ASCII string that represents the time.

Description

The **ASCII Local Time** routine converts a binary timestamp to an ASCII string that expresses local time.

OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys$timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example converts the current time to local time.

```
char   localTime[UTC_MAX_STR_LEN];

/*
 *   Convert the current time...
 */

utc_asclocaltime(localTime,      /* Out: Converted time          */
                 UTC_MAX_STR_LEN, /* In:  Length of string       */
                 (utc_t*) NULL);  /* In:  Time to convert        */
                                  /*      Default is current time */
```

Related Functions

utc_ascanytime, *utc_ascgmtime*

utc_ascreltime

utc_ascreltime — Converts a relative binary timestamp to an ASCII string that represents the time.

Format

```
#include <utc.h>
int utc_ascreltime( *cp, stringlen, *utc)

char *cp;
const size_t stringlen ;
const utc_t *utc;
```

Parameter

Input

utc

Relative binary timestamp.

stringlen

Length of the cp buffer.

Output

cp

ASCII string that represents the time.

Description

The **ASCII Relative Time** routine converts a relative binary timestamp to an ASCII string that represents the time.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

See the sample program for the *utc_abstime* routine.

Related Functions

utc_mkascreltime

utc_binreltime

utc_binreltime — Converts a relative binary timestamp to two timespec structures that express relative time and inaccuracy.

Format

```
#include <utc.h>
int utc_binreltime( *timesp, *inaccsp, *utc)
```

```
reltimespec_t *timesp;
timespec_t *inaccsp;
const utc_t *utc;
```

Parameter

Input

utc

Relative binary timestamp.

Output

timesp

Time component of the relative binary timestamp, in the form of seconds and nanoseconds since the base time (1970-01-01:00:00:00.0 + 00:00I0).

inaccsp

Inaccuracy component of the relative binary timestamp, in the form of seconds and nanoseconds.

Description

The **Binary Relative Time** routine converts a relative binary timestamp to two *timespec* structures that express relative time and inaccuracy. These *timespec* structures describe a time interval.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example measures the duration of a process, then prints the resulting relative time and inaccuracy.

```
utc_t               before, duration;
reltimespec_t       tduration;
timespec_t          iduration;

/*
 *   Get the time before the start of the operation...
 */

utc_gettime(&before);      /* Out: Before binary timestamp    */

/*
 * ...Later...
 *
 * Subtract, getting the duration as a relative time.
 *
 *    NOTE: The NULL argument is used to obtain the current time.
 */
```

```
utc_subtime(&duration,     /* Out: Duration rel bin timestamp  */
            (utc_t *)0,    /* In:  After binary timestamp      */
            &before);      /* In:  Before binary timestamp     */

/*
 *    Convert the relative times to timespec structures...
 */

utc_binreltime(&tduration, /* Out: Duration time timespec      */
               &iduration, /* Out: Duration inacc timespec     */
               &duration); /* In:  Duration rel bin timestamp  */

/*
 *    Print the duration...
 */

printf("%d.%04d", tduration.tv_sec, (tduration.tv_nsec/10000));

if ((long)iduration.tv_sec == -1)
    printf("Iinf\n");
else
printf("I%d.%04d\n", iduration.tv_sec, (iduration.tv_nsec/100000));
```

Related Functions

utc_mkbinreltime

utc_bintime

utc_bintime — Converts a binary timestamp to a timespec structure.

Format

```
#include <utc.h>
int utc_bintime( *timesp, *inaccsp, *tdf, *utc)

timespec_t *timesp;
timespec_t *inaccsp;
long *tdf;
const utc_t *utc;
```

Parameter

Input

utc

Binary timestamp.

Output

timesp

Time component of the binary timestamp, in the form of seconds and nanoseconds since the base time.

inaccsp

Inaccuracy component of the binary timestamp, in the form of seconds and nanoseconds.

tdf

TDF component of the binary timestamp in the form of signed number of seconds east or west of GMT.

Description

The **Binary Time** routine converts a binary timestamp to a *timespec* structure. The TDF information contained in the timestamp is returned.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_anytime* routine.

Related Functions

utc_binreltime, *utc_mkbintime*

utc_boundtime

utc_boundtime — Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event.

Format

```
#include <utc.h>
int utc_boundtime( *result, *utc1, *utc2)

utc_t *result;
const utc_t *utc1;
const utc_t *utc2;
```

Parameter

Input

utc1

Before binary timestamp or relative binary timestamp.

utc2

After binary timestamp or relative binary timestamp.

Output

result

Spanning timestamp.

Description

Given two UTC times, the **Bound Time** routine returns a single UTC time whose inaccuracy bounds the two input times. This is useful for timestamping events; the routine gets the *utc* values before and after the event, then calls *utc_boundtime* to build a timestamp that includes the event.

The TDF in the output UTC value is copied from the *utc2* input. If one or both input values have infinite inaccuracies, the returned time value also has an infinite inaccuracy and is the average of the two input values.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid parameter order.

Example

The following example records the time of an event and constructs a single timestamp, which includes the time of the event. Note that the *utc_getusertime* routine is called so the time zone information that is included in the timestamp references the user's environment rather than the system's default time zone.

OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys$timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

```
utc_t               before, after, evnt;

/*
 *   Get the time before the event...
 */

utc_getusertime(&before);  /* Out: Before binary timestamp     */

/*
 *   Get the time after the event...
 */

utc_getusertime(&after);   /* Out: After binary timestamp      */

/*
 *   Construct a single timestamp that describes the time of the
 *   event...
 */

utc_boundtime(&evnt,   /* Out: Timestamp that bounds event */
              &before, /* In:  Before binary timestamp     */
              &after); /* In:  After binary timestamp      */
```

Related Functions

utc_gettime, *utc_pointtime*, *utc_spantime*

utc_cmpintervaltime

utc_cmpintervaltime — Compares two binary timestamps or two relative binary timestamps.

Format

```
#include <utc.h>
int utc_cmpintervaltime( *relation, *utc1, *utc2)

enum utc_cmptype *relation;
const utc_t *utc1;
const utc_t *utc2;
```

Parameter

Input

utc1

Binary timestamp or relative binary timestamp.

utc2

Binary timestamp or relative binary timestamp.

Output

relation

Receives the result of the comparison of utc1:utc2, where the result is an enumerated type with one of the following values:

- *utc_equalTo*
- *utc_lessThan*
- *utc_greaterThan*
- *utc_indeterminate*

Description

The **Compare Interval Time** routine compares two binary timestamps and returns a flag indicating that the first time is greater than, less than, equal to, or overlapping with the second time. Two times overlap if the intervals (time - inaccuracy, time + inaccuracy) of the two times intersect.

The input binary timestamps express two absolute or two relative times. Do not compare relative binary timestamps and binary timestamps. If you do, no meaningful results and no errors are returned.

This routine does a temporal ordering of the time intervals.

```
utc1 is utc_lessThan utc2 iff
```

```
        utc1.time + utc1.inacc < utc2.time - utc2.inacc

utc1 is utc_greaterThan utc2 iff
        utc1.time - utc1.inacc > utc2.time + utc2.inacc

utc1 utc_equalTo utc2 iff
        utc1.time == utc2.time and
        utc1.inacc == 0 and
        utc2.inacc == 0

utc1 is utc_indeterminate with respect to utc2 if the intervals
overlap.
```

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument.

Example

The following example checks to see if the current time is definitely after 1:00 P.M. today GMT.

```
struct tm           tmtime, tmzero;
enum utc_cmptype    relation;
utc_t               testtime;

/*
 *   Zero the tm structure for inaccuracy...
 */

memset(&tmzero, 0, sizeof(tmzero));

/*
 *   Get the current time, mapped to a tm structure...
 *
 *        NOTE: The NULL argument is used to get the current time.
 */

utc_gmtime(&tmtime,      /* Out: Current GMT time in tm struct   */
          (long *)0,     /* Out: Nanoseconds of time             */
          (struct tm *)0, /* Out: Current inaccuracy in tm struct */
          (long *)0,     /* Out: Nanoseconds of inaccuracy       */
          (utc_t *)0);   /* In:  Current timestamp               */

/*
 *   Construct a tm structure that corresponds to 1:00 PM...
 */

tmtime.tm_hour = 13;
tmtime.tm_min = 0;
tmtime.tm_sec = 0;

/*
 *   Convert to a binary timestamp...
 */

utc_mkgmtime(&testtime,  /* Out: Binary timestamp of 1:00 PM  */
```

```
            &tmtime,    /* In:  1:00 PM in tm struct         */
            0,          /* In:  Nanoseconds of time          */
            &tmzero,    /* In:  Zero inaccuracy in tm struct */
            0);         /* In:  Nanoseconds of inaccuracy    */

/*
 *   Compare to the current time, noting the use of the
 *   NULL argument...
 */

utc_cmpintervaltime(&relation,    /* Out: Comparison relation */
                    (utc_t *)0,   /* In:  Current timestamp   */
                    &testtime);   /* In:  1:00 PM timestamp   */

/*
 *   If it is not later - wait, print a message, etc.
 */

if (relation != utc_greaterThan) {

/*
 *     Note: It could be earlier than 1:00 PM or it could be
 *           indeterminate. If indeterminate, for some applications
 *           it might be worth waiting.
 */
}
```

Related Functions

utc_cmpmidtime

utc_cmpmidtime

utc_cmpmidtime — Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies.

Format

```
#include <utc.h>
int utc_cmpmidtime( *relation, *utc1, *utc2)

enum utc_cmptype *relation;
const utc_t *utc1;
const utc_t *utc2;
```

Parameter

Input

utc1

Binary timestamp or relative binary timestamp.

utc2

Binary timestamp or relative binary timestamp.

- *utc_equalTo*
- *utc_lessThan*
- *utc_greaterThan*

Output

relation

Result of the comparison of utc1:utc2, where the result is an enumerated type with one of the following values:

- *utc_equalTo*
- *utc_lessThan*
- *utc_greaterThan*

Description

The **Compare Midpoint Times** routine compares two binary timestamps and returns a flag indicating that the first timestamp is greater than, less than, or equal to the second timestamp. Inaccuracy information is ignored for this comparison; the input values are, therefore, equivalent to the midpoints of the time intervals described by the input binary timestamps.

The input binary timestamps express two absolute or two relative times. Do not compare relative binary timestamps and binary timestamps. If you do, no meaningful results and no errors are returned.

The following routine does a lexical ordering on the time interval midpoints.

```
utc1 is utc_lessThan utc2 iff
        utc1.time < utc2.time

utc1 is utc_greaterThan utc2 iff
        utc1.time > utc2.time

utc1 is utc_equalTo utc2 iff
        utc1.time == utc2.time
```

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument.

Example

The following example checks if the current time (ignoring inaccuracies) is after 1:00 P.M. today local time.

```
struct tm           tmtime, tmzero;
enum utc_cmptype    relation;
utc_t               testtime;

/*
```

```
 *   Zero the tm structure for inaccuracy...
 */

memset(&tmzero, 0, sizeof(tmzero));

/*
 *   Get the current time, mapped to a tm structure...
 *
 *        NOTE:  The NULL argument is used to get the current time.
 */

utc_localtime(&tmtime,   /* Out: Current local time in tm struct */
         (long *)0,      /* Out: Nanoseconds of time             */
         (struct tm *)0, /* Out: Current inacc in tm struct      */
         (long *)0,      /* Out: Nanoseconds of inaccuracy       */
         (utc_t *)0);    /* In:  Current timestamp               */

/*
 *   Construct a tm structure that corresponds to 1:00 P.M....
 */

tmtime.tm_hour = 13;
tmtime.tm_min = 0;
tmtime.tm_sec = 0;

/*
 *   Convert to a binary timestamp...
 */

utc_mklocaltime(&testtime, /* Out: Binary timestamp of 1:00 P.M. */
                &tmtime,   /* In:  1:00 P.M. in tm struct        */
                0,         /* In:  Nanoseconds of time           */
                &tmzero,   /* In:  Zero inaccuracy in tm struct  */
                0);        /* In:  Nanoseconds of inaccuracy     */

/*
 *   Compare to the current time, noting the use of the
 *   NULL argument...
 */

utc_cmpmidtime(&relation,    /* Out: Comparison relation         */
               (utc_t *)0,   /* In:  Current timestamp           */
               &testtime);   /* In:  1:00 P.M. timestamp         */

/*
 *   If the time is not later - wait, print a message, etc.
 */

if (relation != utc_greaterThan) {

/*          It is not later then 1:00 P.M. local time. Note that
 *          this depends on the setting of the user's environment.
 */
}
```

Related Functions

utc_cmpintervaltime

utc_gettime

utc_gettime — Returns the current system time and inaccuracy as a binary timestamp.

Format

```
#include <utc.h>
int utc_gettime( *utc)

utc_t *utc;
```

Parameter

Input

None.

Output

utc

System time as a binary timestamp.

Description

The **Get Time** routine returns the current system time and inaccuracy in a binary timestamp. The routine takes the TDF from the operating system's kernel; the TDF is specified in a system-dependent manner.

Returns

0	Indicates that the routine executed successfully.
--1	Generic error that indicates the time service cannot be accessed.

Example

See the sample program for the *utc_binreltime* routine.

utc_getusertime

utc_getusertime — Returns the time and process-specific TDF, rather than the system-specific TDF.

Format

```
#include <utc.h>
int utc_getusertime( *utc)

utc_t *utc;
```

Parameter

Input

None.

Output

utc

System time as a binary timestamp.

Description

The **Get User Time** routine returns the system time and inaccuracy in a binary timestamp. The routine takes the TDF from the user's environment, which determines the time zone rule. OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys$timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

Returns

0	Indicates that the routine executed successfully.
--1	Generic error that indicates the time service cannot be accessed.

Example

See the sample program for the *utc_boundtime* routine.

Related Functions

utc_gettime

utc_gmtime

utc_gmtime — Converts a binary timestamp to a tm structure that expresses GMT or the equivalent UTC.

Format

```
#include <utc.h>
int utc_gmtime( *timetm, *tns, *inacctm, *ins, *utc)

struct tm *timetm;
long *tns;
struct tm *inacctm;
long *ins;
const utc_t *utc;
```

Parameter

Input

utc

Binary timestamp to be converted to tm structure components.

Output

timetm

Time component of the binary timestamp.

tns

Nanoseconds since time component of the binary timestamp.

inacctm

Seconds of inaccuracy component of the binary timestamp. If the inaccuracy is finite, then tm_mday returns a value of --1 and tm_mon and tm_year return values of zero. The field tm_yday contains the inaccuracy in days. If the inaccuracy is infinite, all tm structure fields return values of --1.

ins

Nanoseconds of inaccuracy component of the binary timestamp. If the inaccuracy is infinite, ins returns a value of --1.

Description

The **Greenwich Mean Time** (GMT) routine converts a binary timestamp to a *tm* structure that expresses GMT (or the equivalent UTC). Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_cmpintervaltime* routine.

Related Functions

utc_anytime, *utc_gmtzone*, *utc_localtime*, *utc_mkgmtime*

utc_gmtzone

utc_gmtzone — Gets the time zone label for GMT.

Format

```
#include <utc.h>
int utc_gmtzone( *tzname, tzlen, *tdf, *isdst, *utc)

char *tzname;
size_t tzlen ;
long *tdf;
int *isdst;
const utc_t *utc;
```

Parameter

Input

tzlen

Length of buffer tzname.

utc

Binary timestamp. This parameter is ignored.

Output

tzname

Character string long enough to hold the time zone label.

tdf

Longword with differential in seconds east or west of GMT. A value of zero is always returned.

isdst

Integer with a value of zero, indicating that daylight saving time is not in effect. A value of zero is always returned.

Description

The **Greenwich Mean Time Zone** routine gets the time zone label and zero offset from GMT. Outputs are always *tdf* = 0 and *tzname* = GMT. This routine exists for symmetry with the **Any Zone** (`utc_anyzone`) and the **Local Zone** (`utc_localzone`) routines.

All of the output parameters are optional. No value is returned and no error occurs if the *tzname* pointer is NULL.

Returns

0	Indicates that the routine executed successfully (always returned).

Example

The following example prints out the current time in both local time and GMT time.

```
utc_t       now;
struct tm   tmlocal, tmgmt;
long        tzoffset;
int         tzdaylight;
char        tzlocal[80], tzgmt[80];

/*
 *   Get the current time once, so both conversions use the same
 *   time...
 */

utc_gettime(&now);

/*
 *   Convert to local time, using the process TZ environment
 *   variable...
 */
```

```
utc_localtime(&tmlocal,       /* Out: Local time tm structure  */
              (long *)0,      /* Out: Nanosec of time          */
              (struct tm *)0, /* Out: Inaccuracy tm structure  */
              (long *)0,      /* Out: Nanosec of inaccuracy    */
              &now);          /* In:  Current binary timestamp */

/*
 *   Get the local time zone name, offset from GMT, and current
 *   daylight savings flag...
 */

utc_localzone(tzlocal,     /* Out: Local time zone name          */
              80,          /* In:  Length of loc time zone name  */
              &tzoffset,   /* Out: Loc time zone offset in secs  */
              &tzdaylight, /* Out: Local time zone daylight flag */
              &now);       /* In:  Current binary timestamp      */

/*
 *   Convert to GMT...
 */

utc_gmtime(&tmgmt,         /* Out: GMT tm structure              */
           (long *)0,      /* Out: Nanoseconds of time           */
           (struct tm *)0, /* Out: Inaccuracy tm structure       */
           (long *)0,      /* Out: Nanoseconds of inaccuracy     */
           &now);          /* In:  Current binary timestamp      */

/*
 *   Get the GMT time zone name...
 */

utc_gmtzone(tzgmt,         /* Out: GMT time zone name            */
            80,            /* In:  Size of GMT time zone name    */
            (long *)0,     /* Out: GMT time zone offset in secs  */
            (int *)0,      /* Out: GMT time zone daylight flag   */
            &now);         /* In:  Current binary timestamp      */

/*
 *   Print out times and time zone information in the following
 *   format:
 *
 *          12:00:37 (EDT) = 16:00:37 (GMT)
 *          EDT is -240 minutes ahead of Greenwich Mean Time.
 *          Daylight savings time is in effect.
 */

printf("%d:%02d:%02d (%s) = %d:%02d:%02d (%s)\n",
       tmlocal.tm_hour, tmlocal.tm_min, tmlocal.tm_sec, tzlocal,
       tmgmt.tm_hour, tmgmt.tm_min, tmgmt.tm_sec, tzgmt);
printf("%s is %d minutes ahead of Greenwich Mean Time\n",
       tzlocal, tzoffset/60);
if (tzdaylight != 0)
    printf("Daylight savings time is in effect\n");
```

Related Functions

utc_anyzone, *utc_gmtime*, *utc_localzone*

utc_localtime

utc_localtime — Converts a binary timestamp to a tm structure that expresses local time.

Format

```
#include <utc.h>
int utc_localtime( *timetm, *tns, *inacctm, *ins, *utc)

struct tm *timetm;
long *tns;
struct tm *inacctm;
long *ins;
const utc_t *utc;
```

Parameter

Input

utc

Binary timestamp.

Output

timetm

Time component of the binary timestamp, expressing local time.

tns

Nanoseconds since time component of the binary timestamp.

inacctm

Seconds of inaccuracy component of the binary timestamp. If the inaccuracy is finite, then *tm_mday* returns a value of --1 and *tm_mon* and *tm_year* return values of zero. The field *tm_yday* contains the inaccuracy in days. If the inaccuracy is infinite, all *tm* structure fields return values of --1.

ins

Nanoseconds of inaccuracy component of the binary timestamp. If the inaccuracy is infinite, ins returns a value of --1.

Description

The **Local Time** routine converts a binary timestamp to a *tm* structure that expresses local time.

OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys $timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_gmtzone* routine.

Related Functions

utc_anytime, *utc_gmtime*, *utc_localzone*, *utc_mklocaltime*

utc_localzone

utc_localzone — Gets the local time zone label and offset from GMT, given utc .

Format

```
#include <utc.h>
int utc_localzone( *tzname, tzlen, *tdf, *isdst, *utc)

char *tzname;
size_t tzlen ;
long *tdf;
int *isdst;
const utc_t *utc;
#include <utc.h>

int utc_localzone( *tzname, tzlen, *tdf, *isdst, *utc)
```

Parameter

Input

tzlen

Length of the tzname buffer.

utc

Binary timestamp.

Output

tzname

Character string long enough to hold the time zone label.

tdf

Longword with differential in seconds east or west of GMT.

isdst

Integer with a value of zero if standard time is in effect or a value of 1 if daylight savings time is in effect.

Description

The **Local Zone** routine gets the local time zone label and offset from GMT, given *utc*.

OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys$timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

All of the output parameters are optional. No value is returned and no error occurs if the pointer is null.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or an insufficient buffer.

Example

See the sample program for the *utc_gmtzone* routine.

Related Functions

utc_anyzone, *utc_gmtzone*, *utc_localtime*

utc_mkanytime

utc_mkanytime — Converts a tm structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.

Format

```
#include <utc.h>
int utc_mkanytime( *utc, *timetm, tns, *inacctm, ins, tdf)

utc_t *utc;
const struct tm *timetm;
long tns ;
const struct tm *inacctm;
long ins ;
long tdf ;
```

Parameter

Input

timetm

A *tm* structure that expresses the local time; *tm_wday* and *tm_yday* are ignored on input.

tns

Nanoseconds since time component.

inacctm

A *tm* structure that expresses days, hours, minutes, and seconds of inaccuracy. If *tm_yday* is negative, the inaccuracy is considered to be infinite; *tm_mday*, *tm_mon*, *tm_wday*, *tm_isdst*, *tm_gmtoff*, and *tm_zone* are ignored on input.

ins

Nanoseconds of inaccuracy component.

tdf

Time differential factor to use in conversion.

Output

utc

Resulting binary timestamp.

Description

The **Make Any Time** routine converts a *tm* structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp. Required inputs include nanoseconds since time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example converts a string ISO format time in an arbitrary time zone to a binary timestamp. This may be part of an input timestamp routine, although a real implementation will include range checking.

```
utc_t       utc;
struct tm   tmtime, tminacc;
float       tsec, isec;
double      tmp;
long        tnsec, insec;
int         i, offset, tzhour, tzmin, year, mon;
char        *string;

/*  Try to convert the string...                                    */

if(sscanf(string, "%d-%d-%d-%d:%d:%e+%d:%dI%e",
          &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
          &tmtime.tm_min, &tsec, &tzhour, &tzmin, &isec) != 9) {

/*  Try again with a negative TDF...                                */
```

```
if (sscanf(string, "%d-%d-%d-%d:%d:%e-%d:%dI%e",
           &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
           &tmtime.tm_min, &tsec, &tzhour, &tzmin, &isec) != 9) {

/*  ERROR                                                       */

        exit(1);
    }

/*  TDF is negative                                             */

    tzhour = -tzhour;
    tzmin = -tzmin;

}

/*  Fill in the fields...                                       */

tmtime.tm_year = year - 1900;
tmtime.tm_mon = --mon;
tmtime.tm_sec = tsec;
tnsec = (modf(tsec, &tmp)*1.0E9);
offset = tzhour*3600 + tzmin*60;
tminacc.tm_sec = isec;
insec = (modf(isec, &tmp)*1.0E9);

/* Convert to a binary timestamp...                             */

utc_mkanytime(&utc,     /* Out: Resultant binary timestamp      */
              &tmtime,  /* In:  tm struct that represents input */
              tnsec,    /* In:  Nanoseconds from input          */
              &tminacc, /* In:  tm struct that represents inacc */
              insec,    /* In:  Nanoseconds from input          */
              offset);  /* In:  TDF from input                  */
```

Related Functions

utc_anytime, *utc_anyzone*

utc_mkascreltime

utc_mkascreltime — Converts a null-terminated character string that represents a relative timestamp to a binary timestamp.

Format

```
#include <utc.h>
int utc_mkascreltime( *utc, *string)

utc_t *utc;
char *string;
```

Parameter

Input

string

A null-terminated string that expresses a relative timestamp in its ISO format.

Output

utc

Resulting binary timestamp.

Description

The **Make ASCII Relative Time** routine converts a null-terminated string, which represents a relative timestamp, to a binary timestamp.

The ASCII string must be null-terminated.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example converts an ASCII relative time string to its binary equivalent.

```
utc_t      utc;
char       str[UTC_MAX_STR_LEN];

/*
 * Relative time of 333 days, 12 hours, 1 minute, 37.223 seconds
 * Inaccuracy of 50.22 sec. in the format: -333-12:01:37.223I50.22
 */

(void)strcpy((void *)str,
             "-333-12:01:37.223I50.22");

utc_mkascreltime(&utc,   /* Out: Binary utc               */
                 str);   /* In:  String                   */
```

Related Functions

utc_ascreltime

utc_mkasctime

utc_mkasctime — Converts a null-terminated character string that represents an absolute time to a binary timestamp.

Format

```
#include <utc.h>
int utc_mkasctime( *utc, *string)
```

```
utc_t *utc;
char *string;
```

Parameter

Input

string

A null-terminated string that expresses an absolute time.

Output

utc

Resulting binary timestamp.

Description

The **Make ASCII Time** routine converts a null-terminated string that represents an absolute time to a binary timestamp.

The ASCII string must be null-terminated.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time parameter or invalid results.

Example

The following example converts an ASCII time string to its binary equivalent.

```
utc_t     utc;
char      str[UTC_MAX_STR_LEN];

/*
 *   July 4, 1776, 12:01:37.223 local time
 *   TDF of -5:00 hours
 *   Inaccuracy of 3600.32 seconds
 */

(void)strcpy((void *)str,
             "1776-07-04-12:01:37.223-5:00 I 3600.32");

utc_mkasctime(&utc,    /* Out: Binary utc           */
              str);    /* In:  String               */
```

Related Functions

utc_ascanytime, *utc_ascgmtime*, *utc_asclocaltime*

utc_mkbinreltime

utc_mkbinreltime — Converts a timespec structure expressing a relative time to a binary timestamp.

Format

```
#include <utc.h>
int utc_mkbinreltime( *utc, *timesp, *inaccsp)

utc_t *utc;
const reltimespec_t *timesp;
const timespec_t *inaccsp;
```

Parameter

Input

timesp

A *reltimespec* structure that expresses a relative time.

inaccsp

A *timespec* structure that expresses inaccuracy. If *tv_sec* is set to a value of --1, the inaccuracy is considered to be infinite.

Output

utc

Resulting relative binary timestamp.

Description

The **Make Binary Relative Time** routine converts a *timespec* structure that expresses relative time to a binary timestamp.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_addtime* routine.

Related Functions

utc_binreltime, *utc_mkbintime*

utc_mkbintime

utc_mkbintime — Converts a timespec structure to a binary timestamp.

Format

```
#include <utc.h>
```

```
int utc_mkbintime( *utc, *timesp, *inaccsp)

utc_t *utc;
const timespec_t *timesp;
const timespec_t *inaccsp;
long tdf ;
```

Parameter

Input

timesp

A *timespec* structure that expresses time since 1970-01-01:00:00:00.0+0:00I0.

inaccsp

A *timespec* structure that expresses inaccuracy. If *tv_sec* is set to a value of --1, the inaccuracy is considered to be infinite.

tdf

TDF component of the binary timestamp.

Output

utc

Resulting binary timestamp.

Description

The Make Binary Time routine converts a timespec structure time to a binary timestamp. The TDF input is used as the TDF of the binary timestamp.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example obtains the current time from *time()*, converts it to a binary timestamp with an inaccuracy of 5.2 seconds, and specifies GMT.

```
timespec_t   ttime, tinacc;
utc_t        utc;

/*
 *   Obtain the current time (without the inaccuracy)...
 */

ttime.tv_sec = time((time_t *)0);
ttime.tv_nsec = 0;
```

```
/*
 *   Specify the inaccuracy...
 */

tinacc.tv_sec = 5;
tinacc.tv_nsec = 200000000;

/*
 *   Convert to a binary timestamp...
 */

utc_mkbintime(&utc,       /* Out: Binary timestamp          */
              &ttime,     /* In:  Current time in timespec  */
              &tinacc,    /* In:  5.2 seconds in timespec   */
              0);         /* In:  TDF of GMT                */
```

Related Functions

utc_bintime, *utc_mkbinreltime*

utc_mkgmtime

utc_mkgmtime — Converts a tm structure that expresses GMT or UTC to a binary timestamp.

Format

```
#include <utc.h>
int utc_mkgmtime( *utc, *timetm, tns, *inacctm, ins)

utc_t *utc;
const struct tm *timetm;
long tns ;
const struct tm *inacctm;
long ins ;
```

Parameter

Input

timetm

A tm structure that expresses GMT. On input, tm_wday and tm_yday are ignored.

tns

Nanoseconds since time component.

inacctm

A tm structure that expresses days, hours, minutes, and seconds of inaccuracy. If tm_yday is negative, the inaccuracy is considered to be infinite. On input, tm_mday , tm_mon , tm_wday , tm_isdst , tm_gmtoff , and tm_zone are ignored.

ins

Nanoseconds of inaccuracy component.

Output

utc

Resulting binary timestamp.

Description

The **Make Greenwich Mean Time** routine converts a *tm* structure that expresses GMT or UTC to a binary timestamp. Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_cmpintervaltime* routine.

Related Functions

utc_gmtime

utc_mklocaltime

utc_mklocaltime — Converts a tm structure that expresses local time to a binary timestamp.

Format

```
#include <utc.h>
int utc_mklocaltime( *utc, *timetm, tns, *inacctm, ins)

utc_t *utc;
const struct tm *timetm;
long tns ;
const struct tm *inacctm;
long ins ;
```

Parameter

Input

timetm

A tm structure that expresses the local time. On input, tm_wday and tm_yday are ignored.

tns

Nanoseconds since time component.

inacctm

A tm structure that expresses days, hours, minutes, and seconds of inaccuracy. If tm_yday is negative, the inaccuracy is considered to be infinite. On input, tm_mday , tm_mon , tm_wday , tm_isdst , tm_gmtoff , and tm_zone are ignored.

ins

Nanoseconds of inaccuracy component.

Output

utc

Resulting binary timestamp.

Description

The **Make Local Time** routine converts a *tm* structure that expresses local time to a binary timestamp.

OpenVMS systems do not have a default time zone rule. You select a time zone by defining *sys$timezone_rule* during the *sys$manager:net$configure.com* procedure, or by explicitly defining *sys$timezone_rule*.

Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_cmpmidtime* routine.

Related Functions

utc_localtime

utc_mkreltime

utc_mkreltime — Converts a tm structure that expresses relative time to a relative binary timestamp.

Format

```
#include <utc.h>
int utc_mkreltime( *utc, *timetm, tns, *inacctm, ins)

utc_t *utc;
const struct tm *timetm;
long tns ;
```

```
const struct tm *inacctm;
long ins ;
```

Parameter

Input

timetm

A tm structure that expresses a relative time. On input, tm_wday and tm_yday are ignored.

tns

Nanoseconds since time component.

inacctm

A tm structure that expresses seconds of inaccuracy. If tm_yday is negative, the inaccuracy is considered to be infinite. On input, tm_mday , tm_mon , tm_year , tm_wday , tm_isdst , and tm_zone are ignored.

ins

Nanoseconds of inaccuracy component.

Output

utc

Resulting relative binary timestamp.

Description

The **Make Relative Time** routine converts a *tm* structure that expresses relative time to a relative binary timestamp. Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example converts a string relative time in the format (1991-04-01-12:12:12.12I12.12) to a binary timestamp. This may be part of an input relative timestamp routine, though a real implementation will include range checking.

```
utc_t       utc;
struct tm   tmtime, tminacc;
float       tsec, isec;
double      tmp;
long        tnsec, insec;
```

```
int         i, tzhour, tzmin, year, mon;
char        *string;

/*
 *   Try to convert the string...
 */

if(sscanf(string, "%d-%d-%d-%d:%d:%eI%e",
          &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
          &tmtime.tm_min, &tsec, &isec) != 7) {

/*
 *   ERROR...
 */
    exit(1);

}

/*
 *   Fill in the fields...
 */

tmtime.tm_year = year - 1900;
tmtime.tm_mon = --mon;
tmtime.tm_sec = tsec;
tnsec = (modf(tsec, &tmp)*1.0E9);
tminacc.tm_sec = isec;
insec = (modf(isec, &tmp)*1.0E9);

/*
 * Convert to a binary timestamp...
 */

utc_mkreltime(&utc,     /* Out: Resultant binary timestamp      */
              &tmtime,  /* In:  tm struct that represents input */
              tnsec,    /* In:  Nanoseconds from input          */
              &tminacc, /* In:  tm struct that represents inacc */
              insec);   /* In:  Nanoseconds from input          */
```

Related Functions

utc_reltime

utc_mkvmsanytime

utc_mkvmsanytime — Converts a binary OpenVMS format time and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.

Format

```
#include <utc.h>
int utc_mkvmsanytime( *utc, *timadr, tdf)

utc_t *utc;
const long *timadr;
const long tdf ;
```

Parameter

Input

***timadr**

Binary OpenVMS format time.

tdf

Time differential factor to use in conversion.

Output

***utc**

Binary timestamp.

Description

The **Make VMS Any Time** routine converts a binary time in the OpenVMS (Smithsonian) format and an arbitrary TDF to a UTC-based binary timestamp. Because the input and output values are based on different time standards, any input representing a value after A.D. 30,000 returns an error.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example shows how to convert between OpenVMS format binary timestamps and UTC binary timestamps, while specifying the TDF for each. The TDF value determines the offset from GMT and the local time.

```
/*****
 start example mkvmsanytime,vmsanytime
 *****/
#include <utc.h>

main()
{
struct utc utcTime;
int vmsTime[2];

SYS$GETTIM(vmsTime);    /* read the current time */

/*
 * convert the VMS local time to a UTC, applying a TDF of
 * -300 minutes (the timezone is -5 hours from GMT)
 */
if (utc_mkvmsanytime(&utcTime,vmsTime,-300))
    exit(1);

/*
```

```
 * convert UTC back to VMS local time.  A TDF of -300 is applied
 * to the UTC, since utcTime was constructed with that same value.
 * This effectively gives us the same VMS time value we started
 * with.
 */
if (utc_vmsanytime(vmsTime,&utcTime))
    exit(2);
}
/****
  end example
 ****/
```

Related Functions

utc_vmsanytime

utc_mkvmsgmtime

utc_mkvmsgmtime — Converts a binary OpenVMS format time expressing GMT (or the equivalent UTC) into a binary timestamp.

Format

```
#include <utc.h>
int utc_mkvmsgmtime( *utc, *timadr)

utc_t *utc;
const long *timadr;
```

Parameter

Input

***timadr**

Binary OpenVMS format time representing GMT or the UTC equivalent.

Output

***utc**

Binary timestamp.

Description

The **Make VMS Greenwich Mean Time** routine converts an OpenVMS format binary time representing GMT to a binary timestamp with the equivalent UTC value. Since the input and output values are based on different time standards, any input representing a value after A.D. 30,000 returns an error.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the **vmsgmtime** routine.

Related Functions

utc_vmsgmtime

utc_mkvmslocaltime

utc_mkvmslocaltime — Converts a local binary OpenVMS format time to a binary timestamp, using the host system's time differential factor.

Format

```
#include <utc.h>
int utc_mkvmslocaltime( *utc, *timadr)

const long *timadr;
utc_t *utc;
```

Parameter

Input

***timadr**

Binary OpenVMS format time expressing local time.

Output

***utc**

Binary timestamp expressing the system's local time.

Description

The **Make VMS Local Time** routine converts a binary OpenVMS format time, representing the local time of the host system, to a binary timestamp. The system's local time value is defined by the time zone rule in *sys$timezone_rule*, which is created by the system configuration process *sys$manager:net$configure.com*.

If the routine call is made during a seasonal time zone change when the local time is indeterminate, an error is returned. For example, if the time zone change occurs at the current local time of 2:00 A.M. to a new local time of 1:00 A.M., and the routine is called between 1:00 A.M. and 2:00 A.M., it cannot be determined which TDF applies.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument, invalid results, or invalid routine call during a time zone change.

Example

The following example shows how to retrieve the current local time of the system in the binary OpenVMS format, convert the OpenVMS format time to a UTC-based binary timestamp (using the system's TDF), and print an ASCII representation of the binary timestamp.

```
/*********
  start example mkvmslocaltime
 *********/
#include <utc.h>

main()
{
char outstring[UTC_MAX_STR_LEN];
struct utc utcTime;
int vmsTime[2];

SYS$GETTIM(vmsTime);                        /* read current time      */

if (utc_mkvmslocaltime(&utcTime,vmsTime)) /* convert the local time */
    exit(1);                                /* vmsTime to UTC using   */
                                            /* the system tdf.        */

/* convert to ISO ascii*/
   utc_asclocaltime(outstring,UTC_MAX_STR_LEN,&utcTime);
/* format and print */
   printf("Current time=> %s\n",outstring);
}
/*****
 end example
 *****/
```

Related Functions

utc_vmslocaltime

utc_mulftime

utc_mulftime — Multiplies a relative binary timestamp by a floating-point value.

Format

```
#include <utc.h>
int utc_mulftime( *result, *utc1, factor)

utc_t *result;
const utc_t *utc1;
const double factor ;
```

Parameter

Input

utc1

Relative binary timestamp.

factor

Real scale factor (double-precision floating-point)

Output

result

Resulting relative binary timestamp.

Description

The **Multiply a Relative Time by a Real Factor** routine multiplies a relative binary timestamp by a floating-point value. Either or both may be negative; the resulting relative binary timestamp has the appropriate sign. The unsigned inaccuracy in the relative binary timestamp is also multiplied by the absolute value of the floating-point value.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example scales and prints a relative time.

```
utc_t       relutc, scaledutc;
struct tm   sacledreltm;
char        timstr[UTC_MAX_STR_LEN];

/*
 * Assume relutc contains the time to scale.
 * Scale it by a factor of 17...
 */

utc_multime(&scaledutc,          /* Out: Scaled rel time     */
            &relutc,             /* In:  Rel time to scale   */
            17L);                /* In:  Scale factor        */

utc_ascreltime(timstr,           /* Out: ASCII rel time      */
               UTC_MAX_STR_LEN,  /* In:  Length of input str */
               &scaledutc);      /* In:  Rel time to convert */

printf("%s\n",timstr);

/*
 *   Scale it by a factor of 17.65...
 */

utc_mulftime(&scaledutc,         /* Out: Scaled rel time     */
             &relutc,            /* In:  Rel time to scale   */
             17.65);             /* In:  Scale factor        */

utc_ascreltime(timstr,           /* Out: ASCII rel time      */
               UTC_MAX_STR_LEN,  /* In:  Input str length    */
               &scaledutc);      /* In:  Rel time to convert */
```

```
printf("%s\n",timstr);

/*
 *    Convert it to a tm structure and print it.
 */

utc_reltime(&scaledreltm,        /* Out: Scaled rel tm       */
            (long *)0,           /* Out: Scaled rel nano-sec */
            (struct tm *)0,      /* Out: Scaled rel inacc tm */
            (long *)0,           /* Out: Scd rel inacc nanos */
            &scaledutc);         /* In:  Rel time to convert */

printf("Approximately %d days, %d hours and %d minutes\n",
 scaledreltm.tm_yday, scaledreltm.tm_hour, scaledreltm.tm_min);
```

Related Functions

utc_multime

utc_multime

utc_multime — Multiplies a relative binary timestamp by an integer factor.

Format

```
#include <utc.h>
int utc_multime( *result, *utc1, factor)

utc_t *result;
const utc_t *utc1;
long factor ;
```

Parameter

Input

utc1

Relative binary timestamp.

factor

Integer scale factor.

Output

result

Resulting relative binary timestamp.

Description

The Multiply Relative Time by an Integer Factor routine multiplies a relative binary timestamp by an integer. Either or both may be negative; the resulting binary timestamp has the appropriate sign. The unsigned inaccuracy in the binary timestamp is also multiplied by the absolute value of the integer.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_mulftime* routine.

Related Functions

utc_mulftime

utc_pointtime

utc_pointtime — Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time.

Format

```
#include <utc.h>
int utc_pointtime( *utclp, *utcmp, *utchp, *utc)

utc_t *utclp;
utc_t *utcmp;
utc_t *utchp;
const utc_t *utc;
```

Parameter

Input

utc

Binary timestamp or relative binary timestamp.

Output

utclp

Lowest (earliest) possible time that the input binary timestamp or shortest possible relative time that the relative binary timestamp can represent.

utcmp

Midpoint of the input binary timestamp or the midpoint of the input relative binary timestamp.

utchp

Highest (latest) possible time that the input binary timestamp or the longest possible relative time that the relative binary timestamp can represent.

Description

The **Point Time** routine converts a binary timestamp to three binary timestamps that represent the earliest, latest, and most likely (midpoint) times. If the input is a relative binary time, the outputs represent relative binary times.

All outputs have zero inaccuracy. An error is returned if the input binary timestamp has an infinite inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument.

Example

See the sample program for the *utc_addtime* routine.

Related Functions

utc_boundtime, *utc_spantime*

utc_reltime

utc_reltime — Converts a relative binary timestamp to a tm structure.

Format

```
#include <utc.h>
int utc_reltime( *timetm, *tns, *inacctm, *ins, *utc)

struct tm *timetm;
long *tns;
struct tm *inacctm;
long *ins;
const utc_t *utc;
```

Parameter

Input

utc

Relative binary timestamp.

Output

timetm

Relative time component of the relative binary timestamp. The field tm_mday returns a value of --1 and the fields tm_year and tm_mon return values of zero. The field tm_yday contains the number of days of relative time.

tns

Nanoseconds since time component of the relative binary timestamp.

inacctm

Seconds of inaccuracy component of the relative binary timestamp. If the inaccuracy is finite, then tm_mday returns a value of --1 and tm_mon and tm_year return values of zero. The field tm_yday contains the inaccuracy in days. If the inaccuracy is infinite, all tm structure fields return values of --1.

ins

Nanoseconds of inaccuracy component of the relative binary timestamp.

Description

The **Relative Time** routine converts a relative binary timestamp to a *tm* structure. Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_mulftime* routine.

Related Functions

utc_mkreltime

utc_spantime

utc_spantime — Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps.

Format

```
#include <utc.h>
int utc_spantime( *result, *utc1, *utc2)

utc_t *result;
const utc_t *utc1;
const utc_t *utc2;
```

Parameter

Input

utc1

Binary timestamp.

utc2

Binary timestamp.

Output

result

Spanning timestamp.

Description

Given two binary timestamps, the **Span Time** routine returns a single UTC time interval whose inaccuracy spans the two input timestamps (that is, the interval resulting from the earliest possible time of either timestamp to the latest possible time of either timestamp).

The *tdf* in the output UTC value is copied from the *utc2* input. If either input binary timestamp has an infinite inaccuracy, an error is returned.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument.

Example

The following example computes the earliest and latest times for an array of 10 timestamps.

```
utc_t               time_array[10], testtime, earliest, latest;
int                 i;

/*
 *   Set the running timestamp to the first entry...
 */

testtime = time_array[0];

for (i=1; i<10; i++) {

   /*
    *   Compute the minimum and the maximum against the next
    *   element...
    */

utc_spantime(&testtime,       /* Out: Resultant interval        */
             &testtime,       /* In:  Largest previous interval */
             &time_array[i]); /* In:  Element under test        */
}

/*
 *   Compute the earliest possible time...
 */

utc_pointtime(&earliest,    /* Out: Earliest poss time in array */
             (utc_t *)0,    /* Out: Midpoint                    */
             &latest,       /* Out: Latest poss time in array   */
             &testtime);    /* In:  Spanning interval           */
```

Related Functions

utc_boundtime, *utc_gettime*, *utc_pointtime*

utc_subtime

utc_subtime — Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times.

Format

```
#include <utc.h>
int utc_subtime( *result, *utc1, *utc2)

utc_t *result;
const utc_t *utc1;
const utc_t *utc2;
```

Parameter

Input

utc1

Binary timestamp or relative binary timestamp.

utc2

Binary timestamp or relative binary timestamp.

Output

result

Resulting binary timestamp or relative binary timestamp, depending on the operation performed:

- *absolute time* - *absolute time* = **relative time**
- *relative time* - *relative time* = **relative time**
- *absolute time* - *relative time* = **absolute time**
- *relative time* - *absolute time* is undefined. See **NOTES**.

Description

The **Subtract Time** routine subtracts one binary timestamp from another. The resulting timestamp is *utc1* minus *utc2*. The inaccuracies of the two input timestamps are combined and included in the output timestamp. The TDF in the first timestamp is copied to the output.

Although no error is returned, do **not** use the combination *relative time* - *absolute time*.

Returns

0	Indicates that the routine executed successfully.

--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *utc_binreltime* routine.

Related Functions

utc_addtime

utc_vmsanytime

utc_vmsanytime — Converts a binary timestamp to a binary OpenVMS format time. The TDF encoded in the input timestamp determines the TDF of the output.

Format

```
#include <utc.h>
int utc_vmsanytime( *timadr, *utc)

const utc_t *utc;
long *timadr;
```

Parameter

Input

***utc**

Binary timestamp.

Output

***timadr**

Binary OpenVMS format time.

Description

The **VMS Any Time** routine converts a UTC-based binary timestamp to a 64-bit binary time in the OpenVMS (Smithsonian) format. Because the input and output values are based on different time standards, any input representing a value before the Smithsonian base time of November 17, 1858 returns an error.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *mkvmsanytime* routine.

Related Functions

utc_mkvmsanytime

utc_vmsgmtime

utc_vmsgmtime — Converts a binary timestamp to a binary OpenVMS format time expressing GMT or the equivalent UTC.

Format

```
#include <utc.h>
int utc_vmsgmtime( *timadr, *utc)

const utc_t *utc;
long *timadr;
```

Parameter

Input

***utc**

Binary timestamp to be converted.

Output

***timadr**

Binary OpenVMS format time representing GMT or the UTC equivalent.

Description

The **OpenVMS Greenwich Mean Time** routine converts a UTC-based binary timestamp to a 64-bit binary time in the OpenVMS (Smithsonian) format. The OpenVMS format time represents Greenwich Mean Time or the equivalent UTC. Because the input and output values are based on different time standards, any input representing a value before the Smithsonian base time of November 17, 1858 returns an error.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

The following example shows the following time zone and time format conversions:

1. Retrieve a binary timestamp representing UTC with the *sys$getutc* system service.
2. Convert the binary timestamp to a OpenVMS format binary time representing GMT
3. Convert the OpenVMS format binary time representing GMT back to a UTC-based binary timestamp with a TDF of 0 (zero)

4. Convert the UTC-based binary time to a binary OpenVMS format time representing the local time; use the TDF from the system

```
/*****
  start example vmsgmtime, mkvmsgmtime, vmslocaltime
 *****/
#include <utc.h>

main()
{
int status;
struct utc utcTime;
int vmsTime[2];

 if (!((status=SYS$GETUTC(&utcTime))&1))
     exit(status);              /* read curr time as a utc */

/*
 * convert the utcvalue into a vms time, with a timezone of 0
 * (GMT).  Printing the resultant vmstime yields the time at
 * the prime meridian in Greenwich, not (necessarily) the
 * local time.
 */
 if (utc_vmsgmtime(vmsTime,&utcTime))
    exit(1);

/*
 * Convert the vmstime (which is in GMT) to a utc
 */
if (utc_mkvmsgmtime(&utcTime, vmsTime))
    exit(2);

/*
 *  convert the UTC to local 64-bit time.  Note that this is the
 *  value we would have read if we had issued a 'SYS$GETTIM' in
 *  the initial statement.
 */
if (utc_vmslocaltime(vmsTime, &utcTime))
    exit(3);
}
/*****
 end example
 *****/
```

Related Functions

utc_mkvmsgmtime

utc_vmslocaltime

utc_vmslocaltime — Converts a binary timestamp to a local binary OpenVMS format time, using the host system's time differential factor.

Format

```
#include <utc.h>
```

```
int utc_vmslocaltime( *timadr, *utc)

const utc_t *utc;
long *timadr;
```

Parameter

Input

***utc**

Binary timestamp.

Output

***timadr**

Binary OpenVMS format time expressing local time.

Description

The **VMS Local Time** routine converts a binary timestamp to a binary OpenVMS format time; the output value represents the local time of the host system. The system's offset from UTC and the local time value are defined by the time zone rule in *sys$timezone_rule*, which is created by the system configuration process *sys$manager:net$configure.com*.

Returns

0	Indicates that the routine executed successfully.
--1	Indicates an invalid time argument or invalid results.

Example

See the sample program for the *vmsgmtime* routine.

Related Functions

utc_vmsmklocaltime

9.6. Example Using the DECdts API Routines

This section contains a C programming example showing a practical application of the DECdts API programming routines. The program performs the following actions:

- Prompts the user to enter time coordinates.
- Stores those coordinates in a *tm* structure.
- Converts the *tm* structure to a *utc* structure.
- Determines which event occurred first.
- Determines if Event 1 may have caused Event 2 by comparing the intervals.

- Prints out the *utc* structure in ISO text format.

```
#include <time.h>   /* time data structures                    */
#include <utc.h>    /* utc structure definitions               */

void ReadTime();
void PrintTime();

/*
 * This program requests user input about events, then prints out
 * information about those events.
 */

main()
{
    struct utc event1,event2;
    enum utc_cmptype relation;

    /*
     * Read in the two events.
     */

    ReadTime(&event1);
    ReadTime(&event2);

    /*
     * Print out the two events.
     */

    printf("The first event is : ");
    PrintTime(&event1);
    printf("\nThe second event is : ");
    PrintTime(&event2);
    printf("\n");

    /*
     * Determine which event occurred first.
     */
    if (utc_cmpmidtime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing midpoints: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing midpoints: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing midpoints: Event1 == Event2\n");
        break;
        default:
        exit(1);
        break;
    }

    /*
```

```
     * Could Event 1 have caused Event 2?  Compare the intervals.
     */

    if (utc_cmpintervaltime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing intervals: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing intervals: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing intervals: Event1 == Event2\n");
        break;
        case utc_indeterminate:
        printf("comparing intervals: Event1 ? Event2\n");
        default:
        exit(1);
        break;
    }

}

/*
 * Print out a utc structure in ISO text format.
 */

void PrintTime(utcTime)
struct utc *utcTime;
{

    char   string[50];

    /*
     * Break up the time string.
     */

    if (utc_ascgmtime(string,      /* Out: Converted time    */
                      50,          /* In:  String length     */
                      utcTime))    /* In:  Time to convert   */
        exit(1);
    printf("%s\n",string);
}


/*
 * Prompt the user to enter time coordinates.  Store the
 * coordinates in a tm structure and then convert the
 * tm structure to a utc structure.
 */

void ReadTime(utcTime)
struct utc *utcTime;
{
struct tm tmTime,tmInacc;
```

```
    (void)memset((void *)&tmTime,  0,sizeof(tmTime));
    (void)memset((void *)&tmInacc, 0,sizeof(tmInacc));
    (void)printf("Year? ");
    (void)scanf("%d",&tmTime.tm_year);
    tmTime.tm_year -= 1900;
    (void)printf("Month? ");
    (void)scanf("%d",&tmTime.tm_mon);
    tmTime.tm_mon -= 1;
    (void)printf("Day? ");
    (void)scanf("%d",&tmTime.tm_mday);
    (void)printf("Hour? ");
    (void)scanf("%d",&tmTime.tm_hour);
    (void)printf("Minute? ");
    (void)scanf("%d",&tmTime.tm_min);
    (void)printf("Inacc Secs? ");
    (void)scanf("%d",&tmInacc.tm_sec);

    if (utc_mkanytime(utcTime,
                      &tmTime,
                      (long)0,
                      &tmInacc,
                      (long)0,
                      (long)0))
        exit(1);

}
```

Assume the preceding program is named *compare_events.c*. To compile and link the program on a DECnet-Plus for OpenVMS system, enter the following command:

```
$ cc compare_events.c/output=compare_events.obj
$ link compare_events.obj, sys$input:/options[Return]
sys$library:dtss$shr.exe/share[Ctrl-z]
$
```

Chapter 10. EDT Routines

On OpenVMS operating systems, the EDT editor can be called from a program written in any language that generates calls using the OpenVMS Calling Standard.

You can set up your call to EDT so the program handles all the editing work, or you can make EDT run interactively so you can edit a file while the program is running.

This chapter on callable EDT assumes that you know how to call an external facility from the language you are using. Callable EDT is a shareable image, which means that you save physical memory and disk space by having all processes access a single copy of the image.

10.1. Introduction to EDT Routines

You must include a statement in your program accessing the EDT entry point. This reference statement is similar to a library procedure reference statement. The EDT entry point is referenced as EDT$EDIT. You can pass arguments to EDT$EDIT; for example, you can pass EDT$FILEIO or your own routine. When you refer to the routines you pass, call them FILEIO, WORKIO, and XLATE. Therefore, FILEIO can be either a routine provided by EDT (named EDT$FILEIO) or a routine that you write.

10.2. Using the EDT Routines: An Example

Example 10.1 shows a VAX BASIC program that calls EDT. All three routines (FILEIO, WORKIO, and XLATE) are called. Note the reference to the entry point EDT$EDIT in line number 500.

Example 10.1. Using the EDT Routines in a VAX BASIC Program

```
100  EXTERNAL INTEGER EDT$FILEIO  ❶
200  EXTERNAL INTEGER EDT$WORKIO
250  EXTERNAL INTEGER AXLATE
300  EXTERNAL INTEGER FUNCTION EDT$EDIT
400  DECLARE INTEGER RESULT


450  DIM INTEGER PASSFILE(1%)  ❷
460  DIM INTEGER PASSWORK(1%)
465  DIM INTEGER PASSXLATE(1%)
470  PASSFILE(0%) = LOC(EDT$FILEIO)
480  PASSWORK(0%) = LOC(EDT$WORKIO)
485  PASSXLATE(0%) = LOC(AXLATE)

500  RESULT = EDT$EDIT('FILE.BAS','','EDTINI','',0%,  ❸
     PASSFILE(0%)BY REF, PASSWORK(0%) BY REF, ❹
     PASSXLATE(0%) BY REF)  ❺
600  IF (RESULT AND 1%) = 0%
     THEN
       PRINT "SOMETHING WRONG"
       CALL LIB$STOP(RESULT BY VALUE)
900  PRINT "EVERYTHING O.K."
1000 END
```

❶ The external entry points EDT$FILEIO, EDT$WORKIO, and AXLATE are defined so they can be passed to callable EDT.

❷ Arrays are used to construct the two-longword structure needed for data type BPV.
❸ Here is the call to EDT. The input file is FILE.BAS, the output and journal files are defaulted, and the command file is EDTINI. A *0* is passed for the options word to get the default EDT options.
❹ The array PASSFILE points to the entry point for all file I/O, which is set up in this example to be the EDT-supplied routine with the entry point EDT$FILEIO. Similarly, the array PASSWORK points to the entry point for all work I/O, which is the EDT-supplied routine with the entry point EDT$WORKIO.
❺ PASSXLATE points to the entry point that EDT will use for all XLATE processing. PASSXLATE points to a user-supplied routine with the entry point AXLATE.

10.3. EDT Routines

This section describes the individual EDT routines.

EDT$EDIT

Edit a File — The EDT$EDIT routine invokes the EDT editor.

Format

```
EDT$EDIT in_file [,out_file] [,com_file] [,jou_file] [,options] [,fileio]
 [,workio] [,xlate]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

in_file

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

File specification of the input file that EDT$EDIT is to edit. The *in_file* argument is the address of a descriptor pointing to this file specification. The string that you enter in this calling sequence is passed to the FILEIO routine to open the primary input file. This is the only required argument.

out_file

OpenVMS usage:	char_string

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

File specification of the output file that EDT$EDIT creates. The *out_file* argument is the address of a descriptor pointing to this file specification. The default is that the input file specification is passed to the FILEIO routine to open the output file for the EXIT command.

com_file

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

File specification of the startup command file to be executed when EDT is invoked. The *com_file* argument is the address of a descriptor pointing to this file specification. The *com_file* string is passed to the FILEIO routine to open the command file. The default is the same as that for EDT command file defaults.

jou_file

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

File specification of the journal file to be opened when EDT is invoked. The *jou_file* argument is the address of a descriptor pointing to this file specification. The *jou_file* string is passed to the FILEIO routine to open the journal file. The default is to use the same file name as *in_file*.

options

OpenVMS usage:	mask_longword
type:	aligned bit string
access:	read only
mechanism:	by reference

Bit vector specifying options for the edit operation. The *options* argument is the address of an aligned bit string containing this bit vector. Only bits <5:0> are currently defined; all others must be *0*. The default options have all bits set to *0*. This is the same as the default setting when you invoke EDT to edit a file from DCL.

Symbols and their descriptions follow:

Symbol	Description
EDT$M_RECOVER	If set, bit <0> causes EDT to read the journal file and execute the commands in it, except for the EXIT or QUIT commands, which are ignored.

Symbol	Description
	After the journal file commands are processed, editing continues normally. If bit <0> is set, the FILEIO routine is asked to open the journal file for both input and output; otherwise FILEIO is asked only to open the journal file for output. Bit <0> corresponds to the /RECOVER qualifier on the EDT command line.
EDT$M_COMMAND	If set, bit <1> causes EDT to signal if the startup command file cannot be opened. When bit <1> is *0*, EDT intercepts the signal from the FILEIO routine indicating that the startup command file could not be opened. Then, EDT proceeds with the editing session without reading any startup command file. If no command file name is supplied with the call to the EDT$EDIT routine, EDT tries to open SYS$LIBRARY:EDTSYS.EDT or, if that fails, EDTINI.EDT. Bit <1> corresponds to the /COMMAND qualifier on the EDT command line. If EDT$M_NOCOMMAND (bit <4>) is set, bit <1> is overridden because bit <4> prevents EDT from trying to open a command file.
EDT$M_NOJOURNAL	If set, bit <2> prevents EDT from opening the journal file. Bit <2> corresponds to the /NOJOURNAL or /READ_ONLY qualifier on the EDT command line.
EDT$M_NOOUTPUT	If set, bit <3> prevents EDT from using the input file name as the default output file name. Bit <3> corresponds to the /NOOUTPUT or /READ_ONLY qualifier on the EDT command line.
EDT$M_NOCOMMAND	If set, bit <4> prevents EDT from opening a startup command file. Bit <4> corresponds to the /NOCOMMAND qualifier on the EDT command line.
EDT$M_NOCREATE	If set, bit <5> causes EDT to return to the caller if the input file is not found. The status returned is the error code EDT$_INPFILNEX.

fileio

OpenVMS usage:	vector_longword_unsigned
type:	bound procedure value
access:	function call
mechanism:	by reference

User-supplied routine called by EDT to perform file I/O functions. The `fileio` argument is the address of a bound procedure value containing the user-supplied routine. When you do not need to intercept any file I/O, either use the entry point EDT$FILEIO for this argument or omit it. When you only need to intercept some amount of file I/O, call the EDT$FILEIO routine for the other cases.

To avoid confusion, note that EDT$FILEIO is a routine provided by EDT whereas FILEIO is a routine that you provide.

In order to accommodate routines written in high-level languages that do up-level addressing, this argument must have a data type of BPV (bound procedure value). BPV is a two-longword entity in which the first longword contains the address of a procedure value and the second longword is the environment value. When the bound procedure is called, EDT loads the second longword into R1. If you use EDT$FILEIO for this argument, set the second longword to <0>. You can pass a <0> for the argument, and EDT will set up EDT$FILEIO as the default and set the environment word to *0*.

workio

OpenVMS usage:	vector_longword_unsigned
type:	bound procedure value
access:	function call
mechanism:	by reference

User-supplied routine called by EDT to perform I/O between the work file and EDT. The *workio* argument is the address of a bound procedure value containing the user-supplied routine. Work file records are addressed only by number and are always 512 bytes long. If you do not need to intercept work file I/O, you can either use the entry point EDT$WORKIO for this argument or omit it.

In order to accommodate routines written in high-level languages that do up-level addressing, this argument must have a data type of BPV (bound procedure value). This means that EDT loads R1 with the second longword addressed before calling it. If EDT$WORKIO is used for this argument, set the second longword to *0*. You can pass a *0* for this argument, and EDT will set up EDT$WORKIO as the default and set the environment word to *0*.

xlate

OpenVMS usage:	vector_longword_unsigned
type:	bound procedure value
access:	function call
mechanism:	by reference

User-supplied routine that EDT calls when it encounters the nokeypad command XLATE. The *xlate* argument is the address of a bound procedure value containing the user-supplied routine. The XLATE routine allows you to gain control of your EDT session. If you do not need control of EDT during the editing session, you can either use the entry point EDT$XLATE for this argument or omit it.

In order to accommodate routines written in high-level languages that do up-level addressing, this argument must have a data type of BPV (bound procedure value). This means that EDT loads R1 with the second longword addressed before calling it. If EDT$XLATE is used for this argument, set the second longword to *0*. You can pass a *0* for this argument, and EDT will set up EDT$XLATE as the default and set the environment word to *0*.

Description

If the EDT session is terminated by EXIT or QUIT, the status will be a successful value (bit <0> = 1). If the session is terminated because the file was not found and if the /NOCREATE qualifier was in effect, the failure code EDT$_INPFILNEX is returned. In an unsuccessful termination caused by an EDT error,

a failure code corresponding to that error is returned. Each error status from the FILEIO and WORKIO routines is explained separately.

Three of the arguments to the EDT$EDIT routine, *fileio*, *workio*, and *xlate* are the entry point names of user-supplied routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

EDT$_INPFILNEX

/NOCREATE specified and input file does not exist.

This routine also returns any condition values returned by user-supplied routines.

FILEIO

FILEIO — The user-supplied FILEIO routine performs file I/O functions. Call it by specifying it as an argument in the EDT$EDIT routine. It cannot be called independently.

Format

```
FILEIO code ,stream ,record ,rhb
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

A status code that your FILEIO routine returns to EDT$EDIT. The *fileio* argument is a longword containing the status code. The only failure code that is normally returned is RMS$_EOF from a GET call. All other OpenVMS RMS errors are signaled, not returned. The RMS signal should include the file name and both longwords of the RMS status. Any errors detected with the FILEIO routine can be indicated by setting status to an error code. That special error code will be returned to the program by the EDT$EDIT routine. There is a special status value EDT$_NONSTDFIL for nonstandard file opening.

Condition values are returned in R0.

Arguments

code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only

mechanism:	by reference

A code from EDT that specifies what function the FILEIO routine is to perform. The *code* argument is the address of a longword integer containing this code. Following are the valid function codes:

Function Code	Description
EDT$K_OPEN_INPUT	The *record* argument names a file to be opened for input. The *rhb* argument is the default file name.
EDT$K_OPEN_OUTPUT_SEQ	The *record* argument names a file to be opened for output as a sequenced file. The *rhb* argument is the default file name.
EDT$K_OPEN_OUTPUT_NOSEQ	The *record* argument names a file to be opened for output. The *rhb* argument is the default file name.
EDT$K_OPEN_IN_OUT	The *record* argument names a file to be opened for both input and output. The *rhb* argument is the default file name.
EDT$K_GET	The *record* argument is to be filled with data from the next record of the file. If the file has record prefixes, *rhb* is filled with the record prefix. If the file has no record prefixes, *rhb* is not written. When you attempt to read past the end of file, *status* is set to RMS$_EOF.
EDT$K_PUT	The data in the *record* argument is to be written to the file as its next record. If the file has record prefixes, the record prefix is taken from the *rhb* argument. For a file opened for both input and output, EDT$K_PUT is valid only at the end of the file, indicating that the *record* is to be appended to the file.
EDT$K_CLOSE_DEL	The file is to be closed and then deleted. The *record* and *rhb* arguments are not used in the call.
EDT$K_CLOSE	The file is to be closed. The *record* and *rhb* arguments are not used in the call.

stream

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

A code from EDT that indicates which file is being used. The *stream* argument is the address of a longword integer containing the code. Following are the valid codes:

Function Code	Description
EDT$K_COMMAND_FILE	The command file.

Function Code	Description
EDT$K_INPUT_FILE	The primary input file.
EDT$K_INCLUDE_FILE	The secondary input file. Such a file is opened in response to an INCLUDE command. It is closed when the INCLUDE command is complete and will be reused for subsequent INCLUDE commands.
EDT$K_JOURNAL_FILE	The journal file. If bit 0 of the options is set, it is opened for both input and output and is read completely. Otherwise, it is opened for output only. After it is read or opened for output only, it is used for writing. On a successful termination of the editing session, the journal file is closed and deleted. EXIT/SAVE and QUIT/SAVE close the journal file without deleting it.
EDT$K_OUTPUT_FILE	The primary output file. It is not opened until you enter the EXIT command.
EDT$K_WRITE_FILE	The secondary output file. Such a file is opened in response to a WRITE or PRINT command. It is closed when the command is complete and will be reused for subsequent WRITE or PRINT commands.

record

OpenVMS usage:	char_string
type:	character-coded text string
access:	modify
mechanism:	by descriptor

Text record passed by descriptor from EDT to the user-supplied FILEIO routine; the *code* argument determines how the *record* argument is used. The *record* argument is the address of a descriptor pointing to this argument. When the *code* argument starts with EDT$K_OPEN, the *record* is a file name. When the *code* argument is EDT$K_GET, the *record* is a place to store the record that was read from the file. For *code* argument EDT$K_PUT, the *record* is a place to find the record to be written to the file. This argument is not used if the *code* argument starts with EDT$K_CLOSE.

Note that for EDT$K_GET, EDT uses a dynamic or varying string descriptor; otherwise, EDT has no way of knowing the length of the record being read. EDT uses only string descriptors that can be handled by the Run-Time Library routine STR$COPY_DX.

rhb

OpenVMS usage:	char_string
type:	character-coded text string
access:	modify
mechanism:	by descriptor

Text record passed by descriptor from EDT to the user-supplied FILEIO routine; the *code* argument determines how the *rhb* argument is used. When the *code* argument starts with EDT$K_OPEN, the

rhb argument is the default file name. When the *code* is EDT$K_GET and the file has record prefixes, the prefixes are put in this argument. When the *code* is EDT$K_PUT and the file has record prefixes, the prefixes are taken from this argument. Like the *record* argument, EDT uses a dynamic or varying string descriptor for EDT$K_GET and uses only string descriptors that can be handled by the Run-Time Library routine STR$COPY_DX.

Description

If you do not need to intercept any file I/O, you can use the entry point EDT$FILEIO for this argument or you can omit it. If you need to intercept only some file I/O, call the EDT$FILEIO routine for the other cases.

When you use EDT$FILEIO as a value for the *fileio* argument, files are opened as follows:

- The *record* argument is always the RMS file name.
- The *rhb* argument is always the RMS default file name.
- There is no related name for the input file.
- The related name for the output file is the input file with OFP (output file parse). EDT passes the input file name, the output file name, or the name from the EXIT command in the *record* argument.
- The related name for the journal file is the input file name with the OFP RMS bit set.
- The related name for the INCLUDE file is the input file name with the OFP set. This is unusual because the file is being opened for input.

EDT contains support for VFC files. Normally, EDT will zero the length of the RHB field if the file is not a VFC file. However, when the user supplies the FILEIO routines, they are responsible for performing this operation.

EDT checks for a VFC file with the following algorithm:

```
IF FAB$B_RFM = FAB$C_VFC
AND FAB$B_RAT <> FAB$M_PRN
THEN
        VFC file
ELSE
        not VFC file, zero out RHB descriptor length field.
```

Condition Values Returned

SS$_NORMAL

Normal successful completion.

EDT$_NONSTDFIL

File is not in standard text format.

RMS$_EOF

End of file on a GET.

WORKIO

WORKIO — The user-supplied WORKIO routine is called by EDT when it needs temporary storage for the file being edited. Call it by specifying it as an argument in the EDT$EDIT routine. It cannot be called independently.

Format

```
WORKIO code ,recordno ,record
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by immediate value

Longword value returned as a status code. It is generally a success code, because all OpenVMS RMS errors should be signaled. The signal should include the file name and both longwords of the RMS status. Any errors detected within work I/O can be indicated by setting status to an error code, which will be returned by the EDT$EDIT routine.

The condition value is returned in R0.

Arguments

code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

A code from EDT that specifies the operation to be performed. The *code* argument is the address of a longword integer containing this argument. The valid function codes are as follows:

Function Code	Description
EDT$K_OPEN_IN_OUT	Open the work file for both input and output. Neither the *record* nor *recordno* argument is used.
EDT$K_GET	Read a record. The *recordno* argument is the number of the record to be read. The *record* argument gives the location where the record is to be stored.
EDT$K_PUT	Write a record. The *recordno* argument is the number of the record to be written. The *record* argument tells the location of the record to be written.

Function Code	Description
EDT$K_CLOSE_DEL	Close the work file. After a successful close, the file is deleted. Neither the *record* nor *recordno* argument is used.

recordno

OpenVMS usage:	longword_signed
type:	longword integer (signed)
access:	read only
mechanism:	by reference

Number of the record to be read or written. The *recordno* argument is the address of a longword integer containing this argument. EDT always writes a record before reading that record. This argument is not used for open or close calls.

record

OpenVMS usage:	char_string
type:	character string
access:	modify
mechanism:	by descriptor

Location of the record to be read or written. This argument always refers to a 512-byte string during GET and PUT calls. This argument is not used for open or close calls.

Description

Work file records are addressed only by number and are always 512 bytes long. If you do not need to intercept work file I/O, you can use the entry point EDT$WORKIO for this argument or you can omit it.

Condition Value Returned

SS$_NORMAL

Normal successful completion.

XLATE

XLATE — The user-supplied XLATE routine is called by EDT when it encounters the nokeypad command XLATE. You cause it to be called by specifying it as an argument in the EDT$EDIT routine. It cannot be called independently.

Format

```
XLATE string
```

Returns

OpenVMS usage:	cond_value

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

Longword value returned as a status code. It is generally a success code. If the XLATE routine cannot process the passed string for some reason, it sets status to an error code. Returning an error code from the XLATE routine aborts the current key execution and displays the appropriate error message.

The condition value is returned in R0.

Argument

string

OpenVMS usage:	char_string
type:	character-coded text string
access:	modify
mechanism:	by descriptor

Text string passed to the nokeypad command XLATE. You can use the nokeypad command XLATE by defining a key to include the following command in its definition:

```
XLATEtext^Z
```

The text is passed by the *string* argument. The *string* argument can be handled by the Run-Time Library routine STR$COPY_DX.

This argument is also a text string returned to EDT. The string is made up of nokeypad commands that EDT is to execute.

Description

The nokeypad command XLATE allows you to gain control of the EDT session. (See the *OpenVMS EDT Reference Manual*[1] for more information about the XLATE command.) If you do not need to gain control of EDT during the editing session, you can use the entry point EDT$XLATE for this argument or you can omit it.

Condition Value Returned

SS$_NORMAL

Normal successful completion.

[1]This manual has been archived but is available on the *OpenVMS Documentation CD-ROM*.

Chapter 11. Encryption (ENCRYPT) Routines

The encryption routines (APIs) allow you to program encryption operations into applications. OpenVMS Version 8.3 Integrity servers and Alpha systems support the Advanced Encryption Standard (AES) algorithm, which allows any OpenVMS user, system manager, security manager, or programmer to secure their files, save sets, or application data with AES Encryption. The former DES algorithm is also supported for complete backward compatibility. This allows updating archived data encrypted with DES to the more secure AES encryption algorithm.

Note

The DES encryption standard, reviewed and approved by the National Bureau of Standards (NBS) every five years, remained the popular standard until 1992. The National Institute of Standards and Technology (NIST) later declared the minimum encryption standard to be Triple-DES (or TDEA). Triple-DES typically uses at least two or three different secret keys. Since 1999, the older single DES standard is used only for legacy government systems.

Since 2001, the Advanced Encryption Standard (AES) (FIPS PUB 197[5]) is the approved symmetric encryption algorithm that replaced DES.

Encryption is used to convert sensitive or otherwise private data to an unintelligible form called **cipher text**. **Decryption** reverses this process, taking the unintelligible cipher text and converting data back to its original form, called **plain text**. Encryption and decryption are also known as cipher and decipher.

Note

OpenVMS Version 8.3 integrates the former Encryption for OpenVMS software product into the operating system, eliminating the requirement for a separate installation and product license.

11.1. Introduction to Encryption Routines

Encryption provides the following routines, listed by function:

- Defining, generating, and deleting keys:
  - ENCRYPT$DEFINE_KEY
  - ENCRYPT$GENERATE_KEY
  - ENCRYPT$DELETE_KEY
- Encrypting and decrypting files:
  - ENCRYPT$ENCRYPT
  - ENCRYPT$ENCRYPT_FILE
  - ENCRYPT$DECRYPT
- Initializing and terminating the context area:

    - ENCRYPT$INIT
    - ENCRYPT$FINI
- Returning statistics:
    - ENCRYPT$STATISTICS

11.2. Encrypt AES Features

AES encryption, like DES, is a symmetric block cipher. However, its algorithm is very different, its key scheduling and number of rounds is based on key size (10, 12, or 14 rounds for 128, 192, and 256 bit keys), making AES much stronger cryptographically. AES features allows any user, system manager, security manager, or programmer to secure their files, save-sets, or application data with strong AES Encryption. It is integrated with OpenVMS Version 8.3 and does not require a separate product license or installation.

Encrypt-AES provides the following features and compatibility:

- The former data encryption standard (DES) algorithm is maintained for use with existing DES data and their applications. All the functions that existed with DES continue to provide that same level of DES support.
- Encrypt-AES is integrated with BACKUP for encrypting and decrypting save sets with AES or DES.
- Command-line use of Encrypt-AES is the same as Encrypt-DES, with minor changes to qualifiers (see the encryption routines later in this chapter).
- Changes to the ENCRYPT$ application programming interface (API) are minimal, with only textual parameter or flag changes required to use the AES algorithm.
- Encrypt-AES supports the AES algorithm with four different cipher modes. With each mode, you can specify a secret key in three different lengths (128, 192, and 256 bits), for a total of 12 different cipher and decipher operations:
    - Cipher block chaining:

      AESCBC128
      AESCBC192
      AESCBC256

    - Electronic code book:

      AESECB128
      AESECB192
      AESECB256

    - Cipher feedback:

      AESCFB128
      AESCFB192
      AESCFB256

    - Output feedback:

AESOFB128
AESOFB192
AESOFB256

- The additional AES algorithm, modes, and key sizes are specified in the *algorithm* parameter to the ENCRYPT$ENCRYPT_FILE and the ENCRYPT$INIT routine, or specified in the *algorithm-name* parameter for the ENCRYPT$GENERATE_KEY routine.

- AES Key-Length Requirements--- The AES key requirements are the actual number of bits utilized for each of the AES modes. This is actually the minimum number of bytes needed for the encryption or decryption operation. The minimum required key sizes are as follows:

  - 128 bit mode = 16 byte key

  - 192 bit mode = 24 byte key

  - 256 bit mode = 32 byte key

  For more information in encryption keys, see Section 11.3.1.

11.2.1. ENCRYPT-AES Key, Flag Mask, and Value

There are no new Encrypt-AES API routines in OpenVMS V8.3. However, to accommodate the AES algorithm and the various key-length values, an additional AES key and AES file flag mask and value are added to OpenVMS Version 8.3:

- AES key flag

  The KEY_AES mask value specified an AES key (as a longword by reference) to the ENCRYPT $DEFINE_KEY, ENCRYPT$DELETE_KEY, and ENCRYPT$GENERATE_KEY routines.

  - ENCRYPT$M_KEY_AES

  - ENCRYPT$V_KEY_AES

- AES file flag

  An additional FILE_AES flag mask (and value) is used with the ENCRYPT$ENCRYPT_FILE routine when encrypting files that use an AES algorithm.

  The ENCRYPT$ENCRYPT_FILE_FLAGS flags are used to control file operations such as cipher direction, file compression, and so on. The FILE_AES flag controls file AES initialization and encryption operations and also flags AES keys.

  - ENCRYPT$M_FILE_AES

  - ENCRYPT$V_FILE_AES

  The AES algorithm, mode, and a key length (128, 192, or 256 bits) are specified in the algorithm parameter for the ENCRYPT$ENCRYPT_FILE and ENCRYPT$INIT routines, or the are specified in the algorithm-name parameter for the ENCRYPT$GENERATE_KEY routine. This parameter is in the form of a character string descriptor reference (pointer), as follows:

  - Block mode ciphers

AESCBC128 - Cipher Block Chaining
AESCBC192 - Cipher Block Chaining
AESCBC256 - Cipher Block Chaining
AESECB128 - Electronic Code Book
AESECB192 - Electronic Code Book
AESECB256 - Electronic Code Book

- Stream mode ciphers

  AESCFB128 - Cipher Feedback
  AESCFB192 - Cipher Feedback
  AESCFB256 - Cipher Feedback
  AESOFB128 - Output Feedback
  AESOFB192 - Output Feedback
  AESOFB256 - Output Feedback

Note

AESCBC128 is the default cipher and is also used for encryption and decryption of the users key for storage of logical names. These ciphers are looked up in the order in which they are stored in their algorithm table with the new image file SYS$SHARE:ENCRYPT$ALG$AES.EXE file.

11.3. How the Routines Work

You can call the Encryption for OpenVMS routines from any language that supports the OpenVMS Calling Standard in 32 bit mode. After it is called, each routine:

- Performs its function
- Returns a 32-bit status code value for the calling program to determine success or failure
- Returns control to the calling program

The callable routines do not provide all the options of the file selection qualifiers available with the DCL ENCRYPT and DECRYPT commands. The functions of /BACKUP, /BEFORE, /BY_OWNER, /CONFIRM, /EXCLUDE, /EXPIRED, /SINCE, and /SHOW are supported only at the DCL-interface level. For more information, see the *Guide to Creating OpenVMS Modular Procedures*.

11.3.1. Encryption Keys

This section provides information about encryptions for AES and DES.

- AES Keys are created, encrypted (always with AESCBC128 and a master key), and stored in a logical name table. During an encrypt operation, the key is fetched, decrypted, and used as a 16-, 24- or 32-byte key, depending on the chosen algorithm/key size for the cipher operation.
- Non-literal DES keys are compressed, that is, converted to uppercase. Only the characters A-Z, 0-9, dollar sign ($), period (.), and underscore (_) are allowed. All others are converted to spaces, and multiple spaces are removed. AES ASCII key values are not compressed.
- Use caution when creating keys to ensure they meet the minimum key length when later used for the algorithm/key size selected. This condition was not a problem with 8-byte DES keys. Any key (literal

or nonliteral) that is longer in length than necessary is folded for the proper 16-, 24- or 32-byte key size.

- The key name is a logical name for the key as stored in the logical name table (SYSTEM, JOB, GROUP, or PROCESS - the default). The value can be ASCII (normal text keys), or hexadecimal/binary. When creating a literal key (key-flags = ENCRYPT$M_LITERAL_KEY), the value is stored as a literal value and it is not compressed.

- Errors can result when using the ENCRYPT$GENERATE_KEY routine to generate AES keys and specifying key lengths that are not multiples of 16.

- Exercise care when supplying the key to the ENCRYPT$INIT routine; it must match the key stored in the logical name table. The descriptor type determines how the DES key is handled:

  - As text to be compressed, or

  - As a binary value not to be compressed

  AES key values are not compressed. The key flag (1 = literal, 0 = name) determines how the key-name parameter is interpreted:

  - As a literal value passed directly to INIT, or

  - As a key name for logical name lookup, translation, and decryption.

  Note that errors can result if you use an incorrect key type. For example, an error occurs if the key flag = 0 (name) and a literal key value is provided instead of a key name. An error could also occur if you attempt to provide a key name to be used as a literal value.

  For the ENCRYPT$INIT routine, key name descriptors of type DSC$K_DTYPE_T, DSC $K_DTYPE_VT, and DSC$K_DTYPE_Z specify that the key value should be compressed for DES keys. AES key values are not compressed.

11.3.1.1. Deleting AES Keys

Like DES keys, AES keys are deleted or removed with the encrypt command-line qualifier /REMOVE_KEY or with the ENCRYPT$DELETE_KEY routine:

```
$ ENCRYPT/REMOVE_KEY KEYNAME /AES
```

The user's secret key is encrypted with a master key and stored in a logical name table (PROCESS, JOB, GROUP or SYSTEM-ENCRYP$SYSTEM table), the default is the PROCESS logical name table. To delete a key in a table other than the PROCESS logical name table, the appropriate qualifier (/JOB, /GROUP, or /SYSTEM) must also be specified in the ENCRYPT /REMOVE_KEY command.

Because the users secret key name is unique, only one key with the same name can exist in the same logical name table, whether this is a DES key or an AES key. This means that the /AES qualifier is unnecessary, although it is implemented nevertheless.

11.3.1.2. DES Key and Data Semantics

The National Bureau of Standards (NBS) document FIPS-PUB-46 describes the operation of the DES algorithm in detail. The bit-numbering conventions in the NBS document are different from OpenVMS numbering conventions.

Note

For the AES algorithm, see the National Institute of Standards and Technology (NIST) document FIPS-PUB-197, pages 7 through 9.

If you are using Encryption for OpenVMS routines in conjunction with an independently developed DES encryption system, ensure that you are familiar with the relationship between the NBS and OpenVMS numbering conventions. Table 11.1 highlights the differences.

Table 11.1. Comparison of NBS and OpenVMS Numbering Conventions

NBS	Encryption for OpenVMS
Numbers bits from left to right.	Numbers bits from right to left.
Displays bytes in memory from left to right.	Displays bytes in memory from right to left.
Handles keys and data in 8-byte blocks.	Handles 8-byte blocks in OpenVMS display order.
Treats keys and data as byte strings.	Treats keys and data as character strings.
The "most significant byte" is byte 1.	Same.
In DES keys, the parity bits are DES bits 8, 16, 24, and so forth.	In DES keys, the parity bits are OpenVMS bits 0, 8, 16, and so forth.
DES keys, when expressed as strings of hexadecimal digits, are given starting with the high digit of byte 1, then the low digit of byte 1, then the high digit of byte 2, and so forth, through the low digit of byte 8.	Same.

To convert a hexadecimal key string into the 8-byte binary key, convert from hex to binary one byte at a time. For example, a quadword hex-to-binary conversion, using the library subroutine OTS$CVT_TZ_L, yields an incorrect, byte-reversed key.

Figure 11.1. OpenVMS Numbering Overlay on FIPS-46 Numbering

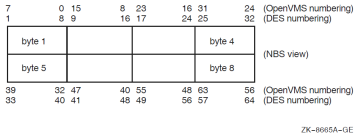
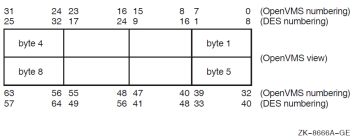


Figure 11.2. NBS Numbering Overlay on an OpenVMS Quadword



Note

On OpenVMS Integrity server systems, AES uses an OpenVMS numbering overlay on FIPS-197 numbering. For a description of AES key and data semantics, see the National Institute of Standards and Technology (NIST) document FIPS-PUB-197, pages 7 through 9.

11.3.2. File Encryption and Decryption

Once a key has been created, you can encrypt and decrypt files. This can be accomplished at the command line with the ENCRYPT and DECRYPT commands, or by using the ENCRYPT $ENCRYPT_FILE routine.

File encryption encrypts RMS files in fixed-length, 512- byte records. The file characteristics and attributes of the file are preserved, for example, the file creation and modify date, and whether the file was organized as sequential or indexed, and its record format (STREAM_LF, VAR, or other). You specify a key to be used for the encrypting a file and a data algorithm. However, the user key is used to encrypt the random key, initialization vector (IV), and data algorithm in the random key record. The random key encrypts the files attributes and feature records, and its data records, using the data algorithm that you specify.

When decrypting the file, the key specified decrypts the random key record, which retrieves the random (data) key, IV, and data algorithm file. Then the file's attributes, feature records, and data records are decrypted with the random key, IV, and data algorithm from the fixed-length 512-byte records, and then restored to its original format and creation date. The modified (or revised) file date is then updated.

11.4. Maintaining Keys

When you use AES or DES symmetric key encryption routines, first define the key that will be used in the encryption operation. Similarly, to decrypt a file specify the same key. Table 11.2 describes the callable routines that maintain keys.

Table 11.2. Routines for Maintaining Keys

Routine	Description
ENCRYPT$DEFINE_KEY	Creates a key definition with a key name and a key value. Puts the definition into a key storage table. Similar to the ENCRYPT /CREATE_KEY command.
ENCRYPT$DELETE_KEY	Removes a key definition from a key storage table. Uses the key name to identify the key to be removed. Similar to the ENCRYPT / REMOVE_KEY command.
ENCRYPT$GENERATE_KEY	Generates random key values.

When you call these routines, use the following arguments:

- With ENCRYPT$DEFINE_KEY
  - To pass the values for the key name and key value, use the **key-name** and the **key-value** arguments.
  - To specify a key storage table, use the **key-flags** argument.
  - To specify other key options, use the **key-flags** argument.
  - On DES, to override key compression, use the **key-flags** argument. (AES keys are not compressed.)
- With ENCRYPT$DELETE_KEY

  - To pass the key name, use the **key-name** argument.
  - To specify the key storage table in which the key resides, use the **key-flags** argument.
- With ENCRYPT$GENERATE_KEY
  - To define the length of the key, use the **key-length** argument in increments of 8 bytes for DES and 16-bytes for AES (that is, the block size).
  - To specify the buffer into which the generated key is to be placed, use the **key-buffer** argument.
  - To specify the algorithm that will use the key, use the **algorithm-name** argument.
  - To optionally pass three arbitrary values for added security, use the **factor-a**, **factor-b**, and **factor-c** arguments. These values are randomizing factors when the routine generates a key value. For example, the factors might be:
    - Time an operation started
    - Size of a certain stack
    - Copy of the last command line

11.5. Operations on Files

The ENCRYPT$ENCRYPT_FILE routine is similar to the DCL ENCRYPT and DECRYPT commands in that you use this routine with entire files.

The ENCRYPT$ENCRYPT_FILE routine specifies the key, the input file specification, the output file specification, and other file operation information.

Specify the type of operation, either encryption or decryption, with the **file-flags** argument for DES and **file-AES** argument for AES operations.

ENCRYPT$ENCRYPT_FILE does not require a prior call to ENCRYPT$INIT.

11.6. Operations on Records and Blocks

To operate on small records or blocks of data, use the following routines:

- ENCRYPT$ENCRYPT_ONE_RECORD
- ENCRYPT$DECRYPT_ONE_RECORD

These routines are a shorthand form of the ENCRYPT$INIT, ENCRYPT$ENCRYPT, ENCRYPT $DECRYPT, ENCRYPT$FINI sequence of calls.

Do not use these routines for data larger than a few records.

To use AES for one record ciphers, an AES key must first be created and stored in the logical name table (encrypted). The key name of an AES key is specified as an address of a descriptor that contains the ASCII text for the selected AESmmmkkk (mode and key size) algorithm, for example, AESCBC256. Note that the input and output buffers (descriptor addresses) are also provided.

11.7. Routine Descriptions

This section describes the syntax of each callable routine. The routines are listed alphabetically.

11.7.1. Specifying Arguments

Each routine's argument list shows the mandatory arguments first, followed by the optional arguments. Brackets ([ ]) identify optional arguments in the argument list.

For example, this format line shows that the required arguments are **context**, **input**, and **output**, and that the optional arguments are **output-length** and **p1**:

ENCRYPT$DECRYPT context ,input ,output [,output-length] [,p1]

When you specify arguments, follow these guidelines:

- The order is important. Specify arguments in the order in which they appear in the argument list.
- Separate each argument with a comma.
- Pass a zero value for each optional argument that you omit.

11.7.2. Bitmasks

Constants are associated with the symbolic names of the bitmasks used by the Encryption routines. These constants are defined in the ENCRYPT_STRUCTURES files that are provided with the kit.

The examples directory, ENCRYPT$EXAMPLES, has a copy of the ENCRYPT_STRUCTURES file in each supported programming language.

11.7.3. Error Handling

By default, Encryption signals error conditions with messages. To intercept a message that is inappropriate for your application, supply a condition handler.

For information about implementing condition handlers, see your programming language reference manual.

ENCRYPT$DECRYPT

ENCRYPT$DECRYPT — Decrypts the next record of ciphertext according to the algorithm specified in the ENCRYPT$INIT call.

Format

ENCRYPT$DECRYPT context, input, output [,output-length] [,p1]

Argument

context

OpenVMS usage:	
type:	longword integer (signed)

access:	write only
mechanism:	by reference

Context area initialized when ENCRYPT$INIT completes execution. The context argument is the address of a longword of unspecified interpretation that is used to convey context between encryption operations.

input

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Ciphertext record that ENCRYPT$DECRYPT is to decrypt. The input argument is the address of a descriptor pointing to a byte-aligned buffer containing the input record to the decryption operation.

output

OpenVMS usage:	
type:	char_string
access:	write only
mechanism:	by descriptor

Plaintext record that results when ENCRYPT$DECRYPT completes execution. The output argument is the address of a descriptor pointing to a byte-aligned padding buffer that will contain the output record from the decryption operation.

If the descriptor is dynamic and insufficient space is allocated to contain the output record, storage will be allocated from dynamic memory. If insufficient space exists to contain the output of the operation, then an error status is returned.

The ENCRYPT$DECRYPT routine adjusts the length of the output descriptor, if possible, to reflect the actual length of the output string. If the descriptor type is not DSC$K_DTYPE_VS (varying string), DSC$K_DTYPE_V (varying), or DSC$K_DTYPE_D (dynamic), the routine takes the actual output count from the **output-length** argument.

The output buffer must be able to accommodate a padded block to an increment of the block length. For AES this is 16 bytes and for DES, eight bytes.

output-length

OpenVMS usage:	
type:	word integer
access:	write only
mechanism:	by reference

Optional argument.

Number of bytes that ENCRYPT$DECRYPT wrote to the output buffer. The **output-length** argument is the address of a word containing the number of bytes written to the output buffer, including any bytes

of pad characters generated by the selected algorithm to meet length requirements of the input buffer, if any. Output length does not count padding in the case of a fixed-length string.

Some encryption algorithms have specific requirements for the length of the input and output strings. In particular, DESECB and DESCBC pad input data with from 1 to 7 bytes to form complete 64-bit blocks for operation. The values of the pad characters are indeterminate.

When you decrypt fewer than 8 bytes, present the full 8 bytes resulting from the ENCRYPT$ENCRYPT to ENCRYPT$DECRYPT. Retain the byte count of the input data in order to strip trailing pad bytes after a subsequent decryption operation. Note that the AES block mode algorithms (AESCBCxxx and AESECBxxx), pad the data to even 16 byte block boundaries. For AES, one byte encrypts and decrypts to 16 bytes, 72 bytes to 80, and so forth. The AES padding character is a HEX number of bytes indicating the number of bytes padded, for example, the one byte encrypted pad would decrypt to 15 characters of 0F following the one decrypted byte of data. For the 72 bytes of data, eight bytes of padding characters (08 08 ... 08), would follow the 72 bytes of decrypted data. DESECB and DESCBC modes always pad with characters of zeros. The character stream modes (AESCFBxxx, AESOFBxxx, DESCFB), do not pad the data, so the output-length will match the actual number of data bytes.

p1

OpenVMS usage:	
type:	quadword[1](DES), quadword[2](AES)
access:	read only
mechanism:	by reference

Optional argument. The p1 argument is the address of a quadword initialization vector used to seed the two modes of the DES algorithm for which it is applicable (DESECB and DESCFB). (That is, the DES IV initialization vector is a quadword reference, to an eight byte value.)

For AES, the optional P1 argument for the AES IV initialization vector is a reference to a 16 byte (two quadwords) value.

If this argument is omitted, the initialization vector used is the residue of the previous use of the specified context block. ENCRYPT$INIT initializes the context block with an initialization vector of zero.

Description

The ENCRYPT$DECRYPT routine decrypts the next record of ciphertext according to the algorithm specified in the ENCRYPT$INIT call. Any errors encountered in the operation are returned as status values. The message authentication mode (DESMAC) is not supported by ENCRYPT$DECRYPT.

The ENCRYPT$DECRYPT routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Record successfully decrypted.

ENCRYPT$ *xyz*

An error reported by the Encryption software. *xyz* identifies the message.

SS$_*xyz*

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$DECRYPT_ONE_RECORD

ENCRYPT$DECRYPT_ONE_RECORD — Decrypts a small amount of data on a decrypt stream. To use AES for one record ciphers, you must first create an AES key, which is stored in the logical name table (encrypted). The key name of an AES key is specified as an address of a descriptor that contains the ASCII text for the selected AESmmmkkk (mode and key size) algorithm, for example, AESCBC256. The input and output buffers (descriptor addresses) are also provided.

Format

ENCRYPT$DECRYPT_ONE_RECORD input, output, key-name, algorithm

Argument

input

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Ciphertext record to be decrypted. The **input** argument is the address of a string descriptor pointing to a byte-aligned buffer containing the input record to be decrypted.

output

OpenVMS usage:	
type:	char_string
access:	write only
mechanism:	by descriptor

Plaintext record resulting when ENCRYPT$DECRYPT_ONE_RECORD completes execution. The **output** argument is the address of a string descriptor pointing to a byte-aligned buffer that will contain the plaintext record.

If the descriptor is dynamic and insufficient space is allocated to contain the output record, storage is allocated from dynamic memory. If insufficient space exists to contain the output of the operation, an error is returned.

The ENCRYPT$DECRYPT_ONE_RECORD routine adjusts the length of the output descriptor, if possible, to reflect the actual length of the output string.

key-name

OpenVMS usage:	
type:	char_string
access:	read only

mechanism:	by descriptor

Key used to initialize the decrypt stream. The **key-name** argument is the address of a string descriptor pointing to the name of the previously defined user key to be used.

algorithm

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Algorithm used for the decryption operation. The **algorithm** argument is the address of a string descriptor pointing to a code for the selected algorithm. The algorithm code is an ASCII string. Specify the descriptor type value as one of the following:

- DSC$K_DTYPE_T (text)
- DSC$K_DTYPE_VT (varying text)
- DSC$K_DTYPE_Z (unspecified)

For DES, the following algorithms are valid:

- DESCBC (default)
- DESECB
- DESCFB

For AES, the following algorithms are valid:

- Cipher block chaining:

  AESCBC128 (default)
  AESCBC192
  AESCBC256

- Electronic code book:

  AESECB128
  AESECB192
  AESECB256

- Cipher feedback:

  AESCFB128
  AESCFB192
  AESCFB256

- Output feedback:

  AESOFB128
  AESOFB192
  AESOFB256

Description

In some applications, only a small amount of data needs to be decrypted on a particular decrypt stream. The ENCRYPT$DECRYPT_ONE_RECORD routine allows you to perform such a decryption operation.

The ENCRYPT$DECRYPT_ONE_RECORD routine is a shorthand form of the ENCRYPT $INIT, ENCRYPT$DECRYPT, and ENCRYPT$FINI sequence of calls. However, using ENCRYPT $DECRYPT_ONE_RECORD repeatedly to decrypt records of a file is extremely inefficient.

The ENCRYPT$DECRYPT_ONE_RECORD routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Operation performed.

ENCRYPT$ xyz

An error reported by the Encryption software. *xyz* identifies the message.

SS$_ xyz

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$DEFINE_KEY

ENCRYPT$DEFINE_KEY — Places a key definition into the process, group, job, or system key storage table.

Format

`ENCRYPT$DEFINE_KEY` key-name, key-value, key-flags

Argument

key-name

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Name of the key defined when ENCRYPT$DEFINE_KEY completes execution. The **key-name** argument is the address of a string descriptor pointing to a char_string that is interpreted as the name of the key to be defined. A maximum of 243 characters is permitted.

Note

Key names beginning with ENCRYPT$ are reserved for VSI.

key-value

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Key value defined when ENCRYPT$DEFINE_KEY completes execution. The *key-value* argument is the address of a string descriptor pointing to a vector of unsigned byte values that are assigned to the named key. A maximum of 240 bytes may be assigned.

key-flags

OpenVMS usage:	
type:	longword
access:	read only
mechanism:	by reference

Flags that ENCRYPT$DEFINE_KEY uses when defining a key. The **key-flags** argument is the address of a longword containing flags that control the key definition process.

Each flag has a symbolic name. The constants associated with these names are defined in the ENCRYPT $EXAMPLES:ENCRYPT_STRUCTURES files in various programming languages.

Table 11.3 defines the function of each flag.

Table 11.3. ENCRYPT$DEFINE_KEY Flags

Flag	**Function**
Symbolic Name	**Function**
ENCRYPT$M_KEY_PROCESS	Places definition in process table
ENCRYPT$M_KEY_GROUP	Places definition in group table
ENCRYPT$M_KEY_JOB	Places definition in job table
ENCRYPT$M_KEY_SYSTEM	Places definition in system table
ENCRYPT$M_KEY_LITERAL	Stores key without compressing
ENCRYPT$M_KEY_AES	Designates an AES key value

The following AES mask can be used in addition to (OR with) other flags for the key-flags parameter (as a longword by reference). An associated AES key value can be used for testing the bit within the program. Use the KEY_AES key flag to specify an AES key:

- ENCRYPT$M_KEY_AES
- ENCRYPT$V_KEY_AES

Description

The ENCRYPT$DEFINE_KEY routine places a key definition into the process, group, job, or system key storage table. The key value supplied with the routine is processed as specified and placed in the key storage table under the indicated name. The ENCRYPT$DEFINE_KEY routine does not interpret the key value.

By default, DES keys are treated as char_string keys, using the Digital Multinational Character Set and are compressed before being inserted into the key storage table. The compression proceeds as follows:

1. The string is converted to uppercase characters.
2. The digits 0 through 9 are left unchanged.
3. All characters except letters, digits, dollar signs, periods, and underscores are converted to spaces.
4. All sequences of multiple spaces (or characters that have been converted into spaces) are converted into single spaces.

When a char_string key is retrieved from key storage for use as a DES key, it is folded into an 8-byte key by exclusive OR-ing 8-byte segments of the key string together, and then applying odd parity to each byte by modifying the sign bit (bit 7).

The key flag ENCRYPT$M_KEY_LITERAL specifies that the key string supplied is a binary key. A binary key is not compressed, but is placed into key storage as is. When a binary key is used as a DES key, it is likewise folded into an 8-byte key by exclusive OR-ing 8-byte segments together. For DES, odd parity is then applied by modifying the low bit (bit 0) of each byte.

AES key values are not subject to ASCII compression. Therefore, any 8 bit character is allowed for AES keys.

The ENCRYPT$DEFINE_KEY routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Key has been defined.

ENCRYPT$ ***xyz***

An error reported by the Encryption software. *xyz* identifies the message.

SS$_*xyz*

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$DELETE_KEY

ENCRYPT$DELETE_KEY — Deletes a key definition from a key storage table.

Format

ENCRYPT$DELETE_KEY key-name, key-flags

Argument

key-name

OpenVMS usage:

type:	char_string
access:	read only
mechanism:	by descriptor

Name of the key removed from a key storage table when ENCRYPT$DELETE_KEY completes execution. The **key-name** argument is the address of a string descriptor pointing to a char_string that is interpreted as the name of the key to be deleted. A maximum of 243 characters is permitted.

key-flags

OpenVMS usage:	
type:	longword
access:	read only
mechanism:	by reference

Key table from which ENCRYPT$DELETE_KEY removes a key. The key-flags argument is a longword containing flags that control the deletion process. The following flags are available:

ENCRYPT$M_KEY_PROCESS	Deletes a key from process table
ENCRYPT$M_KEY_GROUP	Deletes a key from group table
ENCRYPT$M_KEY_JOB	Deletes a key from job table
ENCRYPT$M_KEY_SYSTEM	Deletes a key from system table
ENCRYPT$M_KEY_AES	Designates an AES key value

The following AES mask can be used in addition to (or with) other flags for the key-flags parameter (as a longword by reference). An associated AES key value can be used for testing the bit within the program. Use the KEY_AES key flag to specify an AES key:

- ENCRYPT$M_KEY_AES
- ENCRYPT$V_KEY_AES

Description

The ENCRYPT$DELETE_KEY routine deletes a key definition from a key storage table. The ENCRYPT$DELETE_KEY routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Key has been deleted.

ENCRYPT$ *xyz*

An error reported by the Encryption software. *xyz* identifies the message.

SS$_ *xyz*

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$ENCRYPT

ENCRYPT$ENCRYPT — Transforms the next record of plaintext according to the algorithm you specify in the ENCRYPT$INIT call. This routine performs either an encryption or decryption operation.

Format

ENCRYPT$ENCRYPT context, input, output [,output-length] [,p1]

Argument

context

OpenVMS usage:	
type:	longword integer (signed)
access:	write only
mechanism:	by reference

Context area initialized when ENCRYPT$INIT completes execution. The **context** argument is the address of a longword of unspecified interpretation that is used to convey context between encryption operations.

input

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Plaintext record to encrypt. The **input** argument is the address of a descriptor pointing to a byte-aligned buffer containing the input record to the encryption operation.

output

OpenVMS usage:	
type:	char_string
access:	write only by descriptor
mechanism:	

Ciphertext record that results when ENCRYPT$ENCRYPT completes execution. The **output** argument is the address of a descriptor pointing to a byte-aligned buffer that will contain the output record from the encryption operation.

If the descriptor is dynamic and insufficient space is allocated to contain the output record, storage is allocated from dynamic memory.

ENCRYPT$ENCRYPT adjusts the length of the output descriptor, if possible, to reflect the actual length of the output string. If the descriptor type is not DSC$K_DTYPE_VS (varying string), DSC $K_DTYPE_V (varying), or DSC$K_DTYPE_D (dynamic), the routine takes the actual output count from the **output-length** argument.

The output buffer must be able to accommodate a padded block to an increment of the block length. For AES this is 16 bytes and for DES, 8 bytes.

output-length

OpenVMS usage:	
type:	word integer
access:	write only
mechanism:	by reference

Optional argument. Number of bytes that ENCRYPT$ENCRYPT wrote to the output buffer. The **output-length** argument is the address of a word containing the number of bytes written to the output buffer.

Some encryption algorithms have specific requirements for the length of the input and output strings. In particular, DESECB and DESCBC pad input data with from 1 to 7 bytes to form complete 64-bit blocks for operation. The values of the pad characters are indeterminate.

When you decrypt fewer than 8 bytes, preserve and present to ENCRYPT$DECRYPT the full 8 bytes resulting from ENCRYPT$ENCRYPT. Retain the byte count of the input data in order to strip trailing pad bytes after a subsequent decryption operation.

Note that the AES block mode algorithms (AESCBCxxx and AESECBxxx) pad the data to even 16 byte block boundaries. For AES, one byte encrypts and decrypts to 16 bytes, 72 bytes to 80, and so forth. The AES padding character is a HEX number of bytes indicating the number of bytes padded. For example, the one-byte encrypted pad would decrypt to 15 characters of 0F following the one encrypted byte of data. For the 72 bytes of data, eight bytes of padding characters (08 08 ... 08), would follow the 72 bytes of encrypted data. DESECB and DESCBC modes always pad with characters of zeros. The character stream modes (AESCFBxxx, AESOFBxxx, DESCFB). In order that the output-length will match the actual number of data bytes, do not pad the data.

p1

OpenVMS usage:	
type:	quadword[1] (DES), quadword[2] (AES)
access:	read only
mechanism:	by reference

Optional argument. The p1 argument is the address of a quadword initialization vector used to seed the three modes (DESECB, DESCFB, and DESMAC) of the DES algorithm for which it is applicable. The DES IV initialization vector is a quadword reference, to an eight byte value.

For AES, the optional P1 argument for the AES IV initialization vector is a reference to a 16 byte (two quadwords) value.

If you omit this argument, the initialization vector used is the residue of the previous use of the specified context block. ENCRYPT$INIT initializes the context block with an initialization vector of zero.

Description

The ENCRYPT$ENCRYPT routine transforms the next record of plaintext according to the algorithm specified in the ENCRYPT$INIT call. Any errors encountered in the operation are returned as status

values. The ENCRYPT$ENCRYPT routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Record successfully encrypted.

ENCRYPT$ ***xyz***

An error reported by the Encryption software. *xyz* identifies the message.

SS$_*xyz*

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$ENCRYPT_FILE

ENCRYPT$ENCRYPT_FILE — Encrypts or decrypts data files.

Format

`ENCRYPT$ENCRYPT_FILE` input-file, output-file, key-name, algorithm, file-flags [,item-list]

Argument

input-file

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Name of the input file that ENCRYPT$ENCRYPT_FILE is to process. The **input-file** argument is the address of a string descriptor pointing to the file specification string for the input file.

Wildcard characters are valid. To specify multiple input files, you must use wildcard characters.

output-file

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Name of the output file that ENCRYPT$ENCRYPT_FILE is to generate. The **output-file** argument is the address of a string descriptor pointing to the file specification for the output file to be processed.

You can use wildcard characters. To specify the same names for the output and input files, use a null character as the **output-file** argument.

key-name

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Name of the key used when ENCRYPT$ENCRYPT_FILE processes files. The **key-name** argument is the address of a string descriptor pointing to the name of the key to be used in initializing the encrypt or decrypt stream used for each file processed.

algorithm

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Name of the algorithm that ENCRYPT$ENCRYPT_FILE uses to initialize the process stream. The **algorithm** argument is the address of a string descriptor pointing to the name of the algorithm.

For DES, the following algorithms are valid:

- DESCBC (default)
- DESECB
- DESCFB

For AES, the following algorithms are valid:

- Cipher block chaining:

  AESCBC128 (default)
  AESCBC192
  AESCBC256

- Electronic code book:

  AESECB128
  AESECB192
  AESECB256

- Cipher feedback:

  AESCFB128
  AESCFB192
  AESCFB256

- Output feedback:

  AESOFB128
  AESOFB192

AESOFB256

file-flags

OpenVMS usage:	
type:	longword
access:	read only
mechanism:	by reference

Flags that specify how ENCRYPT$ENCRYPT_FILE performs the file operation. The **file-flags** argument is the address of a longword containing a mask of flags. Table 11.4 shows the function of each flag.

Table 11.4. ENCRYPT$ENCRYPT_FILE Flags

Flag	Function
ENCRYPT$M_FILE_COMPRESS	Compresses file data before encryption.
ENCRYPT$M_FILE_ENCRYPT	Flag set: encrypts the file. Flag clear: decrypts the file.
ENCRYPT$M_FILE_DELETE	Deletes the input file when the operation completes.
ENCRYPT$M_FILE_ERASE	Erases the file with the security data pattern before deleting it.
ENCRYPT$M_FILE_KEY_VALUE	Flag set: Treats the key value as a literal value and does not compress it. Flag clear: Treats the key value as a text string that can be compressed. If the KEY_NAME parameter is present, this flag is ignored.
ENCRYPT$M_FILE_AES	Flag set: indicates encrypting a file with an AES key and algorithm

There is an additional FILE_AES flag mask (and value) that is used with the ENCRYPT $ENCRYPT_FILE routine when encrypting files using an AES algorithm. The ENCRYPT $ENCRYPT_FILE_FLAGS are used to control file operations such as cipher direction, file compression and so on. The FILE_AES flag controls file AES initialization and cipher operation.

item-list

OpenVMS usage:	
type:	item_list_3
access:	read only
mechanism:	by descriptor

The optional **item-list** argument is used to override the data algorithm parameter. This argument substitutes one algorithm for another that is similar in function but that may be different in its name. In other words, it overrides the name of the algorithm that is found in the random key record with the name of the algorithm you provided in the override descriptor. This process provides a way to open files that were encrypted with an algorithm name that may be different than the algorithm name in the decrypt environment.

ENCRYPT$K_DATA_ALGORITHM

OpenVMS usage:	
type:	3 longwords
access:	read only
mechanism:	by descriptor

Algorithm to be used to encrypt the file. This argument specifies the address and length of the name string of the algorithm.

The following algorithms are valid:

- DESCBC (default)
- DESECB
- DESCFB

For AES, the following algorithms are valid:

- Cipher block chaining:

  AESCBC128 (default)
  AESCBC192
  AESCBC256

- Electronic code book:

  AESECB128
  AESECB192
  AESECB256

- Cipher feedback:

  AESCFB128
  AESCFB192
  AESCFB256

- Output feedback:

  AESOFB128
  AESOFB192
  AESOFB256

Description

The ENCRYPT$ENCRYPT_FILE routine either encrypts or decrypts data files from within an application.

The routine uses the user key and the specified algorithm to protect only the randomly generated key and the initialization vector that are used with the DESCBC algorithm to encrypt the file.

The ENCRYPT$ENCRYPT_FILE routine returns a 32-bit status code indicating the success or failure of the routine's operation.

When you use this routine, do not also use ENCRYPT$INIT or ENCRYPT$FINI.

Condition Values Returned

SS$_NORMAL

Record successfully encrypted.

ENCRYPT$ *xyz*

An error reported by the Encryption software. *xyz* identifies the message.

SS$_ *xyz*

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$ENCRYPT_ONE_RECORD

ENCRYPT$ENCRYPT_ONE_RECORD — Encrypts a small amount of data in an encrypt stream. To use AES for one record ciphers, you must first create an AES key, which is stored in the logical name table (encrypted). The key name of an AES key is specified as an address of a descriptor that contains the ASCII text for the selected AESmmmkkk (mode and key size) algorithm, for example, AESCBC256. The input and output buffers (descriptor addresses) are also provided.

Format

ENCRYPT$ENCRYPT_ONE_RECORD input, output, key-name, algorithm

Argument

input

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Plaintext record to be encrypted. The input argument is the address of a string descriptor pointing to a byte-aligned buffer containing the input record to be encrypted.

output

OpenVMS usage:	
type:	char_string
access:	write only
mechanism:	by descriptor

Ciphertext record resulting when the routine completes execution. The **output** argument is the address of a string descriptor pointing to a byte-aligned buffer that will contain the ciphertext record.

If the descriptor is dynamic, and insufficient space is allocated to contain the output record, storage is allocated from dynamic memory. If insufficient space exists to contain the output of the operation, an error is returned.

The ENCRYPT$ENCRYPT_ONE_RECORD routine adjusts the length of the output descriptor, if possible, to reflect the actual length of the output string.

key-name

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Key used to initialize the encrypt stream. The **key-name** argument is the address of a string descriptor pointing to the name of the previously defined user key to be used.

algorithm

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Algorithm used for the encryption operation. The **algorithm** argument is the address of a string descriptor pointing to a code for the selected algorithm. The algorithm code is an ASCII string. For descriptor type value, use one of the following:

- DSC$K_DTYPE_T (text)
- DSC$K_DTYPE_VT (varying text)
- DSC$K_DTYPE_Z (unspecified)

For DES, the following algorithms are valid:

- DESCBC (default)
- DESECB
- DESCFB

For AES, the following algorithms are valid:

- Cipher block chaining:

  AESCBC128 (default)
  AESCBC192
  AESCBC256

- Electronic code book:

  AESECB128
  AESECB192
  AESECB256

- Cipher feedback:

AESCFB128
AESCFB192
AESCFB256

- Output feedback:

  AESOFB128
  AESOFB192
  AESOFB256

Description

To encrypt only a small amount of data, use the ENCRYPT$ENCRYPT_ONE_RECORD routine.

The ENCRYPT$ENCRYPT_ONE_RECORD routine is a shorthand form of the ENCRYPT $INIT, ENCRYPT$ENCRYPT, and ENCRYPT$FINI sequence of calls. However, using ENCRYPT $ENCRYPT_ONE_RECORD repeatedly to encrypt records of a file is extremely inefficient.

The ENCRYPT$ENCRYPT_ONE_RECORD routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Operation performed.

ENCRYPT$ xyz

An error reported by the Encryption software. *xyz* identifies the message.

SS$_xyz

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$FINI

ENCRYPT$FINI — Disassociates the encryption context and releases it.

Format

ENCRYPT$FINI context

Argument

context

OpenVMS usage:	
type:	longword integer (signed)
access:	read/write
mechanism:	by reference

Context area terminated when ENCRYPT$FINI completes execution. The **context** argument is the address of a longword initialized by the ENCRYPT$INIT routine.

Description

The ENCRYPT$FINI routine disassociates the indicated encryption context and releases it. The ENCRYPT$FINI routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Encryption context successfully terminated.

ENCRYPT$ xyz

An error reported by the Encryption software. xyz identifies the message.

SS$_ xyz

A return status from a called system service. xyz identifies the return status.

ENCRYPT$GENERATE_KEY

ENCRYPT$GENERATE_KEY — Generates a random key value.

Format

ENCRYPT$GENERATE_KEY algorithm-name, key-length [,factor-a] [,factor-b] [,factor-c] [,key buffer]

Argument

algorithm-name

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

The name of the algorithm that will use the generated key.

key-length

OpenVMS usage:	
type:	word unsigned
access:	read only
mechanism:	by reference

Unsigned integer indicating the size of the key to be generated. The **key-length** argument is the address of an unsigned word containing a value that indicates the length of the key.

For AES, the key-length argument takes values as increments of AES block size: 16 bytes, 32, bytes, and 48 bytes, and so on.

factor-a, factor-b, factor-c

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Optional arguments. The **factor-a**, **factor-b**, and **factor-c** arguments are operation-dependent data used as randomizing factors when the routine generates a key value. For example, the factors might be:

- Time an operation started
- Size of a certain stack
- Copy of the last command line

key-buffer

OpenVMS usage:	
type:	char_string
access:	write
mechanism:	by descriptor

Buffer into which the generated key is to be placed. The **key-buffer** argument is the address of a string descriptor referencing the appropriate buffer.

If you specify a class D descriptor, dynamic memory is allocated to contain the entire key.

Description

The ENCRYPT$GENERATE_KEY routine generates a random key value. The ENCRYPT $GENERATE_KEY routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Key has been created.

ENCRYPT$ xyz

An error reported by the Encryption software. *xyz* identifies the message.

SS$_ xyz

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$INIT

ENCRYPT$INIT — Initializes the context for the encryption operation.

Format

ENCRYPT$INIT context, algorithm, key-type, key-name [,p1]

Argument

context

OpenVMS usage:	
type:	longword integer signed
access:	write only
mechanism:	by reference

Context area that is initialized. The **context** argument is the address of a longword of unspecified interpretation that is used to convey context between encryption operations. An uninitialized context longword is defined to be zero and is initialized to nonzero by this routine. The context area itself is allocated from process dynamic memory.

algorithm

OpenVMS usage:	
type:	char_string
access:	read/write
mechanism:	by descriptor

Algorithm used for the encryption operation. The **algorithm** argument is the address of a string descriptor pointing to a code for the selected algorithm. The algorithm code is an ASCII string. For descriptor type value, use one of the following:

DSC$K_DTYPE_T (text)
DSC$K_DTYPE_VT (varying text)
DSC$K_DTYPE_Z (unspecified)

For DES, the following algorithms are valid:

- DESCBC (default)
- DESECB
- DESCFB

For AES, the following algorithms are valid:

- Cipher block chaining:

  AESCBC128 (default)
  AESCBC192
  AESCBC256

- Electronic code book:

  AESECB128
  AESECB192
  AESECB256

- Cipher feedback:

  AESCFB128

AESCFB192
AESCFB256

- Output feedback:

  AESOFB128
  AESOFB192
  AESOFB256

key-type

OpenVMS usage:	
type:	longword logical unsigned
access:	read only
mechanism:	by reference

Code specifying how ENCRYPT$INIT is to interpret the **key-name** argument. The **key-type** argument is the address of an unsigned longword indicating whether key-name is the name of the key or the key value. If you specify:

Key-type as 0	ENCRYPT$INIT interprets **key-name** as a descriptor pointing to the key name string.
Key-type as 1	ENCRYPT$INIT interprets **key-name** as the descriptor for the value of the key to be used.

key-name

OpenVMS usage:	
type:	char_string
access:	read only
mechanism:	by descriptor

Key that ENCRYPT$INIT passes to the selected encryption routine. The **key-name** argument is the address of a character string descriptor containing the name of the key or the address of the actual key value. ENCRYPT$INIT interprets this argument based on the value of key-type. If this argument is:

The key name	Actual key value is retrieved from key storage by the selected encryption routine.
A key value	It is stored with a temporary name, which is passed to the selected encryption routine.

If the **key-name** argument is used to specify a key value (that is, if key-type has been specified as 1), the key-name string descriptor type field determines whether the key value is to be treated as a char_string or as a binary value to be used exactly as specified.

If the descriptor type is DSC$K_DTYPE_T (char_string), DSC$K_DTYPE_VT (varying char_string), or DSC$K_DTYPE_Z (unspecified), the value is treated as a text string to be compressed for DES key values. ASCII compression converts lowercase characters to uppercase, only A--Z, 0--9, $, . (period), and _ (underscore) are allowed. Other characters are converted to spaces, and the extra spaces are removed. AES ASCII key values are not subject to ASCII compression, allowing any 8-bit ASCII character.

All other descriptor types are treated as though the key value is to be used exactly as specified.

Note

The key name descriptors of type DSCK_DTYPE_T, DSCK_DTYPE_VT, and DSC$K_DTYPE_Z all specify that the key value should be compressed. For OpenVMS V8.3, this functionality applies only to DES, not AES. AES keys are not compressed.

p1

OpenVMS usage:	
type:	quadword[1] (DES), quadword[2] (AES)
access:	read only
mechanism:	by reference

Optional argument. The p1 argument is the address of a quadword initialization vector used to seed the three modes of the DES algorithm that uses an initialization vector. These modes are: DESCBC (default), DESCFB, and DESMAC. That is, the DES IV initialization vector is a quadword reference, to an eight byte value.

For AES, the optional P1 argument for the AES IV initialization vector is a reference to a 16 byte (two quadwords) value.

If you omit this argument, the initialization vector used is the residue of the previous use of the specified context block. ENCRYPT$INIT initializes the context block with an initialization vector of zero.

Description

ENCRYPT$INIT initializes the context for the encryption operation. ENCRYPT$INIT creates pre-initialized key tables in the context area to speed the encryption or decryption process. Before you can re-use a context with a new algorithm, key, or other values specified with ENCRYPT$INIT, terminate the old context with a call to ENCRYPT$FINI.

Note

Always initialize the context with ENCRYPT$INIT when you change the operation from encryption to decryption, or from decryption to encryption.

ENCRYPT$INIT returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Initialization successfully completed.

ENCRYPT$ xyz

An error reported by the Encryption software. *xyz* identifies the message.

SS$_ xyz

A return status from a called system service. *xyz* identifies the return status.

ENCRYPT$STATISTICS

ENCRYPT$STATISTICS — Gains access to the statistics maintained by the Encryption software.

Format

ENCRYPT$STATISTICS context, code, destination, return-length

Argument

context

OpenVMS usage:	
type:	longword
access:	read only
mechanism:	by reference

Context area initialized by ENCRYPT$INIT. The **context** argument is the address of a longword initialized by the ENCRYPT$INIT routine.

code

OpenVMS usage:	
type:	longword
access:	read only
mechanism:	by reference

Code specifying the desired statistic. The **code** argument is the address of a longword containing the code. The only accepted value is 1, which indicates that ENCRYPT$STATISTICS is to return all statistics to the destination buffer.

destination

OpenVMS usage:	
type:	char_string
access:	write only
mechanism:	by descriptor

Buffer into which ENCRYPT$STATISTICS places the statistics. The **destination** argument is the address of a string descriptor describing the buffer. Ensure that the destination buffer is at least 20 bytes long and contains:

- One longword indicating the number of times the primitive has been entered referencing this encryption stream
- One quadword indicating the total bytes processed for this stream
- One quadword indicating the total CPU time, in OpenVMS time format, spent on processing requests for this stream

return-length

OpenVMS usage:	
type:	longword
access:	write only
mechanism:	by reference

Number of bytes written to the destination buffer. The **return-length** argument is the address of a word containing the number of bytes.

Description

To track the progress and performance of an encryption operation, the Encryption for OpenVMS software maintains statistics in the context area. You can access these statistics with the ENCRYPT $STATISTICS routine. The ENCRYPT$STATISTICS routine returns a 32-bit status code indicating the success or failure of the routine's operation.

Condition Values Returned

SS$_NORMAL

Statistics returned.

ENCRYPT$ xyz

An error reported by the Encryption software. *xyz* identifies the message.

SS$_ xyz

A return status from a called system service. *xyz* identifies the return status.

Chapter 12. File Definition Language (FDL) Routines

This chapter describes the File Definition Language (FDL) routines. These routines perform many of the functions of the File Definition Language that define file characteristics. Typically, you use FDL to perform the following operations:

- Specify file characteristics otherwise unavailable from your language.
- Examine or modify the file characteristics of an existing data file to improve program or system interaction with that file.

12.1. Introduction to FDL Routines

You specify FDL attributes for a data file when you use FDL to create the data file, set the desired file characteristics, and close the file. You can then use the appropriate language statement to reopen the file. Because the data file is closed between the time the FDL attributes are set and the time your program accesses the file, you cannot use FDL to specify run-time attributes (attributes that are ignored or deleted when the associated data file is closed).

The FDL$CREATE routine is the one most likely to be called from a high-level language. It creates a file from an FDL specification and then closes the file. The following VSI Fortran program segment creates an empty data file named INCOME93.DAT using the file characteristics specified by the FDL file INCOME.FDL. The `STATEMENT` variable contains the number of the last FDL statement processed by FDL$CREATE; this argument is useful for debugging an FDL file.

```
INTEGER STATEMENT
INTEGER STATUS,
2       FDL$CREATE

STATUS = FDL$CREATE ('INCOME.FDL',
2                    'INCOME93.DAT',
2                    ,,,,
2                    STATEMENT,
2                    ,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
    .
    .
    .
```

The following three FDL routines provide a way to specify all the options OpenVMS RMS allows when it executes create, open, or connect operations. They also allow you to specify special processing options required for your applications.

- The FDL$GENERATE routine produces an FDL specification by interpreting a set of RMS control blocks in an existing data file. It then writes the FDL specification either to an FDL file or to a character string. If your programming language does not provide language statements that access RMS control blocks (for example, VSI Fortran), you must use FDL$GENERATE from within the context of a user-open routine to generate an FDL file.
- The FDL$PARSE routine parses an FDL specification, allocates RMS control blocks, and fills in the relevant fields.

- The FDL$RELEASE routine deallocates the virtual memory used by the RMS control blocks created by FDL$PARSE.

These routines cannot be called from asynchronous system trap (AST) level. In addition, in order to function properly, these routines require ASTs to remain enabled.

An FDL specification can be in either a file or a character string. When specifying an FDL specification in a character string, use semicolons to delimit the statements of the FDL specification.

12.2. Using the FDL Routines: Examples

This section provides examples that demonstrate the use of the FDL routines in various programming scenarios.

- Example 12.1 shows how to use the FDL$CREATE routine in a Fortran program.
- Example 12.2 shows how to use the FDL$PARSE and FDL$RELEASE routines in a C program.
- Example 12.3 shows a VSI Pascal program that uses the FDL$PARSE routine to fill in the RMS control blocks in a data file. The program then uses the FDL$GENERATE routine to create an FDL file using the information in the control blocks.

Example 12.1. Using FDL$CREATE in a Fortran Program

```
*       This program calls the FDL$CREATE routine.  It
*       creates an indexed output file named NEW_MASTER.DAT
*       from the specifications in the FDL file named
*       INDEXED.FDL.  You can also supply a default filename
*       and a result name (that receives the name of the
*       created file).  The program also returns all the
*       statistics.
*
        IMPLICIT        INTEGER*4       (A - Z)
        EXTERNAL        LIB$GET_LUN,    FDL$CREATE
        CHARACTER       IN_FILE*11      /'INDEXED.FDL'/,
        1               OUT_FILE*14     /'NEW_MASTER.DAT'/,
        1               DEF_FILE*11     /'DEFAULT.FDL'/,
        1               RES_FILE*50
        INTEGER*4       FIDBLK(3)       /0,0,0/
        I = 1
        STATUS = FDL$CREATE (IN_FILE,OUT_FILE,
                 DEF_FILE,RES_FILE,FIDBLK,,)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        STATUS=LIB$GET_LUN(LOG_UNIT)
        OPEN (UNIT=LOG_UNIT,FILE=RES_FILE,STATUS='OLD')
        CLOSE (UNIT=LOG_UNIT, STATUS='KEEP')

        WRITE (6,1000) (RES_FILE)
        WRITE (6,2000) (FIDBLK (I), I=1,3)

1000    FORMAT  (1X,'The result filename is: ',A50)
```

```
2000    FORMAT  (/1X,'FID-NUM: ',I5/,
        1         1X,'FID-SEQ: ',I5/,
        1         1X,'FID-RVN: ',I5)

        END
```

Example 12.2 shows how to use the FDL$PARSE and FDL$RELEASE routines in a C program.

Example 12.2. Using FDL$PARSE and FDL$RELEASE in a C Program

```
/*  FDLEXAM.C
**  This program calls the FDL utility routines FDL$PARSE and
**  FDL$RELEASE.  First, FDL$PARSE parses the FDL specification
**  PART.FDL.  Then the data file named in PART.FDL is accessed
**  using the primary key.  Last, the control blocks allocated
**  by FDL$PARSE are released by FDL$RELEASE.
**  Note; to try this program use the following command on any
**  file with textual data:  $ANALYZE/RMS/FDL/OUT=PART.FDL
*/

#include <descrip>
#include <rms>
#define REC_SIZE 80     /* as appropriate for files used */

FDLEXAM ()
{

struct FAB *fab_ptr;  /* variable to hold pointer to FAB structure */
struct RAB *rab_ptr;  /* variable to hold pointer to RAB structure */
$DESCRIPTOR (fdl_file, "PART.FDL");         /* free choice of name */
char record_buffer[REC_SIZE+1];       /* allow for null terminator */
int stat;

/*
** Read and parse FDL file allocating and initializing RAB and
** and FAB accordingly, returning pointers to the FAB & RAB.
*/
stat = FDL$PARSE ( &fdl_file, &fab_ptr, &rab_ptr );
if (!(stat & 1)) LIB$STOP ( stat );

/*
** Try to open file as described by information in the FAB.
** Signal open errors. Note the usage of STAT, instead of
** FAB_PTR->FAB$L_STS because just in case the FAB is invalid,
** the only status returned is STAT.
*/
stat = SYS$OPEN ( fab_ptr );
if (!(stat & 1)) LIB$STOP ( stat, fab_ptr->fab$l_stv );

stat = SYS$CONNECT ( rab_ptr );
if (!(stat & 1)) LIB$STOP ( stat, rab_ptr->rab$l_stv );

/*
** Opened the file and connect some internal buffers.
** Fill in the record output buffer information which is the only
** missing information in the RAB that was created for us by FDL.
** Print a header recod and perform the initial $GET.
*/
```

```
rab_ptr->rab$w_usz = REC_SIZE;
rab_ptr->rab$l_ubf = record_buffer;
printf ("------------------- start of records -------------- \n");
stat = SYS$GET ( rab_ptr );
while (stat & 1)      /* As long as the $GET is successful */
    {
    record_buffer[rab_ptr->rab$w_rsz] = 0;  /* Terminate for printf */
    printf ("%s\n", record_buffer);     /* Current record */
    stat = SYS$GET ( rab_ptr );      /* Try to get next one */
    }

/*
** At this point in the execution, the status should be EOF indicating
** Successfully read the file to end. If not, signal real error.
*/
if (stat != RMS$_EOF) LIB$STOP ( rab_ptr->rab$l_sts, rab_ptr->rab$l_stv );

printf ("-------------------- end of records --------------- \n");
stat = SYS$CLOSE ( fab_ptr );  /* implicit $DISCONNECT */
if (!(stat & 1)) LIB$STOP ( fab_ptr->fab$l_sts, fab_ptr->fab$l_stv );

/*
** Allow FDL to release the FAB and RAB structures and any other
** structures (XAB) that it allocated on behalf of the program.
** Return with its status as final status (success or failure).
*/
return FDL$RELEASE ( &fab_ptr, &rab_ptr );
}
```

Example 12.3 shows a VSI Pascal program that uses the FDL$PARSE routine to fill in the RMS control blocks in a data file, and then uses the FDL$GENERATE routine to create an FDL file.

Example 12.3. Using FDL$PARSE and FDL$GENERATE in a VSI Pascal Program

```
[INHERIT ('SYS$LIBRARY:STARLET')]
PROGRAM FDLexample (input,output,order_master);

(* This program fills in its own FAB, RAB, and        *)
(* XABs by calling FDL$PARSE and then generates       *)
(* an FDL specification describing them.              *)
(* It requires an existing input FDL file             *)
(* (TESTING.FDL) for FDL$PARSE to parse.              *)
TYPE
(*+                                                   *)
(* FDL CALL INTERFACE CONTROL FLAGS                   *)
(*-                                                   *)
        $BIT1 = [BIT(1),UNSAFE] BOOLEAN;

        FDL2$TYPE = RECORD CASE INTEGER OF
        1: (FDL$_FDLDEF_BITS : [BYTE(1)] RECORD END;
            );
        2: (FDL$V_SIGNAL : [POS(0)] $BIT1;
              (* Signal errors; don't return          *)
            FDL$V_FDL_STRING : [POS(1)] $BIT1;
              (* Main FDL spec is a char string       *)
            FDL$V_DEFAULT_STRING : [POS(2)] $BIT1;
              (* Default FDL spec is a char string    *)
```

```
            FDL$V_FULL_OUTPUT : [POS(3)] $BIT1;
              (* Produce a complete FDL spec          *)
            FDL$V_$CALLBACK : [POS(4)] $BIT1;
              (* Used by EDIT/FDL on input (DEC only) *)
            )
        END;

    mail_order =  RECORD
                  order_num : [KEY(0)] INTEGER;
                  name : PACKED ARRAY[1..20] OF CHAR;
                  address : PACKED ARRAY[1..20] OF CHAR;
                  city : PACKED ARRAY[1..19] OF CHAR;
                  state : PACKED ARRAY[1..2] OF CHAR;
                  zip_code : [KEY(1)] PACKED ARRAY[1..5]
                       OF CHAR;
                  item_num : [KEY(2)] INTEGER;
                  shipping : REAL;
                  END;

    order_file  = [UNSAFE] FILE OF mail_order;
    ptr_to_FAB  = ^FAB$TYPE;
    ptr_to_RAB  = ^RAB$TYPE;
    byte = 0..255;

VAR
    order_master : order_file;
    flags        : FDL2$TYPE;
    order_rec    : mail_order;
    temp_FAB     : ptr_to_FAB;
    temp_RAB     : ptr_to_RAB;
    status       : integer;

FUNCTION FDL$PARSE
    (%STDESCR FDL_FILE : PACKED ARRAY [L..U:INTEGER]
         OF CHAR;
    VAR FAB_PTR : PTR_TO_FAB;
    VAR RAB_PTR : PTR_TO_RAB) : INTEGER; EXTERN;

FUNCTION FDL$GENERATE
    (%REF FLAGS : FDL2$TYPE;
    FAB_PTR : PTR_TO_FAB;
    RAB_PTR : PTR_TO_RAB;
    %STDESCR FDL_FILE_DST : PACKED ARRAY [L..U:INTEGER]
         OF CHAR) : INTEGER;
    EXTERN;

BEGIN

    status := FDL$PARSE ('TESTING',TEMP_FAB,TEMP_RAB);
    flags::byte := 0;
    status := FDL$GENERATE (flags,
                            temp_FAB,
                            temp_RAB,
                            'SYS$OUTPUT:');

END.
```

12.3. FDL Routines

This section describes the individual FDL routines.

Note that the **fdl_desc** and the **default_fdl_desc** arguments that are used as part of these routine calls are character strings that can be either of the following:

- A string descriptor pointing to a file that contains a specification
- A character string that is the actual specification

For additional details, see the descriptions of the individual routine calls.

FDL$CREATE

Create a File from an FDL Specification and Close the File — The FDL$CREATE routine creates a file from an FDL specification and then closes the file.

Format

```
FDL$CREATE fdl_desc [,filename] [,default_name] [,result_name] [,fid_block]
 [,flags] [,stmnt_num] [,retlen] [,sts] [,stv] [,default_fdl_desc]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

fdl_desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

The *fdl_desc* argument is one of the following:

- A character string descriptor pointing to a file containing the FDL specification to be parsed
- A character string containing the actual FDL specification

The choice depends on the application making the call. For example, if the application wants to create data files that are compatible with a PC application, it might create the following FDL file and name it TRANSFER.FDL:

```
FILE
        ORGANIZATION            sequential
RECORD
        FORMAT                  stream_lf
```

The application could then include the address of the FDL file as the *fdl_desc* argument to the FDL $PARSE call:

```
call fdl$parse transfer.fdl ,…
```

Optionally, the application might code the FDL specification itself into the call using a quoted character string as the *fdl_desc* argument:

```
call fdl$parse  "FILE; ORG SEQ; FORMAT STREAM_LF;" ,…
```

Note that directly including the FDL specification into the call requires you to do the following:

- Enclose the *fdl_desc* argument in quotation marks
- Use a semicolon to delimit statements within the *fdl_desc* argument
- Assign the symbol FDL$M_FDL_STRING as the *flags* mask value

filename

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

Name of the OpenVMS RMS file to be created using the FDL specification. The *filename* argument is the address of a character string descriptor pointing to the RMS file name. This name overrides the *default_name* parameter given in the FDL specification.

default_name

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

Default name of the file to be created using the FDL specification. The *default_name* argument is the address of a character string descriptor pointing to the default file name. This name overrides any name given in the FDL specification.

result_name

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor—fixed-length string descriptor

Resultant name of the file created by FDL$CREATE. The *result_name* argument is the address of a character string descriptor that receives the resultant file name.

fid_block

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

File identification of the RMS file created by FDL$CREATE. The *fid_block* argument is the address of an array of longwords that receives the RMS file identification information. The first longword contains the FID_NUM, the second contains the FID_SEQ, and the third contains the FID_RVN. They have the following definitions:

FID_NUM	The location of the file on the disk. Its value can range from *1* up to the number of files the disk can hold.
FID_SEQ	The file sequence number, which is the number of times the file number has been used.
FID_RVN	The relative volume number, which is the volume number of the volume on which the file is stored. If the file is not stored on a volume set, the relative volume number is *0*.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the *fdl_desc* argument is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing the control flags (or a mask). If you omit this argument or specify it as 0, no flags are set. The following table shows the flags and their meanings:

Flag	**Function**
FDL$V_FDL_STRING	Interprets the *fdl_desc* argument as an FDL specification in string form. By default, the *fdl_desc* argument is interpreted as the file name of an FDL file.
FDL$V_LONG_NAMES	Returns the *RESULT_NAME* using the long result name from a long name access block (NAML). By default, the *RESULT_NAME* is returned from the short fields of a name access block (NAM) and thus may have a generated specification. This flag is valid for OpenVMS Alpha only.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

stmnt_num

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

FDL statement number. The *stmnt_num* argument is the address of a longword that receives the FDL statement number. If the routine finishes successfully, the *stmnt_num* argument is the number of statements in the FDL specification. If the routine does not finish successfully, the *stmnt_num* argument receives the number of the statement that caused the error. Note that line numbers and statement numbers are not the same and that an FDL specification in string form has no "lines."

retlen

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Number of characters returned in the *result_name* argument. The *retlen* argument is the address of a longword that receives this number.

sts

OpenVMS usage:	longword_unsigned
type:	longword_unsigned
access:	write only
mechanism:	by reference

RMS status value FAB$L_STS. The *sts* argument is the address of a longword that receives the status value FAB$L_STS from the $CREATE system service.

stv

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

RMS status value FAB$L_STV. The *stv* argument is the address of a longword that receives the status value FAB$L_STV from the $CREATE system service.

default_fdl_desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only

mechanism: by descriptor—fixed-length string descriptor

The *default_fdl_desc* argument is one of the following:

- A character string descriptor pointing to a file containing the default FDL specification to be parsed
- A character string containing the actual default FDL specification

See the description of the *fdl_desc* argument for details.

This argument allows you to specify default FDL attributes. In other words, FDL$CREATE processes the attributes specified in this argument unless you override them with the attributes you specify in the *fdl_desc* argument.

You can code the FDL defaults directly into your program, typically with an FDL specification in string form.

Description

FDL$CREATE calls the FDL$PARSE routine to parse the FDL specification. The FDL specification can be in a file or a character string.

Source of FDL Specification	Advantages	Disadvantages
FDL file	Variability; for example, if the specification changes regularly, you can revise the file without revising the calling program.	File must be in default directory. Slower.
Character string	You do not have to be concerned with locating a file.	Program must be recoded to change FDL specification.
	Faster access.	

If the FDL specification is relatively simple and is not going to change, put the FDL specification in a character string as the *fdl_desc* argument to the call.

FDL$CREATE opens (creates) the specified RMS file and then closes it without putting any data in it.

FDL$CREATE does not create the output file if an error status is either returned or signaled.

Condition Values Returned

RMS$_NORMAL

Normal successful completion.

FDL$_ABKW

Ambiguous keyword in statement *number* <CRLF> *reference-text*.

FDL$_ABPRIKW

Ambiguous primary keyword in statement *number* <CRLF> *reference-text*.

FDL$_BADLOGIC

Internal logic error detected.

FDL$_CLOSEIN

Error closing *filename* as input.

FDL$_CLOSEOUT

Error closing *filename* as output.

FDL$_CREATE

Error creating *filename*.

FDL$_CREATED

Filename created.

FDL$_CREATED_STM

Filename created in stream format.

FDL$_FDLERROR

Error parsing FDL file.

FDL$_ILL_ARG

Wrong number of arguments.

FDL$_INSVIREM

Insufficient virtual memory.

FDL$_INVBLK

Invalid RMS control block at virtual address 'hex-offset '.

FDL$_MULPRI

Multiple primary definition in statement *number*.

FDL$_OPENFDL

Error opening *filename*.

FDL$_OPENIN

Error opening *filename* as input.

FDL$_OPENOUT

Error opening *filename* as output.

FDL$_OUTORDER

Key or area primary defined out of order in statement *number*.

FDL$_READERR

Error reading *filename*.

FDL$_RFLOC

Unable to locate related file.

FDL$_SYNTAX

Syntax error in statement *number reference-text*.

FDL$_UNPRIKW

Unrecognized primary keyword in statement *number* <CRLF> *reference-text*.

FDL$_UNQUAKW

Unrecognized qualifier keyword in statement *number* <CRLF> *reference-text*.

FDL$_UNSECKW

Unrecognized secondary keyword in statement *number* <CRLF> *reference-text*.

FDL$_VALERR

Specified value is out of legal range.

FDL$_VALPRI

Value required on primary in statement *number*.

FDL$_WARNING

Parsed with warnings.

FDL$_WRITEERR

Error writing *filename*.

RMS$_ACT

File activity precludes operation.

RMS$_CRE

Ancillary control process (ACP) file create failed.

RMS$_CREATED

File was created, not opened.

RMS$_DNF

Directory not found.

RMS$_DNR

Device not ready or not mounted.

RMS$_EXP

File expiration date not yet reached.

RMS$_FEX

File already exists, not superseded.

RMS$_FLK

File currently locked by another user.

RMS$_PRV

Insufficient privilege or file protection violation.

RMS$_SUPERSEDE

Created file superseded existing version.

RMS$_WLK

Device currently write locked.

FDL$GENERATE

Generate an FDL Specification — The FDL$GENERATE routine produces an FDL specification and writes it to either an FDL file or a character string.

Format

```
FDL$GENERATE flags ,fab_pointer ,rab_pointer [,fdl_file_dst]
 [,fdl_file_resnam] [,fdl_str_dst] [,bad_blk_addr] [,retlen]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the *fdl_str_dst* argument is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing the control flags (or a mask).

If you omit this argument or specify it as zero, no flags are set. The flags and their meanings are as follows:

Flag	Function
FDL$V_FDL_STRING	Interprets the *fdl_str_dst* argument as an FDL specification in string form. By default, the *fdl_str_dst* argument is interpreted as the file name of an FDL file.
FDL$V_FULL_OUTPUT	Includes the FDL attributes to describe all the bits and fields in the OpenVMS RMS control blocks, including run-time options. If this flag is set, every field is inspected before being written. By default, only the FDL attributes that describe permanent file attributes are included (producing a much shorter FDL specification).
FDL$V_LONG_NAMES	Returns the *FDL_FILE_RESNAME* using the long result name from a long name access block (NAML). By default, the *FDL_FILE_RESNAM* is returned from the short fields of a name access block (NAM) and thus may have a generated specification. This flag is valid for OpenVMS Alpha only.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

fab_pointer

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

RMS file access block (FAB). The *fab_pointer* argument is the address of a longword containing the address of a FAB.

rab_pointer

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

RMS record access block (RAB). The *rab_pointer* argument is the address of a longword containing the address of a RAB.

fdl_file_dst

OpenVMS usage:	char_string

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

Name of the FDL file to be created. The *fdl_file_dst* argument is the address of a character-string descriptor containing the file name of the FDL file to be created. If the FDL$V_FDL_STRING flag is set in the *flags* argument, this argument is ignored; otherwise, it is required. The FDL specification is written to the file named in this argument.

fdl_file_resnam

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor—fixed-length string descriptor

Resultant name of the FDL file created. The *fdl_file_resnam* argument is the address of a variable character-string descriptor that receives the resultant name of the FDL file created (if FDL $GENERATE is directed to create an FDL file).

fdl_str_dst

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor—fixed-length string descriptor

FDL specification. The *fdl_str_dst* argument is the address of a variable character string descriptor that receives the FDL specification created. If the FDL$V_FDL_STRING bit is set in the *flags* argument, this argument is required; otherwise, it is ignored.

bad_blk_addr

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of an invalid RMS control block. The *bad_blk_addr* argument is the address of a longword that receives the address of an invalid control block (a fatal error). If an invalid control block is detected, this argument is returned; otherwise, it is ignored.

retlen

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Number of characters received in either the *fdl_file_resnam* or the *fdl_str_dst* argument. The *retlen* argument is the address of a longword that receives this number.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

FDL$_INVBLK

Invalid block.

RMS$_ACT

File activity precludes operation.

RMS$_CONTROLC

Operation completed under Ctrl/C.

RMS$_CONTROLO

Output completed under Ctrl/O.

RMS$_CONTROLY

Operation completed under Ctrl/Y.

RMS$_DNR

Device not ready or mounted.

RMS$_EXT

ACP file extend failed.

RMS$_OK_ALK

Record already locked.

RMS$_OK_DUP

Record inserted had duplicate key.

RMS$_OK_IDX

Index update error occurred.

RMS$_PENDING

Asynchronous operation pending completion.

RMS$_PRV

Insufficient privilege or file protection violation.

RMS$_REX

Record already exists.

RMS$_RLK

Target record currently locked by another stream.

RMS$_RSA

Record stream currently active.

RMS$_WLK

Device currently write locked.

SS$_ACCVIO

Access violation.

STR$_FATINERR

Fatal internal error in run-time library.

STR$_ILLSTRCLA

Illegal string class.

STR$_INSVIRMEM

Insufficient virtual memory.

FDL$PARSE

Parse an FDL Specification — The FDL$PARSE routine parses an FDL specification, allocates OpenVMS RMS control blocks (FABs, RABs, or XABs), and fills in the relevant fields.

Format

```
FDL$PARSE fdl_desc ,fdl_fab_pointer ,fdl_rab_pointer [,flags]
 [,default_fdl_desc] [,stmnt_num]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

fdl_desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

Name of the FDL file or the actual FDL specification to be parsed. See the description of the *fdl_desc* argument for the FDL$CREATE routine for details.

fdl_fab_pointer

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of an RMS file access block (FAB). The *fdl_fab_pointer* argument is the address of a longword that receives the address of the FAB. FDL$PARSE both allocates the FAB and fills in its relevant fields.

fdl_rab_pointer

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of an RMS record access block (for Alpha, it is the RAB64). The *fdl_rab_pointer* argument is the address of a longword that receives the address of the RAB or RAB64. FDL$PARSE both allocates the RAB or RAB64 and fills in any fields designated in the FDL specification.

For Alpha, the 64-bit record access block (RAB64) consists of the traditional 32-bit RAB followed by some 64-bit fields. The RAB64 is automatically allocated for Alpha users, who can either use it as a RAB64 or overlay it with the 32-bit RAB definition and use it as a traditional 32-bit RAB.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags (or masks) that control how the *default_fdl_desc* argument is interpreted and how errors are signaled. The *flags* argument is the address of a longword containing the control flags. If you omit this argument or specify it as zero, no *flags* are set. The *flags* and their meanings are as follows:

Flag	Function
FDL$V_DEFAULT_STRING	Interprets the *default_fdl_desc* argument as an FDL specification in string form. By default, the

Flag	Function
	default_fdl_desc argument is interpreted as the file name of an FDL file.
FDL$V_FDL_STRING	Interprets the *fdl_desc* argument as an FDL specification in string form. By default, the *fdl_desc* argument is interpreted as the file name of an FDL file.
FDL$V_LONG_NAMES	Allocates and returns a long name access block (NAML) linked to the returned RMS file access block (FAB). The appropriate values are set in the NAML and FAB blocks so that the long file name fields of the NAML block will be used. By default, a name block is not allocated and the file name fields of FAB are used. If the FDL$V_LONG_NAMES flag is set, then the FDL$V_LONG_NAMES bit must also be set in the *flags* argument to the FDL$RELEASE routine to ensure that memory allocated for the NAML block is deallocated properly. This flag is valid for OpenVMS Alpha only.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

By default, an error status is returned rather than signaled.

default_fdl_desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor—fixed-length string descriptor

The *default_fdl_desc* argument is the address of a character-string descriptor pointing to either the default FDL file or the default FDL specification. See the description of the *fdl_desc* argument for the FDL$CREATE routine for details.

This argument allows you to specify default FDL attributes. In other words, FDL$PARSE processes the attributes specified in this argument unless you override them with the attributes you specify in the *fdl_desc* argument.

You can code the FDL defaults directly into your program, typically with an FDL specification in string form.

stmnt_num

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only

mechanism:	by reference

FDL statement number. The *stmnt_num* argument is the address of a longword that receives the FDL statement number. If the routine finishes successfully, the *stmnt_num* argument is the number of statements in the FDL specification. If the routine does not finish successfully, the *stmnt_num* argument receives the number of the statement that caused the error. Note that line numbers and statement numbers are not the same and that an FDL specification in string form has no “lines.”

By default, an error status is returned rather than signaled.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

LIB$_BADBLOADR

Bad block address.

LIB$_BADBLOSIZ

Bad block size.

LIB$_INSVIRMEM

Insufficient virtual memory.

RMS$_DNF

Directory not found.

RMS$_DNR

Device not ready or not mounted.

RMS$_WCC

Invalid wildcard context (WCC) value.

FDL$RELEASE

Free Virtual Memory Obtained By FDL$PARSE — The FDL$RELEASE routine deallocates the virtual memory used by the OpenVMS RMS control blocks created by FDL$PARSE. You must use FDL $PARSE to populate the control blocks if you plan to deallocate memory later with FDL$RELEASE.

Format

```
FDL$RELEASE [fab_pointer] [,rab_pointer] [,flags] [,badblk_addr]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

fab_pointer

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

File access block (FAB) to be deallocated using the LIB$FREE_VM routine. The *fab_pointer* argument is the address of a longword containing the address of the FAB. The FAB must be the same one returned by the FDL$PARSE routine. Any name blocks (NAMs) and extended attribute blocks (XABs) connected to the FAB are also released.

If you omit this argument or specify it as zero, the FAB (and any associated NAMs and XABs) is not released.

rab_pointer

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Record access block (RAB) to be deallocated using the LIB$FREE_VM system service. The *rab_pointer* argument is the address of a longword containing the address of the RAB. The address of the RAB must be the same one returned by the FDL$PARSE routine. Any XABs connected to the RAB are also released.

If you omit this argument or specify it as zero, the RAB (and any associated XABs) is not released.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flag (or mask) that controls how errors are signaled. The *flags* argument is the address of a longword containing the control flag (or a mask). If you omit this argument or specify it as zero, no flag is set. The flag is defined as follows:

FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

FDL$V_LONG_NAMES	Deallocates any virtual memory used for a long name access block (NAML) created by the FDL $PARSE routine. This flag is valid for OpenVMS Alpha only.

badblk_addr

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of an invalid RMS control block. The *badblk_addr* argument is the address of a longword that receives the address of an invalid control block. If an invalid control block (a fatal error) is detected, this argument is returned; otherwise, it is ignored.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

FDL$_INVBLK

Invalid RMS control block at virtual address 'hex-offset '.

LIB$_BADBLOADR

Bad block address.

RMS$_ACT

File activity precludes operation.

RMS$_RNL

Record not locked.

RMS$_RSA

Record stream currently active.

SS$_ACCVIO

Access violation.

Chapter 13. Librarian (LBR) Routines

The Librarian (LBR) routines let you create and maintain libraries and their modules, and use the data stored in library modules. You can also create and maintain libraries at the DCL level by using the DCL command LIBRARY. For more information, see the *VSI OpenVMS DCL Dictionary*.

13.1. Introduction to LBR Routines

This section briefly describes the types of libraries you can create and maintain using LBR routines and how the libraries are structured. This section also lists and briefly describes the LBR routines. Section 13.2 provides sample programs showing how to use various LBR routines. Section 13.3 is a reference section that provides details about each of the LBR routines.

13.1.1. Types of Libraries

You can use the LBR routines to maintain the following types of libraries:

- Object libraries, including Integrity servers (ELF) object libraries and Alpha object libraries, contain the object modules of frequently called routines. The Linker utility searches specified object module libraries when it encounters a reference it cannot resolve in one of its input files. For more information about how the linker uses libraries, see the description of the Linker utility in the *VSI OpenVMS Linker Utility Manual*.

  An object library has a default file type of .OLB and defaults the file type of input files to .OBJ.

- Macro libraries contain macro definitions used as input to the assembler. The assembler searches specified macro libraries when it encounters a macro that is not defined in the input file. For information on porting code to Integrity server systems, see the *Porting Applications from VSI OpenVMS Alpha to VSI OpenVMS Industry Standard 64 for Integrity Servers*.

  A macro library has a default file type of .MLB and defaults the file type of input files to .MAR.

- Help libraries contain modules of help messages that provide user information about a program. You can retrieve help messages at the DCL level by using the DCL command HELP, or in your program by calling the appropriate LBR routines. For information about creating help modules for insertion into help libraries, see the description of the Librarian utility in the *VSI OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

  A help library has a default file type of .HLB and defaults the file type of input files to .HLP.

- Text libraries contain any sequential record files that you want to retrieve as data for a program. For example, some compilers can retrieve program source code from text libraries. Each text file inserted into the library corresponds to one library module. Your programs can retrieve text from text libraries by calling the appropriate LBR routines.

  A text library has a default file type of .TLB and defaults the file type of input files to .TXT.

- Shareable image libraries, including Integrity servers (ELF) shareable image libraries and Alpha shareable symbol table libraries contain the symbol tables of shareable images used as input to the linker. For information about how to create a shareable image library, see the descriptions of the Librarian and Linker utilities in the *VSI OpenVMS Command Definition, Librarian, and Message Utilities Manual* and the *VSI OpenVMS Linker Utility Manual*, respectively.

  A shareable image library has a default type of .OLB and defaults the file type of input files to .EXE.

- National character set (NCS) libraries contain definition modules that define collating sequences and conversion functions. NCS libraries have the default file type .NLB. For information about how to create an NCS library, see the *OpenVMS National Character Set Utility Manual*. [1]
- User-developed libraries have characteristics specified when you call the LBR$OPEN routine to create a new library. User-developed libraries allow you to use the LBR routines to create and maintain libraries that are not structured in the form assigned by default to the other library types. Note that you cannot use the DCL command LIBRARY to access user-developed libraries.

Table 13.1shows the libraries that are created by the Librarian utility for each OpenVMS platform.

Table 13.1. Libraries Created by OpenVMS Platforms

OpenVMS Alpha	OpenVMS Integrity servers
Alpha object	Integrity servers object
Alpha shareable image	Integrity servers shareable image
Macro	Macro
Text	Text
Help	Help

13.1.2. Structure of Libraries

You create libraries by executing the DCL command LIBRARY or by calling the LBR$OPEN routine. When object, macro, text, help, or shareable image libraries are created, the Librarian utility structures them as described in Figure 13.1 and Figure 13.2. You can create user-developed libraries only by calling LBR$OPEN; they are structured as described in Figure 13.3.

13.1.2.1. Library Headers

Every library contains a library header that describes the contents of the library, for example, its type, size, version number, creation date, and number of indexes. You can retrieve data from a library's header by calling the LBR$GET_HEADER routine.

13.1.2.2. Modules

Each library module consists of a header and data. The data is the information you inserted into the library; the header associated with the data is created by the LBR routine and provides information about the module, including its type, attributes, and date of insertion into the library. You can read and update a module's header by calling the LBR$SET_MODULE routine.

13.1.2.3. Indexes and Keys

Libraries contain one or more indexes, which can be thought of as directories of the library's modules. The entries in each index are keys, and each key consists of a key name and a module reference. The module reference is a pointer to the module's header record and is called that record's file address (RFA). Macro, text, and help libraries (see Figure 13.1) contain only one index, called the module name table. The names of the keys in the index are the names of the modules in the library.

Object and shareable image libraries (see Figure 13.2) contain two indexes: the module name table and a global symbol table. The global symbol table consists of all the global symbols defined in the modules in the library. Each global symbol is a key in the index and points to the module in which it was defined.

[1]This manual has been archived but is available on the *VSI OpenVMS Documentation CD*.

If you need to point to the same module with several keys, you should create a user-developed library, which can have up to eight indexes (see Figure 13.3). Each index consists of keys that point to the library's modules.

The LBR routines differentiate library indexes by numbering them, starting with 1. For all but user-developed libraries, the module name table is index number 1 and the global symbol table, if present, is index number 2. You number the indexes in user-developed libraries. When you access libraries that contain more than one index, you may have to call LBR$SET_INDEX to tell the LBR routines which index to use.

Figure 13.1. Structure of a Macro, Text, or Help Library

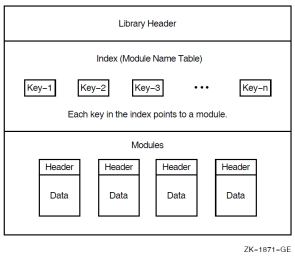


Figure 13.2. Structure of an Object or Shareable Image Library

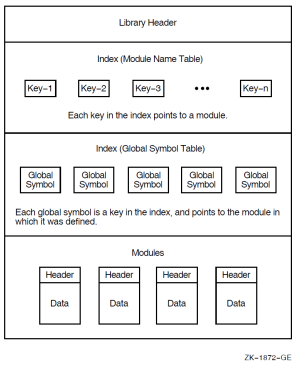
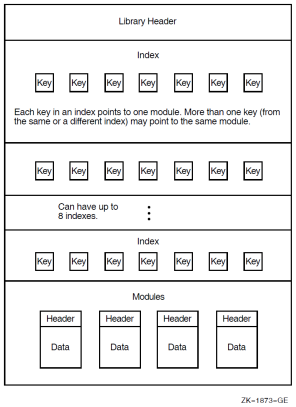


Figure 13.3. Structure of a User-Developed Library



13.1.3. Summary of LBR Routines

All the LBR routines begin with the characters LBR$. Your programs can call these routines by using the OpenVMS Calling Standard. When you call an LBR routine, you must provide all required

arguments. Upon completion, the routine returns its completion status as a condition value. In addition to the listed condition values, some routines may return the success code SS$_NORMAL as well as various OpenVMS RMS or system status (SS) error codes.

When you link programs that contain calls to LBR routines, the linker locates the routines during its default search of SYS$SHARE:LBRSHR. Table 13.2 lists the routines and summarizes their functions.

Table 13.2. LBR Routines

Routine Name	Function
LBR$CLOSE	Closes an open library.
LBR$DELETE_DATA	Deletes a specified module's header and data.
LBR$DELETE_KEY	Deletes a key from a library index.
LBR$FIND	Finds a module by using an address returned by a preceding call to LBR$LOOKUP_KEY.
LBR$FLUSH	Writes the contents of modified blocks to the library file and returns the virtual memory that contained those blocks.
LBR$GET_HEADER	Retrieves information from the library header.
LBR$GET_HELP	Retrieves help text from a specified library.
LBR$GET_HISTORY	Retrieves library update history records and calls a user-supplied routine with each record returned.
LBR$GET_INDEX	Calls a routine to process modules associated with some or all of the keys in an index.
LBR$GET_RECORD	Reads a data record from the module associated with a specified key.
LBR$INI_CONTROL	Initializes a control index that the Librarian uses to identify a library.
LBR$INSERT_KEY	Inserts a new key in the current library index.
LBR$LOOKUP_KEY	Looks up a key in the current index.
LBR$LOOKUP_TYPE	Searches the index for the key from a particular module (RFA) and returns the key's type for that module.
LBR$MAP_MODULE	Integrity servers only. Maps a module in P2 space.
LBR$OPEN	Opens an existing library or creates a new one.
LBR$OUTPUT_HELP	Retrieves help text from an explicitly named library or from user-supplied default libraries, and optionally prompts you for additional help queries.
LBR$PUT_END	Terminates the writing of a sequence of records to a module using the LBR$PUT_RECORD routine.
LBR$PUT_HISTORY	Inserts a library update history record.
LBR$PUT_MODULE	Integrity servers only. Puts an entire module, with the module's file address (RFA), from memory space into the current library.
LBR$PUT_RECORD	Writes a data record to the module associated with the specified key.

Routine Name	Function
LBR$REPLACE_KEY	Replaces an existing key in the current library index.
LBR$RET_RMSSTV	Returns the last RMS status value.
LBR$SEARCH	Finds index keys that point to specified data.
LBR$SET_INDEX	Sets the index number to be used during processing of the library.
LBR$SET_LOCATE	Sets Librarian subroutine record access to locate mode.
LBR$SET_MODULE	Reads and optionally updates a module header associated with a given record's file address (RFA).
LBR$SET_MOVE	Sets Librarian subroutine record access to move mode.
LBR$UNMAP_MODULE	Integrity servers only. Unmaps a module from process P2 space.

13.2. Using the LBR Routines: Examples

This section provides programming examples that call LBR routines. Although the examples do not illustrate all the LBR routines, they do provide an introduction to the various data structures and the calling syntax.

The program examples are written in VSI Pascal and the subroutine examples are written in VSI Fortran. The listing of each program example contains comments and is followed by notes about the program. The highlighted numbers in the notes are keyed to the highlighted numbers in the examples.

Each sample program calls the LBR$INI_CONTROL routine and the LBR$OPEN routine before calling any other routine.

Note

The one exception is that when you call the LBR$OUTPUT_HELP routine, you need not call the LBR $INI_CONTROL routine and the LBR$OPEN routine.

The sample programs require access to various symbols derived from definition macros. Use the INHERIT attribute to access these symbols from definition macros in SYS$LIBRARY:STARLET.PEN.

The LBR$INI_CONTROL routine sets up a control index; do not confuse this with a library index. The control index is used in subsequent LBR routine calls to identify the applicable library (because you may want your program to work with more than one library at a time).

Note

Do not alter the control index value.

LBR$INI_CONTROL specifies the library function, which can be to either create and update a new library (LIB$C_CREATE), modify an existing library (LIB$C_UPDATE), or read an existing library without updating it (LIB$C_READ).

Upon completion of the LBR$INI_CONTROL routine, call the LBR$OPEN routine to open the library. Open an existing library, or create and open a new library, in either the UPDATE or READ mode,

checking for an error status value of RMS$_FNF. If this error occurs, open the library in CREATE mode.

When you open the library, specify the library type and pass the file specification or partial file specification of the library file.

If you are creating a new library, pass the create options array. The CRE symbols identify the significant longwords of the array by their byte offsets into the array. Convert these values to subscripts for an array of integers (longwords) by dividing by 4 and adding 1. If you do not load the significant longwords before calling LBR$INI_CONTROL, the library may be corrupted upon creation.

Finally, pass any defaults for the file specification. If you omit the device and directory parts of the file specification, the current default device and directory are used.

When you finish working with a library, call LBR$CLOSE to close the library by providing the control index value. You must close a library explicitly before updates can be posted. Remember to call LBR $INI_CONTROL again if you want to reopen the library. LBR$CLOSE deallocates all the memory associated with the library, including the control index.

The order in which you call the routines between LBR$OPEN and LBR$CLOSE depends upon the library operations you need to perform. You may want to call LBR$LOOKUP_KEY or LBR $GET_INDEX to find a key, then perform some operation on the module associated with the key. You can think of a module as being both the module itself and its associated keys. To access a module, you first need to access a key that points to it; to delete a module, you first need to delete any keys that point to it.

Note

Do not use LBR$INI_CONTROL, LBR$OPEN, and LBR$CLOSE for writing help text with LBR $OUTPUT_HELP. Simply invoke LBR$OUTPUT_HELP.

13.2.1. Creating, Opening, and Closing a Text Library

Example 13.1 is a sample VSI Pascal program that creates, opens, and then closes a text library. The program is summarized in the following steps:

1. Initialize the library—Call LBR$INI_CONTROL to initialize the library.
2. Open the library—Call LBR$OPEN to open the library.
3. Close the library—Call LBR$CLOSE to close the library.

Example 13.1. Creating a New Library Using VSI Pascal

```
PROGRAM createlib(INPUT,OUTPUT);
                (*This program creates a text library*)
TYPE                                              (*Data type of*)
    Create_Array = ARRAY [1..20] OF INTEGER;     (*create options array*)
VAR                                        (*Constants and return status error
                                           codes for LBR$_OPEN & LBR
$INI_CONTROL.
                                           These are defined in $LBRDEF
 macro*)
    LBR$C_CREATE,LBR$C_TYP_TXT,LBR$_ILLCREOPT,LBR$_ILLCTL,     ❶
    LBR$_ILLFMT,LBR$_NOFILNAM,LBR$_OLDMISMCH,LBR$_TYPMISMCH :
```

```
                                [EXTERNAL] INTEGER;
                                        (*Create options array codes.
 These
                                        are defined in $CREDEF macro*)
    CRE$L_TYPE,CRE$L_KEYLEN,CRE$L_ALLOC,CRE$L_IDXMAX,CRE$L_ENTALL,
    CRE$L_LUHMAX,CRE$L_VERTYP,CRE$L_IDXOPT,CRE$C_MACTXTCAS,
    CRE$C_VMSV3 :               [EXTERNAL]INTEGER;
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library to create*)
    Options : Create_Array;             (*Create options array*)
    File_Type : PACKED ARRAY [1..4]     (*Character string that is
 default*)
                OF CHAR := '.TLB';        (*file type of created lib file*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
    status : UNSIGNED;                  (*Return Status for function
 calls*)
        (*-*-*-*-Function and Procedure Definitions-*-*-*-*)
                                        (*Function that returns library
                                        control index used by Librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;    ❷
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[l..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                        (*Function that creates/opens
 library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                    fns: [class_s]PACKED ARRAY[l..u:INTEGER] OF CHAR;
                    create_options: Create_Array;
                    dns: [CLASS_S] PACKED ARRAY [l3..u3:INTEGER] OF CHAR;
                    rlfna: ARRAY [l4..u4:INTEGER] OF INTEGER := %IMMED 0;
                    rns: [CLASS_S] PACKED ARRAY [l5..u5:INTEGER] OF CHAR :=
                           %IMMED 0;
                    VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                        (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;
                                        (*Error handler to check error
 codes
                                        if open/create not successful*)
PROCEDURE Open_Error;     ❸
    BEGIN
        WRITELN('Open Not Successful'); (*Now check specific error codes*)
        IF status = IADDRESS(LBR$_ILLCREOPT) THEN
            WRITELN('    Create Options Not Valid Or Not Supplied');
        IF status = IADDRESS(LBR$_ILLCTL) THEN
            WRITELN('    Invalid Library Index');
        IF status = IADDRESS(LBR$_ILLFMT) THEN
            WRITELN('    Library Not In Correct Format');
        IF status = IADDRESS(LBR$_NOFILNAM) THEN
            WRITELN('    Library Name Not Supplied');
        IF status = IADDRESS(LBR$_OLDMISMCH) THEN
            WRITELN('    Old Library Conflict');
        IF status = IADDRESS(LBR$_TYPMISMCH) THEN
            WRITELN('    Library Type Mismatch')
    END; (*of procedure Open_Error*)
BEGIN  (* *************** DECLARATIONS COMPLETE ************************
```

```
        *************** MAIN PROGRAM BEGINS HERE ********************** *)
                                        (*Prompt for Library Name*)
    WRITE('Library Name: '); READLN(Lib_Name);
                                        (*Fill Create Options Array. Divide
                                       by 4 and add 1 to get proper
 subscript*)
    Options[IADDRESS(CRE$L_TYPE) DIV 4 + 1]   := IADDRESS(LBR$C_TYP_TXT);
    Options[IADDRESS(CRE$L_KEYLEN) DIV 4 + 1] := 31;      ❹
    Options[IADDRESS(CRE$L_ALLOC) DIV 4 + 1]  := 8;
    Options[IADDRESS(CRE$L_IDXMAX) DIV 4 + 1] := 1;
    Options[IADDRESS(CRE$L_ENTALL) DIV 4 + 1] := 96;
    Options[IADDRESS(CRE$L_LUHMAX) DIV 4 + 1] := 20;
    Options[IADDRESS(CRE$L_VERTYP) DIV 4 + 1] := IADDRESS(CRE$C_VMSV3);
    Options[IADDRESS(CRE$L_IDXOPT) DIV 4 + 1] := IADDRESS(CRE$C_MACTXTCAS);
                                        (*Initialize library control
 index*)
    status := LBR$INI_CONTROL   (lib_index_ptr,      ❺
                                IADDRESS(LBR$C_CREATE),    (*Create
 access*)
                                IADDRESS(LBR$C_TYP_TXT));  (*Text library*)
    IF NOT ODD(status) THEN             (*Check return status*)
        WRITELN('Initialization Failed')
    ELSE                                (*Initialization was successful*)
        BEGIN                           (*Create and open the library*)
            status := LBR$OPEN  (lib_index_ptr,
                                Lib_Name,
                                Options,      ❻
                                File_Type);
                IF NOT ODD(status) THEN (*Check return status*)
                    Open_Error          (*Call error handler*)    ❼
                ELSE                    (*Open/create was successful*)
                    BEGIN               (*Close the library*)
                        status := LBR$CLOSE(lib_index_ptr);
                        IF NOT ODD(status) THEN (*Check return status*)
                            WRITELN('Close Not Successful')
                    END
        END
END. (*of program creatlib*)
```

Each item in the following list corresponds to a number highlighted in Example 13.1:

❶ Use the INHERIT attribute to access the LBR and CRE symbols from SYS $LIBRARY:STARLET.PEN.

❷ Start the declarations of the LBR routines that are used by the program. Each argument to be passed to the Librarian is specified on a separate line and includes the name (which just acts as a placeholder) and data type (for example: UNSIGNED, which means an unsigned integer value, and PACKED ARRAY OF CHAR, which means a character string). If the argument is preceded by VAR, then a value for that argument is returned by the LBR to the program.

❸ Declare the procedure Open_Error, which is called in the executable section if the Librarian returns an error when LBR$OPEN is called. Open_Error checks the Librarian's return status value to determine the specific cause of the error. The return status values for each routine are listed in the descriptions of the routines.

❹ Initialize the array called Options with the values the Librarian needs to create the library.

❺ Call LBR$INI_CONTROL, specifying that the function to be performed is create and that the library type is text.

❻ Call LBR$OPEN to create and open the library; pass the Options array initialized in item 5 to the Librarian.
❼ If the call to LBR$OPEN was unsuccessful, call the procedure Open_Error (see item 4) to determine the cause of the error.

13.2.2. Inserting a Module

Example 13.2 illustrates the insertion of a module into a library from a VSI Pascal program. The program is summarized in the following steps:

1. Ensure that the module does not already exist by calling LBR$LOOKUP_KEY. The return status should be LBR$_KEYNOTFND. This step is optional.

2. Construct the module by calling LBR$PUT_RECORD once for each record going into the module. Pass the contents of the record as the second argument. LBR$PUT_RECORD returns the record file address (RFA) in the library file as the third argument on the first call. On subsequent calls, you pass the RFA as the third argument, so do not alter its value between calls.

3. Call LBR$PUT_END after the last call to LBR$PUT_RECORD.

4. Call LBR$INSERT_KEY to catalog the records you have just put in the library. The second argument is the name of the module.

To replace an existing module, save the RFA of the module header returned by LBR$LOOKUP_KEY in Step 1 in one variable and the new RFA returned by the first call to LBR$PUT_RECORD (Step 2) in another variable. In Step 4, invoke LBR$REPLACE_KEY instead of LBR$INSERT_KEY, pass the old RFA as the third argument, and the new RFA as the fourth argument.

Example 13.2. Inserting a Module into a Library Using VSI Pascal

```
PROGRAM insertmod(INPUT,OUTPUT);
                (*This program inserts a module into a library*)
TYPE
    Rfa_Ptr = ARRAY [0..1] OF INTEGER;  (*Data type of RFA of module*)
VAR
    LBR$C_UPDATE,                       (*Constants for LBR$INI_CONTROL*)
    LBR$C_TYP_TXT,                        (*Defined in $LBRDEF macro*)
    LBR$_KEYNOTFND : [EXTERNAL] INTEGER;(*Error code for LBR$LOOKUP_KEY*)
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library receiving
 module*)
    Module_Name : VARYING [31] OF CHAR; (*Name of module to insert*)
    Text_Data_Record : VARYING [255] OF CHAR; (*Record in new module*)
    Textin : FILE OF VARYING [255] OF CHAR; (*File containing new module*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
    status : UNSIGNED;                  (*Return status for function
 calls*)
    txtrfa_ptr : Rfa_Ptr;               (*For key lookup and insertion*)
    Key_Not_Found : BOOLEAN := FALSE;   (*True if new mod not already in
 lib*)
                (*-*-*-*-Function Definitions-*-*-*-*)
                                        (*Function that returns library
                                        control index used by Librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[1..u:INTEGER]
```

```
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                        (*Function that creates/opens
 library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                    fns: [class_s]PACKED ARRAY[l..u:INTEGER] OF CHAR;
                    create_options: ARRAY [l2..u2:INTEGER] OF INTEGER :=
                         %IMMED 0;
                    dns: [CLASS_S] PACKED ARRAY [l3..u3:INTEGER] OF CHAR
                         := %IMMED 0;
                    rlfna: ARRAY [l4..u4:INTEGER] OF INTEGER := %IMMED 0;
                    rns: [CLASS_S] PACKED ARRAY [l5..u5:INTEGER] OF CHAR :=
                         %IMMED 0;
                    VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                        (*Function that finds a key in
 index*)
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR;
                         VAR txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                        (*Function that inserts key in
 index*)
FUNCTION LBR$INSERT_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR;
                         txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                        (*Function that writes data
 records*)
FUNCTION LBR$PUT_RECORD (library_index: UNSIGNED;              (*to
 modules*)
                         textline:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR;
                        txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                        (*Function that marks end of a
 module*)
FUNCTION LBR$PUT_END (library_index: UNSIGNED):
        INTEGER; EXTERN;
                                        (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;
BEGIN  (* *************** DECLARATIONS COMPLETE ************************
        *************** MAIN PROGRAM BEGINS HERE ********************* *)
                                        (*Prompt for library name and
                                        module to insert*)
    WRITE('Library Name: '); READLN(Lib_Name);
    WRITE('Module Name:  '); READLN(Module_Name);
                                        (*Initialize lib for update
 access*)
    status := LBR$INI_CONTROL   (lib_index_ptr,     ❶
                                IADDRESS(LBR$C_UPDATE),    (*Update
 access*)
                                IADDRESS(LBR$C_TYP_TXT));  (*Text library*)
    IF NOT ODD(status) THEN             (*Check error status*)
        WRITELN('Initialization Failed')
```

```
    ELSE                                (*Initialization was successful*)
        BEGIN
            status := LBR$OPEN  (lib_index_ptr, (*Open the library*)
                                 Lib_Name);
            IF NOT ODD(status) THEN     (*Check error status*)
                WRITELN('Open Not Successful')
            ELSE                        (*Open was successful*)
                BEGIN                   (*Is module already in the library?
*)
                    status := LBR$LOOKUP_KEY    (lib_index_ptr,     ❷
                                                 Module_Name,
                                                 txtrfa_ptr);
                    IF ODD(status) THEN (*Check status.  Should not be
 odd*)
                        WRITELN('Lookup key was successful.',
                                'The module is already in the library.')
                    ELSE   (*Did lookup key fail because key not found?*)
                            IF status = IADDRESS(LBR$_KEYNOTFND) THEN ❸
                                Key_Not_Found := TRUE
                END
        END;
         (******If LBR$LOOKUP_KEY failed because the key was not found
          (as expected), we can open the file containing the new module,
          and write the module's records to the library file*******)
    IF Key_Not_Found THEN
        BEGIN
            OPEN(Textin,Module_Name,old);
            RESET(Textin);
            WHILE NOT EOF(Textin) DO            (*Repeat until end of
 file*)
                BEGIN                           ❹
                    READ(Textin,Text_Data_Record);      (*Read record from
                                                         external file*)
                    status := LBR$PUT_RECORD    (lib_index_ptr,   (*Write*)
                                                 Text_Data_Record, (*record
 to*)
                                                 txtrfa_ptr);
 (*library*)
                    IF NOT ODD(status) THEN
                        WRITELN('Put Record Routine Not Successful')
                END; (*of WHILE statement*)
            IF ODD(status) THEN (*True if all the records have been
                                 successfully written into the library*)
                BEGIN
                    status := LBR$PUT_END (lib_index_ptr); (*Write end of
                                                            module record*)
                    IF NOT ODD(status) THEN
                        WRITELN('Put End Routine Not Successful')
                    ELSE                (*Insert key for new module*)
                        BEGIN                       ❺
                            status := LBR$INSERT_KEY    (lib_index_ptr,
                                                         Module_Name,
                                                         txtrfa_ptr);
                            IF NOT ODD(status) THEN
                                WRITELN('Insert Key Not Successful')
                        END
                END
        END;
```

```
    status := LBR$CLOSE(lib_index_ptr);
    IF NOT ODD(status) THEN
        WRITELN('Close Not Successful')
END. (*of program insertmod*)
```

Each item in the following list corresponds to a number highlighted in Example 13.2:

❶ Call LBR$INI_CONTROL, specifying that the function to be performed is update and that the library type is text.
❷ Call LBR$LOOKUP_KEY to see whether the module to be inserted is already in the library.
❸ Call LBR$LOOKUP_KEY to see whether the lookup key failed because the key was not found. (In this case, the status value is LBR$_KEYNOTFND.)
❹ Read a record from the input file, then use LBR$PUT_RECORD to write the record to the library. When all the records have been written to the library, use LBR$PUT_END to write an end-of-module record.
❺ Use LBR$INSERT_KEY to insert a key for the module into the current index.

13.2.3. Extracting a Module

Example 13.3 illustrates the extraction of a library module from a VSI Pascal program. The program is summarized in the following steps:

1. Call LBR$LOOKUP_KEY to locate the module. Specify the name of the module as the second argument. LBR$LOOKUP_KEY returns the RFA of the module as the third argument; do not alter this value.

2. Call LBR$GET_RECORD once for each record in the module. Specify a character string to receive the extracted record as the second argument. LBR$GET_RECORD returns a status value of RMS $_EOF after the last record in the module is extracted.

Example 13.3. Extracting a Module from a Library Using VSI Pascal

```
PROGRAM extractmod(INPUT,OUTPUT,Textout);
                (*This program extracts a module from a library*)
TYPE
    Rfa_Ptr = ARRAY [0..1] OF INTEGER;  (*Data type of RFA of module*)
VAR
    LBR$C_UPDATE,                       (*Constants for LBR$INI_CONTROL*)
    LBR$C_TYP_TXT,                        (*Defined in $LBRDEF macro*)
    RMS$_EOF : [EXTERNAL] INTEGER;      (*RMS return status; defined in
                                        $RMSDEF macro*)
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library receiving
 module*)
    Module_Name : VARYING [31] OF CHAR; (*Name of module to insert*)
    Extracted_File : VARYING [31] OF CHAR;      (*Name of file to hold
                                                extracted module*)
    Outtext : PACKED ARRAY [1..255] OF CHAR;    (*Extracted mod put here,*)
    Outtext2 : VARYING [255] OF CHAR;           (*     then moved to here*)
    i : INTEGER;                        (*For loop control*)
    Textout : FILE OF VARYING [255] OF CHAR;    (*File containing extracted
                                                module*)
    nullstring : CHAR;                  (*nullstring, pos, and len used
 to*)
    pos, len : INTEGER;                 (*find string in extracted file
 recd*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
```

```
    status : UNSIGNED;                  (*Return status for function
 calls*)
    txtrfa_ptr : Rfa_Ptr;               (*For key lookup and insertion*)
                (*-*-*-*-Function Definitions-*-*-*-*)
                        (*Function that returns library
                        control index used by Librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[l..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                        (*Function that creates/opens library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                    fns: [class_s]PACKED ARRAY[l..u:INTEGER] OF CHAR;
                    create_options: ARRAY [l2..u2:INTEGER] OF INTEGER :=
                         %IMMED 0;
                    dns: [CLASS_S] PACKED ARRAY [l3..u3:INTEGER] OF CHAR
                         := %IMMED 0;
                    rlfna: ARRAY [l4..u4:INTEGER] OF INTEGER := %IMMED 0;
                    rns: [CLASS_S] PACKED ARRAY [l5..u5:INTEGER] OF CHAR :=
                         %IMMED 0;
                    VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                        (*Function that finds a key in an index*)
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR;
                         VAR txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
(*Function that retrieves records from modules*)
FUNCTION LBR$GET_RECORD (library_index: UNSIGNED;
                         var textline:[CLASS_S] PACKED ARRAY [l..u:INTEGER]
 OF
                               CHAR):
                        INTEGER;
EXTERN;
                        (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;
BEGIN  (* *************** DECLARATIONS COMPLETE ************************
        *************** MAIN PROGRAM BEGINS HERE ********************** *)
 (* Get Library Name, Module To Extract, And File To Hold Extracted Module
 *)
    WRITE('Library Name: '); READLN(Lib_Name);
    WRITE('Module Name:  '); READLN(Module_Name);
    WRITE('Extract Into File: '); READLN(Extracted_File);

    status := LBR$INI_CONTROL   (lib_index_ptr,     ❶
                                 IADDRESS(LBR$C_UPDATE),
                                 IADDRESS(LBR$C_TYP_TXT));
    IF NOT ODD(status) THEN
        WRITELN('Initialization Failed')
    ELSE
        BEGIN
            status := LBR$OPEN  (lib_index_ptr,
                                 Lib_Name);
            IF NOT ODD(status) THEN
```

```
            WRITELN('Open Not Successful')
        ELSE
            BEGIN                                ❷
                status := LBR$LOOKUP_KEY    (lib_index_ptr,
                                            Module_Name,
                                            txtrfa_ptr);
                IF NOT ODD(status) THEN
                    WRITELN('Lookup Key Not Successful')
                ELSE
                    BEGIN                        ❸
                        OPEN(Textout,Extracted_File,new);
                        REWRITE(Textout)
                    END
            END
    END;
WHILE ODD(status) DO
    BEGIN
        nullstring := ''(0);
        FOR i := 1 TO 255 DO                     ❹
            Outtext[i] := nullstring;
        status := LBR$GET_RECORD    (lib_index_ptr,
                                    Outtext);
        IF NOT ODD(status) THEN
            BEGIN                                ❺
                IF status = IADDRESS(RMS$_EOF) THEN
                    WRITELN('  RMS end of file')
            END
        ELSE
            BEGIN                                ❻
                pos := INDEX(Outtext, nullstring);  (*find first null
                                                    in Outtext*)
                len := pos - 1;     (*length of Outtext to first null*)
                IF len >= 1 THEN
                    BEGIN
                        Outtext2 := SUBSTR(Outtext,1,LEN);
                        WRITE(Textout,Outtext2)
                    END
            END
    END; (*of WHILE*)
status := LBR$CLOSE(lib_index_ptr);
IF NOT ODD(status) THEN
    WRITELN('Close Not Successful')
END. (*of program extractmod*)
```

Each item in the following list corresponds to a number highlighted in Example 13.3:

❶ Call LBR$INI_CONTROL, specifying that the function to be performed is update and that the library type is text.
❷ Call LBR$LOOKUP_KEY to find the key that points to the module you want to extract.
❸ Open an output file to receive the extracted module.
❹ Initialize the variable that is to receive the extracted records to null characters.
❺ Call LBR$GET_RECORD to see if there are more records in the file (module). A failure indicates that the end of the file has been reached.
❻ Write the extracted record data to the output file. This record should consist only of the data up to the first null character.

13.2.4. Deleting a Module

Example 13.4 illustrates the deletion of library module from a VSI Pascal program. The program is summarized in the following steps:

1. Call LBR$LOOKUP_KEY, and specify the name of the module as the second argument. LBR $LOOKUP_KEY returns the RFA of the module as the third argument; do not alter this value.
2. Call LBR$DELETE_KEY to delete the module key. Specify the name of the module as the second argument.
3. Call LBR$DELETE_DATA to delete the module itself. Specify the RFA of the module obtained in Step 1 as the second argument.

Example 13.4. Deleting a Module from a Library Using VSI Pascal

```
PROGRAM deletemod(INPUT,OUTPUT);
        (*This program deletes a module from a library*)
TYPE
    Rfa_Ptr = ARRAY [0..1] OF INTEGER;  (*Data type of RFA of module*)
VAR
    LBR$C_UPDATE,                       (*Constants for LBR$INI_CONTROL*)
    LBR$C_TYP_TXT,                        (*Defined in $LBRDEF macro*)
    LBR$_KEYNOTFND : [EXTERNAL] INTEGER;(*Error code for LBR$LOOKUP_KEY*)
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library receiving
 module*)
    Module_Name : VARYING [31] OF CHAR; (*Name of module to insert*)
    Text_Data_Record : VARYING [255] OF CHAR; (*Record in new module*)
    Textin : FILE OF VARYING [255] OF CHAR; (*File containing new module*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
    status : UNSIGNED;                  (*Return status for function
 calls*)
    txtrfa_ptr : Rfa_Ptr;               (*For key lookup and insertion*)
    Key_Not_Found : BOOLEAN := FALSE;   (*True if new mod not already in
 lib*)
                (*-*-*-*-Function Definitions-*-*-*-*)
                                (*Function that returns library
                                control index used by Librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[l..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                (*Function that creates/opens library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                    fns: [class_s]PACKED ARRAY[l..u:INTEGER] OF CHAR;
                    create_options: ARRAY [l2..u2:INTEGER] OF INTEGER :=
                         %IMMED 0;
                    dns: [CLASS_S] PACKED ARRAY [l3..u3:INTEGER] OF CHAR
                         := %IMMED 0;
                    rlfna: ARRAY [l4..u4:INTEGER] OF INTEGER := %IMMED 0;
                    rns: [CLASS_S] PACKED ARRAY [l5..u5:INTEGER] OF CHAR :=
                         %IMMED 0;
                    VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                (*Function that finds a key in index*)
```

```
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR;
                         VAR txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                (*Function that removes a key from an
 index*)
FUNCTION LBR$DELETE_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR):
                         INTEGER;
EXTERN;
(*Function that deletes all the records
                                associated with a module*)
FUNCTION LBR$DELETE_DATA (library_index: UNSIGNED;
                         txtrfa: Rfa_Ptr):
                         INTEGER;
EXTERN;
                                (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;

BEGIN  (* *************** DECLARATIONS COMPLETE ************************
        *************** MAIN PROGRAM BEGINS HERE ********************* *)
                                (* Get Library Name and Module to Delete *)
    WRITE('Library Name: '); READLN(Lib_Name);
    WRITE('Module Name:  '); READLN(Module_Name);
                                        (*Initialize lib for update
 access*)
    status := LBR$INI_CONTROL   (lib_index_ptr,         ❶
                                IADDRESS(LBR$C_UPDATE),     (*Update
 access*)
                                IADDRESS(LBR$C_TYP_TXT));  (*Text library*)
    IF NOT ODD(status) THEN             (*Check error status*)
        WRITELN('Initialization Failed')
    ELSE                                (*Initialization was successful*)
        BEGIN
            status := LBR$OPEN  (lib_index_ptr, (*Open the library*)
                                Lib_Name);
            IF NOT ODD(status) THEN     (*Check error status*)
                WRITELN('Open Not Successful')
            ELSE                        (*Open was successful*)
                BEGIN        ❷  (*Is module in the library?*)
                    status := LBR$LOOKUP_KEY    (lib_index_ptr,
                                                Module_Name,
                                                txtrfa_ptr);
                    IF NOT ODD(status) THEN (*Check status*)
                        WRITELN('Lookup Key Not Successful')
                END
        END;
    IF ODD(status) THEN                 (*Key was found; delete it*)
        BEGIN
            status := LBR$DELETE_KEY    (lib_index_ptr,     ❸
                                        Module_Name);
            IF NOT ODD(status) THEN
                WRITELN('Delete Key Routine Not Successful')
            ELSE                        (*Delete key was successful*)
```

```
                BEGIN                   (*Now delete module's data
 records*)
                    status := LBR$DELETE_DATA   (lib_index_ptr,     ❹
                                                txtrfa_ptr);
                    IF NOT ODD(status) THEN
                        WRITELN('Delete Data Routine Not Successful')
                END
        END;
    status := LBR$CLOSE(lib_index_ptr); (*Close the library*)
    IF NOT ODD(status) THEN
        WRITELN('Close Not Successful');
END. (*of program deletemod*)
```

Each item in the following list corresponds to a number highlighted in Example 13.4:

❶ Call LBR$INI_CONTROL, specifying that the function to be performed is update and the library type is text.
❷ Call LBR$LOOKUP_KEY to find the key associated with the module you want to delete.
❸ Call LBR$DELETE_KEY to delete the key associated with the module you want to delete. If more than one key points to the module, you need to call LBR$LOOKUP_KEY and LBR $DELETE_KEY for each key.
❹ Call LBR$DELETE_DATA to delete the module (the module header and data) from the library.

13.2.5. Using Multiple Keys and Multiple Indexes

You can point to the same module with more than one key. The keys can be in the primary index (index 1) or alternate indexes (indexes 2 through 10). The best method is to reserve the primary index for module names. In system-defined object libraries, index 2 contains the global symbols defined by the various modules.

Example 13.5 illustrates the way that keys can be associated with modules.

Example 13.5. Associating Keys with Modules

```
SUBROUTINE ALIAS (INDEX)
! Catalogs modules by alias

INTEGER STATUS,         ! Return status
        INDEX,          ! Library index
        TXTRFA (2)      ! RFA of module
CHARACTER*31 MODNAME,   ! Name of module
             ALIASNAME  ! Name of alias
INTEGER MODNAME_LEN     ! Length of module name
INTEGER ALIASNAME_LEN   ! Length of alias name
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
        LBR$SET_INDEX,
        LBR$INSERT_KEY,
        LIB$GET_INPUT,
        LIB$GET_VALUE
        LIB$LOCC
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
         LBR$_DUPKEY,    ! Duplicate key
         RMS$_EOF,       ! End of text in module
         DOLIB_NOMOD     ! No such module
```

```
! Get module name from /ALIAS on command line
CALL CLI$GET_VALUE ('ALIAS', MODNAME)
! Calculate length of module name
MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
! Look up module name in library index
STATUS = LBR$LOOKUP_KEY (INDEX,
                         MODNAME (1:MODNAME_LEN),
                         TXTRFA)
END IF
! Insert aliases if module exists
IF (STATUS) THEN
  ! Set to index 2
  STATUS = LBR$SET_INDEX (INDEX, 2)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Get alias name from /ALIAS on command line
  STATUS = CLI$GET_VALUE ('ALIAS', ALIASNAME)
  ! Insert aliases in index 2 until bad return status
  ! which indicates end of qualifier values
  DO WHILE (STATUS)
    ! Calculate length of alias name
    ALIASNAME_LEN = LIB$LOCC (' ', ALIASNAME) - 1
    ! Put alias name in index
    STATUS = LBR$INSERT_KEY (INDEX,
                             ALIASNAME (1:ALIASNAME_LEN),
                             TXTRFA)
    IF ((.NOT. STATUS) .AND.
        (STATUS .NE. %LOC (LBR$_DUPKEY)) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
    ! Get another alias
    STATUS = CLI$GET_VALUE ('ALIAS', ALIASNAME)
  END DO

  ! Issue warning if module does not exist
ELSE IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
  CALL LIB$SIGNAL (DOLIB_NOMOD,
                   %VAL (1),
                   MODNAME (1:MODNAME_LEN))
ELSE
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Exit
END
```

You can look up a module using any of the keys associated with it. The following code fragment checks index 2 for a key if the lookup in the primary index fails:

```
STATUS = LBR$SET_INDEX (INDEX, 1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LBR$LOOKUP_KEY (INDEX,
                         MODNAME (1:MODNAME_LEN),
                         TXTRFA)
IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
  STATUS = LBR$SET_INDEX (INDEX, 2)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  STATUS = LBR$LOOKUP_KEY (INDEX,
                           MODNAME (1:MODNAME_LEN),
```

```
                            TXTRFA)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

There are two ways to identify the keys associated with a module:

- Use the LBR$LOOKUP_KEY routine to look up the module using one of the keys.
- Use LBR$SEARCH to search applicable indexes for the keys. LBR$SEARCH calls a user-written routine each time it retrieves a key. The routine must be an integer function defined as external that returns a success (odd number) or failure (even number) status. LBR$SEARCH stops processing on a return status of failure.

The subroutine in Example 13.6 lists the names of keys in index 2 (the aliases) that point to a module identified on the command line by the module's name in the primary index.

Example 13.6. Listing Keys Associated with a Module

```
    .
    .
    .
SUBROUTINE SHOWAL (INDEX)
! Lists aliases for a module

INTEGER STATUS,      ! Return status
        INDEX,       ! Library index
        TXTRFA (2)   ! RFA for module text
CHARACTER*31 MODNAME ! Name of module
INTEGER MODNAME_LEN  ! Length of module name
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
        LBR$SEARCH,
        LIB$LOCC
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
         DOLIB_NOMOD     ! No such module
! Search routine
EXTERNAL SEARCH
INTEGER SEARCH
! Get module name and calculate length
CALL CLI$GET_VALUE ('SHOWALIAS', MODNAME)
MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
! Look up module in index 1
  STATUS = LBR$LOOKUP_KEY (INDEX,
                           MODNAME (1:MODNAME_LEN),
                           TXTRFA)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Search for alias names in index 2
  STATUS = LBR$SEARCH (INDEX,
                       2,
                       TXTRFA,
                       SEARCH)

END
INTEGER FUNCTION SEARCH (ALIASNAME, RFA)
! Function called for each alias name pointing to MODNAME
! Displays the alias name
INTEGER STATUS_OK,        ! Good return status
```

```
        RFA (2)           ! RFA of module
PARAMETER (STATUS_OK = 1) ! Odd number
CHARACTER*(*) ALIASNAME   ! Name of module
! Display module name
TYPE *, MODNAME

! Exit
SEARCH = STATUS_OK
END
```

13.2.6. Accessing Module Headers

You can store user information in the header of each module up to the total size of the header specified at library creation time in the CRE$L_UHDMAX option. The total size of each header in bytes is the value of MHD$B_USRDAT plus the value assigned to the CRE$L_UHDMAX option. The value of MHD$B_USRDAT is defined by the macro $MHDDEF; the default value is 16 bytes.

To put user data into a module header, first locate the module with LBR$LOOKUP_KEY; then move the data to the module header by invoking LBR$SET_MODULE, specifying the first argument (index value returned by LBR$INI_CONTROL), the second argument (RFA returned by LBR $LOOKUP_KEY), and the fifth argument (character string containing the user data).

To read user data from a module header, first locate the module with LBR$LOOKUP_KEY; then, retrieve the entire module header by invoking LBR$SET_MODULE, specifying the first, second, third (character string to receive the contents of the module header), and fourth (length of the module header) arguments. The user data starts at the byte offset defined by MHD$B_USRDAT. Convert this value to a character string subscript by adding 1.

Example 13.7 displays the user data portion of module headers on SYS$OUTPUT and applies updates from SYS$INPUT.

Example 13.7. Displaying the Module Header

```
    .
    .
    .
SUBROUTINE MODHEAD (INDEX)
! Modifies module headers

INTEGER STATUS,         ! Return status
        INDEX,          ! Library index
        TXTRFA (2)      ! RFA of module
CHARACTER*31 MODNAME    ! Name of module
INTEGER MODNAME_LEN     ! Length of module name
CHARACTER*80 HEADER     ! Module header
INTEGER HEADER_LEN      ! Length of module header
INTEGER USER_START      ! Start of user data in header
CHARACTER*64 USERDATA   ! User data part of header
INTEGER*2 USERDATA_LEN ! Length of user data
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
        LBR$SET_MODULE,
        LIB$GET_INPUT,
        LIB$PUT_OUTPUT,
        CLI$GET_VALUE,
        LIB$LOCC
! Offset to user data --- defined in $MHDDEF
```

```
EXTERNAL MHD$B_USRDAT
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
         DOLIB_NOMOD     ! No such module
! Calculate start of user data in header
USER_START = %LOC (MHD$B_USRDAT) + 1
! Get module name from /MODHEAD on command line
STATUS = CLI$GET_VALUE ('MODHEAD', MODNAME)
! Get module headers until bad return status
! which indicates end of qualifier values
DO WHILE (STATUS)

  ! Calculate length of module name
  MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
  ! Look up module name in library index
  STATUS = LBR$LOOKUP_KEY (INDEX,
                           MODNAME (1:MODNAME_LEN),
                           TXTRFA)

  ! Get header if module exists
  IF (STATUS) THEN
    STATUS = LBR$SET_MODULE (INDEX,
                             TXTRFA,
                             HEADER,
                             HEADER_LEN)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Display header and solicit replacement
    STATUS = LIB$PUT_OUTPUT
    ('User data for module '//MODNAME (1:MODNAME_LEN)//':')
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$PUT_OUTPUT
    (HEADER (USER_START:HEADER_LEN))
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$PUT_OUTPUT
    ('Enter replacement text below or just hit return:')
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$GET_INPUT (USERDATA,, USERDATA_LEN)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Replace user data
    IF (USERDATA_LEN .GT. 0) THEN
      STATUS = LBR$SET_MODULE (INDEX,
                               TXTRFA,,,
                               USERDATA (1:USERDATA_LEN))
    END IF

    ! Issue warning if module does not exist
  ELSE IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
    CALL LIB$SIGNAL (DOLIB_NOMOD,
                     %VAL (1),
                     MODNAME (1:MODNAME_LEN))
  ELSE
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF

  ! Get another module name
  STATUS = CLI$GET_VALUE ('MODHEAD', MODNAME)
END DO
```

```
! Exit
END
```

13.2.7. Reading Library Headers

Call LBR$GET_HEADER to obtain general information concerning the library. Pass the value returned by LBR$INI_CONTROL as the first argument. LBR$GET_HEADER returns the information to the second argument, which must be an array of 128 longwords. The LHI symbols identify the significant longwords of the array by their byte offsets into the array. Convert these values to subscripts by dividing by 4 and adding 1.

Example 13.8 reads the library header and displays some information from it.

Example 13.8. Reading Library Headers

```
    .
    .
    .
SUBROUTINE TYPEINFO (INDEX)
! Types the type, major ID, and minor ID
! of a library to SYS$OUTPUT

INTEGER STATUS             ! Return status
        INDEX,             ! Library index
        HEADER (128),      ! Structure for header information
        TYPE,              ! Subscripts for header structure
        MAJOR_ID,
        MINOR_ID
CHARACTER*8 MAJOR_ID_TEXT, ! Display info in character format
            MINOR_ID_TEXT
! VMS library procedures
INTEGER LBR$GET_HEADER,
        LIB$PUT_OUTPUT
! Offsets for header --- defined in $LHIDEF
EXTERNAL LHI$L_TYPE,
         LHI$L_MAJORID,
         LHI$L_MINORID
! Library type values --- defined in $LBRDEF
EXTERNAL LBR$C_TYP_OBJ,
         LBR$C_TYP_MLB,
         LBR$C_TYP_HLP,
         LBR$C_TYP_TXT
! Get header information
STATUS = LBR$GET_HEADER (INDEX, HEADER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Calculate subscripts for header structure
TYPE = %LOC (LHI$L_TYPE) / 4 + 1
MAJOR_ID = %LOC (LHI$L_MAJORID) / 4 + 1
MINOR_ID = %LOC (LHI$L_MINORID) / 4 + 1
! Display library type
IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_OBJ)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: object')
ELSE IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_MLB)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: macro')
ELSE IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_HLP)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: help')
ELSE IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_TXT)) THEN
```

```
  STATUS = LIB$PUT_OUTPUT ('Library type: text')
ELSE
  STATUS = LIB$PUT_OUTPUT ('Library type: unknown')
END IF
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Convert and display major ID
WRITE (UNIT=MAJOR_ID_TEXT,
       FMT='(I)') HEADER (MAJOR_ID)
STATUS = LIB$PUT_OUTPUT ('Major ID: '//MAJOR_ID_TEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Convert and display minor ID
WRITE (UNIT=MINOR_ID_TEXT,
       FMT='(I)') HEADER (MINOR_ID)
STATUS = LIB$PUT_OUTPUT ('Minor ID: '//MINOR_ID_TEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

! Exit
END
```

13.2.8. Displaying Help Text

You can display text from a help library by calling the LBR$OUTPUT_HELP routine and specifying the output routine, the keywords, and the name of the library. You must also specify the input routine if the prompting mode flag is set or if the *flags* argument is omitted.

Note

If you specify subprograms in an argument list, they must be declared as external.

You can use the LIB$PUT_OUTPUT and LIB$GET_INPUT routines to specify the output routine and the input routine. (If you use your own routines, make sure the argument lists are the same as for LIB $PUT_OUTPUT and LIB$GET_INPUT.) Do not call LBR$INI_CONTROL and LBR$OPEN before calling LBR$OUTPUT_HELP.

Example 13.9 solicits keywords from SYS$INPUT and displays the text associated with those keywords on SYS$OUTPUT, thus inhibiting the prompting facility.

Example 13.9. Displaying Text from a Help Library

```
PROGRAM GET_HELP

! Prints help text from a help library
CHARACTER*31 LIBSPEC   ! Library name
CHARACTER*15 KEYWORD   ! Keyword in help library
INTEGER*2 LIBSPEC_LEN, ! Length of name
          KEYWORD_LEN  ! Length of keyword
INTEGER FLAGS,         ! Help flags
        STATUS         ! Return status
! VMS library procedures
INTEGER LBR$OUTPUT_HELP,
        LIB$GET_INPUT,
        LIB$PUT_OUTPUT
EXTERNAL LIB$GET_INPUT,
         LIB$PUT_OUTPUT
! Error codes
EXTERNAL RMS$_EOF,      ! End-of-file
         LIB$_INPSTRTRU ! Input string truncated
```

```
! Flag values --- defined in $HLPDEF
EXTERNAL HLP$M_PROMPT,
         HLP$M_PROCESS,
         HLP$M_GROUP,
         HLP$M_SYSTEM,
         HLP$M_LIBLIST,
         HLP$M_HELP
! Get library name
STATUS = LIB$GET_INPUT (LIBSPEC,
                        'Library: ',
                        LIBSPEC_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
IF (LIBSPEC_LEN .EQ. 0) THEN
  LIBSPEC = 'HELPLIB'
  LIBSPEC_LEN = 7
END IF
! Set flags for no prompting
FLAGS = %LOC (HLP$_PROCESS) +
        %LOC (HLP$_GROUP) +
        %LOC (HLP$_SYSTEM)

! Get first keyword
STATUS = LIB$GET_INPUT (KEYWORD,
                        'Keyword or Ctrl/Z: ',
                        KEYWORD_LEN)
IF ((.NOT. STATUS) .AND.
    (STATUS .NE. %LOC (LIB$_INPSTRTRU)) .AND.
    (STATUS .NE. %LOC (RMS$_EOF))) THEN
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
! Display text until end-of-file
DO WHILE (STATUS .NE. %LOC (RMS$_EOF))
  STATUS = LBR$OUTPUT_HELP (LIB$PUT_OUTPUT,,
                            KEYWORD (1:KEYWORD_LEN),
                            LIBSPEC (1:LIBSPEC_LEN),
                            FLAGS,
                            LIB$GET_INPUT)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Get another keyword
  STATUS = LIB$GET_INPUT (KEYWORD,
                          'Keyword or Ctrl/Z: ',
                          KEYWORD_LEN)
  IF ((.NOT. STATUS) .AND.
      (STATUS .NE. %LOC (LIB$_INPSTRTRU)) .AND.
      (STATUS .NE. %LOC (RMS$_EOF))) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END DO

! Exit
END
```

13.2.9. Listing and Processing Index Entries

You can process index entries an entry at a time by invoking LBR$GET_INDEX. The fourth argument specifies a match name for the entry or entries in the index to be processed: you can include the asterisk (*) and percent (%) characters in the match name for generic processing. For example, MOD* means all

entries whose names begin with MOD; and MOD% means all entries whose names are four characters and begin with MOD.

The third argument names a user-written routine that is executed once for each index entry specified by the fourth argument. The routine must be a function declared as external that returns a success (odd number) or failure (even number) status. LBR$GET_INDEX processing stops on a return status of failure. Declare the first argument passed to the function as a passed-length character argument; this argument contains the name of the index entry. Declare the second argument as an integer array of two elements.

Example 13.10 obtains a match name from the command line and displays the names of the matching entries from index 1 (the index containing the names of the modules).

Example 13.10. Displaying Index Entries

```
SUBROUTINE LIST (INDEX)
! Lists modules in the library

INTEGER STATUS,         ! Return status
        INDEX,          ! Library index
CHARACTER*31 MATCHNAME ! Name of module to list
INTEGER MATCHNAME_LEN  ! Length of match name
! VMS library procedures
INTEGER address LBR$GET_INDEX,
        LIB$LOCC
! Match routine
INTEGER MATCH
EXTERNAL MATCH
! Get module name and calculate length
CALL CLI$GET_VALUE ('LIST', MATCHNAME)
MATCHNAME_LEN = LIB$LOCC (' ', MATCHNAME) - 1
! Call routine to display module names
STATUS = LBR$GET_INDEX (INDEX,
                        1, ! Primary index
                        MATCH,
                        MATCHNAME (1:MATCHNAME_LEN))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

! Exit
END
INTEGER FUNCTION MATCH (MODNAME, RFA)
! Function called for each module matched by MATCHNAME
! Displays the module name
INTEGER STATUS_OK,         ! Good return status
        RFA (2)            ! RFA of module name in index
PARAMETER (STATUS_OK = 1) ! Odd value
CHARACTER*(*) MODNAME      ! Name of module
! Display the name
TYPE *, MODNAME ! Display module name

! Exit
MATCH = STATUS_OK
END
```

13.3. LBR Routines

This section describes the individual LBR routines.

LBR$CLOSE

Close a Library — The LBR$CLOSE routine closes an open library.

Format

```
LBR$CLOSE library_index
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

Description

When you are finished working with a library, you should call LBR$CLOSE to close it. Upon successful completion, LBR$CLOSE closes the open library and deallocates all of the memory used for processing it.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$DELETE_DATA

Delete Module Data from the Library — The LBR$DELETE_DATA routine deletes module data from the library.

Format

```
LBR$DELETE_DATA library_index, txtrfa [,flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Record's file address (RFA) of the module header for the module you want to delete. The *txtrfa* argument is the address of the 2-longword array that contains the RFA. You can obtain the RFA of a module header by calling LBR$LOOKUP_KEY or LBR$PUT_RECORD.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

The contents of the flag are ignored. The purpose of this argument is to indicate to this routine that the application knows about the new index structure for ELF object and ELF shareable image libraries.

Description

To delete a library module, first call LBR$DELETE_KEY to delete all keys that point to it. If no library index keys are pointing to the module header, LBR$DELETE_DATA deletes the module header and associated data records; otherwise, this routine returns the error LBR$_STILLKEYS.

Note that other library routines can reuse data blocks that contain no data.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_INVRFA

Specified RFA not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$_STILLKEYS

Keys in other indexes still point to the module header. Therefore, the specified module was not deleted.

LBR$DELETE_KEY

Delete a Key — The LBR$DELETE_KEY routine removes a key from the current library index.

Format

```
LBR$DELETE_KEY library_index, key_name[, txtrfa] [, flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of a longword that contains the index.

key_name

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The key to be deleted from the library index. For libraries with binary keys, the *key_name* argument is the address of an unsigned longword containing the key number.

For libraries with ASCII keys, the *key_name* argument is the address of the string descriptor pointing to the key with the following argument characteristics:

Argument Characteristics	Entry
OpenVMS usage	char_string
type	character string
access	read only
mechanism	by descriptor

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The *txtrfa* argument is the address of the 2-longword array that contains the record file address (RFA). If present and if the *flags* argument is not present, the routine scans for all types of the key for the specified *txtrfa* and delete those entries.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

If present, this argument indicates that a particular type of the key or all types of the key is to be deleted. The flags bits are as follows:

Flag Bits	Description
LBR$M_SYM_WEAK = 0x1	UNIX-style weak symbol attribute
LBR$M_SYM_GROUP = 0x2	Group symbol attribute
LBR$M_SYM_ALL = 0x80000000	All symbols

If the *txtrfa* argument is not present or if its value is zero, the type indicated by *flags* is deleted. If *txtrfa* specifies a nonzero value, the entry of the type indicated, with the *txtrfa* supplied, is removed. Note that only one type or all types can be specified.

Description

If LBR$DELETE_KEY finds the key specified by *key_name* in the current index, it deletes the key. Note that if you want to delete a library module, you must first use LBR$DELETE_KEY to delete all the keys that point to it, then use LBR$DELETE_DATA to delete the module's header and associated data. You cannot call LBR$DELETE_KEY from within the user-supplied routine specified in LBR$SEARCH or LBR$GET_INDEX.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_KEYNOTFND

Specified key not found.

LBR$_LIBNOTOPN

Specified library not open.

LBR$_UPDIRTRAV

Specified index update not valid in a user-supplied routine specified in LBR$SEARCH or LBR $GET_INDEX.

LBR$FIND

Look Up a Module by Its RFA — The LBR$FIND routine sets the current internal read context for the library to the library module specified.

Format

```
LBR$FIND library_index ,txtrfa
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)

access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Record's file address (RFA) of the module header for the module you want to access. The *txtrfa* argument is the address of a 2-longword array containing the RFA. You can obtain the RFA of a module header by calling LBR$LOOKUP_KEY or LBR$PUT_RECORD.

Description

Use the LBR$FIND routine to access a module that you had accessed earlier in your program. For example, if you look up several keys with LBR$LOOKUP_KEY, you can save the RFAs returned by LBR$LOOKUP_KEY and later use LBR$FIND to reaccess the modules. Thus, you do not have to look up the module header's key every time you want to access the module. If the specified RFA is valid, LBR$FIND initializes internal tables so you can read the associated data.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_INVRFA

Specified RFA not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$FLUSH

Recover Virtual Memory — The LBR$FLUSH routine writes modified blocks back to the library file and frees the virtual memory the blocks had been using.

Format

```
LBR$FLUSH library_index ,block_type
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

block_type

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Extent of the flush operation. The *block_type* argument contains the longword value that indicates how the flush operation proceeds. If you specify LBR$C_FLUSHDATA, the data blocks are flushed. If you specify LBR$C_FLUSHALL, first the data blocks and then the current library index are flushed.

Each programming language provides an appropriate mechanism for accessing these symbols.

Description

LBR$FLUSH cannot be called from other LBR routines that reference cache addresses or by routines called by LBR routines.

Condition Values Returned

LBR$_NORMAL

Operation completed successfully.

LBR$_BADPARAM

Error. A value passed to the LBR$FLUSH routine was either out of range or an illegal value.

LBR$_WRITERR

Error. An error occurred during the writing of the cached update blocks to the library file.

LBR$GET_HEADER

Retrieve Library Header Information — The LBR$GET_HEADER routine returns information from the library's header to the caller.

Format

```
LBR$GET_HEADER library_index ,retary
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

retary

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Array of 128 longwords that receives the library header. The *retary* argument is the address of the array that contains the header information. The information returned in the array is listed in the following table. Each programming language provides an appropriate mechanism for accessing this information.

Offset in Longwords	Symbolic Name	Contents
0	LHI$L_TYPE	Library type (see LBR$OPEN for possible values)
1	LHI$L_NINDEX	Number of indexes

Offset in Longwords	Symbolic Name	Contents
2	LHI$L_MAJORID	Library format major identification
3	LHI$L_MINORID	Library format minor identification
4	LHI$T_LBRVER	ASCIC version of Librarian
12	LHI$L_CREDAT	Creation date/time
14	LHI$L_UPDTIM	Date/time of last update
16	LHI$L_UPDHIS	Virtual block number (VBN) of start of update history
17	LHI$L_FREEVBN	First logically deleted block
18	LHI$L_FREEBLK	Number of deleted blocks
19	LHI$B_NEXTRFA	Record file address (RFA) of end of library
21	LHI$L_NEXTVBN	Next VBN to allocate at end of file
22	LHI$L_FREIDXBLK	Number of free preallocated index blocks
23	LHI$L_FREEIDX	List head for preallocated index blocks
24	LHI$L_HIPREAL	VBN of highest preallocated block
25	LHI$L_IDXBLKS	Number of index blocks in use
26	LHI$L_IDXCNT	Number of index entries (total)
27	LHI$L_MODCNT	Number of entries in index 1 (module names)
28	LHI$L_MHDUSZ	Number of bytes of additional information reserved in module header
29	LHI$L_MAXLUHREC	Maximum number of library update history records maintained
30	LHI$L_NUMLUHREC	Number of library update history records in history
31	LHI$L_LIBSTATUS	Library status (false if there was an error closing the library)
32-128		Reserved by VSI

Description

On successful completion, LBR$GET_HEADER places the library header information into the array of 128 longwords.

Note that the offset is the byte offset of the value into the header structure. You can convert the offset to a longword subscript by dividing the offset by 4 and adding 1 (assuming that subscripts in your programming language begin with 1).

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$GET_HELP

Retrieve Help Text — The LBR$GET_HELP routine retrieves help text from a help library, displaying it on SYS$OUTPUT or calling your routine for each record returned.

Format

```
LBR$GET_HELP library_index [,line_width] [,routine] [,data] [,key_1]
 [,key_2…,key_10]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

line_width

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Width of the help text line. The *line_width* argument is the address of a longword containing the width of the listing line. If you do not supply a line width or if you specify 0, the line width defaults to 80 characters per line.

routine

OpenVMS usage:	procedure

type:	procedure value
access:	read only
mechanism:	by reference

Routine called for each line of text you want output. The *routine* argument is the address of the procedure value for this user-written routine.

If you do not supply a *routine* argument, LBR$GET_HELP calls the Run-Time Library procedure LIB$PUT_OUTPUT to send the help text lines to the current output device (SYS$OUTPUT). However, if you want SYS$OUTPUT for your program to be a disk file rather than the terminal, you should supply a routine to output the text.

If the user-written routine returns an error status with low bit clear, the LBR$GET_HELP routine passes this status to the caller. If the user-written routine returns a success status with low bit set, the LBR $GET_HELP routine returns 1 to the caller.

The routine you specify is called with an argument list of four longwords:

1. The first argument is the address of a string descriptor for the output line.
2. The second argument is the address of an unsigned longword containing flag bits that describe the contents of the text being passed. The possible flags are as follows:

HLP$M_NOHLPTXT	Specified help text cannot be found.
HLP$M_KEYNAMLIN	Text contains key names of the printed text.
HLP$M_OTHERINFO	Text is part of the information provided on additional help available.

   Each programming language provides an appropriate mechanism for accessing these flags. Note that, if no flag bit is set, help text is passed.
3. The third argument is the address stipulated in the data argument specified in the call to LBR $GET_HELP (or the address of a 0 constant if the data argument is zero or was omitted).
4. The fourth argument is a longword containing the address of the current key level.

The routine you specify must return with success or failure status. A failure status (low bit = 0) terminates the current call to LBR$GET_HELP.

data

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Data passed to the routine specified in the *routine* argument. The *data* argument is the address of data for the routine. The address is passed to the routine specified in the *routine* argument. If you omit this argument or specify it as zero, then the argument passed in your routine will be the address of a zero constant.

key_1,key_2, …,key_10

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by descriptor

Level of the help text to be output. Each *key_1*,*key_2*, …,*key_10* argument is the address of a descriptor pointing to the key for that level.

If the *key_1* descriptor is 0 or if it is not present, LBR$GET_HELP assumes that the *key_1* name is HELP, and it ignores all the other keys. For *key_2* through *key_10*, a descriptor address of 0, or a length of 0, or a string address of 0 terminates the list.

The *key* argument may contain any of the following special character strings:

String	Meaning
*	Return all level 1 help text in the library.
KEY…	Return all help text associated with the specified key and its subkeys (valid for level 1 keys only).
*...	Return all help text in the library.

Description

LBR$GET_HELP returns all help text in the same format as the output returned by the DCL command HELP; that is, it indents two spaces for every key level of text displayed. (Because of this formatting, you may want to make your help messages shorter than 80 characters, so they fit on one line on terminal screens with the width set to 80.) If you do not want the help text indented to the appropriate help level, you must supply your own routine to change the format.

Note that most application programs use LBR$OUTPUT_HELP instead of LBR$GET_HELP.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$_NOTHLPLIB

Specified library not a help library.

LBR$GET_HISTORY

Retrieve a Library Update History Record — The LBR$GET_HISTORY routine returns each library update history record to a user-specified action routine.

Format

```
LBR$GET_HISTORY library_index ,action_routine
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

action_routine

OpenVMS usage:	procedure
type:	procedure value
access:	modify
mechanism:	by reference

User-supplied routine for processing library update history records. The *action_routine* argument is the address of the procedure value of this user-supplied routine. The routine is invoked once for each update history record in the library. One argument is passed to the routine, namely, the address of a descriptor pointing to a history record.

Description

This routine retrieves the library update history records written by the routine LBR$PUT_HISTORY.

Condition Values Returned

LBR$_NORMAL

Normal exit from the routine.

LBR$_EMPTYHIST

History empty. This is an informational code, not an error code.

LBR$_INTRNLERR

Internal Librarian routine error occurred.

LBR$_NOHISTORY

No update history. This is an informational code, not an error code.

LBR$GET_INDEX

Call a Routine for Selected Index Keys — The LBR$GET_INDEX routine calls a user-supplied routine for selected keys in an index.

Format

```
LBR$GET_INDEX library_index ,index_number ,routine_name [,match_desc] [,
 flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

index_number

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Number of the library index. The *index_number* argument is the address of a longword containing the index number. This is the index number associated with the keys you want to use as input to the user-supplied routine.

routine_name

OpenVMS usage:	procedure

type:	procedure value
access:	read only
mechanism:	by reference

User-supplied routine called for each of the specified index keys. The *routine_name* argument is the address of the procedure value for this user-supplied routine.

LBR$GET_INDEX passes two arguments to the routine on OpenVMS Alpha; and passes three arguments to the routine on OpenVMS Integrity servers:

- A key name.
  - For libraries with ASCII keys, the *key_name* argument is the address of a string descriptor pointing to the key. Note that the string and the string descriptor passed to the routine are valid only for the duration of that call. The string must be copied privately if you need it again for more processing.
  - For libraries with binary keys, the *key_name* argument is the address of an unsigned longword containing the key number.
- The record file address (RFA) of the module's header for this key name. The RFA argument is the address of a 2-longword array that contains the RFA.
- The key's type whose bits are as follows:

Flag Bits	Description
LBR$M_SYM_WEAK = 1	UNIX-style weak symbol attributes
LBR$M_SYM_GROUP = 2	Group symbol attribute

  This parameter is passed only on OpenVMS Integrity servers.

The user routine must return a value to indicate success or failure. If the user routine returns a false value (low bit = 0), LBR$GET_INDEX stops searching the index and returns the status value of the user-specified routine to the calling program.

The routine cannot contain calls to either LBR$DELETE_KEY or LBR$INSERT_KEY.

match_desc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Key matching identifier. The *match_desc* argument is the address of a string descriptor pointing to a string used to identify which keys result in calls to the user-supplied routine. Wildcard characters are allowed in this string. If you omit this argument, the routine is called for every key in the index. The *match_desc* argument is valid only for libraries that have ASCII keys.

flags

OpenVMS usage:	mask_longword

type:	longword (unsigned)
access:	read only
mechanism:	by value

If present and non-zero, this argument specifies the type, or all types, of the key provided. The flag bits are:

Flag Bits	Description
LBR$M_SYM_WEAK = 0x1	UNIX-style weak symbol attribute
LBR$M_SYM_GROUP = 0x2	Group symbol attribute
LBR$M_SYM_ALL = 0x80000000	All symbols

The user routine will be provided the key's type through an additional third parameter.

Description

LBR$GET_INDEX searches through the specified index for keys that match the *match_desc* argument. Each time it finds a match, it calls the user routine specified by the *routine_name* argument. If you do not specify the *match_desc* argument, LBR$GET_INDEX calls the user routine for every key in the index.

For example, if you call LBR$GET_INDEX on an object library with *match_desc* equal to TR* and *index_number* set to 1 (module name table), then LBR$GET_ INDEX calls *routine_name* for each module whose name begins with TR.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_ILLIDXNUM

Specified index number not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$_NULIDX

Specified library empty.

LBR$GET_RECORD

Read a Data Record — The LBR$GET_RECORD routine returns the next data record in the module associated with a specified key.

Format

```
LBR$GET_RECORD library_index [,inbufdes] [,outbufdes]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index. The library must be open and LBR $LOOKUP_KEY or LBR$FIND must have been called to find the key associated with the module whose records you want to read.

inbufdes

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

User buffer to receive the record. The *inbufdes* argument is the address of a string descriptor that points to the buffer that receives the record from LBR$GET_RECORD. This argument is required when the Librarian subroutine record access is set to move mode (which is the default). This argument is not used if the record access mode is set to locate mode. The Description section contains more information about the locate and move modes.

outbufdes

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

String descriptor that receives the actual length and address of the data for the record returned. The *outbufdes* argument is the address of the string descriptor for the returned record. The length and address fields of the string descriptor are filled in by the LBR$GET_RECORD routine. This parameter must be specified when Librarian subroutine record access is set to locate mode. This parameter is

optional if record access mode is set to move mode. The Description section contains more information about the locate and move modes.

Description

Before calling LBR$GET_RECORD, you must first call LBR$LOOKUP_KEY or LBR$FIND to set the internal library read context to the record's file address (RFA) of the module header of the module whose records you want to read.

LBR$GET_RECORD uses two record access modes: locate mode and move mode. Move mode is the default. The LBR$SET_LOCATE and LBR$SET_MOVE subroutines set these modes. The record access modes are mutually exclusive; that is, when one is set, the other is turned off. If move mode is set, LBR$GET_RECORD copies the record to the user-specified buffer described by *inbufdes*. If you have optionally specified the output buffer string descriptor, *outbufdes*, the Librarian fills it with the actual length and address of the data. If locate mode is set, LBR$GET_RECORD returns the record by way of an internal subroutine buffer, pointing the *outbufdes* descriptor to the internal buffer. The second parameter, *inbufdes*, is not used when locate mode is set.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$_LKPNOTDON

Requested key lookup not done.

RMS$_EOF

Error. An attempt has been made to read past the logical end of the data in the module.

LBR$INI_CONTROL

Initialize a Library Control Structure — The LBR$INI_CONTROL routine initializes a control structure, called a library control index, to identify the library for use by other LBR routines.

Format

```
LBR$INI_CONTROL library_index ,func [,type] [,namblk]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of a longword that is to receive the index.

func

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library function to be performed. The *func* argument is the address of the longword that contains the library function. Valid functions are LBRC_CREATE, LBRC_READ, and LBR$C_UPDATE. Each programming language provides an appropriate mechanism for accessing these symbols.

type

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library type. The *type* argument is the address of the longword containing the library type. Valid library types include the following:

- LBR$C_TYP_EOBJ (Alpha object)
- LBR$C_TYP_ESHSTB (Alpha shareable image)
- LBR$C_TYP_MLB (macro)
- LBR$C_TYP_HLP (help)
- LBR$C_TYP_TXT (text)
- LBR$C_TYP_UNK (unknown)
- LBR$C_TYP_NCS (NCS library)
- For user-developed libraries, a type in the range of LBR$C_TYP_USRLW through LBR $C_TYP_USRHI.

namblk

OpenVMS usage:	nam
type:	longword (unsigned)
access:	read only
mechanism:	by reference

OpenVMS RMS name block (NAM). The *namblk* argument is the address of a variable-length data structure containing an RMS NAM block. The LBR$OPEN routine fills in the information in the NAM block so it can be used later to open the library. If the NAM block has this file identification in it from previous use, the LBR$OPEN routine uses the open-by-NAM block option. This argument is optional and should be used if the library will be opened many times during a single run of the program. For a detailed description of RMS NAM blocks, see the *VSI OpenVMS Record Management Services Reference Manual*.

Description

Except for the LBR$OUTPUT_HELP routine, you must call LBR$INI_CONTROL before calling any other LBR routine. After you initialize the library control index, you must open the library or create a new one using the LBR$OPEN routine. You can then call other LBR routines that you need. After you finish working with a library, close it with the LBR$CLOSE routine.

LBR$INI_CONTROL initializes a library by filling the longword referenced by the *library_index* argument with the control index of the library. Upon completion of the call, the index can be used to refer to the current library in all future routine calls. Therefore, your program must not alter this value.

You can have up to 16 libraries open simultaneously in your program.

Condition Values Returned

LBR$_NORMAL

Library control index initialized successfully.

LBR$_ILLFUNC

Requested function not valid.

LBR$_ILLTYP

Specified library type not valid.

LBR$_TOOMNYLIB

Error. An attempt was made to allocate more than 16 control indexes.

LBR$INSERT_KEY

Insert a New Key — The LBR$INSERT_KEY routine inserts a new key in the current library index.

Format

```
LBR$INSERT_KEY library_index ,key_name ,txtrfa [, flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL library routine. The *library_index* argument is the address of the longword that contains the index.

key_name

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Name of the new key you are inserting.

If the library uses binary keys, the *key_name* argument is the address of an unsigned longword containing the value of the key.

If the library uses ASCII keys, the *key_name* argument is the address of a string descriptor of the key with the following argument characteristics:

Argument Characteristics	Entry
OpenVMS usage	char_string
type	character string
access	read only
mechanism	by descriptor

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	modify
mechanism:	by reference

The record file address (RFA) of the module associated with the new key you are inserting. The *txtrfa* argument is the address of a 2-longword array containing the RFA. You can use the RFA returned by the first call to LBR$PUT_RECORD.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

If present, specifies the key's type. The flag bits are as follows:

Flag Bits	Description
LBR$M_SYM_WEAK = 0x1	UNIX-style weak symbol attribute
LBR$M_SYM_GROUP = 0x2	Group symbol attribute

If this argument is not present, the normal NonGroup-Global type is the assumed type.

Description

The LBR$INSERT_KEY routine inserts a new key in the current library index. You cannot call LBR $INSERT_KEY within the user-supplied routine specified in LBR$SEARCH or LBR$GET_INDEX.

Condition Values Returned

LBR$_DUPKEY

Index already contains the specified key.

LBR$_ILLCTL

Specified library control index not valid.

LBR$_INVRFA

Specified RFA does not point to valid data.

LBR$_LIBNOTOPN

Specified library not open.

LBR$_UPDURTRAV

LBR$INSERT_KEY is called by the user-defined routine specified in LBR$SEARCH or LBR $GET_INDEX.

LBR$LOOKUP_KEY

Look Up a Library Key — The LBR$LOOKUP_KEY routine looks up a key in the library's current index and prepares to access the data in the module associated with the key.

Format

```
LBR$LOOKUP_KEY library_index ,key_name ,txtrfa [, flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

key_name

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Name of the library key. If the library uses binary keys, the *key_name* argument is the address of the unsigned longword value of the key.

If the library uses ASCII keys, the *key_name* argument is the address of a string descriptor for the key with the following argument characteristics:

Argument Characteristics	Entry
OpenVMS usage	char_string
type	character string
access	read only
mechanism	by descriptor

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)

access:	write only
mechanism:	by reference

The record file address (RFA) of the library module header. The *txtrfa* argument is the address of the 2-longword array that receives the RFA of the module header.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	write only
mechanism:	by reference

The *flags* argument, if present and not zero, receives the type of key returned. The flag bits are as follows:

Flag Bits	Description
LBR$SYM_WEAK = 0x1	UNIX-style weak symbol attribute
LBR$SYM_GROUP = 0x2	Group symbol attribute

The key returned is the highest precedent definition type present.

Description

If LBR$LOOKUP_KEY finds the specified key, it initializes internal tables so you can access the associated data.

This routine returns the RFA to the 2-longword array referenced by *txtrfa*.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_INVRFA

RFA obtained not valid.

LBR$_KEYNOTFND

Specified key not found.

LBR$_LIBNOTOPN

Specified library not open.

LBR$LOOKUP_TYPE

Searches index and returns key type for the module — The LBR$LOOK_TYPE routine searches the index for the key from a particular module (RFA) and returns that key's type for that module.

Format

```
LBR$LOOKUP_TYPE library_index, key_name, txtrfa, ret_types
```

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

key_name

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The *key_name* argument is the address of the string descriptor pointing to the key with the following argument characteristics:

Argument Characteristics	Entry
OpenVMS usage	char_string
type	character string
access	read only
mechanism	by descriptor

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The module's record file address (RFA) of the library module header. The *txtrfa* argument is the address of the 2-longword array that specifies the RFA of the module header.

ret_types

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	write only

mechanism:	by reference

The address of a longword to receive the symbol types found for the specified module (*txtrfa*). The return type bits are as follows:

LBR$M_SYM_NGG = 1
LBR$M_SYM_UXWK = 2
LBR$M_SYM_GG = 4
LBR$M_SYM_GUXWK = 8

Description

This routine searches the index for the key from a particular module (RFA) and returns that key's type for that module, if present. Otherwise, it returns LBR$_ KEYNOTFND.

LBR$MAP_MODULE

Maps a module into process P2 space (Integrity servers only) — The LBR$MAP_MODULE routine maps a module into process P2 space.

Format

```
LBR$MAP_MODULE library_index, ret_va_addr, ret_mod_len, txtrfa
```

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL library routine. The *library_index* argument is the address of the longword that contains the index.

ret_va_addr

OpenVMS usage:	address
type:	quadword address
access:	write only
mechanism:	by 32-bit or 64-bit reference

The 32-bit or 64-bit virtual address of a naturally aligned quadword into which the routine returns the virtual address at which the routine mapped the library module.

ret_mod_len

OpenVMS usage:	byte_count
type:	quadword (unsigned)

access:	read only
mechanism:	by reference

The address of a naturally aligned quadword into which the library routine returns the module length.

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The module's record file address (RFA) of the library module header. The txtrfa argument is the address of the 2-longword array that specifies the RFA of the module header.

Description

This routine maps a module, with the given txtrfa, into process P2 memory space and returns the virtual address where the module is mapped and the module size.

Unlike other LBR services that use RMS services, LBR$MAP_MODULE also uses system services. Because of this, the secondary status for error returns is placed in LBR$$GL_SUBSTS. Use this secondary status to find additional status when an error is returned.

LBR$OPEN

Open or Create a Library — The LBR$OPEN routine opens an existing library or creates a new one.

Format

```
LBR$OPEN library_index [,fns] [,create_options] [,dns] [,rlfna] [,rns]
 [,rnslen]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only

mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of a longword containing the index.

fns

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

File specification of the library. The *fns* argument is the address of a string descriptor pointing to the file specification. Unless the OpenVMS RMS NAM block address was previously supplied in the LBR $INI_CONTROL routine and contained a file specification, this argument must be included. Otherwise, the Librarian returns an error (LBR$_NOFILNAM).

create_options

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library characteristics. The *create_options* argument is the address of an array of 20 longwords that define the characteristics of the library you are creating. If you are creating a library with LBR $C_CREATE, you must include the *create_options* argument. The following table shows the entries that the array must contain. Each programming language provides an appropriate mechanism for accessing the listed symbols.

Offset in Longwords	Symbolic Name	Contents
0	CRE$L_TYPE	Library type:
	LBR$C_TYP_UNK (0)	Unknown/unspecified
	LBR$C_TYP_OBJ (1)	VAX object
	LBR$C_TYP_MLB (2)	Macro
	LBR$C_TYP_HLP (3)	Help
	LBR$C_TYP_TXT (4)	Text
	LBR$C_TYP_SHSTB (5)	VAX shareable image
	LBR$C_TYP_NCS (6)	NCS
	LBR$C_TYP_EOBJ (7)	Alpha object
	LBR$C_TYP_ESHSTB (8)	Alpha shareable image
	(9–127)	Reserved by VSI
	LBR$C_TYP_USRLW (128)	User library types — low end of range
	LBR$C_TYP_USRHI (255)	User library types — high end of range

Offset in Longwords	Symbolic Name	Contents
1	CRE$L_KEYLEN	Maximum length of ASCII keys or, if 0, indicates 32-bit unsigned keys (binary keys)
2	CRE$L_ALLOC	Initial library file allocation
3	CRE$L_IDXMAX	Number of library indexes (maximum of eight)
4	CRE$L_UHDMAX	Number of additional bytes to reserve in module header
5	CRE$L_ENTALL	Number of index entries to preallocate
6	CRE$L_LUHMAX	Maximum number of library update history records to maintain
7	CRE$L_VERTYP	Format of library to create:
	CRE$C_VMSV2	VMS Version 2.0
	CRE$C_VMSV3	VMS Version 3.0
8	CRE$L_IDXOPT	Index key casing option:
	CRE$C_HLPCASING	Treat character case as it is for help libraries
	CRE$C_OBJCASING	Treat character case as it is for object libraries
	CRE$C_MACTXTCAS	Treat character case as it is for macro and text libraries
9–19		Reserved by VSI

The input of uppercase and lowercase characters is treated differently for help, object, macro, and text libraries. For details, see the *VSI OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

dns

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Default file specification. The `dns` argument is the address of the string descriptor that points to the default file specification. See the *VSI OpenVMS Record Management Services Reference Manual* for details about how defaults are processed.

rlfna

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only

mechanism:	by reference

Related file name. The *rlfna* argument is the address of an RMS NAM block pointing to the related file name. You must specify *rlfna* for related file name processing to occur. If a related file name is specified, only the file name, type, and version fields of the NAM block are used for related name block processing. The device and directory fields are not used. See the *VSI OpenVMS Record Management Services Reference Manual* for details on processing related file names.

rns

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Resultant file specification returned. The *rns* argument is the address of a string descriptor pointing to a buffer that is to receive the resultant file specification string. If an error occurs during an attempt to open the library, the expanded name string is returned instead.

rnslen

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	write only
mechanism:	by reference

Length of the resultant or expanded file name. The *rnslen* argument is the address of a longword receiving the length of the resultant file specification string (or the length of the expanded name string if there was an error in opening the library).

Description

You can call this routine only after you call LBR$INI_CONTROL and before you call any other LBR routine except LBR$OUTPUT_HELP.

When the library is successfully opened, the LBR routine reads the library header into memory and sets the default index to 1.

If the library cannot be opened because it is already open for a write operation, LBR$OPEN retries the open operation every second for a maximum of 30 seconds before returning the RMS error, RMS $_FLK, to the caller.

Condition Values Returned

LBR$_ERRCLOSE

Error. When the library was last modified while opened for write access, the write operation was interrupted. This left the library in an inconsistent state.

LBR$_ILLCREOPT

Requested create options not valid or not supplied.

LBR$_ILLCTL

Specified library control index not valid.

LBR$_ILLFMT

Specified library format not valid.

LBR$_ILLFUNC

Specified library function not valid.

LBR$_LIBOPN

Specified library already open.

LBR$_NOFILNAM

Error. The *fns* argument was not supplied or the RMS NAM block was not filled in.

LBR$_OLDLIBRARY

Success. The specified library has been opened; the library was created with an old library format.

LBR$_OLDMISMCH

Requested library function conflicts with old library type specified.

LBR$_TYPMISMCH

Library type does not match the requested type.

LBR$OUTPUT_HELP

Output Help Messages — The LBR$OUTPUT_HELP routine outputs help text to a user-supplied output routine. The text is obtained from an explicitly named help library or, optionally, from user-specified default help libraries. An optional prompting mode is available that enables LBR$OUTPUT_HELP to interact with you and continue to provide help information after the initial help request has been satisfied.

Format

```
LBR$OUTPUT_HELP output_routine [,output_width] [,line_desc] [,library_name]
 [,flags] [,input_routine]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

output_routine

OpenVMS usage:	procedure
type:	procedure value
access:	write only
mechanism:	by reference

Name of a routine that writes help text a line at a time. The *output_routine* argument is the address of the procedure value of the routine to call. You should specify either the address of LIB $PUT_OUTPUT or a routine of your own that has the same calling format as LIB$PUT_OUTPUT.

output_width

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Width of the help-text line to be passed to the user-supplied output routine. The *output_width* argument is the address of a longword containing the width of the text line to be passed to the user-supplied output routine. If you omit *output_width* or specify it as 0, the default output width is 80 characters per line.

line_desc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Contents of the help request line. The *line_desc* argument is the address of a string descriptor pointing to a character string containing one or more help keys defining the help requested, for example, the HELP command line minus the HELP command and HELP command qualifiers. The default is a string descriptor for an empty string.

library_name

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Name of the main library. The *library_name* argument is the address of a string descriptor pointing to the main library file specification string. The default is a null string, which means you should use the default help libraries. If you omit the device and directory specifications, the default is SYS$HELP. The default file type is .HLB.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags specifying help output options. Each programming language provides an appropriate mechanism for accessing these flags. The *flags* argument is the address of an unsigned longword that contains the following flags, when set:

Flag	Description
HLP$M_PROMPT	Interactive help prompting is in effect.
HLP$M_PROCESS	The process logical name table is searched for default help libraries.
HLP$M_GROUP	The group logical name table is searched for group default help libraries.
HLP$M_SYSTEM	The system logical name table is searched for system default help libraries.
HLP$M_LIBLIST	The list of default libraries available is output with the list of topics available.
HLP$M_HELP	The list of topics available in a help library is preceded by the major portion of the text on help.

If you omit this longword, the default is for prompting and all default library searching to be enabled, but no library list is generated and no help text precedes the list of topics.

input_routine

OpenVMS usage:	procedure
type:	procedure value
access:	read only
mechanism:	by reference

Routine used for prompting. The *input_routine* argument is the address of the procedure value of the prompting routine. You should specify either the address of LIB$GET_INPUT or a routine of your own that has the same calling format as LIB$GET_INPUT. This argument must be supplied when the HELP command is run in prompting mode (that is, HLP$M_PROMPT is set or defaulted).

Description

The LBR$OUTPUT_HELP routine provides a simple, one-call method to initiate an interactive help session. Help library bookkeeping functions, such as LBR$INI_CONTROL and LBR$OPEN, are handled internally. You should not call LBR$INI_CONTROL or LBR$OPEN before you issue a call to LBR$OUTPUT_HELP.

LBR$OUTPUT_HELP accepts help keys in the same format as LBR$GET_HELP, with the following qualifications:

- If the keyword HELP is supplied, help text on HELP is output, followed by a list of HELP subtopics available.

If no help keys are provided or if the *line_desc* argument is 0, a list of topics available in the root library is output.

- If the *line_desc* argument contains a list of help keys, then each key must be separated from its predecessor by a slash (/) or by one or more spaces.
- The first key can specify a library to replace the main library as the root library (the first library searched) in which LBR$OUTPUT_HELP searches for help. A key used for this purpose must have the form *<@filespec>*, where *filespec* is subject to the same restrictions as the *library_name* argument. If the specified library is an enabled user-defined default library, then *filespec* can be abbreviated as any unique substring of that default library's logical name translation.

In default library searches, you can define one or more default libraries for LBR$OUTPUT_HELP to search for help information not contained in the root library. Do this by equating logical names (HLP $LIBRARY, HLP$LIBRARY_1, …, HLP$LIBRARY_999) to the file specifications of the default help libraries. You can define these logical names in the process, group, or system logical name table.

If default library searching is enabled by the *flags* argument, LBR$OUTPUT_HELP uses those flags to determine which logical name tables are enabled and then automatically searches any user default libraries that have been defined in those logical name tables. The library search order proceeds as follows: root library, main library (if specified and different from the root library), process libraries (if enabled), group libraries (if enabled), system libraries (if enabled). If the requested help information is not found in any of these libraries, LBR$OUTPUT_HELP returns to the root library and issues a “help not found” message.

To enter an interactive help session (after your initial request for help has been satisfied), you must set the HLP$M_PROMPT bit in the *flags* argument.

You can encounter four different types of prompt in an interactive help session. Each type represents a different level in the hierarchy of help available to you.

1. If the root library is the main library and you are not currently examining HELP for a particular topic, the prompt *Topic?* is output.
2. If the root library is a library other than the main library and if you are not currently examining HELP for a particular topic, a prompt of the form @ <*library-spec*> *Topic?* is output.
3. If you are currently examining HELP for a particular topic (and subtopics), a prompt of the form <*keyword...*>*subtopic?* is output.
4. A combination of 2 and 3.

When you encounter one of these prompt messages, you can respond in any one of several ways. Each type of response and its effect on LBR$OUTPUT_HELP in each prompting situation is described in the following table:

Response	**Action in the Current Prompt Environment** [1]
keyword […]	(1,2) Search all enabled libraries for these keys.
	(3,4) Search additional help for the current topic (and subtopic) for these keys.
@filespec [keyword[…]]	(1,2) Same as above, except that the root library is the library specified by *filespec*. If the specified

Response	Action in the Current Prompt Environment [1]
	library does not exist, treat @*filespec* as a normal key.
	(3,4) Same as above; treat @*filespec* as a normal key.
?	(1,2) Display a list of topics available in the root library.
	(3,4) Display a list of subtopics of the current topic (and subtopics) for which help exists.
Carriage Return	(1) Exit from LBR$OUTPUT_HELP.
	(2) Change root library to main library.
	(3,4) Strip the last keyword from a list of keys defining the current topic (and subtopic) environment.
Ctrl/Z	(1,2,3,4) Exit from LBR$OUTPUT_HELP.

[1]Keyed to the prompt in the preceding list.

Condition Values Returned

LBR$_ILLINROU

Input routine improperly specified or omitted.

LBR$_ILLOUTROU

Output routine improperly specified or omitted.

LBR$_NOHLPLIS

Error. No default help libraries can be opened.

LBR$_TOOMNYARG

Error. Too many arguments were specified.

LBR$_USRINPERR

Error. An error status was returned by the user-supplied input routine.

LBR$PUT_END

Write an End-of-Module Record — The LBR$PUT_END routine marks the end of a sequence of records written to a library by the LBR$PUT_RECORD routine.

Format

```
LBR$PUT_END library_index
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of a longword containing the index.

Description

Call LBR$PUT_END after you write data records to the library with the LBR$PUT_RECORD routine. LBR$PUT_END terminates a module by attaching a 3-byte logical end-of-file record (hexadecimal 77,00,77) to the data.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$PUT_HISTORY

Write an Update History Record — The LBR$PUT_HISTORY routine adds an update history record to the end of the update history list.

Format

```
LBR$PUT_HISTORY library_index ,record_desc
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

record_desc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Library history record. The *record_desc* argument is the address of a string descriptor pointing to the record to be added to the library update history.

Description

LBR$PUT_HISTORY writes a new update history record. If the library already contains the maximum number of history records (as specified at creation time by CRE$L_LUHMAX; see LBR$OPEN for details), the oldest history record is deleted before the new record is added.

Condition Values Returned

LBR$_NORMAL

Normal exit from the routine.

LBR$_INTRNLERR

Internal Librarian error.

LBR$_NOHISTORY

No update history. This is an informational code, not an error code.

LBR$_RECLNG

Record length greater than that specified by LBR$C_MAXRECSIZ. The record was not inserted or truncated.

LBR$PUT_MODULE

Puts a module and module's RFA from memory space into current library (Integrity servers only) — The LBR$PUT_MODULE routine puts an entire module, with the module's record file address (RFA), from memory space into the current library.

Format

```
LBR$PUT_MODULE library_index, mod_addr, mod_len, txtrfa
```

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL library routine. The *library_index* argument is the address of the longword that contains the index.

mod_addr

OpenVMS usage:	address
type:	quadword address
access:	read only
mechanism:	by 32-bit or 64-bit reference

The address from which the Library service obtains the 64-bit address of where the module is mapped in memory. The *mod_addr* argument is the 32- or 64-bit virtual address of a naturally aligned quadword containing the virtual address location of the module to write to the library.

mod_len

OpenVMS usage:	byte_count
type:	quadword (unsigned)
access:	read only
mechanism:	by 32- or 64-bit reference

The 64-bit virtual address of a naturally aligned quadword containing the length of the module that the Library service is to write into the library.

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

The module's record file address (RFA) of the library module header. The *txtrfa* argument is the address of the 2-longword array receiving the RFA of the newly created module header.

Description

The LBR$PUT_MODULE routine puts an entire module, with the module's record file address (RFA), from memory space into the current library. LBR$PUT_END is not required when you write an entire module to the current library.

LBR$PUT_RECORD

Write a Data Record — The LBR$PUT_RECORD routine writes a data record beginning at the next free location in the library.

Format

```
LBR$PUT_RECORD library_index ,bufdes ,txtrfa [, mod_size]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

bufdes

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Record to be written to the library. The *bufdes* argument is the address of a string descriptor pointing to the buffer containing the output record. On Integrity servers and Alpha libraries, the symbolic maximum record size is ELBR$_MAXRECSIZ.

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Record's file address (RFA) of the module header. The *txtrfa* argument is the address of a 2-longword array receiving the RFA of the newly created module header upon the first call to LBR $PUT_RECORD.

mod_size

OpenVMS usage:	byte_count
type:	longword (unsigned)
access:	read only
mechanism:	by value

The value from *mod_size* is read on the first call to this routine and ignored otherwise. The value specifies the size of the module to be entered so that contiguous space is allocated within the library for that module. This argument is ignored for non-ELF object libraries and for data-reduced ELF object libraries. The LBR$PUT_END routine is still required to terminate the byte stream and close off the module.

Description

If this is the first call to LBR$PUT_RECORD, this routine first writes a module header and returns its RFA to the 2-longword array pointed to by *txtrfa*. LBR$PUT_RECORD then writes the supplied data record to the library. On subsequent calls to LBR$PUT_RECORD, this routine writes the data record beginning at the next free location in the library (after the previous record). The last record written for the module should be followed by a call to LBR$PUT_END.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$REPLACE_KEY

Replace a Library Key — The LBR$REPLACE_KEY routine modifies or inserts a key into the library.

Format

```
LBR$REPLACE_KEY library_index ,key_name ,oldrfa ,newrfa  [, flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

key_name

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

For libraries with ASCII keys, the *key_name* argument is the address of a string descriptor for the key.

For libraries with binary keys, the *key_name* argument is the address of an unsigned longword value for the key.

oldrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Old record file address (RFA). The *oldrfa* argument is the address of a 2-longword array containing the original RFA (returned by LBR$LOOKUP_KEY) of the module header associated with the key you are replacing.

newrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)

access:	read only
mechanism:	by reference

New RFA. The *newrfa* argument is the address of a 2-longword array containing the RFA (returned by LBR$PUT_RECORD) of the module header associated with the new key.

flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

If present, the *flags* argument specifies the type of key being replaced. The flag bits are as follows:

Flag Bits	Description
LBR$SYM_WEAK = 0x1	UNIX-style weak symbol attribute
LBR$SYM_GROUP = 0x2	Group symbol attribute

If this argument is not present, NonGroup-Global is the assumed type. In this case, all type lists are searched and the entries removed. The new symbol is placed in the new NonGroup-Global definition with *newrfa* as the defining module.

If this parameter is present, it represents the flags set for the type of symbol being replaced. The replacement is done in place without losing its position in the type list. If the symbol does not exist when the call to this routine is made, the new definition is placed at the end of the type list for the specified type.

Because there are now different symbol definition types, VSI advises using the LBR$DELETE_KEY routine followed by the LBR$INSERT_KEY routine when the old key and new key differ in definition type.

Description

If LBR$REPLACE_KEY does not find the key in the current index, it calls the LBR$INSERT_KEY routine to insert the key. If LBR$REPLACE_KEY does find the key, it modifies the key entry in the index so that it points to the new module header.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_INVRFA

Specified RFA not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$RET_RMSSTV

Return OpenVMS RMS Status Value — The LBR$RET_RMSSTV routine returns the status value of the last OpenVMS RMS function performed by any LBR subroutine.

Format

```
LBR$RET_RMSSTV
```

Returns

OpenVMS 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 under Condition Values Returned.

Description

The LBR$RET_RMSSTV routine returns, as the status value, the status of the last RMS operation performed by the Librarian. Each programming language provides an appropriate mechanism for accessing RMS status values.

Condition Values Returned

This routine returns any condition values returned by RMS routines.

LBR$SEARCH

Search an Index — The LBR$SEARCH routine finds index keys that point to specified data.

Format

```
LBR$SEARCH library_index ,index_number ,rfa_to_find ,routine_name [, flags]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

index_number

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library index number. The *index_number* argument is the address of a longword containing the number of the index you want to search.

rfa_to_find

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Record file address (RFA) of the module whose keys you are searching for. The *rfa_to_find* argument is the address of a 2-longword array containing the RFA (returned earlier by LBR $LOOKUP_KEY or LBR$PUT_RECORD) of the module header.

routine_name

OpenVMS usage:	procedure
type:	procedure value
access:	read only
mechanism:	by reference

Name of a user-supplied routine to process the keys. The *routine_name* argument is the address of the procedure value of a user-supplied routine to call for each key entry containing the RFA (in other words, for each key that points to the same module header).

This user-supplied routine cannot contain any calls to LBR$DELETE_KEY or LBR$INSERT_KEY.

flags

OpenVMS usage:	mask_longword
type:	longword unsigned
access:	read only
mechanism:	by reference

If present and nonzero, the *flags* argument specifies the type, or all types, of the key provided. The flag bits are as follows:

Flag Bits	Description
LBR$M_SYM_WEAK = 0x1	UNIX-style weak symbol attribute
LBR$M_SYM_GROUP = 0x2	Group symbol attribute
LBR$M_SYM_ALL = 0x80000000	All symbols

The user routine is provided the symbol's type through an additional third parameter.

Description

The LBR$SEARCH routine searches the library index for symbols with the given RFA and calls the supplied routine with those symbols.

Use LBR$SEARCH to find index keys that point to the same module header. Generally, in index number 1 (the module name table), just one key points to any particular module; thus, you would probably use this routine only to search library indexes where more than one key points to a module. For example, you might call LBR$SEARCH to find all the symbols in the symbol index that are associated with an object module in an object library.

If LBR$SEARCH finds an index key associated with the specified RFA, it calls a user-supplied routine with two arguments:

- The key argument, which is the address of either of the following items:
  - A string descriptor for the key name (libraries with ASCII key names)
  - An unsigned longword for the key value (libraries with binary keys)
- The RFA argument, which is the address of a 2-longword array containing the RFA of the module header
- The key's type, whose flag bits are as follows:

Flag Bits	Description
LBR$M_SYM_WEAK = 1	UNIX-style weak symbol attribute
LBR$M_SYM_GROUP = 2	Group symbol attribute

The user routine must return a value to indicate success or failure. If the specified user routine returns a false value (low bit = 0), then the index search terminates.

Note that the key found by LBR$SEARCH is valid only during the call to the user-supplied routine. If you want to use the key later, you must copy it.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_ILLIDXNUM

Specified library index number not valid.

LBR$_KEYNOTFND

Library routine did not find any keys with the specified RFA.

LBR$_LIBNOTOPN

Specified library not open.

LBR$SET_INDEX

Set the Current Index Number — The LBR$SET_INDEX routine sets the index number to use when processing libraries that have more than one index.

Format

```
LBR$SET_INDEX library_index ,index_number
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

index_number

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Index number you want to establish as the current index number. The *index_number* argument is the address of the longword that contains the number of the index you want to establish as the current index. Refer to Section 13.1.2.3.

Description

When you call LBR$INI_CONTROL, the Librarian sets the current library index to 1 (the module name table, unless the library is a user-developed library). If you need to process another library index, you must use LBR$SET_INDEX to change the current library index.

Note that macro, help, and text libraries contain only one index; therefore, you do not need to call LBR $SET_INDEX. Object libraries contain two indexes. If you want to access the global symbol table, you must call the LBR$SET_INDEX routine to set the index number. User-developed libraries can contain more than one index; therefore, you may need to call LBR$SET_INDEX to set the index number.

Upon successful completion, LBR$SET_INDEX sets the current library index to the requested index number. LBR routines number indexes starting with 1.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_ILLIDXNUM

Library index number specified not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$SET_LOCATE

Set Record Access to Locate Mode — The LBR$SET_LOCATE routine sets the record access of LBR subroutines to locate mode.

Format

```
LBR$SET_LOCATE library_index
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

library_index

OpenVMS usage:	longword_unsigned

type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

Description

Librarian record access may be set to move mode (the default set by LBR$SET_MOVE) or locate mode. The setting affects the operation of the LBR$GET_RECORD routine.

If move mode is set (the default), LBR$GET_RECORD copies the requested record to the specified user buffer. If locate mode is set, the record is not copied. Instead, the **outbufdes** descriptor is set to reference the internal LBR subroutine buffer that contains the record.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$SET_MODULE

Read or Update a Module Header — The LBR$SET_MODULE routine reads, and optionally updates, the module header associated with a given record's file address (RFA).

Format

```
LBR$SET_MODULE library_index ,rfa [,bufdesc] [,buflen] [,updatedesc]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

library_index

OpenVMS usage:	longword_unsigned

type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

rfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Record's file address (RFA) associated with the module header. The *rfa* argument is the address of a 2-longword array containing the RFA returned by LBR$PUT_RECORD or LBR$LOOKUP_KEY.

bufdesc

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Buffer that receives the module header. The *bufdesc* argument is the address of a string descriptor pointing to the buffer that receives the module header. The buffer must be the size specified by the symbol MHD$B_USRDAT plus the value of the CRE$L_UHDMAX create option. The MHD$ and CRE$ symbols are defined in the modules $MHDDEF and $CREDEF, which are stored in SYS $LIBRARY:STARLET.MLB.

buflen

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	write only
mechanism:	by reference

Length of the module header. The *buflen* argument is the address of a longword receiving the length of the returned module header.

updatedesc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Additional information to be stored with the module header. The *updatedesc* argument is the address of a string descriptor pointing to additional data that the Librarian stores with the module header. If you include this argument, the Librarian updates the module header with the additional information.

Description

If you specify *bufdesc*, the LBR routine returns the module header into the buffer. If you specify *buflen*, the routine also returns the buffer's length. If you specify *updatedesc*, the routine updates the header information.

You define the maximum length of the update information (by specifying a value for CRE $L_UHDMAX) when you create the library. The Librarian zero-fills the information if it is less than the maximum length or truncates it if it exceeds the maximum length.

Condition Values Returned

LBR$_HDRTRUNC

Buffer supplied to hold the module header was too small.

LBR$_ILLCTL

Specified library control index not valid.

LBR$_ILLOP

Error. The *updatedesc* argument was supplied and the library was a Version 1.0 library or the library was opened only for read access.

LBR$_INVRFA

Specified RFA does not point to a valid module header.

LBR$_LIBNOTOPN

Specified library not open.

LBR$SET_MOVE

Set Record Access to Move Mode — The LBR$SET_MOVE routine sets the record access of LBR subroutines to move mode.

Format

```
LBR$SET_MOVE library_index
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL routine. The *library_index* argument is the address of the longword that contains the index.

Description

Librarian record access may be set to move mode (the default, set by LBR$SET_MOVE) or locate mode. The setting affects the operation of the LBR$GET_RECORD routine. If move mode is set, LBR $GET_RECORD copies the requested record to the specified user buffer. For details, see the description of LBR$GET_RECORD.

Condition Values Returned

LBR$_ILLCTL

Specified library control index not valid.

LBR$_LIBNOTOPN

Specified library not open.

LBR$UNMAP_MODULE

Unmaps a module from process P2 space (Integrity servers only) — The LBR$UNMAP_MODULE routine unmaps a module from process P2 space.

Format

```
LBR$PUT_MODULE library_index, txtrfa
```

Arguments

library_index

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Library control index returned by the LBR$INI_CONTROL library routine. The *library_index* argument is the address of the longword that contains the index.

txtrfa

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

The module's record file address (RFA) of the library module header. The *txtrfa* argument is the address of the 2-longword array that specifies the RFA of the module header.

Description

The LBR$UNMAP_MODULE routine unmaps the module, with the record file address in *txtrfa*, from process P2 space. This action releases the resources used to map the module.

Unlike other LBR services that use RMS services, LBR$UNMAP_MODULE also uses system services. Because of this, the secondary status for error returns is placed in LBR$GL_SUBSTS. Use this to find further status when an error is returned.

Chapter 14. Lightweight Directory Access Protocol (LDAP) Routines

14.1. Introduction

This chapter describes the C language application programming interface (API) to the Lightweight Directory Access Protocol (LDAP). This API supports Version 3 of the LDAP API (LDAPv3), and includes support for controls, information hiding, and thread safety. The LDAP API is available on OpenVMS Alpha only.

The C LDAP API is designed to be powerful, yet simple to use. It defines compatible synchronous and asynchronous interfaces to LDAP to support a wide variety of applications. This chapter gives a brief overview of the LDAP model, and describes how the application program uses the API to obtain LDAP information. The API calls are described in detail, followed by a section that provides some example code demonstrating the use of the API.

14.1.1. Overview of the LDAP Model

LDAP is the lightweight directory access protocol, which is based on a client-server model. In this model, a client makes a TCP connection to an LDAP server, over which it sends requests and receives responses.

The LDAP information model is based on the entry, which contains information about some object (for example, a person). Entries are composed of attributes, which have a type and one or more values. Each attribute has a syntax that determines what kinds of values are allowed in the attribute (for example, ASCII characters or a jpeg photograph) and how those values behave during directory operations (for example, whether case is significant during comparisons).

Entries may be organized in a tree structure, usually based on political, geographical, or organizational boundaries. Each entry is uniquely named relative to its sibling entries by its relative distinguished name (RDN) consisting of one or more distinguished attribute values from the entry. At most, one value from each attribute may be used in the RDN. For example, the entry for the person Babs Jensen might be named with the Barbara Jensen value from the commonName attribute.

A globally unique name for an entry, called a distinguished name or DN, is constructed by concatenating the sequence of RDNs from the entry up to the root of the tree. For example, if Babs worked for the University of Michigan, the DN of her U-M entry might be the following:

```
cn=Barbara Jensen, o=University of Michigan, c=US
```

Operations are provided to authenticate, search for and retrieve information, modify information, and add and delete entries from the tree. The next sections give an overview of how the API is used and provide detailed descriptions of the LDAP API calls that implement all of these functions.

14.1.2. Overview of LDAP API Use

An application generally uses the C LDAP API in four simple steps.

- Initialize an LDAP session with a primary LDAP server. The *ldap_init()* function returns a handle to the session, allowing multiple connections to be open at once.

- Authenticate to the LDAP server. The *ldap_bind()* function supports a variety of authentication methods.
- Perform some LDAP operations and obtain some results. The *ldap_search()* function returns results that can be parsed by *ldap_parse_result()*, *ldap_first_entry()*, and *ldap_next_entry()*.
- Close the session. The *ldap_unbind()* function closes the connection.

Operations can be performed either synchronously or asynchronously. The names of the synchronous functions end in _s. For example, a synchronous search can be completed by calling *ldap_search_s()*. An asynchronous search can be initiated by calling *ldap_search()*. All synchronous functions return an indication of the outcome of the operation (for example, the constant LDAP_SUCCESS or some other error code). The asynchronous functions make available to the caller the message id of the operation initiated. This id can be used in subsequent calls to `ldap_result()` to obtain the result(s) of the operation. An asynchronous operation can be abandoned by calling *ldap_abandon()* or *ldap_abandon_ext()*.

Results and errors are returned in an opaque structure called LDAPMessage. Functions are provided to parse this structure, step through entries and attributes returned. Functions are also provided to interpret errors. Later sections of this chapter describe these functions in more detail.

LDAPv3 servers may return referrals to other servers. By default, implementations of this API will attempt to follow referrals automatically for the application. This behavior can be disabled globally (using the *ldap_set_option()* call) or on a per-request basis through the use of a server control.

As in the LDAPv3 protocol, all DNs and string values that are passed into or produced by the C LDAP API are represented as UTF-8 characters. Conversion functions are described in Section 14.20.

For compatibility with existing applications, implementations of this API will, by default, use Version 2 of the LDAP protocol. Applications that intend to take advantage of LDAPv3 features will need to use the *ldap_set_option()* call with a LDAP_OPT_PROTOCOL_VERSION switch set to Version 3.

The file LDAP_EXAMPLE.C in SYS$EXAMPLES contains an example program that demonstrates how to use the LDAP API on OpenVMS.

14.1.3. LDAP API Use on OpenVMS Systems

This release of the LDAP API provides support for client applications written in C or C++.

In order to use the LDAP API, a program must use an include statement of the form:

```
#include <ldap.h>
```

The LDAP.H header file includes prototypes and data structures for all of the functions that are available in the LDAP API.

The shareable image LDAP$SHR.EXE includes run-time support for LDAP applications. This shareable image resides in SYS$LIBRARY and should be included in the library IMAGELIB.OLB, which means that no special action is necessary to link or run your programs. For example:

```
$ type myprog.c

/* A not very useful program */
#include <stdio.h>
#include <ldap.h>
```

```
void main(int argc, char *argv[])
{
  LDAP *ld;
  if (argc != 2) {
    printf("usage: %s <hostname>\n",argv[0]);
    return;
  }
  ld = ldap_init(argv[1],LDAP_PORT);
  if (ld != NULL) {
    printf("ldap_init returned 0x%p\n",ld);
  } else {
    printf("ldap_init failed\n");
  }
}

$ cc myprog
$ link myprog
$ myprog :== $mydisk:[mydir]myprog.exe
$ myprog fred
ldap_init returned 0xA6748
$
```

14.1.4. 64-bit Addressing Support

This section describes the LDAP 64-bit addressing support.

14.1.4.1. Background

OpenVMS Alpha provides support for 64-bit virtual memory addressing. Applications that are built using a suitable compiler may take advantage of the 64-bit virtual address space to map and access large amounts of data.

The OpenVMS LDAP API supports both 32- and 64-bit client applications. In order to allow this, separate entry points are provided in the library for those functions that are sensitive to pointer size.

When a user module is compiled, the header file LDAP.H determines the pointer size in effect and uses the C preprocessor to map the function names into the appropriate library entry point. This mapping is transparent to the user application and is effected by setting the /POINTER_SIZE qualifier at compilation time.

For LDAP API users, switching between different pointer sizes should need only a recompilation - no code changes are necessary.

This means that programs using the specification for the C LDAP API, as described in the Internet Engineering Task Force (IETF) documentation, can be built on OpenVMS with either 32-bit or 64-bit pointer size, without having to change the source code.

14.1.4.2. Implementation

The OpenVMS LDAP library uses 64-bit pointers internally and is capable of dealing with data structures allocated by the caller from 64-bit address space.

Applications that use 32-bit pointers will use the 32-bit function entry points in the library. This means they can pass arguments that are based on 32-bit pointers and can assume that any pointers returned by the library will be 32-bit safe.

While the mapping performed by LDAP.H is designed to be transparent, there may be occasions where it is useful (for example in debugging) to understand the consequences of having both 32- and 64-bit support in the same library.

14.1.4.2.1. Library Symbol Names

The symbols exported by the LDAP$SHR OpenVMS run-time library differ from those specified in the IETF C LDAP API specification.

The header file LDAP.H maps user references to LDAP API function names to the appropriate LDAP $SHR symbol name. Therefore, any application wishing to use the OpenVMS LDAP API must include the version of LDAP.H that ships with OpenVMS.

All of the functions in the OpenVMS LDAP library are prefixed with the facility code "LDAP$".

For those functions where the caller's pointer size is significant, the name of the 64-bit entry point will have a "_64" suffix, while the name of the 32-bit jacket will have a "_32" suffix. Functions that are not sensitive to pointer size have no special suffix.

For example, the function *ldap_modify()* is sensitive to the caller's pointer size (because one of its arguments is an array of pointers). Therefore, the library exports symbols for LDAP $LDAP_MODIFY_64 and LDAP$LDAP_MODIFY_32. For the function *ldap_simple_bind()*, which is not sensitive to the caller's pointer size, a single entry point, LDAP$LDAP_SIMPLE_BIND, exists in the library.

Because OpenVMS imposes a 31-character limit on the length of symbol names, certain functions in the library have names which are abbreviated versions of the public API name. For example, in the case of the function *ldap_parse_sasl_bind_result()*, the library provides two entry points, namely LDAP$LDAP_PRS_SASL_BIND_RES_32 and LDAP$LDAP_PRS_SASL_BIND_RES_64.

14.1.4.2.2. LDAP Data Structures

The LDAP API defines various data structures which are used to pass information to and from a client application. Some of these structures are opaque; that is, their internal layout is not visible to a client application. In such cases, the API may return a pointer to such a structure, but the only use of such a pointer to a client application is as a parameter to subsequent library calls.

Some structures are public. Their contents are defined by the API, and client applications may allocate and manipulate such structures or use them as parameters to LDAP functions.

All data structures used by the API are defined with "natural" alignment; that is, each member of a data structure will be aligned on an address boundary appropriate to its type.

Opaque Data Structures

The following data structures are opaque. Applications should not make any assumptions about the contents or size of such data structures.

```
typedef struct ldap
        LDAP;

typedef struct ldapmsg
        LDAPMessage;

typedef struct berelement
        BerElement;
```

Public Data Structures

The following data structures are described in the IETF documents relating to the LDAP API, and definitions are provided for them in LDAP.H. Applications may allocate and manipulate such structures, as well as use them in calls to the LDAP API.

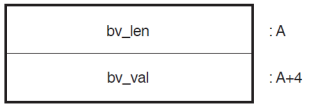
```
typedef struct berval { .. }
        BerValue;

typedef struct ldapapiinfo { .. }
        LDAPAPIInfo;

typedef struct ldap_apifeature_info { .. }
        LDAPAPIFeatureInfo;

typedef struct ldapcontrol { .. }
        LDAPControl;

typedef struct ldapmod { .. }
        LDAPMod;
```

Note that the pointer size in effect at compilation time determines the layout of data structures, which themselves contain pointer fields. Since all of the public data structures listed here contain one or more pointers, their size and layout will differ depending on the pointer size.

For example, in the case of the structure berval, the API provides the following definition:

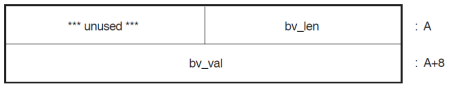
```
struct berval {
     ber_len_t      bv_len;
     char           *bv_val;
} BerValue;
```

(where *ber_len_t* is equivalent on OpenVMS to an unsigned 32-bit integer). For a module compiled using 32-bit pointer size, the layout of a BerValue at address A would look like this:



VM-0729A-AI

In the case of a 64-bit compilation, the layout would be:



VM-0730A-AI

The following code would therefore work correctly regardless of pointer size:

```
#include <ldap.h>
.
.
.
  char       *buff;
  BerValue   val;
.
.
.
```

```
      buff = (char *)malloc(255);
    .
    .
    .
      val.bv_len = 255;
      val.bv_val = buff;
    .
    .
    .
```

14.1.4.3. Mixing Pointer Sizes

Two modules that include LDAP.H can be compiled with different pointer sizes and linked together. While each module may use the LDAP API on its own, it may not be possible for both modules to share LDAP-related data.

None of the public LDAP data structures is directly compatible between 32- and 64-bit modules. For example, a BerValue that has been allocated by a 32-bit module does not have the same layout as a BerValue which a 64-bit module expects to see, and consequently cannot be exchanged between two such modules without some sort of data conversion taking place.

Opaque data structures (such as LDAP *) have only a single structure definition inside the library, and so pointers to such structures may be exchanged between 32- and 64-bit callers. Note that these structures are allocated only by the library itself, and, in the case of a 64-bit caller, these structures may be allocated in 64-bit space. So while the LDAP handle returned to a 32-bit caller of *ldap_init()* could safely be used by a 64-bit module, the reverse may not be true.

14.1.5. Multithreading Support

The OpenVMS LDAP API may be used by a multi-threaded application. Two of the functions in the library, *ldap_perror()* and *ldap_result2error()*, are not thread-safe.

14.2. Common Data Structures and Memory Handling

The following are definitions of some data structures that are common to several LDAP API functions.

```
typedef struct ldap LDAP;

typedef struct berelement BerElement;

typedef struct ldapmsg LDAPMessage;

typedef struct berval {
    ber_len_t       bv_len;
    char            *bv_val;
} BerValue;


struct timeval;
```

The LDAP structure is an opaque data type that represents an LDAP session. Typically, this corresponds to a connection to a single server, but it may encompass several server connections in LDAPv3 referrals.

The LDAPMessage structure is an opaque data type that is used to return entry, reference, result, and error information. An LDAPMessage structure may represent the beginning of a list or a chain of messages that contain a series of entries, references, and result messages that are returned by LDAP operations, such as search. LDAP API functions, such as ldap_parse_result() , that operate on message chains which may contain more than one result message, always operate on the first result message in the chain. See Section 14.17 for more information.

The BerElement structure is an opaque data type that is used to hold data and state information about encoded data.

The berval structure is used to represent arbitrary binary data, and its fields have the following meanings:

bv_len	Length of data in bytes.
bv_val	A pointer to the data itself.

The timeval structure is used to represent an interval of time, and its fields have the following meanings:

tv_sec	Seconds component of time interval.
tv_usec	Microseconds component of time interval.

All memory that is allocated by a function in this C LDAP API and returned to the caller should be disposed of by calling the appropriate free function provided by this API. The correct free function to call is documented in each section of this chapter where a function that allocates memory is described.

Memory that is allocated outside of the C LDAP API must not be disposed of using a function provided by this API.

The following is a complete list of free functions that are used to dispose of allocated memory:

```
ber_bvecfree()

ber_bvfree()

ber_free()

ldap_control_free()

ldap_controls_free()

ldap_memfree()

ldap_msgfree()

ldap_value_free()

ldap_value_free_len()
```

14.3. LDAP Error Codes

Many of the LDAP API functions return LDAP error codes, some of which indicate local errors and some of which may be returned by servers. All of the LDAP error codes returned will be positive integers; those between 0x00 and 0x50 are returned from the LDAP server, those above 0x50 are

generated by the API itself. Supported error codes are as follows (hexadecimal values are given in parentheses after the constant):

```
LDAP_SUCCESS (0x00)

LDAP_OPERATIONS_ERROR (0x01)

LDAP_PROTOCOL_ERROR (0x02)

LDAP_TIMELIMIT_EXCEEDED (0x03)

LDAP_SIZELIMIT_EXCEEDED (0x04)

LDAP_COMPARE_FALSE (0x05)

LDAP_COMPARE_TRUE (0x06)

LDAP_STRONG_AUTH_NOT_SUPPORTED (0x07)

LDAP_STRONG_AUTH_REQUIRED (0x08)

LDAP_REFERRAL (0x0a)                          -- new in LDAPv3

LDAP_ADMINLIMIT_EXCEEDED (0x0b)               -- new in LDAPv3

LDAP_UNAVAILABLE_CRITICAL_EXTENSION (0x0c)  -- new in LDAPv3

LDAP_CONFIDENTIALITY_REQUIRED (0x0d)        -- new in LDAPv3

LDAP_SASL_BIND_IN_PROGRESS (0x0e)           -- new in LDAPv3

LDAP_NO_SUCH_ATTRIBUTE (0x10)

LDAP_UNDEFINED_TYPE (0x11)

LDAP_INAPPROPRIATE_MATCHING (0x12)

LDAP_CONSTRAINT_VIOLATION (0x13)

LDAP_TYPE_OR_VALUE_EXISTS (0x14)

LDAP_INVALID_SYNTAX (0x15)

LDAP_NO_SUCH_OBJECT (0x20)

LDAP_ALIAS_PROBLEM (0x21)

LDAP_INVALID_DN_SYNTAX (0x22)

LDAP_IS_LEAF (0x23)                     -- not used in LDAPv3

LDAP_ALIAS_DEREF_PROBLEM (0x24)

LDAP_INAPPROPRIATE_AUTH (0x30)

LDAP_INVALID_CREDENTIALS (0x31)

LDAP_INSUFFICIENT_ACCESS (0x32)
```

```
LDAP_BUSY (0x33)

LDAP_UNAVAILABLE (0x34)

LDAP_UNWILLING_TO_PERFORM (0x35)

LDAP_LOOP_DETECT (0x36)

LDAP_NAMING_VIOLATION (0x40)

LDAP_OBJECT_CLASS_VIOLATION (0x41)

LDAP_NOT_ALLOWED_ON_NONLEAF (0x42)

LDAP_NOT_ALLOWED_ON_RDN (0x43)

LDAP_ALREADY_EXISTS (0x44)

LDAP_NO_OBJECT_CLASS_MODS (0x45)

LDAP_RESULTS_TOO_LARGE (0x46)            -- reserved for CLDA

LDAP_AFFECTS_MULTIPLE_DSAS (0x47)        -- new in LDAPv3

LDAP_OTHER (0x50)

LDAP_SERVER_DOWN (0x51)

LDAP_LOCAL_ERROR (0x52)

LDAP_ENCODING_ERROR (0x53)

LDAP_DECODING_ERROR (0x54)

LDAP_TIMEOUT (0x55)

LDAP_AUTH_UNKNOWN (0x56)

LDAP_FILTER_ERROR (0x57)

LDAP_USER_CANCELLED (0x58)

LDAP_PARAM_ERROR (0x59)

LDAP_NO_MEMORY (0x5a)

LDAP_CONNECT_ERROR (0x5b)

LDAP_NOT_SUPPORTED (0x5c)

LDAP_CONTROL_NOT_FOUND (0x5d)

LDAP_NO_RESULTS_RETURNED (0x5e)

LDAP_MORE_RESULTS_TO_RETURN (0x5f)

LDAP_CLIENT_LOOP (0x60)
```

```
LDAP_REFERRAL_LIMIT_EXCEEDED (0x61)
```

14.4. Initializing an LDAP Session

The *ldap_init()* function initializes a session with an LDAP server. The server is not actually contacted until an operation is performed that requires it, allowing various options to be set after initialization.

```
LDAP *ldap_init(
      const char    *hostname,
      int           portno);
```

Use of the following function is deprecated.

```
LDAP *ldap_open(
      const char    *hostname,
      int           portno);
```

Unlike *ldap_init()*, the `ldap_open()` function attempts to make a server connection before returning to the caller. A more complete description can be found in RFC 1823.

Parameters are as follows:

hostname	Contains a space-separated list of hostnames or dotted strings representing the IP address of hosts running an LDAP server to connect to. Each hostname in the list can include an optional port number which is separated from the host itself with a colon (:) character. The hosts are tried in the order listed, stopping with the first one to which a successful connection is made. Note that only `ldap_open()` attempts to make the connection before returning to the caller. *ldap_init()* does not connect to the LDAP server.
portno	Contains the TCP port number to connect to. The default LDAP port of 389 can be obtained by supplying the constant LDAP_PORT. If a host includes a port number, then this parameter is ignored.

The *ldap_init()* and `ldap_open()` functions both return a session handle, a pointer to an opaque structure that should be passed to subsequent calls pertaining to the session. These functions return NULL if the session cannot be initialized, in which case the operating system error reporting mechanism can be checked to see why the call failed.

Note that if you connect to an LDAP Version 2 server, one of the ldap_bind() calls must be completed before other operations can be performed on the session. LDAPv3 does not require that a bind operation be completed before other operations can be performed.

The calling program can set various attributes of the session by calling the functions described in the next section.

14.5. LDAP Session Handle Options

The LDAP session handle returned by *ldap_init()* is a pointer to an opaque data type representing an LDAP session. Formerly, this data type was a structure exposed to the caller, and various fields in the structure could be set to control aspects of the session, such as size and time limits on searches.

To insulate callers from inevitable changes to this structure, these aspects of the session are now accessed through a pair of accessor functions.

The *ldap_get_option()* function is used to access the current value of various session-wide parameters. The *ldap_set_option()* function is used to set the value of these parameters. Note that some options are READ-ONLY and cannot be set; it is an error to call *ldap_set_option()* and attempt to set a READ-ONLY option.

```
int ldap_get_option(
        LDAP            *ld,
        int             option,
        void            *outvalue
);

int ldap_set_option(
        LDAP            *ld,
        int             option,
        const void      *invalue
);
```

Parameters are as follows:

ld	The session handle. If this is NULL, a set of global defaults is accessed. New LDAP session handles created with *ldap_init()* or `ldap_open()` inherit their characteristics from these global defaults.	
option	The name of the option being accessed or set. This parameter should be one of the following constants, which have the indicated meanings. After the constant, the actual hexadecimal value of the constant is listed in parentheses.	
	LDAP_OPT_DESC (0x01)	Type for invalue parameter: not applicable (option is read-only). Type for outvalue parameter: int * Description: The underlying socket descriptor corresponding to the primary LDAP connection. This option is read-only and cannot be set.
	LDAP_OPT_DEREF (0x02)	Type for invalue parameter: int *Type for outvalue parameter: int * Description: Determines how aliases are handled during search. It can have one of the following values: LDAP_DEREF_NEVER (0x00), LDAP_DEREF_SEARCHING (0x01), LDAP_DEREF_FINDING (0x02), or LDAP_DEREF_ALWAYS (0x03). The LDAP_DEREF_SEARCHING value means aliases should be dereferenced during the search but not when locating the base object of the search. The LDAP_DEREF_FINDING value means aliases should be dereferenced when locating the base object but not during the search.
	LDAP_OPT_SIZELIMIT (0x03)	Type for invalue parameter: int *Type for outvalue parameter: int *

		Description: A limit on the number of entries to return from a search. A value of LDAP_NO_LIMIT (0) means no limit.
	LDAP_OPT_TIMELIMIT (0x04)	Type for invalue parameter: int *Type for outvalue parameter: int * Description: A limit on the number of seconds to spend on a search. A value of LDAP_NO_LIMIT (0) means no limit.
	LDAP_OPT_REFERRALS (0x08)	Type for invalue parameter: int (LDAP_OPT_ON or LDAP_OPT_OFF)Type for outvalue parameter: int * Description: Determines whether the LDAP library automatically follows referrals returned by LDAP servers. It can be set to one of the constants LDAP_OPT_ON (1) or LDAP_OPT_OFF (0).
	LDAP_OPT_RESTART (0x09)	Type for invalue parameter: int (LDAP_OPT_ON or LDAP_OPT_OFF)Type for outvalue parameter: int * Description: Determines whether LDAP I/O operations should automatically be restarted if they abort prematurely. It should be set to one of the constants LDAP_OPT_ON or LDAP_OPT_OFF. This option is useful if an LDAP I/O operation is interrupted prematurely, (for example, by a timer going off) or other interrupt.
	LDAP_OPT_PROTOCOL_VERSION (0x11)	Type for invalue parameter: int *Type for outvalue parameter: int * Description: This option indicates the version of the LDAP protocol used when communicating with the primary LDAP server. It must be one of the constants LDAP_VERSION2 (2) or LDAP_VERSION3 (3). If no version is set, the default is LDAP_VERSION2 (2).
	LDAP_OPT_SERVER_CONTROLS (0x12)	Type for invalue parameter: LDAPControl **Type for outvalue parameter: LDAPControl ***

		Description: A default list of LDAP server controls to be sent with each request. See Section 14.6 for more information.
	LDAP_OPT_CLIENT_CONTROLS (0x13)	Type for invalue parameter: LDAPControl **Type for outvalue parameter: LDAPControl *** Description: A default list of client controls that affect the LDAP session. See Section 14.6 for more information.
	LDAP_OPT_HOST_NAME (0x30)	Type for invalue parameter: char *Type for outvalue parameter: char ** Description: The host name (or list of host) for the primary LDAP server.
	LDAP_OPT_ERROR_NUMBER (0x31)	Type for invalue parameter: int *Type for outvalue parameter: int * Description: The code of the most recent LDAP error that occurred for this session.
	LDAP_OPT_ERROR_STRING (0x32)	Type for invalue parameter: char *Type for outvalue parameter: char ** Description: The message returned with the most recent LDAP error that occurred for this session.
outvalue	The address of a place to put the value of the option. The actual type of this parameter depends on the setting of the option parameter. For outvalues of type char ** and LDAPControl **, a pointer to data that is associated with the LDAP session ld is returned; callers should dispose of the memory by calling *ldap_memfree()* or *ldap_controls_free()*.	
invalue	A pointer to the value the option is to be given. The actual type of this parameter depends on the setting of the option parameter. The constants LDAP_OPT_ON and LDAP_OPT_OFF can be given for options that have on or off *settings*. Both *ldap_get_option()* and *ldap_set_option()* return 0 if successful and -1 if an error occurs.	

14.6. Working with Controls

LDAPv3 operations can be extended through the use of controls. Controls may be sent to a server or returned to the client with any LDAP message. These controls are referred to as server controls.

The LDAP API also supports a client-side extension mechanism through the use of client controls. These controls affect the behavior of the LDAP API only and are never sent to a server. A common data structure is used to represent both types of controls:

```
typedef struct ldapcontrol {
        char                            *ldctl_oid;
        struct berval                   ldctl_value;
        char                            ldctl_iscritical;
```

```
        } LDAPControl, *PLDAPControl;
```

The fields in the ldapcontrol structure have the following meanings:

ldctl_oid	The control type, represented as a string.
ldctl_value	The data associated with the control (if any). To specify a zero-length value, set ldctl_value.bv_len to zero and ldctl_value.bv_val to a zero-length string. To indicate that no data is associated with the control, set ldctl_value.bv_val to NULL.
ldctl_iscritical	Indicates whether the control is critical or not. If this field is non-zero, the operation will only be carried out if the control is recognized by the server and/or client.

Some LDAP API calls allocate an ldapcontrol structure or a NULL-terminated array of ldapcontrol structures. The following functions can be used to dispose of a single control or an array of controls:

```
    void ldap_control_free( LDAPControl *ctrl );

    void ldap_controls_free( LDAPControl **ctrls );
```

A set of controls that affect the entire session can be set using the `ldap_set_option()` function. A list of controls can also be passed directly to some LDAP API calls, such as `ldap_search_ext()`, in which case any controls set for the session through the use of `ldap_set_option()` are ignored. Control lists are represented as a NULL-terminated array of pointers to ldapcontrol structures.

Server controls are defined by LDAPv3 protocol extension documents; for example, a control has been proposed to support paging of search results. No client controls are currently implemented in this version of the API.

14.7. Authenticating to the Directory

The following functions are used to authenticate an LDAP client to an LDAP directory server.

The `ldap_sasl_bind()` and `ldap_sasl_bind_s()` functions can be used to do general and extensible authentication over LDAP through the use of the Simple Authentication Security Layer. The functions both take the DN to bind as, the method to use, as a dotted-string representation of an OID identifying the method, and a struct berval holding the credentials. The special constant value LDAP_SASL_SIMPLE (NULL) can be passed to request simple authentication, or the simplified functions `ldap_simple_bind()` or `ldap_simple_bind_s()` can be used.

```
int ldap_sasl_bind(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *mechanism,
        const struct berval             *cred,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls,
        int                             *msgidp
);

int ldap_sasl_bind_s(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *mechanism,
        const struct berval             *cred,
        LDAPControl                     **serverctrls,
```

```
        LDAPControl                                **clientctrls,
        struct berval                              **servercredp
);

int ldap_simple_bind(
        LDAP                                       *ld,
        const char                                 *dn,
        const char                                 *passwd
);

 int ldap_simple_bind_s(
        LDAP                                       *ld,
        const char                                 *dn,
        const char                                 *passwd
      );
```

The use of the following functions is deprecated:

```
int ldap_bind( LDAP *ld, char *dn, char *cred, int method );

int ldap_bind_s( LDAP *ld, char *dn, char *cred, int method );
```

Parameters are as follows:

ld	The session handle.
dn	The name of the entry to bind as.
mechanism	Either LDAP_SASL_SIMPLE (NULL) to get simple authentication, or a text string identifying the SASL method.
cred	The credentials with which to authenticate. Arbitrary credentials can be passed using this parameter. The format and content of the credentials depends on the setting of the mechanism parameter.
passwd	For `ldap_simple_bind()`, the password to compare to the entry's user Password attribute.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_sasl_bind()` call succeeds.
servercredp	This result parameter will be filled in with the credentials passed back by the server for mutual authentication, if given. An allocated berval structure is returned that should be disposed of by calling `ber_bvfree()`. NULL may be passed to ignore this field.

Additional parameters for the deprecated functions are not described. See the RFC 1823 documentation for more information.

The `ldap_sasl_bind()` function initiates an asynchronous bind operation and returns the constant LDAP_SUCCESS if the request was successfully sent or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_sasl_bind()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the bind.

The `ldap_simple_bind()` function initiates a simple asynchronous bind operation and returns the message id of the operation initiated. A subsequent call to `ldap_result()` can be used to obtain the

result of the bind. In case of error, `ldap_simple_bind()` will return -1, setting the session error parameters in the LDAP structure appropriately.

The synchronous `ldap_sasl_bind_s()` and `ldap_simple_bind_s()` functions both return the result of the operation, either the constant LDAP_SUCCESS if the operation was successful, or another LDAP error code if it was not. See Section 14.18 for more information about possible errors and how to interpret them.

Note that if an LDAP Version 2 server is contacted, no other operations over the connection should be attempted before a bind call has successfully completed.

Subsequent bind calls can be used to reauthenticate over the same connection, and multistep SASL sequences can be accomplished through a sequence of calls to `ldap_sasl_bind()` or `ldap_sasl_bind_s()`.

14.8. Closing the Session

The following functions are used to unbind from the directory, close the connection, and dispose of the session handle.

```
int ldap_unbind( LDAP *ld );
int ldap_unbind_s( LDAP *ld );
```

Parameter is as follows:

ld	The session handle.

The `ldap_unbind()` and `ldap_unbind_s()` functions both work synchronously, unbinding from the directory, closing the connection, and freeing up the ld structure before returning. There is no server response to an unbind operation. The `ldap_unbind()` function returns LDAP_SUCCESS (or another LDAP error code if the request cannot be sent to the LDAP server). After a call to `ldap_unbind()` or `ldap_unbind_s()`, the session handle ld is invalid and it is illegal to make any further LDAP API calls using ld.

14.9. Searching

The following functions are used to search the LDAP directory, returning a requested set of attributes for each entry matched. There are five variations.

```
int ldap_search_ext(
        LDAP                    *ld,
        const char              *base,
        int                     scope,
        const char              *filter,
        char                    **attrs,
        int                     attrsonly,
        LDAPControl             **serverctrls,
        LDAPControl             **clientctrls,
        struct timeval          *timeout,
        int                     sizelimit,
        int                     *msgidp
);

int ldap_search_ext_s(
        LDAP                    *ld,
        const char              *base,
```

```
        int                     scope,
        const char              *filter,
        char                    **attrs,
        int                     attrsonly,
        LDAPControl             **serverctrls,
        LDAPControl             **clientctrls,
        struct timeval          *timeout,
        int                     sizelimit,
        LDAPMessage             **res
);

int ldap_search(
        LDAP                    *ld,
        const char              *base,
        int                     scope,
        const char              *filter,
        char                    **attrs,
        int                     attrsonly
);

int ldap_search_s(
        LDAP                    *ld,
        const char              *base,
        int                     scope,
        const char              *filter,
        char                    **attrs,
        int                     attrsonly,
        LDAPMessage             **res
);

int ldap_search_st(
        LDAP                    *ld,
        char                    *base,
        int                     scope,
        char                    *filter,
        char                    **attrs,
        int                     attrsonly,
        struct timeval          *timeout,
        LDAPMessage             **res
   );
```

Parameters are as follows:

ld	The session handle.
base	The dn of the entry at which to start the search.
scope	One of LDAP_SCOPE_BASE (0x00), LDAP_SCOPE_ONELEVEL (0x01), or LDAP_SCOPE_SUBTREE (0x02), indicating the scope of the search.
filter	A character string representing the search filter. The value NULL can be passed to indicate that the filter (objectclass=*) that matches all entries should be used.
attrs	A NULL-terminated array of strings indicating which attributes to return for each matching entry. Passing NULL for this parameter causes all available user attributes to be retrieved. The special constant string LDAP_NO_ATTRS (1.1) can be used as the

	only element in the array to indicate that no attribute types should be returned by the server. The special constant string LDAP_ALL_USER_ATTRS (*), can be used in the attrs array along with the names of some operational attributes to indicate that all user attributes plus the listed operational attributes should be returned.
attrsonly	A boolean value that should be either zero if both attribute types and values are to be returned or non-zero if only types are wanted.
timeout	For the `ldap_search_st()` function, this specifies the local search timeout value (if it is NULL, the timeout is infinite). For the `ldap_search_ext()` and `ldap_search_ext_s()` functions, this specifies both the local search timeout value and the operation time limit that is sent to the server within the search request. For the `ldap_search_ext()` and `ldap_search_ext_s()` functions, passing a NULL value for timeout causes the global default timeout stored in the LDAP session handle to be used (set using `ldap_set_option()` with the LDAP_OPT_TIMELIMIT parameter).
sizelimit	For the `ldap_search_ext()` and `ldap_search_ext_s()` calls, this is a limit on the number of entries to return from the search. A value of LDAP_NO_LIMIT (0) means no limit.
res	For the synchronous calls, this is a result parameter which will contain the results of the search upon completion of the call.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_search_ext()` call succeeds.

There are three options in the session handle ld that potentially affect how the search is performed. They are as follows:

LDAP_OPT_SIZELIMIT	A limit on the number of entries to return from the search. A value of LDAP_NO_LIMIT (0) means no limit. Note that the value from the session handle is ignored when using the `ldap_search_ext()` or `ldap_search_ext_s()` functions.
LDAP_OPT_TIMELIMIT	A limit on the number of seconds to spend on the search. A value of LDAP_NO_LIMIT (0) means no limit. Note that the value from the session handle is ignored when using the `ldap_search_ext()` or `ldap_search_ext_s()` functions.
LDAP_OPT_DEREF	One of LDAP_DEREF_NEVER(0x00), LDAP_DEREF_SEARCHING(0x01), LDAP_DEREF_FINDING (0x02), or LDAP_DEREF_ALWAYS (0x03), specifying how aliases should be handled during the search. The LDAP_DEREF_SEARCHING value means aliases should be dereferenced during the search but not

	when locating the base object of the search. The LDAP_DEREF_FINDING value means aliases should be dereferenced when locating the base object but not during the search.

The `ldap_search_ext()` function initiates an asynchronous search operation and returns either the constant LDAP_SUCCESS if the request was successfully sent or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_search_ext()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the results from the search. These results can be parsed using the result parsing functions described in Section 14.18.

Similar to `ldap_search_ext()`, the `ldap_search()` function initiates an asynchronous search operation and returns the message id of the operation initiated. As for `ldap_search_ext()`, a subsequent call to `ldap_result()` can be used to obtain the result of the search. In case of error, `ldap_search()` will return -1, setting the session error parameters in the LDAP structure appropriately.

The synchronous `ldap_search_ext_s()`, `ldap_search_s()`, and `ldap_search_st()` functions all return the result of the operation, either the constant LDAP_SUCCESS if the operation was successful or another LDAP error code if it was not. See Section 14.18 for more information about possible errors and how to interpret them. Entries returned from the search (if any) are contained in the res parameter. This parameter is opaque to the caller. Entries, attributes, and values should be extracted by calling the parsing functions. The results contained in res should be freed when no longer in use by calling `ldap_msgfree()`.

The `ldap_search_ext()` and `ldap_search_ext_s()` functions support LDAPv3 server controls, client controls, and allow varying size and time limits to be easily specified for each search operation. The `ldap_search_st()` function is identical to `ldap_search_s()` except that it takes an additional parameter specifying a local timeout for the search. The local search timeout is used to limit the amount of time the API implementation will wait for a search to complete. After the local search timeout the search operation will return LDAP_TIMEOUT if the search result has not been removed.

14.9.1. Reading and Listing the Children of an Entry

LDAP does not support a read operation directly. Instead, this operation is emulated by a search with base set to the DN of the entry to read, scope set to LDAP_SCOPE_BASE, and filter set to "(objectclass=*)" or NULL. The attrs parameter contains the list of attributes to return.

LDAP does not support a list operation directly. Instead, this operation is emulated by a search with base set to the DN of the entry to list, scope set to LDAP_SCOPE_ONELEVEL, and filter set to "(objectclass=*)" or NULL. The attrs parameter contains the list of attributes to return for each child entry.

14.10. Comparing a Value Against an Entry

The following functions are used to compare a given attribute value assertion against an LDAP entry. There are four variations.

```
int ldap_compare_ext(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *attr,
```

```
        const struct berval             *bvalue
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls,
        int                             *msgidp
);

int ldap_compare_ext_s(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *attr,
        const struct berval             *bvalue,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls
);



int ldap_compare(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *attr,
        const char                      *value
);

int ldap_compare_s(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *attr,
        const char                      *value
);
```

Parameters are as follows:

ld	The session handle.
dn	The name of the entry to compare against.
attr	The attribute to compare against.
bvalue	The attribute value to compare against those found in the given entry. This parameter is used in the extended functions and is a pointer to a struct berval so it is possible to compare binary values.
value	A string attribute value to compare against, used by the `ldap_compare()` and `ldap_compare_s()` functions. Use `ldap_compare_ext()` or `ldap_compare_ext_s()` if you need to compare binary values.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_compare_ext()` call succeeds.

The `ldap_compare_ext()` function initiates an asynchronous compare operation and returns either the constant LDAP_SUCCESS if the request was successfully sent, or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_compare_ext()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the compare.

Similar to `ldap_compare_ext()`, the `ldap_compare()` function initiates an asynchronous compare operation and returns the message id of the operation initiated. As for `ldap_compare_ext()`, a subsequent call to `ldap_result()` can be used to obtain the result of the compare. In case of error, `ldap_compare()` will return -1, setting the session error parameters in the LDAP structure appropriately.

The synchronous `ldap_compare_ext_s()` and `ldap_compare_s()` functions both return the result of the operation, either the constants LDAP_COMPARE_TRUE or LDAP_COMPARE_FALSE if the operation was successful, or another LDAP error code if it was not. See Section 14.18 for more information about possible errors and how to interpret them.

The `ldap_compare_ext()` and `ldap_compare_ext_s()` functions support LDAPv3 server controls and client controls.

14.11. Modifying an Entry

The following functions are used to modify an existing LDAP entry. There are four variations.

```
typedef struct ldapmod {
        int                                  mod_op;
        char                                 *mod_type;
        union {
                char                         **modv_strvals;
                struct berval                **modv_bvals;
                } mod_vals;
} LDAPMod;
#define mod_values      mod_vals.modv_strvals
#define mod_bvalues     mod_vals.modv_bvals

int ldap_modify_ext(
        LDAP                                 *ld,
        const char                           *dn,
        LDAPMod                              **mods,
        LDAPControl                          **serverctrls,
        LDAPControl                          **clientctrls,
        int                                  *msgidp
);

int ldap_modify_ext_s(
        LDAP                                 *ld,
        const char                           *dn,
        LDAPMod                              **mods,
        LDAPControl                          **serverctrls,
        LDAPControl                          **clientctrls

);

int ldap_modify(
        LDAP                                 *ld,
        const char                           *dn,
        LDAPMod                              **mods
);

int ldap_modify_s(
        LDAP                                 *ld,
        const char                           *dn,
```

```
                LDAPMod                                 **mods
        );
```

Parameters are as follows:

ld	The session handle.
dn	The name of the entry to modify.
mods	A NULL-terminated array of modifications to make to the entry.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_modify_ext()` call succeeds.

The fields in the LDAPMod structure have the following meanings:

mod_op	The modification operation to perform. It should be one of LDAP_MOD_ADD(0x00), LDAP_MOD_DELETE (0x01), or LDAP_MOD_REPLACE(0x02). This field also indicates the type of values included in the mod_vals union. It is logically ORed with LDAP_MOD_BVALUES (0x80) to select the mod_bvalues form. Otherwise, the mod_values form is used.
mod_type	The type of the attribute to modify.
mod_vals	The values (if any) to add, delete, or replace. Only one of the mod_values or mod_bvalues variants should be used, selected by ORing the mod_op field with the constant LDAP_MOD_BVALUES. The mod_values field is a NULL-terminated array of zero-terminated strings and mod_bvalues is a NULL- terminated array of berval structures that can be used to pass binary values such as images.

For LDAP_MOD_ADD modifications, the given values are added to the entry, creating the attribute if necessary.

For LDAP_MOD_DELETE modifications, the given values are deleted from the entry, removing the attribute if no values remain. If the entire attribute is to be deleted, the mod_vals field should be set to NULL.

For LDAP_MOD_REPLACE modifications, the attribute will have the listed values after the modification, having been created if necessary, or removed if the mod_vals field is NULL. All modifications are performed in the order in which they are listed.

The `ldap_modify_ext()` function initiates an asynchronous modify operation and returns the constant LDAP_SUCCESS if the request was successfully sent, or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_modify_ext()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the modify.

Similar to `ldap_modify_ext()`, the `ldap_modify()` function initiates an asynchronous modify operation and returns the message id of the operation initiated. As for `ldap_modify_ext()`, a subsequent call to `ldap_result()` can be used to obtain the result of the modify. In case of error, `ldap_modify()` will return -1, setting the session error parameters in the LDAP structure appropriately.

The synchronous `ldap_modify_ext_s()` and `ldap_modify_s()` functions both return the result of the operation, either the constant LDAP_SUCCESS if the operation was successful, or another LDAP error code if it was not.

See Section 14.18 for more information about possible errors and how to interpret them.

The `ldap_modify_ext()` and `ldap_modify_ext_s()` functions support LDAPv3 server controls and client controls.

14.12. Modifying the Name of an Entry

In LDAP Version 2, the `ldap_modrdn()` and `ldap_modrdn_s()` functions were used to change the name of an LDAP entry. They could only be used to change the least significant component of a name (the RDN or relative distinguished name). LDAPv3 provides the Modify DN protocol operation that allows more general name change access. The `ldap_rename()` and `ldap_rename_s()` functions are used to change the name of an entry, and the use of the `ldap_modrdn()` and `ldap_modrdn_s()` functions is deprecated.

```
int ldap_rename(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *newrdn,
        const char                      *newparent,
        int                             deleteoldrdn,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls,
        int                             *msgidp
);

int ldap_rename_s(
        LDAP                            *ld,
        const char                      *dn,
        const char                      *newrdn,
        const char                      *newparent,
        int                             deleteoldrdn,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls
);
```

Use of the following functions is deprecated.

```
int ldap_modrdn(
        LDAP                            *ld,
        char                            *dn,
        char                            *newrdn,
        int                             deleteoldrdn
);

int ldap_modrdn_s(
        LDAP                            *ld,
        char                            *dn,
        char                            *newrdn,
        int                             deleteoldrdn
);
```

Parameters are as follows:

ld	The session handle.
dn	The name of the entry whose DN is to be changed.
newrdn	The new RDN to give the entry.
newparent	The new parent, or superior entry. If this parameter is NULL, only the RDN of the entry is changed. The root DN may be specified by passing a zero length string, "". The newparent parameter should always be NULL when using Version 2 of the LDAP protocol; otherwise the server's behavior is undefined.
deleteoldrdn	This parameter only has meaning on the rename functions if newrdn is different than the old RDN. It is a boolean value. If it is non-zero, it indicates that the old RDN value(s) should be removed. If it is zero, it indicates that the old RDN value(s) should be retained as non-distinguished values of the entry.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_rename()` call succeeds.

The `ldap_rename()` function initiates an asynchronous modify DN operation and returns the constant LDAP_SUCCESS if the request was successfully sent, or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_rename()` places the DN message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the rename.

The synchronous `ldap_rename_s()` returns the result of the operation, either the constant LDAP_SUCCESS if the operation was successful, or another LDAP error code if it was not. See Section 14.18 for more information about possible errors and how to interpret them.

The `ldap_rename()` and `ldap_rename_s()` functions both support LDAPv3 server controls and client controls.

14.13. Adding an Entry

The following functions are used to add entries to the LDAP directory. There are four variations.

```
int ldap_add_ext(
        LDAP                            *ld,
        const char                      *dn,
        LDAPMod                         **attrs,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls,
        int                             *msgidp
);

int ldap_add_ext_s(
        LDAP                            *ld,
        const char                      *dn,
        LDAPMod                         **attrs,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls
);
```

```
int ldap_add(
        LDAP                                    *ld,
        const char                              *dn,
        LDAPMod                                 **attrs
);

int ldap_add_s(
        LDAP                                    *ld,
        const char                              *dn,
        LDAPMod                                 **attrs
);
```

Parameters are as follows:

ld	The session handle.
dn	The name of the entry to add.
attrs	The entry's attributes, specified using the LDAPMod structure defined for `ldap_modify()`. The mod_type and mod_vals fields should be filled in. The mod_op field is ignored unless ORed with the constant LDAP_MOD_BVALUES, used to select the mod_bvalues case of the mod_vals union.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_add_ext()` call succeeds.

Note that the parent of the entry being added must already exist or the parent must be empty (that is, equal to the root DN) for an add to succeed.

The `ldap_add_ext()` function initiates an asynchronous add operation and returns either the constant LDAP_SUCCESS if the request was successfully sent or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_add_ext()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the add.

Similar to `ldap_add_ext()` , the `ldap_add()` function initiates an asynchronous add operation and returns the message id of the operation initiated. As for `ldap_add_ext()` , a subsequent call to `ldap_result()` can be used to obtain the result of the add. In case of error, `ldap_add()` will return -1, setting the session error parameters in the LDAP structure appropriately.

The synchronous `ldap_add_ext_s()` and `ldap_add_s()` functions both return the result of the operation, either the constant LDAP_SUCCESS if the operation was successful, or another LDAP error code if it was not. See Section 14.18 for more information about possible errors and how to interpret them.

The `ldap_add_ext()` and `ldap_add_ext_s()` functions support LDAPv3 server controls and client controls.

14.14. Deleting an Entry

The following functions are used to delete a leaf entry from the LDAP directory. There are four variations.

```
int ldap_delete_ext(
```

```
        LDAP                                    *ld,
        const char                              *dn,
        LDAPControl                             **serverctrls,
        LDAPControl                             **clientctrls,
        int                                     *msgidp
);

int ldap_delete_ext_s(
        LDAP                                    *ld,
        const char                              *dn,
        LDAPControl                             **serverctrls,
        LDAPControl                             **clientctrls
);

int ldap_delete(
        LDAP                                    *ld,
        const char                              *dn
);

int ldap_delete_s(
        LDAP                                    *ld,
        const char                              *dn
);
```

Parameters are as follows:

ld	The session handle.
dn	The name of the entry to delete.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_delete_ext()` call succeeds.

Note that the entry to delete must be a leaf entry (that is, it must have no children). Deletion of entire subtrees in a single operation is not supported by LDAP.

The `ldap_delete_ext()` function initiates an asynchronous delete operation and returns either the constant LDAP_SUCCESS if the request was successfully sent or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_delete_ext()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the delete.

Similar to `ldap_delete_ext()` , the `ldap_delete()` function initiates an asynchronous delete operation and returns the message id of the operation initiated. As for `ldap_delete_ext()` , a subsequent call to `ldap_result()` can be used to obtain the result of the delete. In case of error, `ldap_delete()` will return -1, setting the session error parameters in the LDAP structure appropriately.

The synchronous `ldap_delete_ext_s()` and `ldap_delete_s()` functions both return the result of the operation, either the constant LDAP_SUCCESS if the operation was successful or another LDAP error code if it was not. See Section 14.18 for more information about possible errors and how to interpret them.

The `ldap_delete_ext()` and `ldap_delete_ext_s()` functions support LDAPv3 server controls and client controls.

14.15. Extended Operations

The `ldap_extended_operation()` and `ldap_extended_operation_s()` functions allow extended LDAP operations to be passed to the server, providing a general protocol extensibility mechanism.

```
int ldap_extended_operation(
        LDAP                            *ld,
        const char                      *requestoid,
        const struct berval             *request data,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls,
        int                             *msgidp
);

int ldap_extended_operation_s(
        LDAP                            *ld,
        const char                      *requestoid,
        const struct berval             *request data,
        LDAPControl                     **serverctrls,
        LDAPControl                     **clientctrls,
        char                            **retoidp,
        struct berval                   **retdatap
);
```

Parameters are as follows:

ld	The session handle.
requestoid	The dotted-OID text string naming the request.
requestdata	The arbitrary data required by the operation (if NULL, no data is sent to the server).
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.
msgidp	This result parameter will be set to the message id of the request if the `ldap_extended_operation()` call succeeds.
retoidp	Pointer to a character string that will be set to an allocated, dotted-OID text string returned by the server. This string should be disposed of using the `ldap_memfree()` function. If no OID was returned, *retoidp is set to NULL.
retdatap	Pointer to a berval structure pointer that will be set to an allocated copy of the data returned by the server. This struct berval should be disposed of using ber_bvfree(). If no data is returned, *retdatap is set to NULL.

The `ldap_extended_operation()` function initiates an asynchronous extended operation and returns either the constant LDAP_SUCCESS if the request was successfully sent or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them. If successful, `ldap_extended_operation()` places the message id of the request in *msgidp. A subsequent call to `ldap_result()` can be used to obtain the result of the extended operation which can be passed to ldap_parse_extended_result() to obtain the OID and data contained in the response.

The synchronous `ldap_extended_operation_s()` function returns the result of the operation, either the constant LDAP_SUCCESS if the operation was successful or another LDAP error code if it

was not. See Section 14.18 for more information about possible errors and how to interpret them. The retoid and retdata parameters are filled in with the OID and data from the response. If no OID or data was returned, these parameters are set to NULL.

The `ldap_extended_operation()` and `ldap_extended_operation_s()` functions both support LDAPv3 server controls and client controls.

14.16. Abandoning an Operation

The following calls are used to abandon an operation in progress:

```
int ldap_abandon_ext(
        LDAP                                    *ld,
        int                                     msgid,
        LDAPControl                             **serverctrls,
        LDAPControl                             **clientctrls
);

int ldap_abandon(
        LDAP                                    *ld,
        int                                     msgid
);
```

Parameters are as follows:

ld	The session handle.
msgid	The message id of the request to be abandoned.
serverctrls	List of LDAP server controls.
clientctrls	List of client controls.

The ldap_abandon_ext() function abandons the operation with message id msgid and returns either the constant LDAP_SUCCESS if the abandon was successful or another LDAP error code if not. See Section 14.18 for more information about possible errors and how to interpret them.

The ldap_abandon() function is identical to ldap_abandon_ext() except that it does not accept client or server controls and it returns zero if the abandon was successful, -1 otherwise and does not support LDAPv3 server controls or client controls.

After a successful call to ldap_abandon() or ldap_abandon_ext() , results with the given message id are never returned from a subsequent call to `ldap_result()` . There is no server response to LDAP abandon operations.

14.17. Obtaining Results and Looking Inside LDAP Messages

The `ldap_result()` function is used to obtain the result of a previous asynchronously initiated operation. Note that depending on how it is called, `ldap_result()` may actually return a list or "chain" of result messages. Once a chain of messages has been returned to the caller, it is no longer tied in any caller-visible way to the LDAP request that produced it. Therefore, a chain of messages returned by calling `ldap_result()` or by calling a synchronous search function will never be affected by subsequent LDAP API calls (except for ldap_msgfree() , which is used to dispose of a chain of messages).

The ldap_msgfree() function frees the result messages (possibly an entire chain of messages) obtained from a previous call to `ldap_result()` or from a call to a synchronous search function.

The ldap_msgtype() function returns the type of an LDAP message. The ldap_msgid() function returns the message ID of an LDAP message.

```
int ldap_result(
        LDAP                                    *ld,
        int                                     msgid,
        int                                     all,
        struct timeval                          *timeout,
        LDAPMessage                             **res
);

int ldap_msgfree( LDAPMessage *res );

int ldap_msgtype( LDAPMessage *res );

int ldap_msgid( LDAPMessage *res );
```

Parameters are as follows:

ld	The session handle.
msgid	The message id of the operation whose results are to be returned, or the constant LDAP_RES_ANY (-1) if any result is desired.
all	Specifies how many messages will be retrieved in a single call to `ldap_result()` . This parameter only has meaning for search results. Pass the constant LDAP_MSG_ONE (0x00) to retrieve one message at a time. Pass LDAP_MSG_ALL (0x01) to request that all results of a search be received before returning all results in a single chain. Pass LDAP_MSG_RECEIVED (0x02) to indicate that all results retrieved so far should be returned in the result chain.
timeout	A timeout specifying how long to wait for results to be returned. A NULL value causes `ldap_result()` to block until results are available. A timeout value of zero seconds specifies a polling behavior.
res	For `ldap_result()` , a result parameter that will contain the result(s) of the operation. For ldap_msgfree() , the result chain to be freed, obtained from a previous call to `ldap_result()` , ldap_search_s() , or ldap_search_st() .

Upon successful completion, `ldap_result()` returns the type of the first result returned in the res parameter. This will be one of the following constants.

```
LDAP_RES_BIND (0x61)

LDAP_RES_SEARCH_ENTRY (0x64)

LDAP_RES_SEARCH_REFERENCE (0x73)      -- new in LDAPv3

LDAP_RES_SEARCH_RESULT (0x65)

LDAP_RES_MODIFY (0x67)
```

```
LDAP_RES_ADD (0x69)

LDAP_RES_DELETE (0x6B)

LDAP_RES_MODDN (0x6D)

LDAP_RES_COMPARE (0x6F)

LDAP_RES_EXTENDED (0x78)              -- new in LDAPv3
```

The `ldap_result()` function returns 0 if the timeout expired and -1 if an error occurs, in which case the error parameters of the LDAP session handle will be set accordingly.

The ldap_msgfree() function frees the result structure pointed to by res and returns the type of the message it freed.

The ldap_msgtype() function returns the type of the LDAP message it is passed as a parameter. The type will be one of the types listed above, or -1 on error.

The ldap_msgid() function returns the message ID associated with the LDAP message passed as a parameter.

14.18. Handling Errors and Parsing Results

The following calls are used to extract information from results and handle errors returned by other LDAP API functions. Note that ldap_parse_sasl_bind_result() and ldap_parse_extended_result() must typically be used in addition to ldap_parse_result() to retrieve all the result information from SASL bind and extended operations, respectively.

```
int ldap_parse_result(
        LDAP                            *ld,
        LDAPMessage                     *res,
        int                             *errcodep,
        char                            **matcheddnp,
        char                            **errmsgp,
        char                            ***referralsp,
        LDAPControl                     ***serverctrlsp,
        int                             freeit
);

int ldap_parse_sasl_bind_result(
        LDAP                            *ld,
        LDAPMessage                     *res,
        struct berval                   **servercredp,
        int                             freeit
);

int ldap_parse_extended_result(
        LDAP                            *ld,
        LDAPMessage                     *res,
        char                            **resultoidp,
        struct berval                   **resultdata,
        int                             freeit
);

char *ldap_err2string( int err );
```

The use of the following functions is deprecated.

```
int ldap_result2error(
        LDAP                                    *ld,
        LDAPMessage                             *res,
        int                                     freeit
);

void ldap_perror( LDAP *ld, const char *msg );
```

Parameters are as follows:

ld	The session handle.
res	The result of an LDAP operation as returned by `ldap_result()` or one of the synchronous API operation calls.
errcodep	This result parameter will be filled in with the LDAP error code field from the LDAPMessage result. This is the indication from the server of the outcome of the operation. NULL may be passed to ignore this field.
matcheddnp	In the case of a return of LDAP_NO_SUCH_OBJECT, this result parameter will be filled in with a DN indicating how much of the name in the request was recognized. NULL may be passed to ignore this field. The matched DN string should be freed by calling `ldap_memfree()` .
errmsgp	This result parameter will be filled in with the contents of the error message field from the LDAPMessage result. The error message string should be freed by calling `ldap_memfree()` . NULL may be passed to ignore this field.
referralsp	This result parameter will be filled in with the contents of the referrals field from the LDAPMessage result, indicating zero or more alternate LDAP servers where the request should be retried. The referrals array should be freed by calling ldap_value_free() . NULL may be passed to ignore this field.
serverctrlsp	This result parameter will be filled in with an allocated array of controls copied out of the LDAPMessage result. The control array should be freed by calling ldap_controls_free() .
freeit	A boolean that determines whether or not the res parameter is disposed of. Pass any non-zero value to have these functions free res after extracting the requested information. This option is provided as a convenience; you can also use ldap_msgfree() to free the result later. If freeit is non-zero, the entire chain of messages represented by res is disposed of.
servercredp	For SASL bind results, this result parameter will be filled in with the credentials passed back by the server for mutual authentication, if given. An allocated berval structure is returned that should be disposed of by calling ber_bvfree(). NULL may be passed to ignore this field.
resultoidp	For extended results, this result parameter will be filled in with the dotted-OID text representation of the name of the extended operation response. This string should be disposed of by calling `ldap_memfree()` . NULL may be passed to ignore this field.

resultdatap	For extended results, this result parameter will be filled in with a pointer to a struct berval containing the data in the extended operation response. It should be disposed of by calling ber_bvfree(). NULL may be passed to ignore this field.
err	For ldap_err2string() , an LDAP error code, as returned by ldap_parse_result() or another LDAP API call.

Additional parameters for the deprecated functions are not described. See RFC 1823 for more information.

All three of the ldap_parse_*_result() functions skip over messages of type LDAP_RES_SEARCH_ENTRY and LDAP_RES_SEARCH_REFERENCE when looking for a result message to parse. They return either the constant LDAP_SUCCESS if the result was successfully parsed or another LDAP error code if not. Note that the LDAP error code that indicates the outcome of the operation performed by the server is placed in the errcodep ldap_parse_result() parameter. If a chain of messages that contains more than one result message is passed to these functions, they always operate on the first result in the chain.

The ldap_err2string() function is used to convert a numeric LDAP error code, as returned by either one of the three ldap_parse_*_result() functions or one of the synchronous API operation calls, into an informative zero-terminated character string message describing the error. It returns a pointer to static data.

14.18.1. Stepping Through a List of Results

The ldap_first_message() and ldap_next_message() functions are used to step through the list of messages in a result chain returned by `ldap_result()` . For search operations, the result chain may actually include referral messages, entry messages, and result messages. The ldap_count_messages() function is used to count the number of messages returned. The ldap_msgtype() function can be used to distinguish between the different message types.

```
LDAPMessage *ldap_first_message( LDAP *ld, LDAPMessage *res );
LDAPMessage *ldap_next_message ( LDAP *ld, LDAPMesage *msg );
int ldap_count_messages( LDAP *ld, LDAPMessage *res );
```

Parameters are as follows:

ld	The session handle.
res	The result chain, as obtained by a call to one of the synchronous search functions or `ldap_result()` .
msg	The message returned by a previous call to ldap_first_message() or ldap_next_message() .

The ldap_first_message() and ldap_next_message() functions will return NULL when no more messages exist in the result set to be returned. NULL is also returned if an error occurs while stepping through the entries, in which case the error parameters in the session handle ld will be set to indicate the error.

The ldap_count_messages() function returns the number of messages contained in a chain of results. It can also be used to count the number of messages that remain in a chain if called with a message, entry, or reference returned by ldap_first_message() , ldap_next_message() , ldap_first_entry() , ldap_next_entry() , ldap_first_reference() , ldap_next_reference() .

14.19. Parsing Search Results

The following calls are used to parse the entries and references returned by ldap_search() . These results are returned in an opaque structure that should only be accessed by calling the functions. Functions are provided to step through the entries and references returned, step through the attributes of an entry, retrieve the name of an entry, and retrieve the values associated with a given attribute in an entry.

14.19.1. Stepping Through a List of Entries

The ldap_first_entry() and ldap_next_entry() functions are used to step through and retrieve the list of entries from a search result chain. The ldap_first_reference() and ldap_next_reference() functions are used to step through and retrieve the list of continuation references from a search result chain. The ldap_count_entries() function is used to count the number of entries returned. The ldap_count_references() function is used to count the number of references returned.

```
LDAPMessage *ldap_first_entry( LDAP *ld, LDAPMessage *res );

LDAPMessage *ldap_next_entry( LDAP *ld, LDAPMessage *entry );

LDAPMessage *ldap_first_reference( LDAP *ld, LDAPMessage *res );

LDAPMessage *ldap_next_reference( LDAP *ld, LDAPMessage *ref );

int ldap_count_entries( LDAP *ld, LDAPMessage *res );

int ldap_count_references( LDAP *ld, LDAPMessage *res );
```

Parameters are as follows:

ld	The session handle.
res	The search result, as obtained by a call to one of the synchronous search functions or `ldap_result()` .
entry	The entry returned by a previous call to ldap_first_entry() or ldap_next_entry() .

The ldap_first_entry() and ldap_next_entry() functions will return NULL when no more entries or references exist in the result set to be returned. NULL is also returned if an error occurs while stepping through the entries, in which case the error parameters in the session handle ld will be set to indicate the error.

The ldap_count_entries() function returns the number of entries contained in a chain of entries. It can also be used to count the number of entries that remain in a chain if called with a message, entry or reference returned by ldap_first_message() , ldap_next_message() , ldap_first_entry() , ldap_next_entry() , ldap_first_reference() , ldap_next_reference() .

The ldap_count_references() function returns the number of references contained in a chain of search results. It can also be used to count the number of references that remain in a chain.

14.19.2. Stepping Through the Attributes of an Entry

The ldap_first_attribute() and ldap_next_attribute() calls are used to step through the list of attribute types returned with an entry.

```
        char *ldap_first_attribute(
                LDAP                                    *ld,
```

```
        LDAPMessage                                     *entry,
        BerElement                                      **ptr
);

char *ldap_next_attribute(
        LDAP                                            *ld,
        LDAPMessage                                     *entry,
        BerElement                                      *ptr
);

void ldap_memfree( char *mem );
```

Parameters are as follows:

ld	The session handle.
entry	The entry whose attributes are to be stepped through, as returned by ldap_first_entry() or ldap_next_entry() .
ptr	In ldap_first_attribute() , the address of a pointer used internally to keep track of the current position in the entry. In ldap_next_attribute() , the pointer returned by a previous call to ldap_first_attribute() .
mem	A pointer to memory allocated by the LDAP library, such as the attribute type names returned by ldap_first_attribute() and ldap_next_attribute() , or the DN returned by ldap_get_dn() .

The ldap_first_attribute() and ldap_next_attribute() functions will return NULL when the end of the attributes is reached, or if there is an error, in which case the error parameters in the session handle ld will be set to indicate the error.

Both functions return a pointer to an allocated buffer containing the current attribute name. This should be freed when no longer in use by calling `ldap_memfree()` .

The ldap_first_attribute() function will allocate and return in ptr a pointer to a BerElement used to keep track of the current position. This pointer should be passed in subsequent calls to ldap_next_attribute() to step through the entry's attributes. After a set of calls to ldap_first_attribute() and ldap_next_attribute() , if ptr is non-NULL, it should be freed by calling ber_free(ptr, 0) . Note that it is very important to pass the second parameter as 0 (zero) in this call, since the buffer associated with the BerElement does not point to separately allocated memory.

The attribute type names returned are suitable for passing in a call to ldap_get_values() to retrieve the associated values.

14.19.3. Retrieving the Values of an Attribute

The ldap_get_values() and ldap_get_values_len() functions are used to retrieve the values of a given attribute from an entry. The ldap_count_values() and ldap_count_values_len() functions are used to count the returned values. The ldap_value_free() and ldap_value_free_len() functions are used to free the values.

```
char **ldap_get_values(
        LDAP            *ld,
        LDAPMessage     *entry,
        char            *attr
);
```

```
struct berval **ldap_get_values_len(
        LDAP            *ld,
        LDAPMessage     *entry,
        char            *attr
);

int ldap_count_values( char **vals )

int ldap_count_values_len( struct berval **vals );

void ldap_value_free( char **vals );

void ldap_value_free_len( struct berval **vals );
```

Parameters are as follows:

ld	The session handle.
entry	The entry from which to retrieve values, as returned by ldap_first_entry() or ldap_next_entry() .
attr	The attribute whose values are to be retrieved, as returned by ldap_first_attribute() or ldap_next_attribute() , or a caller- supplied string (for example, "mail").
vals	The values returned by a previous call to ldap_get_values() or ldap_get_values_len() .

Two forms of the various calls are provided. The first form is only suitable for use with non-binary character string data. The second _len form is used with any kind of data.

The ldap_get_values() and ldap_get_values_len() functions return NULL if no values are found for attr or if an error occurs.

The ldap_count_values() and ldap_count_values_len() functions return -1 if an error occurs such as the vals parameter being invalid.

Note that the values returned are dynamically allocated and should be freed by calling either ldap_value_free() or ldap_value_free_len() when no longer in use.

14.19.4. Retrieving the Name of an Entry

The ldap_get_dn() function is used to retrieve the name of an entry. The ldap_explode_dn() and ldap_explode_rdn() functions are used to break up a name into its component parts. The ldap_dn2ufn() function is used to convert the name into a more user-friendly format.

```
char *ldap_get_dn( LDAP *ld, LDAPMessage *entry );

char **ldap_explode_dn( const char *dn, int notypes );

char **ldap_explode_rdn( const char *rdn, int notypes );

char *ldap_dn2ufn( const char *dn );
```

Parameters are as follows:

ld	The session handle.
entry	The entry whose name is to be retrieved, as returned by ldap_first_entry() or ldap_next_entry() .

dn	The dn to explode, such as returned by ldap_get_dn() .
rdn	The rdn to explode, such as returned in the components of the array returned by ldap_explode_dn() .
notypes	A boolean parameter, if non-zero indicating that the DN or RDN components should have their type information stripped off (i.e., "cn=Babs" would become "Babs").

The ldap_get_dn() function will return NULL if there is some error parsing the dn, setting error parameters in the session handle ld to indicate the error. It returns a pointer to newly allocated space that the caller should free by calling `ldap_memfree()` when it is no longer in use.

The ldap_explode_dn() function returns a NULL-terminated char * array containing the RDN components of the DN supplied, with or without types as indicated by the notypes parameter. The components are returned in the order they appear in the dn. The array returned should be freed when it is no longer in use by calling ldap_value_free() .

The ldap_explode_rdn() function returns a NULL-terminated char * array containing the components of the RDN supplied, with or without types as indicated by the notypes parameter. The components are returned in the order they appear in the rdn. The array returned should be freed when it is no longer in use by calling ldap_value_free() .

The ldap_dn2ufn() function converts the DN into the user friendly format. The UFN returned is newly allocated space that should be freed by a call to `ldap_memfree()` when no longer in use.

14.19.5. Retrieving Controls from an Entry

The ldap_get_entry_controls() function is used to extract LDAP controls from an entry.

```
int ldap_get_entry_controls(
        LDAP                                    *ld,
        LDAPMessage                             *entry,
        LDAPControl                             ***serverctrlsp
);
```

Parameters are as follows:

ld	The session handle.
entry	The entry to extract controls from, as returned by ldap_first_entry() or ldap_next_entry() .
serverctrlsp	This result parameter will be filled in with an allocated array of controls copied out of entry. The control array should be freed by calling ldap_controls_free() . If serverctrlsp is NULL, no controls are returned.

The ldap_get_entry_controls() function returns an LDAP error code that indicates whether the reference could be successfully parsed (LDAP_SUCCESS if all goes well).

14.19.6. Parsing References

The ldap_parse_reference() function is used to extract referrals and controls from a SearchResultReference message.

```
int ldap_parse_reference(
        LDAP                                *ld,
        LDAPMessage                         *ref,
```

```
        char                    ***referralsp,
        LDAPControl             ***serverctrlsp,
        int                     freeit
);
```

Parameters are as follows:

ld	The session handle.
ref	The reference to parse, as returned by `ldap_result()` , ldap_first_reference() , or ldap_next_reference() .
referralsp	This result parameter will be filled in with an allocated array of character strings. The elements of the array are the referrals (typically LDAP URLs) contained in ref. The array should be freed when no longer in used by calling ldap_value_free() . If referralsp is NULL, the referral URLs are not returned.
serverctrlsp	This result parameter will be filled in with an allocated array of controls copied out of ref. The control array should be freed by calling ldap_controls_free() . If serverctrlsp is NULL, no controls are returned.
freeit	A boolean that determines whether or not the ref parameter is disposed of. Pass any non-zero value to have these functions free ref after extracting the requested information. This option is provided as a convenience; you can also use ldap_msgfree() to free the result later.

The ldap_parse_reference() function returns an LDAP error code that indicates whether the reference could be successfully parsed (LDAP_SUCCESS if all goes well).

14.20. Encoded ASN.1 Value Manipulation

This section describes functions that may be used to encode and decode BER-encoded ASN.1 values, which are often used inside of control and extension values.

The following additional integral types are defined for use in manipulation of BER encoded ASN.1 values:

```
typedef unsigned long ber_tag_t; /* for BER tags */

typedef long     ber_int_t; /* for BER ints, enums, and Booleans */
```

With the exceptions of two new functions, ber_flatten() and ber_init() , these functions are compatible with the University of Michigan LDAP 3.3 implementation of BER.

```
typedef struct berval {
        ber_len_t            bv_len;
        char                 *bv_val;
} BerValue;
```

A struct berval contains a sequence of bytes and an indication of its length. The bv_val is not null terminated. A bv_len must always be a nonnegative number. Applications may allocate their own berval structures.

```
typedef struct berelement {
     /* opaque */
} BerElement;
```

The BerElement structure contains not only a copy of the encoded value, but also state information used in encoding or decoding. Applications cannot allocate their own BerElement structures. The internal state is neither thread-specific nor locked, so two threads should not manipulate the same BerElement value simultaneously.

A single BerElement value cannot be used for both encoding and decoding.

```
void ber_bvfree( struct berval *bv );
```

The ber_bvfree() function frees a berval returned from this API. Both the bv->bv_val string and the berval itself are freed. Applications should not use ber_bvfree() with bervals which the application has allocated.

```
void ber_bvecfree ( struct berval **bv );
```

The ber_bvecfree() function frees an array of bervals returned from this API. Each of the bervals in the array are freed using ber_bvfree() , then the array itself is freed.

```
struct berval *ber_bvdup (struct berval *bv );
```

The ber_bvdup() function returns a copy of a berval. The bv_val field in the returned berval points to a different area of memory as the bv_val field in the argument berval. The null pointer is returned on error (for example, out of memory).

```
void ber_free ( BerElement *ber, int fbuf );
```

The ber_free() function frees a BerElement which is returned from the API calls ber_alloc_t() or ber_init() . Each BerElement must be freed by the caller. The second argument fbuf should always be set to 1 to ensure that the internal buffer used by the BER functions is freed as well as the BerElement container itself.

14.20.1. Encoding

The following is an example of encoding:

```
BerElement *ber_alloc_t(int options);
```

The ber_alloc_t() function constructs and returns BerElement. The null pointer is returned on error. The options field contains a bitwise-or of options which are to be used when generating the encoding of this BerElement. One option is defined and must always be supplied:

```
#define LBER_USE_DER 0x01
```

When this option is present, lengths will always be encoded in the minimum number of octets. Note that this option does not cause values of sets and sequences to be rearranged in tag and byte order, so these functions are not sufficient for generating DER output as defined in X.509 and X.680. If the caller takes responsibility for ordering values of sets and sequences correctly, DER output as defined in X.509 and X.680 can be produced.

Unrecognized option bits are ignored.

The BerElement returned by ber_alloc_t() is initially empty. Calls to ber_printf() will append bytes to the end of the BerElement.

```
int ber_printf(BerElement *ber, char *fmt, ... )
```

The ber_printf() function is used to encode a BER element in much the same way that sprintf() works. One important difference, though, is that state information is kept in the BER argument so that multiple

calls can be made to ber_printf() to append to the end of the BER element. BER must be a pointer to a BerElement returned by ber_alloc_t() . The ber_printf() function interprets and formats its arguments according to the format string fmt . The ber_printf() function returns -1 if there is an error during encoding and a positive number if successful. As with sprintf() , each character in fmt refers to an argument to ber_printf() .

The format string can contain the following format characters:

t	Tag. The next argument is a ber_tag_t specifying the tag to override the next element to be written to the ber. This works across calls. The value must contain the tag class, constructed bit, and tag value. The tag value must fit in a single octet (tag value is less than 32). For example, a tag of "[3]" for a constructed type is 0xA3.
b	Boolean. The next argument is a ber_int_t , containing either 0 for FALSE or 0xff for TRUE. A boolean element is output. If this format character is not preceded by the 't' format modifier, the tag 0x01 is used for the element.
e	Enumerated. The next argument is a ber_int_t , containing the enumerated value in the host's byte order. An enumerated element is output. If this format character is not preceded by the 't' format modifier, the tag 0x0A is used for the element.
i	Integer. The next argument is a ber_int_t , containing the integer in the host's byte order. An integer element is output. If this format character is not preceded by the 't' format modifier, the tag 0x02 is used for the element.
B	Bitstring. The next two arguments are a char * pointer to the start of the bitstring, followed by a ber_len_t containing the number of bits in the bitstring. A bitstring element is output, in primitive form. If this format character is not preceded by the 't' format modifier, the tag 0x03 is used for the element.
n	Null. No argument is required. An ASN.1 NULL element is output. If this format character is not preceded by the 't' format modifier, the tag 0x05 is used for the element.
o	Octet string. The next two arguments are a char *, followed by a ber_len_t with the length of the string. The string may contain null bytes and need not by zero-terminated. An octet string element is output, in primitive form. If this format character is not preceded by the 't' format modifier, the tag 0x04 is used for the element.
s	Octet string. The next argument is a char * pointing to a zero-terminated string. An octet string element in primitive form is output, which does not include the trailing '\0' byte. If this format character is not preceded by the 't' format modifier, the tag 0x04 is used for the element.
v	Several octet strings. The next argument is a char **, an array of char * pointers to zero-terminated strings. The last element in the array must be a null pointer. The octet strings do not include the leading SEQUENCE OF octet strings. The 't' format modifier cannot be used with this format character.
V	Several octet strings. A NULL-terminated array of struct berval *'s is supplied. Note that a construct like '{V}' is required to get an actual SEQUENCE OF octet strings. The 't' format modifier cannot be used with this format character.
{	Begin sequence. No argument is required. If this format character is not preceded by the 't' format modifier, the tag 0x30 is used.
}	End sequence. No argument is required. The 't' format modifier cannot be used with this format character.

[	Begin set. No argument is required. If this format character is not preceded by the 't' format modifier, the tag 0x31 is used.
]	End set. No argument is required. The 't' format modifier cannot be used with this format character.

Each use of a '{' format character must be matched by a '}' character, either later in the format string, or in the format string of a subsequent call to ber_printf() for that BerElement. The same applies to the '[' and ']'.

Sequences and sets nest, and implementations of this API must maintain internal state to be able to properly calculate the lengths.

```
int ber_flatten (BerElement *ber, struct berval **bvPtr);
```

The ber_flatten() function allocates a struct berval whose contents are a BER encoding taken from the ber argument. The bvPtr pointer points to the returned berval, which must be freed using ber_bvfree() . This function returns 0 on success and -1 on error.

The ber_flatten() API call is not present in U-M LDAP 3.3.

The use of ber_flatten() on a BerElement in which all '{' and '}' format modifiers have not been properly matched is an error (that is, -1 will be returned by ber_flatten() if this situation is exists).

14.20.1.1. Encoding Example

The following is an example of encoding the following ASN.1 data type:

```
Example1Request ::= SEQUENCE {
        s     OCTET STRING, -- must be printable
        val1  INTEGER,
        val2  [0] INTEGER DEFAULT 0
}

int encode_example1(char *s,ber_int_t val1,ber_int_t val2,
                          struct berval **bvPtr)

{
         BerElement *ber;
        int rc;

        ber = ber_alloc_t(LBER_USE_DER);

        if (ber == NULL) return -1;

        if (ber_printf(ber,"{si",s,val1) == -1) {
                ber_free(ber,1);
                return -1;
}

if (val2 != 0) {
if (ber_printf(ber,"ti",(ber_tag_t)0x80,val2) == -1) {
        ber_free(ber,1);
        return -1;
        }
}

if (ber_printf(ber,"}") == -1) {
```

```
        ber_free(ber,1);
        return -1;
}


rc = ber_flatten(ber,bvPtr);
ber_free(ber,1);
return rc;
}
```

14.20.2. Decoding

The following two symbols are available to applications.

```
#define LBER_ERROR   0xffffffffL
#define LBER_DEFAULT 0xffffffffL

BerElement *ber_init (struct berval *bv);
```

The ber_init() function constructs a BerElement and returns a new BerElement containing a copy of the data in the bv argument. The ber_init() function returns the null pointer on error.

```
ber_tag_t ber_scanf (BerElement *ber, char *fmt, ... );
```

The ber_scanf() function is used to decode a BER element in much the same way that sscanf() works. One important difference, though, is that some state information is kept with the ber argument so that multiple calls can be made to ber_scanf() to sequentially read from the BER element. The ber argument must be a pointer to a BerElement returned by ber_init() . The ber_scanf() function interprets function the bytes according to the format string fmt, and stores the results in its additional arguments. The ber_scanf() function returns LBER_ERROR on error, and a different value on success.

The format string contains conversion specifications which are used to direct the interpretation of the BER element. The format string can contain the following characters:

a	Octet string. A char ** argument should be supplied. Memory is allocated, filled with the contents of the octet string, null- terminated, and the pointer to the string is stored in the argument. The returned value must be freed using `ldap_memfree()` . The tag of the element must indicate the primitive form (constructed strings are not supported) but is otherwise ignored and discarded during the decoding. This format cannot be used with octet strings which could contain null bytes.
O	Octet string. A struct berval ** argument should be supplied, which upon return points to a allocated struct berval containing the octet string and its length. The ber_bvfree() function must be called to free the allocated memory. The tag of the element must indicate the primitive form (constructed strings are not supported) but is otherwise ignored during the decoding.
b	Boolean. A pointer to a ber_int_t should be supplied. The value stored will be 0 for FALSE or nonzero for TRUE. The tag of the element must indicate the primitive form but is otherwise ignored during the decoding.
e	Enumerated value stored will be in host byte order. The tag of the element must indicate the primitive form but is otherwise ignored during the decoding. The ber_scanf() function will return an error if the enumerated value cannot be stored in a ber_int_t .
i	Integer. A pointer to a ber_int_t should be supplied. The value stored will be in host byte order. The tag of the element must indicate the primitive form but is

	otherwise ignored during the decoding. The ber_scanf() function will return an error if the integer cannot be stored in a ber_int_t .
B	Bitstring. A char ** argument should be supplied which will point to the allocated bits, followed by a ber_len_t * argument, which will point to the length (in bits) of the bit-string returned. The `ldap_memfree()` function must be called to free the bit-string. The tag of the element must indicate the primitive form (constructed bitstrings are not supported) but is otherwise ignored during the decoding.
n	Null. No argument is required. The element is simply skipped if it is recognized as a zero-length element. The tag is ignored.
v	Several octet strings. A char *** argument should be supplied, which upon return points to a allocated null-terminated array of char *'s containing the octet strings. NULL is stored if the sequence is empty. The `ldap_memfree()` function must be called to free each element of the array and the array itself. The tag of the sequence and of the octet strings are ignored.
V	Several octet strings (which could contain null bytes). A struct berval *** should be supplied, which upon return points to a allocated null-terminated array of struct berval *'s containing the octet strings and their lengths. NULL is stored if the sequence is empty. The ber_bvecfree() function can be called to free the allocated memory. The tag of the sequence and of the octet strings are ignored.
x	Skip element. The next element is skipped. No argument is required.
{	Begin sequence. No argument is required. The initial sequence tag and length are skipped.
}	End sequence. No argument is required.
[	Begin set. No argument is required. The initial set tag and length are skipped.
]	End set. No argument is required.

```
ber_tag_t ber_peek_tag (BerElement *ber, ber_len_t *lenPtr);
```

The ber_peek_tag() function returns the tag of the next element to be parsed in the BerElement argument. The length of this element is stored in the *lenPtr argument. LBER_DEFAULT is returned if there is no further data to be read. The ber argument is not modified.

```
ber_tag_t ber_skip_tag (BerElement *ber, ber_len_t *lenPtr);
```

The ber_skip_tag() function is similar to ber_peek_tag() , except that the state pointer in the BerElement argument is advanced past the first tag and length, and is pointed to the value part of the next element. This function should only be used with constructed types and situations when a BER encoding is used as the value of an OCTET STRING. The length of the value is stored in *lenPtr.

```
ber_tag_t ber_first_element(BerElement *ber,
        ber_len_t *lenPtr, char **opaquePtr);


ber_tag_t ber_next_element  (BerElement *ber,
        ber_len_t *lenPtr, char *opaque);
```

The ber_first_element() and ber_next_element() functions are used to traverse a SET, SET OF, SEQUENCE or SEQUENCE OF data value. The ber_first_element() function calls ber_skip_tag() , stores internal information in *lenPtr and *opaquePtr, and calls ber_peek_tag() for the first element inside the constructed value. LBER_DEFAULT is returned if the constructed value is empty. The ber_next_element() function positions the state at the start of the next element in the constructed type. LBER_DEFAULT is returned if there are no further values.

The len and opaque values should not be used by applications other than as arguments to ber_next_element() , as shown in the following example.

14.20.2.1. Decoding Example

The following is an example of decoding an ASN.1 data type:

```
        Example2Request ::= SEQUENCE {
                dn OCTET STRING, -- must be printable
                scope ENUMERATED { b (0), s (1), w (2) },
                ali ENUMERATED { n (0), s (1), f (2), a (3) },
                size INTEGER,
                time INTEGER,
                tonly BOOLEAN,
                attrs SEQUENCE OF OCTET STRING, -- must be printable
                [0] SEQUENCE OF SEQUENCE {
                        type OCTET STRING -- must be printable,
                        crit BOOLEAN DEFAULT FALSE,
                        value OCTET STRING
        } OPTIONAL }

        #define TAG_CONTROL_LIST 0xA0U /* context specific cons 0 */

        int decode_example2(struct berval *bv)
{
        BerElement *ber;
        ber_len_t len;
        ber_tag_t res;
        ber_int_t scope, ali, size, time, tonly;
        char *dn = NULL, **attrs = NULL;
        int i,rc = 0;
        ber = ber_init(bv);
        if (ber == NULL) {
                        fputs("ERROR ber_init failed\n", stderr);
                        return -1;
        }

        res = ber_scanf(ber,"{aiiiib{v}",&dn,&scope,&ali,
                                &size,&time,&tonly,&attrs);

        if (res == LBER_ERROR) {
                        fputs("ERROR ber_scanf failed\n", stderr);
                        ber_free(ber,1);
                        return -1;
        }

        /* *** use dn */
        ldap_memfree(dn);

        for (i = 0; attrs != NULL && attrs[i] != NULL; i++) {
                /* *** use attrs[i] */
                ldap_memfree(attrs[i]);
        }
        ldap_memfree(attrs);

        if (ber_peek_tag(ber,&len) == TAG_CONTROL_LIST) {
                char *opaque;
                ber_tag_t tag;
```

```
                for (tag = ber_first_element(ber,&len,&opaque);
                     tag != LBER_DEFAULT;
                     tag = ber_next_element (ber,&len,opaque)) {

                                ber_len_t tlen;
                                ber_tag_t ttag;
                                char *type;
                                ber_int_t crit;
                                struct berval *value;

                                if (ber_scanf(ber,"{a",&type) ==
 LBER_ERROR) {
                                        fputs("ERROR cannot parse type\n",
                                        stderr);
                                        break;
                                }
                                /* *** use type */
                                ldap_memfree(type);

                                ttag = ber_peek_tag(ber,&tlen);
                                if (ttag == 0x01U) { /* boolean */
                                        if (ber_scanf(ber,"b",
                                                    &crit) == LBER_ERROR){
                                             fputs("ERROR cannot parse crit
\n",
                                                    stderr);
                                             rc = -1;
                                             break;
                                        }


                        } else if (ttag == 0x04U) { /* octet string */
                                        crit = 0;
                        } else {
                                        fputs("ERROR extra field in
 controls\n",
                                            stderr );
                                        break;
                        }

                        if (ber_scanf(ber,"O}",&value) == LBER_ERROR) {
                                fputs("ERROR cannot parse value\n",
                                stderr);
                                rc = -1;
                                break;
                        }
                        /* *** use value */
                        ber_bvfree(value);
                }
        }

        if ( rc == 0 ) { /* no errors so far */
                if (ber_scanf(ber,"}") == LBER_ERROR) {
                          rc = -1;
                }
        }
```

```
        ber_free(ber,1);

        return rc;

}
```

14.21. Using LDAP with VSI SSL for OpenVMS

Secure Sockets Layer (SSL) is the open standard security protocol for the secure transfer of sensitive information over the Internet.

You can establish VSI SSL for OpenVMS Alpha on an LDAP session if the server supports such sessions. SSL uses X.509 public key technology to provide the following security functions:

- Integrity and confidentiality of the LDAP dialog

  This is the most common use of VSI SSL. The bytes sent over the wire are encrypted.

- Authentication of the client

  Some servers use SSL to authenticate the client and make access control decisions based on the client identity. In this case, the client must have access to its private key and its certificate. The client certificate subject is a DN.

- Authentication of the server

  It might be important for the client to verify the identity of the server to which it is talking. In this case, the client must have access to the appropriate certification authority (CA) public keys.

There are several versions of SSL: SSLv2 (2.0), SSLv3 (3.0), and TLSv1 (3.1). TLS is the latest Internet standard. It does not require the use of RSA algorithms. Usually the client specifies the highest version it supports, and the server negotiates downward, if necessary. The client library supports all the versions listed here.

You can establish SSL over LDAP two different ways:

- LDAPS

  This older, *de facto* standard uses a separate TCP/IP port (usually 636) specifically for SSL over LDAP. In this case, the second parameter to the ldap_tls_start() function must be set to zero.

- StartTLS

  This proposed Internet standard uses a regular LDAP port (usually 389) and requires the client to request the use of SSL. In this case, the second parameter to the ldap_tls_start() function must be set to 1.

14.21.1. VSI SSL Certificate Options

The following session-handle options are specific to SSL and can be set by the ldap_set_option() function:

- LDAP_OPT_TLS_CERT_REQUIRED (0x7001) void *

  Set to LDAP_OPT_ON if the client library requires a server certificate to be present the next time the ldap_tls_start() function is called. The default value is LDAP_OPT_OFF; a server certificate is not required.

- LDAP_OPT_TLS_VERIFY_REQUIRED (0x7002) void *

  Set to LDAP_OPT_ON if the client library requires that a server certificate path be validated the next time the ldap_tls_start() function is called. The default value is LDAP_OPT_OFF; the server certificate, if any, is not verified.

- LDAP_OPT_TLS_CERT_FILE (0x7003) char *

  Set to the name of a file containing the client's certificate for use by the ldap_tls_start() function.

- LDAP_OPT_TLS_PKEY_FILE (0x7004) char *

  Set to the name of a file containing the client's private key for use by the ldap_tls_start() function.

- LDAP_OPT_TLS_CA_FILE (0x7005) char *

  Set to the name of a file containing CA public keys used for validation of the server by the ldap_tls_start() function.

- LDAP_OPT_TLS_CA_PATH (0x7006) char *

  Set to the name of a directory on disk containing CA public key files used for validation of the server by the ldap_tls_start() function.

- LDAP_OPT_TLS_VERSION (0x7007) int *

  Set to the desired SSL protocol version. This option takes one of the following values:

  1: TLSv1 only
  20: SSLv2 only
  23: SSLv2 or SSLv3
  30: SSLv3 only (default)
  31: TLSv1 only

If LDAP_OPT_TLS_VERIFY_REQUIRED is set to ON, either the LDAP_OPT_TLS_CA_FILE or the LDAP_OPT_TLS_CA_PATH option must be set.

If client authentication is required, both LDAP_OPT_TLS_CERT_FILE and LDAP_OPT_TLS_PKEY_FILE must be set.

14.21.2. Obtaining a Key Pair

In order for TLS to authenticate a client, the client must have a private key and a certificate. Obtain these from either a Certification Authority or a self-sign program. A self-sign program is included in the Open Source Security for OpenVMS product.

14.22. Sample LDAP API Code

The following is a sample of LDAP API code.

```
#include <ldap.h>

main()
{
        LDAP            *ld;
        LDAPMessage     *res, *e;
        int             i, rc;
```

```
          char            *a, *dn;
          BerElement      *ptr;
          char            **vals;

          /* open an LDAP session */
          if ( (ld = ldap_init( "dotted.host.name", ldap_PORT )) == NULL )
                  exit( 1 );

          /* authenticate as nobody */
          if (( rc = ldap_simple_bind_s( ld, NULL, NULL )) !=
ldap_SUCCESS ) {
                  fprintf( stderr, "ldap_simple_bind_s: %s\n",
                      ldap_err2string( rc ));
                  exit( 1 );
          }

          /* search for entries with cn of "Babs Jensen", return all attrs
 */
          if (( rc = ldap_search_s( ld, "o=University of Michigan, c=US",
              ldap_SCOPE_SUBTREE, "(cn=Babs Jensen)", NULL, 0, &res ))
              != ldap_SUCCESS ) {
                  fprintf( stderr, "ldap_search_s: %s\n",
                      ldap_err2string( rc ));
                  exit( 1 );
          }

          /* step through each entry returned */
          for ( e = ldap_first_entry( ld, res ); e != NULL;
              e = ldap_next_entry( ld, e ) ) {
                  /* print its name */
                  dn = ldap_get_dn( ld, e );
                  printf( "dn: %s\n", dn );
                  ldap_memfree( dn );

                  /* print each attribute */
                  for ( a = ldap_first_attribute( ld, e, &ptr ); a !=
NULL;
                      a = ldap_next_attribute( ld, e, ptr ) ) {
                          printf( "attribute: %s\n", a );

                          /* print each value */
                          vals = ldap_get_values( ld, e, a );
                          for ( i = 0; vals[i] != NULL; i++ ) {
                                  printf( "value: %s\n", vals[i] );
                          }
                          ldap_value_free( vals );
                          ldap_memfree( a );
                  }
                  if ( ptr != NULL ) {

                          ber_free( ptr, 0 );
                  }
          }

          /* free the search results */
          ldap_msgfree( res );

          /* close and free connection resources */
```

```
        ldap_unbind( ld );
}
```

Chapter 15. LOGINOUT (LGI) Routines

The information in this chapter is intended for programmers implementing the requirements of site security administrators or third-party security software producers.

This chapter differs from other parts of this book because it does not deal strictly with callable routines that are internal to the OpenVMS system. The LOGINOUT callout routines are designed by site security administrators. The callback routines are invoked by the callout routines.

15.1. Introduction to LOGINOUT

The OpenVMS login security program (LOGINOUT.EXE) supports calls to site-specific routines (LOGINOUT callout routines). These callout routines support custom login security programs such as smart card programs, pocket authenticator programs, and other alternative identification and authentication programs. The callout routines permit sites to combine portions of the LOGINOUT security policy functions with site login security functions to establish a customized login security environment.

15.1.1. The LOGINOUT Process

The site security administrator provides LOGINOUT with the following:

- One or more shareable images comprised of modules that include callout routines
- A list of the shareable images

As login events occur, LOGINOUT invokes the applicable callout, thus enabling the site to replace or augment each event using site-specific modifications.

The site may provide multiple callout images. The images are invoked in the order in which they are declared to the system. Each image contains an independently developed set of policy routines.

Each callout routine may do one of the following:

- Enforce site-specific policy functions
- Defer to subsequent routines
- Use elements of the standard OpenVMS policy functions

Each callout routine may access LOGINOUT's internal state and callback routines using a vector of entry points. The callback routines allow the callout routines to communicate with the user and to incorporate elements of the standard OpenVMS policy functions in a modular fashion.

15.1.2. Using LOGINOUT with External Authentication

The following sections describe LOGINOUT's interaction with the external authentication policy supported by OpenVMS. For more information about single sign-on and user authentication, see the *VSI OpenVMS Guide to System Security*.

Note

The use of LOGINOUT callouts disables external authentication, making only the standard OpenVMS authentication policy available.

Overview of External Authentication

At sites using external authentication, all authentication decisions for users are actually made by the LAN manager rather than OpenVMS; however, OpenVMS account restrictions and quota checks remain in effect.

To access the system, users must provide their LAN manager user ID and password at the login prompt. If local password synchronization is required, one of the following messages is displayed indicating the outcome of the synchronization attempt:

```
OpenVMS password has been synchronized with network password

Not able to synchronize OpenVMS password with network password
```

These messages can be suppressed on a per-user basis by setting the DISREPORT flag.

Specifying Local Authentication

The login command line supports the /LOCAL_PASSWORD qualifier. This qualifier indicates to LOGINOUT that the user intends to override external authentication by using their OpenVMS user name and password. This is considered a temporary means for logging in to the system when the external authentication service is unavailable. To use this qualifier, you must have SYSPRV privilege.

When a user has logged in locally, the following message is displayed:

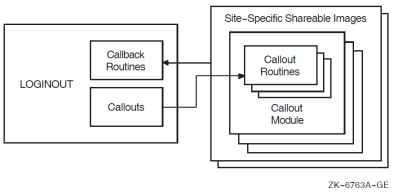
```
Local logon successful; network logon service not used
```

Locally authenticated users are not subject to OpenVMS password policy, since the system manager specified that these users are subject to external authentication policy only.

15.1.3. The LOGINOUT Data Flow

Figure 15.1 provides an overview of the data flow between LOGINOUT, the callout routines, and site-specific shareable images that can include one or more callout modules.

Figure 15.1. LOGINOUT Callout Routines Data Flow



15.2. LOGINOUT Callouts

This section introduces the callouts that LOGINOUT uses to interface with the site-specific callout modules in the shareable images. The section also describes a set of callback routines that the callout routines can use to invoke services provided within LOGINOUT.

15.2.1. LOGINOUT Callout Routines

LOGINOUT calls a different site-provided callout routine at each important step in its execution. Table 15.1 briefly describes the LOGINOUT callouts. See Section 15.4 for detailed descriptions of these routines.

Table 15.1. LOGINOUT Callouts

Callout	Description
LGI$ICR_AUTHENTICATE	Authenticates the user account at login
LGI$ICR_CHKRESTRICT	Checks additional security restrictions
LGI$ICR_DECWINIT	Prepares for interactive contact with DECwindows users
LGI$ICR_FINISH	Gives site-specific code final control of the login process
LGI$ICR_IACT_START	Prepares for interactive contact with users who are not using the DECwindows interface
LGI$ICR_IDENTIFY	Identifies the user at login
LGI$ICR_INIT	Initializes context variable
LGI$ICR_JOBSTEP	Indicates the start of each step in a batch job
LGI$ICR_LOGOUT	Prepares for logout

15.2.2. LOGINOUT Callback Routines

The callback routines enable the site's callout routines to communicate interactively with the user or to invoke other services provided by LOGINOUT. Table 15.2 briefly describes the LOGINOUT callback routines. See Section 15.5 for detailed descriptions of these routines.

Table 15.2. LOGINOUT Callback Routines

Routine	Description
LGI$ICB_ACCTEXPIRED	Checks for account expiration
LGI$ICB_AUTOLOGIN	Verifies that standard rules for autologin apply
LGI$ICB_CHECK_PASS	Checks the entered password against the user authorization file (UAF) record
LGI$ICB_DISUSER	Checks for DISUSER flag
LGI$ICB_GET_INPUT	Enables interaction with the user
LGI$ICB_GET_SYSPWD	Checks system password for character-cell interactive logins
LGI$ICB_MODALHOURS	Checks for restrictions on access modes and access hours
LGI$ICB_PASSWORD	Generates prompts, reads input, and optionally validates input against system user authorization file (SYSUAF.DAT)
LGI$ICB_PWDEXPIRED	Checks for password expiration
LGI$ICB_USERPROMPT	Prompts for and reads input for character-cell interactive logins
LGI$ICB_USERPARSE	Parses input buffer data for character-cell interactive logins
LGI$ICB_VALIDATE	Validates the user name and password against the system user authorization file (SYSUAF.DAT)

15.3. Using Callout Routines

This section describes:

- The calling environment
- The callout routines and how they are organized and activated
- The callout routines interface

Section 15.3.5 contains a sample LOGINOUT program.

15.3.1. Calling Environment

The general form for invoking the callout routines is as follows:

```
return-status = routine (standard_arguments_vector, context,
 routine_specific_args)
```

The call elements include the following:

- Standard argument vector: contains pointers to LOGINOUT data structures and callback routines for communicating with the user
- Context: a longword that the site-specific program may use to store a pointer to local context
- Routine-specific arguments: arguments directly related to the specific routine

The callout routine's return status must be one of the following:

Return Status	**Interpretation**
SS$_NORMAL	Access permitted; continue policy checks. Execute next policy image or OpenVMS policy function associated with this callout, if applicable.
LGI$_SKIPRELATED	Access permitted; discontinue checks. Continue with the login without further processing of login policy functions associated with this callout, including relevant OpenVMS policy functions built into LOGINOUT.
Other	Disallow the login: • Perform break-in detection and intrusion evasion, if appropriate. • Perform security audit. • Allow additional login attempts up to system-specified repeat limit, if appropriate.

Note

When a fatal error occurs, the policy module may terminate the login by signaling a severe error using the BLISS built-in SIGNAL_STOP or by calling LIB$SIGNAL. (See the *VSI OpenVMS RTL Library*

(LIB$) Manual for a description of the LIB$SIGNAL routine.) LOGINOUT will do a security audit, but it will not perform break-in detection or intrusion evasion.

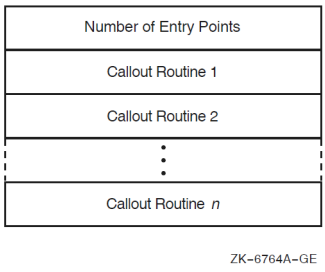
Avoid using a severe error termination unless the LOGINOUT process state is in jeopardy. LOGINOUT should terminate with a clean exit and a disallowed login whenever possible.

15.3.2. Callout Organization

A site may use several callout modules. For example, assume that the site is working with another program that uses logins or the site involves logins for various devices or logins at various security levels.

LOGINOUT invokes the callout routines using a vector of entry points rather than the routine name. Each vector entry point corresponds to a policy function, and the first vector entry contains a count of the entry points in the vector, thus making the vector extendable. Figure 15.2 shows how a callout routine vector is organized.

Figure 15.2. Callout Organization



Note that entry points may be accessed randomly. When a site-provided callout module does not provide a routine for a particular callout, the site must enter a 0 value as a placeholder into the corresponding vector location.

Callout modules may modify the vector during execution so that following events invoke different routines. For example, one of the initialization callout routines could modify the vector in anticipation of a following call to a different terminal or different job type, or it might zero the number of entry points to disable further calls to callout routines contained in the current callout module.

15.3.3. Activating the Callout Routines

A site activates the LOGINOUT callouts by identifying its callout images using the system executive-mode logical name LGI$LOGINOUT_CALLOUTS. The logical name may contain one value or a list of values that identify the callout images using either the:

- File name of a module located in SYS$SHARE:*.EXE
- Name of an executive-mode system logical name representing a full file specification

Note

LOGINOUT is installed with privileges. Therefore, any image containing LOGINOUT callout routines must be installed.

If the identifying logical is a list of several images, the images are sequentially activated in the listed order. If a specified image is not activated, the login fails.

To protect against intrusion, the site uses the system parameter LGI_CALLOUTS to specify the number of callout images. If this value is nonzero and the supplied number of callout images does not correspond to the value, the login fails.

Sites that want to control their job creation process and authenticate each network login by implementing LOGINOUT callouts must set the NET_CALLOUTS system parameter to 255. This ensures that LOGINOUT is called for every network login - bypassing any existing server processes.

The default value of NET_CALLOUTS (0) could bypass the LOGINOUT callouts and allow NET$ACP to perform its own proxy and login authentication. See the file SYS$SYSTEM:NETSERVER.COM for an example of how NET$ACP performs its own authentication and management of server processes.

Parameter values 1 to 254 are reserved by VSI for future use.

Note

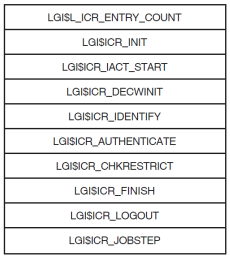
Callouts are not invoked when LOGINOUT initiates the STARTUP process during system bootstrap.

For the logical name LGI$LOGINOUT_CALLOUTS, a clusterwide logical name cannot be used. The number of names in the system logical name LGI$LOGINOUT_CALLOUTS must always match the value of the system parameter LGI_CALLOUTS. LGI$LOGINOUT_CALLOUTS must be in the regular system logical name table and not in a clusterwide logical name table.

When applications that support LGI_CALLOUTS are starting and stopping, they manipulate LGI $LOGINOUT_CALLOUTS as well as LGI_CALLOUTS. A clusterwide logical name would be incorrect since not all nodes in a cluster would have the same LGI_CALLOUTS at the same time. Nodes where the values did not match would experience login and logout failures.

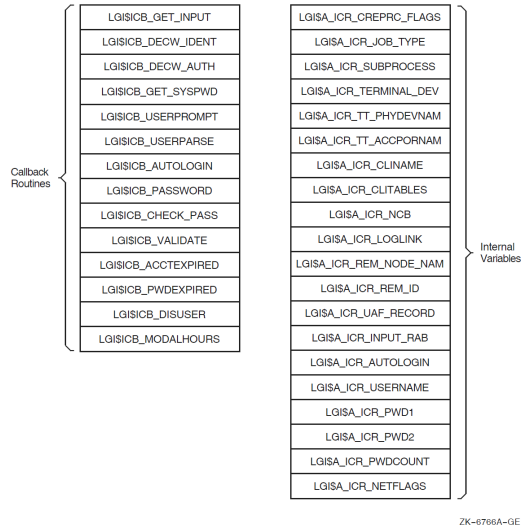
15.3.4. Callout Interface

Each image containing LOGINOUT callouts must define a universal symbol LGI $LOGINOUT_CALLOUTS. This symbol represents a vector of longwords that points to the entry points for the various callout routines, as shown in the following illustration:



The vector is headed by a longword count that delimits the number of callout routines supported by the callout module. Unused vector entries are identified by a 0 value.

Each callout routine has access to a vector of LOGINOUT internal variables, including the addresses of callback routines and other useful information. The vector entries are defined as offsets from the beginning of the vector. The vector has the following format:



Symbols of the form LGI$ICB_*x* are the addresses of the callback routines that the callout routines use to communicate with the user (see Table 15.2). Other offsets are addresses of useful variable information internal to LOGINOUT. These are described in Table 15.3.

Table 15.3. Useful LOGINOUT Internal Variables

Symbols	Definition
LGI$A_ICR_CREPRC_FLAGS	PPD_CREPRC_FLAGS controls program flow based on the major job types of PRCV_BATCH, PRCV_NETWRK, PRC$V_INTER, and other values such as PRC$V_NOPASSWORD (used for interactive jobs created on logged-in terminals).
LGI$A_ICR_JOB_TYPE	The job type from the JIB (byte). LOGINOUT does the following: • Retrieves the job type with a GETJPI during initialization. • Modifies it during execution. (Its value may change between the LGI$ICR_INIT and later callouts.) • Writes it back into the JIB before exiting. For interactive jobs, this flag indicates JIB C_LOCAL, JIBC_REMOTE, or JIB $C_DIALUP.
LGI$A_ICR_SUBPROCESS	The subprocess flag (byte) indicates whether a subprocess is being logged in.
LGI$A_ICR_TERMINAL_DEV	The terminal device flag (byte).
LGI$A_ICR_TT_PHYDEVNAM	A descriptor containing the terminal's physical device name (null if input is not from a terminal).
LGI$A_ICR_TT_ACCPORNAM	A descriptor containing the terminal's access port name (null if input is not from a terminal or is from a terminal without an associated access port).

Symbols	Definition
LGI$A_ICR_CLINAME	A descriptor containing the command language interpreter (CLI) name, parsed from the user name qualifiers. Valid only for interactive jobs.
LGI$A_ICR_CLITABLES	A descriptor containing the CLI tables, parsed from the user name qualifiers. Valid only for interactive jobs.
LGI$A_ICR_NCB	A descriptor containing the network control block. Valid only for network jobs.
LGI$A_ICR_LOGLINK	A longword containing the local link number. Valid only for network jobs and when doing a SET HOST command from a DECnet-Plus remote terminal.
LGI$A_ICR_REM_NODE_NAM	A descriptor containing the remote node name or a printable representation of its node number if the name is not available. Valid only for network jobs and when doing a SET HOST command from a DECnet-Plus remote terminal.
LGI$A_ICR_REM_ID	A descriptor containing the remote ID. This may be the user ID on the remote system if the source operating system sends the user name. Otherwise, it is as defined for the source system. Valid only for network jobs and when doing a SET HOST command from a DECnet-Plus remote terminal.
LGI$A_ICR_UAF_RECORD	Address of the LOGINOUT internal variable containing the address of the user authorization file (UAF) record. Note that because the record will be written back to the UAF record, callout routines must not modify the contents of the UAF record.
LGI$A_ICR_INPUT_RAB	A RAB (record access block) that may be used to communicate with an interactive user.
LGI$A_ICR_AUTOLOGIN	A flag (byte) indicating whether an autologin is being used for this interactive job.
LGI$A_ICR_USERNAME	A descriptor for handling the user name.
LGI$A_ICR_PWD1	A descriptor for handling the primary password.
LGI$A_ICR_PWD2	A descriptor for handling the secondary password.
LGI$A_ICR_PWDCOUNT	A longword containing the count of passwords expected for this user. Valid only for interactive jobs.
LGI$A_ICR_NETFLAGS	A flag (word) containing authorization information. Valid only for network jobs. The bits that have been defined are: • NET_PROXY: A proxy request. • NET_PREAUTH: DECnet-Plus has preauthorized the login.

Symbols	Definition
	• NET_DEFAULT_USER: The session or object database has a default user and no password checking is required. • NET_PROXY_OK: The requested proxy has been allowed by either LOGINOUT or the site-provided callout routines.

15.3.5. Sample Program

The following C program illustrates the use of LOGINOUT callouts. The sample program changes the user name and password prompts to "Who are you?" and "Prove it." The program also adds the message "Goodbye." at logout.

```
#module LGI$CALLOUT_EXAMPLE "TOY LOGINOUT callout example"
/*
**++
**  FACILITY:
**
**      System help
**

** This program can be compiled with the following command
**
**   $ CC/STANDARD=VAXC/LIST/PREFIX_LIBRARY_ENTRIES=ALL LGI
$CALLOUT_EXAMPLE.C
**
** This program can be linked with the following example command procedure
**
**    $ LINK/SHARE=LGI$CALLOUT_EXAMPLE SYS$INPUT/OPT
**    LGI$CALLOUT_EXAMPLE.OBJ

**    SYMBOL_VECTOR=(LGI$LOGINOUT_CALLOUTS=DATA)
**
** The following steps are used to install the program:
**
**    $ DEFINE/SYSTEM/EXEC LGI$LOGINOUT_CALLOUTS LGI$CALLOUT_EXAMPLE
**
** If the program is not located in SYS$SHARE, define it as follows:
**
**    $ DEFINE/SYSTEM/EXEC LGI$CALLOUT_EXAMPLE filespec
**
** [Remember that, without SYSNAM privilege, the /EXEC qualifier is
 ignored.]
**
**    $ INSTALL ADD LGI$CALLOUT_EXAMPLE
**    $ RUN SYS$SYSTEM:SYSGEN
**    SYSGEN> USE ACTIVE
**    SYSGEN> SET LGI_CALLOUTS 1
**    SYSGEN> WRITE ACTIVE
**
** The value of LGI_CALLOUTS is the number of separate callout images
** (of which this example is one) that are to be invoked.  If there is
** more than one image, the logical LGI$LOGINOUT_CALLOUTS must have a
** list of equivalence names, one for each separate callout image.
```

```
**
*/

/*
**
**  INCLUDE FILES
**
*/

#include descrip
#include rms
#include stsdef
#include ssdef
#include prcdef

/* Declare structures for the callout vector and the callout arguments
 vector */

struct LGI$CALLOUT_VECTOR  {
        long int LGI$L_ICR_ENTRY_COUNT;
        int (*LGI$ICR_INIT) ();
        int (*LGI$ICR_IACT_START) ();
        int (*LGI$ICR_DECWINIT) ();
        int (*LGI$ICR_IDENTIFY) ();
        int (*LGI$ICR_AUTHENTICATE) ();
        int (*LGI$ICR_CHKRESTRICT) ();
        int (*LGI$ICR_FINISH) ();
        int (*LGI$ICR_LOGOUT) ();
        int (*LGI$ICR_JOBSTEP) ();
        };

struct LGI$ARG_VECTOR  {
        int (*LGI$ICB_GET_INPUT) ();

        int (*reserved1) ();
        int (*reserved2) ();
        void (*LGI$ICB_GET_SYSPWD) ();
        int (*LGI$ICB_USERPROMPT) ();
        int (*LGI$ICB_USERPARSE) ();
        int (*LGI$ICB_AUTOLOGIN) ();
        int (*LGI$ICB_PASSWORD) ();
        int (*LGI$ICB_CHECK_PASS) ();
        int (*LGI$ICB_VALIDATE) ();
        void (*LGI$ICB_ACCTEXPIRED) ();
        void (*LGI$ICB_PWDEXPIRED) ();
        int (*LGI$ICB_DISUSER) ();
        void (*LGI$ICB_MODALHOURS) ();
        short *LGI$A_ICR_CREPRC_FLAGS;
        char *LGI$A_ICR_JOB_TYPE;
        char *LGI$A_ICR_SUBPROCESS;
        char *LGI$A_ICR_TERMINAL_DEV;
        struct dsc$descriptor_s *LGI$A_ICR_TT_PHYDEVNAM;
        struct dsc$descriptor_s *LGI$A_ICR_TT_ACCPORNAM;
        struct dsc$descriptor_s *LGI$A_ICR_CLINAME;
        struct dsc$descriptor_s *LGI$A_ICR_CLITABLES;
        struct dsc$descriptor_s *LGI$A_ICR_NCB;
        int *LGI$A_ICR_LOGLINK;
        struct dsc$descriptor_s *LGI$A_ICR_REM_NODE_NAM;
```

```
        struct dsc$descriptor_s *LGI$A_ICR_REM_ID;
        unsigned char *LGI$A_ICR_UAF_RECORD;
        struct RAB *LGI$A_ICR_INPUT_RAB;
        char *LGI$A_ICR_AUTOLOGIN;
        struct dsc$descriptor_s *LGI$A_ICR_USERNAME;
        struct dsc$descriptor_s *LGI$A_ICR_PWD1;
        struct dsc$descriptor_s *LGI$A_ICR_PWD2;
        int *LGI$A_ICR_PWDCOUNT;
        short int *LGI$A_ICR_NETFLAGS;
        };

globalvalue int LGI$_SKIPRELATED,       /* callout's return status */
                LGI$_DISUSER,
                LGI$_INVPWD,
                LGI$_NOSUCHUSER,
                LGI$_NOTVALID,
                LGI$_INVINPUT,
                LGI$_CMDINPUT,
                LGI$_FILEACC;


static int callout_logout();
static int callout_decwinit();
static int callout_identify();
static int callout_authenticate();

globaldef struct LGI$CALLOUT_VECTOR LGI$LOGINOUT_CALLOUTS =
                {
                9,
                0,                              /* init */
                0,                              /* iact_start */
                callout_decwinit,               /* decwinit */
                callout_identify,               /* identify */
                callout_authenticate,           /* authenticate */
                0,                              /* chkrestrict */
                0,                              /* finish */
                callout_logout,                 /* logout */
                0,                              /* jobstep */
                };

/* DECwindows initialization */

static int callout_decwinit()
    {
        /* Disable any further calls */
        LGI$LOGINOUT_CALLOUTS.LGI$L_ICR_ENTRY_COUNT = 0;
        /* Return and do standard DECwindows processing */
        return (SS$_NORMAL);
    }

/* Identification */

static int callout_identify(struct LGI$ARG_VECTOR *arg_vector)
    {

    int status;
    $DESCRIPTOR(wru,"\r\nWho are you? ");
```

```
    /* This example deals only with interactive jobs */
    if (!(*arg_vector->LGI$A_ICR_CREPRC_FLAGS & PRC$M_INTER))
        return(SS$_NORMAL);  /* Not interactive, do normal processing */
    if (*arg_vector->LGI$A_ICR_CREPRC_FLAGS & PRC$M_NOPASSWORD)
        return(SS$_NORMAL);  /* Invoked as logged in, don't prompt */
    if (*arg_vector->LGI$A_ICR_SUBPROCESS != 0)
        return(SS$_NORMAL);  /* Don't prompt on subprocesses */

    /* Check for autologin */

    if ($VMS_STATUS_SUCCESS(arg_vector->LGI$ICB_AUTOLOGIN()))
        return (LGI$_SKIPRELATED);       /* Yes, it's an autologin */

    if (!$VMS_STATUS_SUCCESS(status = arg_vector->LGI
$ICB_USERPROMPT(&wru)))
        return (status); /* On error, return error status */

    /* Successful prompt and parse; skip OpenVMS policy */

    return(LGI$_SKIPRELATED);
    }

/* Authentication */

static int callout_authenticate(struct LGI$ARG_VECTOR *arg_vector)
    {

    int status;
    $DESCRIPTOR(proveit,"\r\nProve it: ");

    /* This example deals only with interactive jobs */
    if (!(*arg_vector->LGI$A_ICR_CREPRC_FLAGS & PRC$M_INTER))
        return(SS$_NORMAL);  /* Not interactive, do normal processing */
    if (*arg_vector->LGI$A_ICR_CREPRC_FLAGS & PRC$M_NOPASSWORD)
        return(SS$_NORMAL);  /* Invoked as logged in, don't prompt */
    if (*arg_vector->LGI$A_ICR_SUBPROCESS != 0)
        return(SS$_NORMAL);  /* Don't prompt on subprocesses */

    if (*arg_vector->LGI$A_ICR_PWDCOUNT != 0)
        /* This account has at least one password */
        if (!$VMS_STATUS_SUCCESS(status =
                        arg_vector->LGI$ICB_PASSWORD(0,&proveit)))
        return (status); /* On error, return error status */

    if (*arg_vector->LGI$A_ICR_PWDCOUNT == 2)
        /* This account has two passwords */
        if (!$VMS_STATUS_SUCCESS(status =
                        arg_vector->LGI$ICB_PASSWORD(1,&proveit)))
        return (status); /* On error, return error status */

    /* Successful prompt and password validation; skip OpenVMS policy */

    return(LGI$_SKIPRELATED);
    }

/* LOGOUT command */

static int callout_logout(username, procname,  creprc_flags, write_fao)
```

```
    struct dsc$descriptor_s *username, *procname;
    short *creprc_flags;
    void (*write_fao) ();
    {
        char *Goodbye = "   Goodbye.";          /* This will become an
ASCIC */
        if ((int) write_fao != 0)               /* If output is
permitted... */
        {
            Goodbye[0]=strlen(Goodbye)-1;       /* Fill in ASCIC count */
            write_fao(Goodbye);                 /* and write it */
        }
        return(SS$_NORMAL);
    }
```

15.4. LOGINOUT Callout Routines

The following sections describe the individual callout routines. Each description includes the following:

- The format of the call command
- The anticipated information returned by the called routine
- The arguments presented to the called routine
- A general description of the routine
- Typical condition values that indicate the return status
- Associated OpenVMS policy function, that is, the standard LOGINOUT policy functions developed for OpenVMS compared with the site-provided policy functions

The Typical Condition Values and the Associated OpenVMS Policy Function headings are unique to the LOGINOUT callout routines.

LGI$ICR_AUTHENTICATE

LGI$ICR_AUTHENTICATE — The LGI$ICR_AUTHENTICATE callout routine authenticates passwords.

Format

LGI$ICR_AUTHENTICATE arg_vector, context

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Arguments

arg_vector

OpenVMS usage:	**vector**
type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing callbacks and login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

Description

All logins involving a password invoke the LGI$ICR_AUTHENTICATE callout routine. The routine is not called for subprocesses, network jobs invoked by proxy logins, or logged-in DECterm sessions.

The following pointers are used in password authentication:

- Longword LGI$A_ICR_PWDCOUNT points to a location that contains the number of OpenVMS passwords for a particular account. Nonexistent accounts are assigned a password count of 1 to avoid revealing them by the absence of a password prompt.
- For DECwindows logins only, longword LGI$A_ICR_PWD1 points to a location that contains the user's primary password.
- For DECwindows logins only, longword LGI$A_ICR_PWD2 points to a location that contains the user's secondary password, if applicable.

For all logins except DECwindows logins, the LGI$ICR_AUTHENTICATE callout routine may use the following callback routine sequence:

- Call LGI$ICB_PASSWORD for standard password prompting with an optional nonstandard prompt and the option of checking or just returning the password or other information obtained.
- Call LGI$ICB_GET_INPUT for completely customized prompting for each required piece of authentication information.

For DECwindows logins, neither the LGI$ICB_PASSWORD callback routine nor the LGI $ICB_GET_INPUT callback routine needs to be called. The user enters the password using the DECwindows login dialog box *before* LOGINOUT issues the LGI$ICR_AUTHENTICATE callout.

For a complete description of the DECwindows flow of control, see the description of the LGI $ICR_DECWINIT callout routine.

All logins involving a password may invoke the LGI$ICB_VALIDATE callback routine. This routine validates against SYSUAF.DAT passwords obtained by customized prompting using descriptors for the user name and passwords. Optionally, the login may call the LGI$_ICB_CHECK_PASS callback routine to validate passwords. For interactive jobs, the LGI$ICR_AUTHENTICATE routine should check the DISUSER flag using the LGI$ICB_DISUSER callback routine to preserve the consistency of the invalid user behavior for disabled accounts. For other types of jobs, use the LGI$ICR_CHKRESTRICT callout routine to check the DISUSER flag.

Note

LOGINOUT checks the DISUSER flag as part of the authentication process because, if it is checked later, an intruder could determine that the correct user name and password had been entered and that the account is disabled. This is *deliberately* hidden by keeping the user in the retry loop for a disabled account.

If the DISUSER flag is checked with other access restrictions in the authorization portion, this causes an immediate exit from LOGINOUT.

Break-in detection, intrusion evasion, and security auditing are done in the case of any failure return from LGI$ICR_AUTHENTICATE.

If this routine returns LGI$_SKIPRELATED, the user is fully authenticated, and no further authentication is done by either the site or OpenVMS. If this routine returns an error for an interactive job, the system retries the identification and authentication portions of LOGINOUT. For character-cell terminals, this consists of calling the LGI$ICR_IDENTIFY and LGI$ICR_AUTHENTICATE callout routines; for DECwindows terminals, this consists of calling the LGI$ICR_DECWINIT routine. The number of retries is specified by the SYSGEN parameter LGI_RETRY_LIM.

Typical Condition Values

SS$_NORMAL

Access permitted; continue policy checks.

LGI$_SKIPRELATED

Access permitted; omit calls to the LGI$ICR_AUTHENTICATE callout routine in subsequent images and calls to the associated OpenVMS policy function.

Other

Disallow the login; perform break-in detection, intrusion evasion, and security auditing. For interactive logins, retry identification and authentication portions of LOGINOUT, up to the number specified in the SYSGEN parameter LGI_RETRY_LIM.

Associated OpenVMS Policy Function

1

Perform standard password prompting and validation.

LGI$ICR_CHKRESTRICT

LGI$ICR_CHKRESTRICT — The LGI$ICR_CHKRESTRICT callout routine may be used to check site-specific access restrictions that are not usually included in the OpenVMS login.

Format

LGI$ICR_CHKRESTRICT arg_vector , context

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Argument

arg_vector

OpenVMS usage:	**vector**
type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing callbacks and login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

Description

All logins call this routine after the password is authenticated to allow the site to check other access restrictions. The site may check its own access restrictions and any of the following OpenVMS access restrictions:

Access Restriction	Callback Routine Used to Check Restriction
Account expiration	LGI$ICB_ACCTEXPIRED
Password expiration	LGI$ICB_PWDEXPIRED
Account disabled	LGI$ICB_DISUSER
Access modes and times	LGI$ICB_MODALHOURS

Typical Condition Values

SS$_NORMAL

Access permitted; continue policy checks, including all of the normal OpenVMS policy functions associated with the callback routines used to check restrictions.

LGI$_SKIPRELATED

Access permitted; omit calls to the LGI$ICR_CHKRESTRICT callout routine in subsequent images and calls to the associated OpenVMS policy functions.

Other

Disallow the login.

Associated OpenVMS Policy Functions

1

Check password expiration, check DISUSER flag, check account expiration, and check restrictions on access time.

LGI$ICR_DECWINIT

LGI$ICR_DECWINIT — The LGI$ICR_DECWINIT callout routine enables site-specific initialization functions for logins from the DECwindows session manager.

Format

`LGI$ICR_DECWINIT` arg_vector , context

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Argument

arg_vector

OpenVMS usage:	**vector**
type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing site-specified callbacks and login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

Description

LOGINOUT invokes the LGI$ICR_DECWINIT callout routine at the start of a DECwindows session login. This callout routine does not support a return status of LGI$_SKIPRELATED. Returning LGI $_SKIPRELATED for this callout causes unpredictable results. Use the LGI$ICR_DECWINIT callout routine only to prepare other callout routines for a DECwindows login.

After issuing the LGI$ICR_DECWINIT callout, LOGINOUT performs the following tasks:

- Creates the DECwindows login dialog box and reads the user name and password entered by the user
- Calls the LGI$ICR_IDENTIFY callout
- Obtains the user authorization file (UAF) record

  If the UAF record specifies two passwords, the DECwindows login dialog box is amended to prompt for the second password, and the listed tasks are repeated.
- Issues the LGI$ICR_AUTHENTICATE callout
- If the LGI$ICR_AUTHENTICATE callout routine did not return LGI$_SKIPRELATED, validates the passwords against the UAF record

The LGI$ICR_IDENTIFY and LGI$ICR_AUTHENTICATE callouts may create additional DECwindows dialog boxes to communicate with the user, but the initial dialog box must be created by LOGINOUT.

Typical Condition Values

SS$_NORMAL

Access permitted; continue policy checks.

LGI$_SKIPRELATED

Not supported. Returning this status will cause unpredictable behavior.

Other

Disallow the login.

Associated OpenVMS Policy Function

1

Create dialog box, read user name and password, and call the identification and authentication routines.

LGI$ICR_FINISH

LGI$ICR_FINISH — The LGI$ICR_FINISH callout routine permits the site program to take final local action before exiting from LOGINOUT.

Format

LGI$ICR_FINISH arg_vector , context , user_cond_value

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Argument

arg_vector

OpenVMS usage:	**vector**
type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing callbacks and login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

user_cond_value

OpenVMS usage:	**cond_value**
type:	**longword_unsigned**
access:	**read only**
mechanism:	**by value**

SS$_NORMAL for successful login; otherwise, reason for failure.

Description

The site program calls this routine immediately before exiting to take any final local actions relative to the login process. There is no OpenVMS login security policy associated with LGI$ICR_FINISH.

LGI$ICR_FINISH does not affect login completions because the login is audited before the routine is invoked. The routine has no effect on error recovery when a login fails, and it cannot cause a successful login to fail.

Typical site action may include the following:

- Override job quotas
- Stack CLI command procedures by examining and modifying the logicals PROC1 through PROC9

> **Caution**
>
> For DECwindows session manager logins, be careful modifying the command procedure stack to avoid adversely affecting the command file that invokes the session manager.

- Other postlogin processing

Typical Condition Values

LGI$_SKIPRELATED

Access permitted; omit calls to the LGI$ICR_FINISH callout routine in subsequent images.

Associated OpenVMS Policy Functions

None.

LGI$ICR_IACT_START

LGI$ICR_IACT_START — The LGI$ICR_IACT_START callout routine may perform initialization functions for logins from interactive character-cell terminals.

Format

LGI$ICR_IACT_START arg_vector , context

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Argument

arg_vector

OpenVMS usage:	**vector**
type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing callbacks and login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

Description

This routine makes the first contact for all interactive logins from other than DECwindows terminals after opening the input and output files but before any other dialogue with the user.

At this point, the site should be preparing to augment or replace the OpenVMS system password routine. The callback routine LGI$ICB_GET_SYSPWD provides access to the system password routine. However, because LGI$ICB_GET_SYSPWD returns only on success, the site design should consider what action to take in case LGI$ICB_GET_SYSPWD does not return control to LGI $ICR_IACT_START.

The LGI$ICR_IACT_START routine can use the LGI$ICB_GET_INPUT callback routine to:

- Get input from the user
- Use an OpenVMS RMS record access block (RAB) to establish appropriate terminal mode settings

Typical Condition Values

SS$_NORMAL

Access permitted; continue OpenVMS system password routine.

LGI$_SKIPRELATED

Access permitted; omit calls to the LGI$ICR_IACT_START callout routine in subsequent images and calls to the associated OpenVMS policy function.

Other

Exit quietly to preserve the illusion of an inactive line.

Associated OpenVMS Policy Function

Get the system password.

LGI$ICR_IDENTIFY

LGI$ICR_IDENTIFY — The LGI$ICR_IDENTIFY callout routine identifies the user from the user name input.

Format

LGI$ICR_IDENTIFY arg_vector , context

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Argument

arg_vector

OpenVMS usage:	**vector**
type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing callbacks and useful login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

Description

The LGI$ICR_IDENTIFY callout routine is invoked for all types of login procedures. If the site uses the standard OpenVMS DECwindows dialogue, the identification routine may be called more than once for accounts with two passwords.

If you plan to replace the standard OpenVMS identification processing, consider the following:

- For logins from character-cell terminals, obtain the user name using one of the following:
  - A dialogue with the user. The site can access OpenVMS user name processing to obtain the standard prompt or a specialized prompt by invoking the LGI$ICB_USERPROMPT callback routine. Alternatively, the site may invoke the LGI$ICB_GET_INPUT callback routine to communicate with the user.
  - Site-specific equipment, for example, a card reader or some other authentication device.
  - Autologins. The site may do the identification portion of the standard OpenVMS autologin by invoking the LGI$ICB_AUTOLOGIN callback routine.
- For logins from the DECwindows Session Manager, LOGINOUT invokes the callout module's LGI$ICR_IDENTIFY callout routine after obtaining the user name and putting it in LGI

$A_ICR_USERNAME. The LGI$ICR_IDENTIFY callout routine can provide any additional checking of the user name that may be required.

- For batch jobs, network jobs, logged-in DECterm sessions, and subprocesses, the site may use the LGI$ICR_IDENTIFY routine to verify information without a user dialogue.

Calls to LGI$ICR_IDENTIFY are always followed by validation of the presence of the user name in the system authorization file, unless the routine is invoked for a subprocess.

Typical Condition Values

SS$_NORMAL

Access permitted; continue policy checks.

LGI$_SKIPRELATED

Access permitted; omit calls to the LGI$ICR_IDENTIFY callout routine in subsequent images and calls to the associated OpenVMS policy function.

Other

Disallow the login.

Associated OpenVMS Policy Function

1

Perform standard OpenVMS user name prompting and parsing.

LGI$ICR_INIT

LGI$ICR_INIT — The LGI$ICR_INIT callout routine may perform any required initialization functions.

Format

`LGI$ICR_INIT` arg_vector , context

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns status indicating whether and how to proceed with the login.

Argument

arg_vector

OpenVMS usage:	**vector**

type:	**vector_longword_unsigned**
access:	**modify**
mechanism:	**by reference**

Vector containing callbacks and login information.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

Description

This routine is called for all job types before opening input and output files. If desired, the callout routine may initialize the **context** argument, which LOGINOUT subsequently passes to each callout routine with the address of local storage specific to the callout image.

Typical Condition Values

SS$_NORMAL

Access permitted; continue policy checks.

LGI$_SKIPRELATED

Access permitted; omit calls to the LGI$ICR_INIT callout routine in subsequent images.

Other

Disallow the login.

Associated OpenVMS Policy Functions

None.

LGI$ICR_JOBSTEP

LGI$ICR_JOBSTEP — The LGI$ICR_JOBSTEP callout routine signals the start of each batch job step.

Format

LGI$ICR_JOBSTEP input_file_name , context , [write_fao]

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**

access:	**write only**
mechanism:	**by value**

Not applicable.

Argument

input_file_name

OpenVMS usage:	**descriptor**
type:	**character string**
access:	**read**
mechanism:	**by reference**

The name of the input file.

context

OpenVMS usage:	**context**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Pointer to site's local context.

write_fao (fao_string[,arg1[,arg2][,...]])

OpenVMS usage:	**routine**
type:	**procedure**
access:	**read**
mechanism:	**by reference**

Address of a routine that may be called to format and display output. The routine has **fao_string** as its first argument, followed by a variable number of arguments. (See the $FAO system directive in the *VSI OpenVMS System Services Reference Manual* for more information.)

Description

The LGI$ICR_JOBSTEP routine alerts the site of each job step in a batch job. The routine is invoked as LOGINOUT processes each job step. For the first job step, the LGI$ICR_JOBSTEP callout routine is invoked immediately following the LGI$ICR_IDENTIFY callout routine. For all other job steps, it is the only callout routine that is invoked.

The routine is provided with the input file name, but the input file is not open when the routine is called. For the first job step, the LGI$ICR_INIT callout routine may provide the batch job step routine with context. For other job steps, the **context** argument is a null.

For all job steps except the first, the output file is open, and the routine specified by the **write_fao** argument is available.

There is no OpenVMS policy associated with LGI$ICR_JOBSTEP.

Typical Condition Values

LGI$_SKIPRELATED or any error value

Access permitted; omit calls to the LGI$ICR_JOBSTEP callout routine in subsequent images.

Associated OpenVMS Policy Functions

None.

LGI$ICR_LOGOUT

LGI$ICR_LOGOUT — The LGI$ICR_LOGOUT callout routine permits the site callout images to respond to the DCL command LOGOUT. This routine is not called if the calling process is deleted with STOP/PROCESS ($DELPRC). If the calling terminal is disconnected when logout occurs, this routine must not produce output.

Format

`LGI$ICR_LOGOUT` username , processname , creprc_flags , write_fao

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Returns logout status from the site program.

Argument

username

OpenVMS usage:	**descriptor**
type:	**character string**
access:	**read**
mechanism:	**by reference**

User name.

processname

OpenVMS usage:	**descriptor**
type:	**character string**
access:	**read**
mechanism:	**by reference**

Process name.

creprc_flags

OpenVMS usage:	**mask_longword**
type:	**longword_unsigned**
access:	**read**
mechanism:	**by reference**

Process creation status flags.

write_fao (fao_string[,arg1[,arg2][,...]])

OpenVMS usage:	**routine**
type:	**procedure**
access:	**read**
mechanism:	**by reference**

Procedure for writing data. The value is 0 if output is not permitted.

Address of a routine that may be called to format and display output. The routine has **fao_string** as its first argument, followed by a variable number of arguments. (See the $FAO system directive in the *VSI OpenVMS System Services Reference Manual* for more information.)

Description

The LGI$ICR_LOGOUT routine is invoked after auditing is completed and immediately before LOGOUT prints the logout message. This routine cannot prevent the logout from finishing, but it may prevent display of the standard logout message.

Typical Condition Values

LGI$_SKIPRELATED or any error value

Access permitted; omit calls to the LGI$ICR_LOGOUT callout routine in subsequent images.

Associated OpenVMS Policy Functions

None.

15.5. LOGINOUT Callback Routines

LOGINOUT callout routines use callback routines to interact with the user or to access other LOGINOUT services. This section describes the individual callback routines. The description of each routine includes the following:

- The format of the call command
- The anticipated information returned by the called routine
- The arguments presented to the called routine
- A general description of the routine
- Condition values that indicate the return status of the routine, success or failure

LGI$ICB_ACCTEXPIRED

LGI$ICB_ACCTEXPIRED — The LGI$ICB_ACCTEXPIRED callback routine checks for account expiration.

Format

LGI$ICB_ACCTEXPIRED

Returns

No value. Does not return on failure.

Argument

None.

Description

The site can use this callback routine to determine if the specified account is expired. If the account is expired, the LGI$ICB_ACCTEXPIRED callback routine:

- Writes its standard error message to the user terminal, if a terminal exists
- Does not return control to the caller

Condition Values Returned

None.

LGI$ICB_AUTOLOGIN

LGI$ICB_AUTOLOGIN — The site may use the LGI$ICB_AUTOLOGIN callback routine to determine whether the standard OpenVMS autologin functionality applies for this terminal.

Format

LGI$ICB_AUTOLOGIN

Returns

OpenVMS usage:	**value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

True (logical 1) if autologin enabled; 0 otherwise.

Argument

None.

Description

If the standard OpenVMS autologin functionality applies, the callback routine returns the user name to the site program using the standard argument vector so that the autologin process may continue.

The autologin determination is made *before* the site prompts for the user passwords. The callback routine is applicable only for interactive character-cell logins.

Note

Standard OpenVMS policy uses autologin only on directly connected or LAT connected character-cell terminals. The LGI$ICB_AUTOLOGIN callback routine checks the automatic login file (ALF) SYS $SYSTEM:SYSALF.DAT to make the determination.

A DECwindows callout can include a method for doing a DECwindows autologin. In that case, the callout routine should set the autologin flag to true before returning control to LOGINOUT.

Condition Values Returned

None.

LGI$ICB_CHECK_PASS

LGI$ICB_CHECK_PASS — The LGI$ICB_CHECK_PASS callback routine checks a password against the user authorization file (UAF) record.

Format

LGI$ICB_CHECK_PASS password , uaf_record , pwd_number

Returns

OpenVMS usage:	**value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

The value 1 for a valid password. The value --4 for an invalid password.

Argument

password

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

User-supplied password to be validated.

uaf_record

OpenVMS usage:	**buffer**
type:	**vector_byte (unsigned)**
access:	**read only**
mechanism:	**by reference**

Address of buffer containing UAF record.

pwd_number

OpenVMS usage:	**value**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Password number, 0 (primary) or 1 (secondary).

Description

The site uses this callback routine to check the user-supplied password against the UAF record provided as the second argument. If the password is valid, the routine returns a 1 in R0; if the password is invalid, the routine returns a --4 in R0.

Condition Values Returned

None.

LGI$ICB_DISUSER

LGI$ICB_DISUSER — The LGI$ICB_DISUSER callback routine checks the disabled user account flag.

Format

`LGI$ICB_DISUSER` action

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Condition value in R0.

Argument

action

OpenVMS usage:	**value**

type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

This argument can take two values:

If Value of Action Is...	Then...
LGI$_DISUSER_STOP	Do not return on error.
LGI$_DISUSER_RETURN	Return LGI$_DISUSER or SS$_NORMAL.

Description

The site can use this callback routine to establish the standard OpenVMS action if the DISUSER flag is set.

Condition Values Returned

LGI$_DISUSER

SS$_NORMAL

LGI$ICB_GET_INPUT

LGI$ICB_GET_INPUT — The LGI$ICB_GET_INPUT callback routine enables interaction with the user.

Format

`LGI$ICB_GET_INPUT` rab , flags

Returns

No value. Does not return on failure.

Argument

rab

OpenVMS usage:	**rab**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Data structure used to set up a read-with-prompt OpenVMS RMS operation. Normally you pass the RAB address in LGI$A_ICR_INPUT_RAB.

flags

OpenVMS usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A data structure that determines the error response as follows:

Flags Value	Response
0	Normal error message.
1	LOGINOUT exits quietly.
2	Normal error message; however, the callback routine returns control to the caller rather than exiting on timeout (timeout status is in RAB).

Description

The LGI$ICB_GET_INPUT callback routine invokes the LOGINOUT input routine to enable interaction with character-cell terminal users. The read operation provides a timeout to ensure that the UAF record does not remain locked if the user presses Ctrl/S.

Condition Values Returned

1

No return value. Examine status in RAB to determine the results of the read operation.

LGI$ICB_GET_SYSPWD

LGI$ICB_GET_SYSPWD — The LGI$ICB_GET_SYSPWD callback routine validates the system password.

Format

`LGI$ICB_GET_SYSPWD`

Returns

No value. Does not return on failure.

Argument

None.

Description

This callback routine performs standard system password-checking for interactive logins on character-cell terminals only.

If the system password is validated, this callback routine returns control to the caller. If the system password is not validated, the LOGINOUT image exits, and the login is terminated.

Condition Values Returned

None.

LGI$ICB_MODALHOURS

LGI$ICB_MODALHOURS — The LGI$ICB_MODALHOURS callback routine checks for restrictions on access modes and access hours.

Format

LGI$ICB_MODALHOURS

Returns

No value. Does not return on failure.

Argument

None.

Description

The site uses this callback routine to establish the access modes and access hours available to the user. If the user is not authorized to access the system from this login class (batch, dialup, local, remote, network) at this time (as specified in the UAF), the callback routine:

- Writes its standard error message to the user terminal, if there is a terminal
- Does not return control to the caller

Condition Values Returned

None.

LGI$ICB_PASSWORD

LGI$ICB_PASSWORD — The LGI$ICB_PASSWORD callback routine produces the specified password prompt and then processes the input.

Format

LGI$ICB_PASSWORD password_number , prompt , buffer

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Condition value in R0.

Argument

password_number

OpenVMS usage:	**value**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

A numeric value indicating which password to prompt for and what action to take on it:

Value	Prompt for
0	Primary password and validate it
1	Secondary password and validate it
--1	Primary password but do not validate it
--2	Secondary password but do not validate it
--3	Arbitrary 32-character value returned to buffer specified in **buffer**

If the value is --3, you must specify the **prompt** argument and the **buffer** argument.

prompt

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

String that must begin with "cr,lf". If this argument is not supplied, the standard prompt is used.

buffer

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**modify**
mechanism:	**by reference**

Buffer having at least 32 bytes available to store password when **password_number** argument value is --3.

Description

The site can use this callback routine to interactively prompt for passwords. The routine uses either the standard OpenVMS password prompt or a prompt provided by the caller in the second argument.

The password is returned in one of the following locations, depending on the value of the **password_number** argument:

Value of Password_Number Argument	Location
0 or --1	LGI$A_ICR_PWD1

Value of Password_Number Argument	Location
1 or --2	LGI$A_ICR_PWD2
--3	**buffer** argument

Note

This routine will do overstriking, if necessary, to support echo local terminals. See the *VSI OpenVMS Programming Concepts Manual* for more information about echo terminals.

Condition Values Returned

SS$_NORMAL

Success.

LGI$_INVPWD

Password check failed.

LGI$_NOSUCHUSER

No UAF record found.

LGI$ICB_PWDEXPIRED

LGI$ICB_PWDEXPIRED — The LGI$ICB_PWDEXPIRED callback routine checks for password expiration.

Format

`LGI$ICB_PWDEXPIRED`

Returns

No value. Does not return on failure.

Argument

None.

Description

Use this callback routine to determine whether the account password has expired. If the password is expired, the callback routine:

- Writes its standard error message to the user terminal, if there is a terminal
- Does not return control to the caller

Condition Values Returned

None.

LGI$ICB_USERPARSE

LGI$ICB_USERPARSE — The LGI$ICB_USERPARSE callback routine parses the user name input.

Format

LGI$ICB_USERPARSE input_buffer

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Condition value in R0.

Argument

input_buffer

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

The input buffer must contain the characters LOGIN in the first five character locations, followed by an ASCII space character and then the user name and applicable site-specified qualifiers.

Description

The site can use this callback routine to parse input for interactive logins on character-cell and DECwindows terminals.

Upon completion of this routine, the user name is accessible at the LGI$A_USERNAME entry in the standard arguments vector.

Condition Values Returned

1

True (1) if successful; otherwise, any condition code returned by CLI$PARSE.

LGI$ICB_USERPROMPT

LGI$ICB_USERPROMPT — The LGI$ICB_USERPROMPT callback routine prompts for the user name.

Format

LGI$ICB_USERPROMPT prompt

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Condition value in R0.

Argument

prompt

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

A string that must begin with "cr,lf". For example, to produce the standard user name prompt, use your language equivalent of the following BLISS value:

```
UPLIT(12,UPLIT BYTE(CR,LF,'Username: '))
```

Declare the string in C using the following statement:

```
$DESCRIPTOR(<variable_name>, "lrlnUsername:")
```

You then pass the descriptor using the variable name.

This routine also produces the standard user name prompt if you pass the value 0 for this argument.

Description

Use this callback routine to interactively prompt for the user name on a character-cell terminal. The callback routine reads the response to the prompt and does standard DCL parsing for the user name and any qualifiers provided. Upon completion of this routine, the user name is accessible at the LGI $A_USERNAME entry in the standard arguments vector.

Condition Values Returned

SS$_NORMAL

Success.

LGI$_NOTVALID

Retry count exceeded for user input.

LGI$ICB_VALIDATE

LGI$ICB_VALIDATE — The LGI$ICB_VALIDATE callback routine validates the user name and passwords against the system authorization file.

Format

LGI$ICB_VALIDATE username , pwd1 , pwd2

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Condition value in R0.

Argument

username

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

User name.

pwd1

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

Primary password.

pwd2

OpenVMS usage:	**character string**
type:	**string descriptor**
access:	**read only**
mechanism:	**by reference**

Secondary password.

Description

The site can use this callback routine to validate the user name and the user's primary and secondary passwords against the system authorization file (SYSUAF.DAT). The routine also:

- Updates the user authorization (UAF) record with information about login failures
- Performs security auditing

- Performs break-in detection and intrusion evasion

Condition Values Returned

1

Success, or an error indicating the reason for the failure.

Chapter 16. Mail Utility Routines

The callable interface of the Mail utility (MAIL) lets you send messages to users on your system or on any other computer connected to your system with DECnet. This chapter describes how application programs using callable MAIL routines can perform the following functions:

- Create and access mail files
- Access and manipulate a message or group of messages
- Create and send messages to a user or group of users
- Access and manipulate the user profile database

For information about the DCL interface to the Mail utility, see the *VSI OpenVMS User's Manual*.

16.1. Messages

Messages are files that contain information you want to send to other users. Messages having one or two blocks are part of a mail file, while messages having more than two blocks are external sequential files.

External files reside in the same directory as the mail file that points to them.

Structure of a Message

A message consists of **header** information and the **bodypart**. The message bodypart consists of text records that contain information you want to send to another user.

Example 16.1 illustrates the format of a mail message.

Example 16.1. Standard Message Format

```
From:   MYNODE::USER  "The Celestial Navigator"    ❶
To:     NODE::J_DOE          ❷
CC:     USER                 ❸
Subj:   Perseids ...         ❹

Get ready. Tuesday of this week (August 12th), one    ❺
of the most abundant meteor showers of the year will occur.
The Perseids, also known as the St. Laurence's Tears, stream
across earth's orbit at 319.3 degrees. Radiant 3h4m +58 degrees.
Fine for photography with an average magnitude of 2.27.
There will be some fireballs, fainter white or yellow
meteors, and brighter  green or orange or red ones. About one
third of the meteors, including all the  brightest, leave
yellowish trains, which may be spectacular, up to 2
degrees wide and lasting up to 100 seconds. Brighter
meteors often end in flares or bursts.           ❻
```

The parts of a message are as follows:

- Header information
  - ❶ *From:* field specifies the sender and an optional personal name string

❷ *To:* field specifies the direct addressee
❸ *CC:* field specifies the carbon copy addressee
❹ *Subj:* field specifies the topic of the message

- Bodypart

❺ First line of the bodypart
❻ Last line of the bodypart

External Message Identification Number

In addition, the file name of an external message uses the following format:

```
MAIL$nnnnnnnnnnnnnnn.MAI
```

where *n …n* is the external message identification number.

16.2. Folders

The Mail utility organizes messages by date and time received and, secondarily, by folder name. All messages are associated with a folder name—either default folders or user-specified folders. The Mail utility associates mail messages with one of three default mail folder names. Table 16.1 describes the three default mail folders.

Table 16.1. Default Mail Folders

Folder	Contents
NEWMAIL	Newly received, unread messages
MAIL	Messages that have been read and not deleted
WASTEBASKET	Messages designated for deletion

You can also place messages in any user-defined mail folder and file.

16.3. Mail Files

A mail file is an indexed file that contains the following types of data:

- Header information for all messages
- Text of short messages
- Pointers to long messages

In addition, you can select messages from mail files as well as copy or move messages to or from mail files.

Mail File Format

The indexed mail file format offers two advantages: use of folders and faster access time than sequential access. Indexed mail files use two keys to locate messages—a **primary** key denoting the date and time received and a **secondary** key using the folder name.

16.4. User Profile Database

The Mail utility maintains an indexed data file VMSMAIL_PROFILE.DATA that serves as a systemwide database of **user profile** entries. A user profile entry is a record that contains data describing a Mail user's default processing characteristics and whose primary key is the user name. Table 16.2 summarizes information contained in a user profile entry.

Table 16.2. User Profile Information

Field	Function
Directory	Default MAIL subdirectory
Form	Default print form
Forwarding address	Forwarding address
Personal name string	User-specified character string included in the message header
Queue name	Default print queue name
Flags Automatic purge *CC:* prompt Copy self forward Copy self reply Copy self send	Purging of the wastebasket folder on exiting Carbon copy prompt Copy to self when forwarding a message Copy to self when replying to a message Copy to self when sending a message
Signature file	Text file that is automatically appended to the end of the body of a mail message

Both the callable interface and the user interface access the user profile database to determine default processing characteristics.

16.5. Mail Utility Processing Contexts

The Mail utility defines four discrete levels of processing, or **contexts** for manipulating mail files, messages, folders, and the user profile database as shown in Table 16.3.

Table 16.3. Levels of Mail Utility Processing

Context	Entity
Mail file	Mail files and folders
Message	Mail files, folders, and messages
Send	Messages
User	User profile database

Within each context, your application processes specific entities in certain ways using callable MAIL routines as described in the sections that follow.

Initiating a MAIL Context

You must explicitly begin and end each MAIL context. Each group of routines contains a pair of context-initiating and terminating routines.

When you begin processing in any context, the Mail utility performs the following functions:

1. Allocates sufficient virtual memory to manage context information
2. Initializes context variables and internal structures

Terminating a MAIL Context

Terminating a MAIL processing context deallocates virtual memory. You must explicitly terminate processing in any context by calling a context-terminating routine.

16.5.1. Callable Mail Utility Routines

There are four types of callable Mail utility routines, each corresponding to the context within which they execute. A prefix identifies each functional group:

- MAIL$MAILFILE_
- MAIL$MESSAGE_
- MAIL$SEND_
- MAIL$USER_

Table 16.4 lists Mail utility routines according to context.

Table 16.4. Callable Mail Utility Routines

Context	Routine
Mail file	MAIL$MAILFILE_BEGIN MAIL$MAILFILE_CLOSE MAIL$MAILFILE_COMPRESS MAIL$MAILFILE_END MAIL$MAILFILE_INFO_FILE MAIL$MAILFILE_MODIFY MAIL$MAILFILE_OPEN MAIL$MAILFILE_PURGE_WASTE
Message	MAIL$MESSAGE_BEGIN MAIL$MESSAGE_COPY MAIL$MESSAGE_DELETE MAIL$MESSAGE_END MAIL$MESSAGE_GET

Context	Routine
	MAIL$MESSAGE_INFO MAIL$MESSAGE_MODIFY MAIL$MESSAGE_SELECT
Send	MAIL$SEND_ABORT MAIL$SEND_ADD_ADDRESS MAIL$SEND_ADD_ATTRIBUTE MAIL$SEND_ADD_BODYPART MAIL$SEND_BEGIN MAIL$SEND_END MAIL$SEND_MESSAGE
User	MAIL$USER_BEGIN MAIL$USER_DELETE_INFO MAIL$USER_END MAIL$USER_GET_INFO MAIL$USER_SET_INFO

16.5.2. Single and Multiple Threads

Once you have successfully initiated MAIL processing in a context, you have created a **thread**. A thread is a series of calls to MAIL routines that uses the same context information. Applications can contain one or more threads.

Single Threads

For example, consider an application that begins mail file processing; opens, compresses, and closes a mail file; and ends mail file context processing. This application executes a single thread of procedures that reference the same context variable names and pass the same context information.

Multiple Threads

You can create up to 31 concurrent threads. Applications that contain more than one thread must maintain unique context variables for each thread in order to pass thread-specific context information.

The Mail utility returns the condition value MAIL$_NOMORECTX when your process attempts to exceed the maximum number of allowable threads.

16.6. Programming Considerations

The calling sequence for all MAIL routines consists of a status variable, an entry point name, and an argument list. All arguments within the argument list are required. All callable MAIL routines use the same arguments in their calling sequences as described in the following example:

```
STATUS=MAIL$MAILFILE_BEGIN(CONTEXT, IN_ITEM_LIST, OUT_ITEM_LIST)
```

The variable **status** receives the condition value, and the argument **context** receives the context information. The arguments **in_item_list** and **out_item_list** are input and output item lists that contain one or more input or output item descriptors.

16.6.1. Condition Handling

At run time, a hardware- or software-related event can occur that determines whether or not the application executes successfully. The Mail utility processes such an event, or **condition** in the following ways:

- Signals the condition value
- Returns the error code

You can establish your own condition handler or allow the program to signal the default condition handler.

Returning Condition Values

You can disable signaling for any call by specifying the item code MAIL$_NOSIGNAL as an item in the input item list.

16.6.2. Item Lists and Item Descriptors

Your application passes data to callable MAIL routines and receives data from routines through data structures called **item lists** defined in your program.

16.6.2.1. Structure of an Item Descriptor

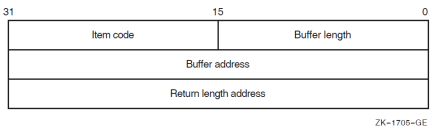
An input or output item list is a data structure that consists of one or more input or output item descriptors.

The following table summarizes the characteristics of item lists:

Item Descriptor	Characteristics
Input	Each descriptor points to a buffer or file from which Mail reads data.
Output	Each descriptor points to a buffer or file to which Mail writes data.

An item descriptor is a data structure consisting of three longwords as described in Figure 16.1.

Figure 16.1. Item Descriptor



Item descriptor fields are described as follows:

Field	Function
Item code	Specifies an action the routine is to perform.

Field	Function	
Buffer length	Specifies the length in bytes of an input or output buffer.	
Buffer address	Specifies the address of the input or output buffer.	
Return length address	Depends on the type of item code specified:	
	Item Code	**Use**
	Input	Not used; specify *0*.
	Output	Address of a longword that receives the length of the result.

Note

You can specify item descriptors in any order within an item list.

Item Codes

The **item code** defines an action that the routine is to perform. Input and output item codes are specified in input and output item descriptors, respectively.

Boolean input and output item codes request an operation but do *not* pass data to the called routine. For example, the item code MAIL$_USER_SET_CC_PROMPT sets the CC prompt flag enabling use of *CC:* field text.

For a complete list of input and output item codes, see Tables 16.10 and 16.11.

16.6.2.2. Null Item Lists

Both the input and output item list arguments in the MAIL routine calling sequence are required. However, there might be situations when you do not want to request an operation or no input or output item codes are listed for the routine. In such cases, you must pass the value *0* in the function call.

16.6.2.3. Declaring Item Lists and Item Descriptors

Depending on the programming language you are using, refer to the appropriate language reference manual for more information about declaring data structures and creating variables.

16.6.2.4. Terminating an Item List

Terminate an item list with a null item descriptor. Assign the value *0* to each field in the item descriptor.

16.6.3. Action Routines

Certain callable MAIL routines allow you to specify an **action routine**. An action routine transfers control to a user-written subroutine that performs specific tasks.

The mail file, message, and send contexts permit the use of action routines for specific reasons. Table 16.5 summarizes the types of action routines and the contexts in which they are used.

Table 16.5. Types of Action Routines

Context	Routine	Action Routine
Mail file	MAIL$MAILFILE_INFO_FILE	Provides information about folder and mail files.

Context	Routine	Action Routine
Message	MAIL$MESSAGE_COPY	Copies messages between files and folders.
Send	MAIL$SEND_MESSAGE	Success and error results; sends a text file to an existing address list.

The preceding table summarizes typical uses of action routines. However, an action routine can perform any task you specify. See the *Guide to Creating OpenVMS Modular Procedures* for more information about action routines.

Mail File and Folder Action Routine Calling Sequence

The main portion of the application calls the action routine and passes values to it using parameters. The calling sequence of a mail file or folder action routine is as follows:

```
entry-point-name(userdata,foldername)
```

The argument **userdata** is the address of a required longword that contains user-specified data, and the argument **foldername** is the address of a descriptor of the foldername.

Send Action Routine Calling Sequence

The calling sequence of a send action routine is as follows:

```
entry-point-name(username,signal-array,userdata)
```

The argument **username** is the address of a descriptor of the user name to which the application successfully sent a message; **signal-array** is the address of a signal array containing the success message; **userdata** is the address of an optional longword that contains user-specified data.

16.7. Managing Mail Files

Using mail files involves opening and closing both default mail files and user-created mail files, displaying folder names, and purging and compressing mail files. Table 16.6 summarizes each mail file routine and its function.

Table 16.6. Mail File Routines

Routine	Description
MAIL$MAILFILE_BEGIN	Initiates mail file processing
MAIL$MAILFILE_CLOSE	Closes a mail file
MAIL$MAILFILE_COMPRESS	Compresses a mail file
MAIL$MAILFILE_END	Terminates mail file processing
MAIL$MAILFILE_INFO_FILE	Obtains information about the mail file
MAIL$MAILFILE_MODIFY	Changes the wastebasket folder name and the default mail file name
MAIL$MAILFILE_OPEN	Opens a mail file
MAIL$MAILFILE_PURGE_WASTE	Purges a mail file

Mail file context processing involves accessing and manipulating one or more mail files.

Initiating the Mail File Context

Your application must call MAIL$MAILFILE_BEGIN to perform mail file context processing.

When you call MAIL$MAILFILE_BEGIN successfully and begin processing in the mail file context, you have created a thread. You must specify the same context variable name in routine calls within the same thread.

Terminating the Mail File Context

Terminate processing in the mail file context calling MAIL routines in the following order:

1. Terminate message context processing (if applicable) using MAIL$MESSAGE_END.
2. Close the currently open mail file using MAIL$MAILFILE_CLOSE.
3. Terminate mail file context processing using MAIL$MAILFILE_END.

The following sections describe these actions in more detail.

16.7.1. Opening and Closing Mail Files

Before you perform any activities on existing messages, folders, and mail files, you must first open a mail file. Whenever you open a mail file, you must do so explicitly using MAIL$MAILFILE_OPEN. You can open only one mail file per mail file thread.

Note that each routine references the same context variable. An open mail file must be explicitly closed with a call to MAIL$MAILFILE_CLOSE.

16.7.1.1. Using the Default Specification for Mail Files

To open a mail file, Mail must first locate it using either a default or a user-specified mail file specification. A mail file specification consists of the following components: disk and directory, file name, and file type.

If you use the default file specification, the Mail utility locates and opens the default mail file using the following information:

Component	Source
User's disk and directory	Retrieved from the user authorization file (UAF)
MAIL subdirectory	Retrieved from the user profile entry
Mail file name and type	MAIL.MAI

16.7.1.2. Specifying an Alternate Mail File Specification

You can use the default specification for mail files or specify all or part of an alternate mail file specification.

When to Specify an Alternate Mail File Specification

The following mail file routines accept alternate mail file specifications when you use the item codes MAIL$_MAILFILE_DEFAULT_NAME or MAIL$_MAILFILE_NAME or both:

- MAIL$MAILFILE_COMPRESS
- MAIL$MAILFILE_INFO_FILE
- MAIL$MAILFILE_MODIFY
- MAIL$MAILFILE_OPEN

How the Mail Utility Creates an Alternate Mail File Specification

The Mail utility constructs an alternate mail file specification by using program-supplied mail file specifications to modify the default specification for mail files in the following order of importance:

1. Program-supplied file specification (MAIL$_MAILFILE_NAME)
    - Program-supplied disk and directory
    - Program-supplied file name and type
2. Program-supplied default file specification (MAIL$_MAILFILE_DEFAULT_NAME)
    - Program-supplied disk and directory
    - Program-supplied file name and type
3. Default specification

If you are using MAIL$_MAILFILE_DEFAULT_NAME and you specify *0* as the buffer size and address, the Mail utility uses the current device and directory.

The default specification for mail files applies unless overridden by your program-supplied mail file specifications. Mail file specifications defined with MAIL$_MAILFILE_NAME override those defined with MAIL$_MAILFILE_DEFAULT_NAME.

For example, an application can override the default specification $DISK0:[USER]MAIL.MAIL by defining an alternate device type $DISK99: using MAIL$_MAILFILE_NAME. The result is $DISK99:[USER]MAIL.MAI. The application can further modify the specification by defining a different mail file MYMAILFILE.MAI using MAIL$_MAILFILE_DEFAULT_NAME. The new mail file specification is $DISK99:[USER]MYMAILFILE.MAI.

16.7.2. Displaying Folder Names

As the size of your mail files increases with messages and folders, you might want to display your folder names. A user-written **folder action** routine lets you do this.

In the mail file context, MAIL$MAILFILE_INFO_FILE can be used to invoke a folder action routine that displays folder names in a mail file. If you specify the item code MAIL$_MAILFILE_FOLDER_ROUTINE, MAIL$MAILFILE_INFO passes a descriptor of a folder name to the action routine repeatedly until it encounters no more folder names and passes a null descriptor.

16.7.3. Purging Mail Files Using the Wastebasket Folder

The Mail utility associates messages designated for deletion with a wastebasket folder. Purging mail files of messages in the wastebasket folder that are designated for deletion is one way to conserve disk space.

You can also use the Mail utility to conserve disk space by reclaiming disk space and compressing mail files, as described in the sections that follow.

Note that purging the wastebasket folder removes the messages from the wastebasket folder but might not reclaim disk space.

16.7.3.1. Reclaiming Disk Space

Simply deleting the messages does not mean you will automatically **reclaim** the disk space. The Mail utility uses a system-defined threshold of bytes designated for deletion to determine when to reclaim disk space. When the total number of total bytes designated for deletion exceeds the threshold, the Mail utility performs a reclaim operation.

You can override the deleted bytes threshold and request a reclaim operation using MAIL $MAILFILE_PURGE_WASTE with the input item code MAIL$_MAILFILE_RECLAIM.

16.7.3.2. Compressing Mail Files

Compressing mail files is a way of conserving disk space. Mail file compression provides faster access to the folders and messages within the mail file. When you call MAIL$MAILFILE_COMPRESS, Mail removes unused space within the specified mail file.

16.8. Message Context

Message context processing involves manipulating existing messages as well as creating and deleting folders and mail files. Table 16.7 summarizes routines used in the message context.

Table 16.7. Message Routines

Routine	Description
MAIL$MESSAGE_BEGIN	Initiates message processing
MAIL$MESSAGE_COPY	Copies messages
MAIL$MESSAGE_DELETE	Deletes messages
MAIL$MESSAGE_END	Terminates message processing
MAIL$MESSAGE_GET	Retrieves a message
MAIL$MESSAGE_INFO	Obtains information about a specified message
MAIL$MESSAGE_MODIFY	Identifies a message as replied, new, or marked
MAIL$MESSAGE_SELECT	Selects a message or messages from the currently open mail file

Initiating the Message Context

Message context processing can begin only after a mail file has been opened. Your application must explicitly call MAIL$MESSAGE_BEGIN in order to execute message context processing.

The Mail utility passes mail file context information to the message context when you call MAIL $MESSAGE_BEGIN with the input item code MAIL$_MESSAGE_FILE_CTX.

Terminating the Message Context

To terminate message-level processing for a specific thread, you must call MAIL$MESSAGE_END to deallocate memory.

16.8.1. Selecting Messages

Applications select messages using MAIL$MESSAGE_SELECT to copy and move messages between folders as well as to read, modify, or delete messages. You must select messages before you can use them. You must specify a folder name when you select messages.

You can select messages based on the following criteria: matching character strings, message arrival date and time, and message characteristics.

Matching Character Strings

You can select a message or set of messages from a mail file by specifying one or more character substrings that you want to match with a character substring in the header information of a message or group of messages. You must specify the specific bodypart in the message header where the substring is located.

- *From:* line
- *To:* line
- *CC:* line
- *Subject:* line

The Mail utility searches the specified folder for message headers that contain the matching character substring. This method of selection is useful when you want to select and use messages from or to a particular user that are associated with many folder names.

When you specify more than one character substring, the Mail utility performs a logical AND operation to find the messages that contain the correct substring.

Message Arrival Date and Time

You can also select a message or group of messages based on their arrival time, that is, when you received them. Applications select messages according to two criteria as follows:

- Messages received before a specified date or time or both
- Messages received on or after a specified date or time or both

The Mail utility searches the mail file and selects messages whose primary key (date and time) matches the date and time specified in your application.

Message Characteristics

You can select messages based on Mail system flag values that indicate the following message characteristics:

- New
- Marked

- Replied

For example, you can select unread messages in order to display them or to display a message you have marked.

16.8.2. Reading and Printing Messages

After a message is selected, an application iteratively retrieves the contents of the bodypart record by record. The message can be retrieved using MAIL$MESSAGE_GET and can then be stored in a buffer or file.

Displaying a Message

To display a message on the terminal screen, you should store the message in a buffer and use the host programming language command that directs data to the screen.

Printing a Message

To print a message on a print queue on your system, you should write the message to an external file and use the $SNDJBC system service to manage print jobs and define queue characteristics.

16.8.3. Modifying Messages

Message modification using MAIL$MESSAGE_MODIFY involves setting flags that identify a message or group of messages as having certain characteristics. The following table summarizes bit offsets that modify flag settings:

Symbol	**Meaning**
MAIL$V_replied	Flagged as answered
MAIL$V_marked	Flagged for display purposes

16.8.4. Copying and Moving Messages

You can copy messages between folders within a mail file or between folders in different mail files using MAIL$MESSAGE_COPY. The Mail utility copies the message from the source folder to the destination folder leaving the original message intact.

Similarly, you can move messages between folders within a mail file or between folders in different mail files using MAIL$MESSAGE_COPY with the item code MAIL$_MESSAGE_DELETE. The Mail utility moves a message by copying the message from the source folder to the destination folder. You must specify a folder name.

When you move a message to another folder within the same mail file, you are changing the message's secondary key—its folder name.

16.8.4.1. Creating Folders

You can create a folder in a specified mail file whenever you attempt to copy or move a message to a nonexistent folder. When you create a folder, you are assigning a previously nonexistent folder name to a message as its secondary key.

Your application can include a user-written folder action routine that notifies you that the folder does not exist and accepts input to create the folder.

16.8.4.2. Deleting Folders

You can delete a folder by moving all of the messages within the source folder to another folder in the same mail file or to a folder in another mail file. In this case, the Mail utility associates messages that are moved with a new folder name.

You can also delete a folder by deleting all of the messages in a folder. The Mail utility associates messages designated for deletion with the wastebasket folder name.

In either case, the original folder name—the secondary key—no longer exists.

16.8.4.3. Creating Mail Files

Similarly, you can create a mail file whenever you attempt to copy or move a message to a nonexistent mail file.

Your application can include a user-written mail file action routine that notifies you that the mail file does not exist and accepts input to create the mail file.

Mail file creation involves creating the mail file and then copying or moving the message to the new mail file. If the message is shorter than 3 blocks, the Mail utility stores the message in the mail file. Otherwise, the Mail utility places a pointer to the message in the newly created mail file.

16.8.5. Deleting Messages

To delete a message, you need to know its message identification number. Applications can retrieve the message identification number by specifying the item code MAIL$_MESSAGE_ID when selecting a message or group of messages with MAIL$MESSAGE_SELECT.

When you delete all messages with the same secondary key (folder name) using MAIL $MESSAGE_DELETE and specifying the item code MAIL$_MESSAGE_ID, you have deleted the folder.

16.9. Send Context

Send context processing involves creating and sending new and existing messages. Table 16.8 summarizes send routines.

Table 16.8. Send Routines

Routine	Description
MAIL$SEND_ABORT	Aborts a send operation
MAIL$SEND_ADD_ADDRESS	Adds an addressee to the address list
MAIL$SEND_ADD_ATTRIBUTE	Constructs the message header
MAIL$SEND_ADD_BODYPART	Constructs the body of the message
MAIL$SEND_BEGIN	Initiates send processing
MAIL$SEND_END	Terminates send processing

Routine	Description
MAIL$SEND_MESSAGE	Sends a message

Initiating the Send Context

You can invoke the send context directly if you are creating a new message. Otherwise, to access an existing message, you must open the mail file that contains the message, select the message, and retrieve it.

Terminating the Send Context

You must terminate the send context explicitly using MAIL$SEND_END.

16.9.1. Sending New Messages

You can send new or existing messages to yourself and other users.

16.9.1.1. Creating a Message

You create new messages using send context routines. If you want to create and send a new message, you do not need to initiate any other context. As mentioned earlier, a message consists of two parts—the message header and the message bodypart.

Constructing a message involves building each part of the message separately using the following routines:

- MAIL$SEND_ADD_ATTRIBUTE
- MAIL$SEND_ADD_BODYPART

16.9.1.1.1. Constructing the Message Header

Each field of the message header is a **message attribute**. You can specify one or more attributes for inclusion in the message header using MAIL$SEND_ADD_ATTRIBUTE. During successive calls to MAIL$SEND_ADD_ATTRIBUTE, an application specifies the specific message attribute to be constructed.

If you do not specify the *From:* or *To:* fields, the Mail utility provides this information from the address list.

16.9.1.1.2. Constructing the Body of the Message

To construct a message, an application must specify a series of calls to MAIL $SEND_ADD_BODYPART to build a message from successive text records contained in a buffer or file.

If the body of the message is located in a file, you can build the bodypart with one call to MAIL $SEND_ADD_BODYPART by specifying its file name.

16.9.1.2. Creating an Address List

You must create an **address list** in order to send a message. The address list is a file or buffer of addressees to whom you want to send the message. Each entry in the address list is a valid user name on your system or on another system connected to your system by DECnet.

Adding User Names to the Address List

User names are added one at a time to the address list using one or more calls to MAIL $SEND_ADD_ADDRESS.

User Name Types

There are two types of user names--- **direct** and **carbon copy** addressees. Direct and carbon copy addressees correspond to user names in the *To:* and *CC:* fields of the message header.

16.9.2. Sending Existing Messages

Sending an existing message involves many tasks as well as initiating the mail file context and message context. The following table summarizes the tasks and routines involved in sending an existing message:

Task	**Routine**
Open a mail file.	MAIL$MAILFILE_OPEN
Select the message.	MAIL$MESSAGE_SELECT
Retrieve the message.	MAIL$MESSAGE_GET
Construct the message. Construct the message header. Construct the message bodypart.	MAIL$SEND_ADD_ATTRIBUTE MAIL$SEND_ADD_BODYPART
Create an address list.	MAIL$SEND_ADD_ADDRESS
Send the message.	MAIL$SEND_MESSAGE

16.9.3. Send Action Routines

Once you have created an address list and constructed a message, you can send the message using MAIL $SEND_MESSAGE. Optional success and error action routines handle signaled success and error events in a synchronous manner.

For example, If DECnet returns messages indicating that it might not be possible to complete a send operation to some users in your address list, a user-specified send action routine might prompt the sender for permission to continue the send operation.

16.9.3.1. Success Action Routines

A success action routine performs a task upon successful completion of a send operation.

16.9.3.2. Error Handling Routines

An error action routine is a user-written error handler that signals error conditions during a send operation.

16.9.3.3. Aborting a Send Operation

Under certain circumstances, you might want to terminate a send operation in progress using MAIL $SEND_ABORT. In this instance, you can use an asynchronous system trap (AST) routine that contains a call to MAIL$SEND_ABORT to abort the send operation whenever the user presses Ctrl/C.

16.10. User Profile Context

The user profile processing context functions as a system management tool for customizing the programming and interactive mail environments. It lets individual users modify their default processing characteristics.

The user profile database VMSMAIL_PROFILE.DATA contains information that application programs and the Mail utility use for processing in any context.

Table 16.9 summarizes the user context routines.

Table 16.9. User Profile Context Routines

Routine	Description
MAIL$USER_BEGIN	Initiates user profile context
MAIL$USER_DELETE_INFO	Deletes a user profile entry
MAIL$USER_END	Terminates user profile context
MAIL$USER_GET_INFO	Retrieves information about a user from the user profile
MAIL$USER_SET_INFO	Adds or modifies a user profile entry

Initiating the User Context

You can invoke the user context directly.

Terminating the User Context

You must terminate the user context with MAIL$USER_END. Terminating the user context deallocates virtual memory.

16.10.1. User Profile Entries

A user profile entry is a dynamic record. The Mail utility creates a user profile entry automatically for the calling process if it does not exist. The callable and user interfaces of the Mail utility use the data contained in the user profile entry. The user profile consists of fields as described in the sections that follow.

MAIL Subdirectory

A MAIL subdirectory is the location—that is, the disk and directory specification—of your mail files. When you define a MAIL subdirectory, you are creating a subdirectory in which the specified mail file and associated external messages are to reside. For example:

```
$DISK5:[MAILUSER.COMMON.MAIL]
```

The subdirectory [.common.mail] represents the MAIL subdirectory specification defined in the user profile entry. This subdirectory contains the mail file (for example, MAIL.MAI) and any external messages associated with the mail file. The disk and directory specification $DISK5:[MAILUSER] is defined in the user authorization file (UAF).

Flags

User profile flags can be set to enable or disable automatic purging of deleted mail, automatic self-copy when forwarding, replying, or sending messages, and use of the *CC* prompt.

Form

The form field of the user profile entry defines the default print form to be used by print batch jobs. The string you specify as the default form must match a valid print form in use on your system.

Forwarding Address

A forwarding address lets you receive messages to your account on another system or to have your messages sent to another user either on your system or another system. You must specify valid node names and user names.

Personal Name

A personal name is a user-specified character string. For example, a personal name might include your entire name and phone number. Any phrase beginning with alphabetic characters up to a maximum of 127 alphanumeric characters is valid. However, consecutive embedded spaces should not be used.

Queue Name

The queue name field defines the default print queue on your system where your print jobs are sent.

16.10.1.1. Adding Entries to the User Profile Database

Ordinarily, the Mail utility creates a user profile entry for the calling process if one does not already exist. A system management application might create entries for other users. When you specify the item code MAIL$_USER_CREATE_IF using MAIL$USER_SET_INFO, the Mail utility creates a user profile entry if it does not already exist.

16.10.1.2. Modifying or Deleting User Profile Entries

The calling process can modify, delete, or retrieve its own user profile entry without privileges.

The following table summarizes the privileges required to modify or delete user profile entries that do not belong to the calling process:

Procedure	Privilege	Function
MAIL$USER_SET_INFO	SYSPRV	Modifies another user's profile entry
MAIL$USER_GET_INFO	SYSNAM or SYSPRV	Retrieves information about another user

16.11. Input Item Codes

Input item codes direct the called routine to read data from a buffer or file and perform a task. Table 16.10 summarizes input item codes.

Table 16.10. Input Item Codes

Item Code	Function
Mail File Context	
MAIL$_MAILFILE_DEFAULT_NAME	Specifies the location (disk and directory) of the default mail file MAIL.MAI.
MAIL$_MAILFILE_FOLDER_ROUTINE	Displays folder names within a specified mail file.
MAIL$_MAILFILE_FULL_CLOSE	Requests that the wastebasket folder be purged and that a convert/reclaim operation be performed, if necessary.
MAIL$_MAILFILE_NAME	Specifies the name of a mail file to be opened.
MAIL$_MAILFILE_RECLAIM	Overrides the deleted bytes threshold and requests a reclaim operation.
MAIL$_MAILFILE_USER_DATA	Passes a longword of user context data to an action routine.
MAIL$_MAILFILE_WASTEBASKET_NAME	Specifies a new name for the wastebasket in a specified mail file.
Message Context	
MAIL$_MESSAGE_AUTO_NEWMAIL	Places newly read messages in the Mail folder automatically.
MAIL$_MESSAGE_BACK	Returns the first record of the preceding message.
MAIL$_MESSAGE_BEFORE	Selects a message before a specified date.
MAIL$_MESSAGE_CC_SUBSTRING	Specifies a character string that must match a node or user name substring in the *CC:* field of the specified message.
MAIL$_MESSAGE_CONTINUE	Returns the next text record of the current message.
MAIL$_MESSAGE_DEFAULT_NAME	Specifies the default mail file specification.
MAIL$_MESSAGE_DELETE	Deletes a message in the current folder after the message has been copied to a new folder.
MAIL$_MESSAGE_FILE_ACTION	Specifies a user-written routine that is called if a mail file is to be created.
MAIL$_MESSAGE_FILE_CTX	Specifies mail file context received from MAIL $MAILFILE_BEGIN.
MAIL$_MESSAGE_FILENAME	Specifies the name of a mail file to which the message is to be moved.
MAIL$_MESSAGE_FOLDER_ACTION	Specifies a user-written routine that is called if a folder is to be created.
MAIL$_MESSAGE_FLAGS	Specifies MAIL system flags to use when selecting messages.
MAIL$_MESSAGE_FLAGS_MBZ	Specifies MAIL system flags that must be zero.
MAIL$_MESSAGE_FOLDER	Specifies the name of the target folder for moving messages.

Item Code	Function
MAIL$_MESSAGE_FROM_SUBSTRING	Specifies a character string that must match a node or user name substring in the *From:* field of the specified message.
MAIL$_MESSAGE_ID	Specifies the message identification number of the message on which an operation is to be performed.
MAIL$_MESSAGE_NEXT	Returns the first record of the message following the current message.
MAIL$_MESSAGE_SINCE	Selects a message received on or after a specified date.
MAIL$_MESSAGE_SUBJ_SUBSTRING	Specifies a character string that must match a node or user name substring in the *Subject:* field of the specified message.
MAIL$_MESSAGE_TO_SUBSTRING	Specifies a character string that must match a substring in the *To:* field of the specified message.
MAIL$_MESSAGE_USER_DATA	Specifies a longword to be passed to the folder and mail file action routines.
Send Context	
MAIL$_SEND_CC_LINE	Specifies the *CC:* field text.
MAIL$_SEND_DEFAULT_NAME	Specifies the default file specification of a text file to be opened.
MAIL$_SEND_ERROR_ENTRY	Specifies a user-written routine to process errors that occur during a send operation.
MAIL$_SEND_FID	Specifies the file identifier.
MAIL$_SEND_FILENAME	Specifies the input file specification of a text file to be opened.
MAIL$_SEND_FROM_LINE	Specifies the *From:* field text.
MAIL$_SEND_PERS_NAME MAIL$_SEND_NO_PERS_NAME	Specifies the personal name string. Specifies that no personal string be used.
MAIL$_SEND_RECORD	Specifies the descriptor of a text record to be added to the body of a message.
MAIL$_SEND_SIGFILE	Specifies a full OpenVMS file specification of the signature file to be used in the message.
MAIL$_SEND_NO_SIGFILE	Specifies that no signature file be used.
MAIL$_SEND_SUBJECT	Specifies the *Subject:* field text.
MAIL$_SEND_SUCCESS_ENTRY	Specifies a user-written routine to process successfully completed events during a send operation.
MAIL$_SEND_TO_LINE	Specifies the *To:* field text.
MAIL$_SEND_USER_DATA	Specifies a longword passed to the send action routines.
MAIL$_SEND_USERNAME	Adds a specified user name to the address list.

Item Code	Function
MAIL$_SEND_USERNAME_TYPE	Specifies the type of user name added to the address list.
MAIL$_SEND_RECIP_FOLDER	Specifies the descriptor of a recipients folder name.
User Context	
MAIL$_USER_CREATE_IF	Creates a user profile entry.
MAIL$_USER_FIRST	Returns information about the first user in the user profile database.
MAIL$_USER_NEXT	Returns information about the next user in the user profile database.
MAIL$_USER_SET_AUTO_PURGE MAIL$_USER_SET_NO_AUTO_PURGE	Sets the automatic purge flag. Clears the automatic purge flag.
MAIL$_USER_SET_CC_PROMPT MAIL$_USER_SET_NO_CC_PROMPT	Sets the *CC* prompt flag. Clears the *CC* prompt flag.
MAIL$_USER_SET_COPY_FORWARD MAIL$_USER_SET_NO_COPY_FORWARD	Sets the copy self forward flag. Clears the copy self forward flag.
MAIL$_USER_SET_COPY_REPLY MAIL$_USER_SET_NO_COPY_REPLY	Sets the copy self reply flag. Clears the copy self reply flag.
MAIL$_USER_SET_COPY_SEND MAIL$_USER_SET_NO_COPY_SEND	Sets the copy self send flag. Clears the copy self send flag.
MAIL$_USER_SET_EDITOR MAIL$_USER_SET_NO_EDITOR	Specifies the default editor. Clears the default editor field.
MAIL$_USER_SET_FORM MAIL$_USER_SET_NO_FORM	Specifies the default print form string. Clears the default print form field.
MAIL$_USER_SET_FORWARDING MAIL$_USER_SET_NO_FORWARDING	Specifies the forwarding address string. Clears the forwarding address field.
MAIL$_USER_SET_NEW_MESSAGES	Specifies the new messages count.
MAIL$_USER_SET_PERSONAL_NAME MAIL$_USER_SET_NO_PERSONAL_NAME	Specifies the personal name string. Clears the personal name field.
MAIL$_USER_SET_QUEUE MAIL$_USER_SET_NO_QUEUE	Specifies the default print queue name string. Clears the default print queue name field.
MAIL$_USER_SET_SIGFILE	Specifies a signature file specification for the specified user.
MAIL$_USER_SET_NO_SIGFILE	Clears a signature file field for the specified user.
MAIL$_USER_SET_SUB_DIRECTORY MAIL$_USER_SET_NO_SUB_DIRECTORY	Specifies a MAIL subdirectory. Clears the MAIL subdirectory field.

Item Code	Function
MAIL$_USER_USERNAME	Points to the user name string to specify the user profile entry to be modified.

16.12. Output Item Codes

Output item codes direct the called routine to return data to a buffer or file which is then available for use by the application. Table 16.11 summarizes output item codes.

Table 16.11. Output Item Codes

Item Code	Function
Mail File Context	
MAIL$_MAILFILE_INDEXED	Determines whether the mail file format is indexed.
MAIL$_MAILFILE_DIRECTORY	Returns the mail file subdirectory specification to the caller.
MAIL$_MAILFILE_RESULTSPEC	Returns the result mail file specification.
MAIL$_MAILFILE_WASTEBASKET	Returns the wastebasket folder name for the specified file.
MAIL$_MAILFILE_DELETED_BYTES	Returns the number of deleted bytes in a specified mail file.
MAIL$_MAILFILE_MESSAGES_DELETED	Returns the number of deleted messages.
MAIL$_MAILFILE_DATA_RECLAIM	Returns the number of data buckets reclaimed.
MAIL$_MAILFILE_DATA_SCAN	Returns the number of data buckets scanned.
MAIL$_MAILFILE_INDEX_RECLAIM	Returns the number of index buckets reclaimed.
MAIL$_MAILFILE_TOTAL_RECLAIM	Returns the total number of bytes reclaimed.
Message Context	
MAIL$_MESSAGE_BINARY_DATE	Returns the date and time received as a binary value.
MAIL$_MESSAGE_CC	Returns the text in the *CC:* field of the current message.
MAIL$_MESSAGE_CURRENT_ID	Returns the message identification number of the current message.
MAIL$_MESSAGE_DATE	Returns the message creation date string.
MAIL$_MESSAGE_EXTID	Returns the external message identification number of the current message.
MAIL$_MESSAGE_FILE_CREATED	Returns the value of the mail file created flag.
MAIL$_MESSAGE_FOLDER_CREATED	Returns the value of the folder created flag.
MAIL$_MESSAGE_FROM	Returns the text in the *From:* field of the current message.
MAIL$_MESSAGE_RECORD	Returns a record from the current message.
MAIL$_MESSAGE_RECORD_TYPE	Returns the record type.
MAIL$_MESSAGE_REPLY_PATH	Returns the reply path.

Item Code	Function
MAIL$_MESSAGE_RESULTSPEC	Returns the resultant mail file specification.
MAIL$_MESSAGE_RETURN_FLAGS	Returns the MAIL system flag value of the current message.
MAIL$_MESSAGE_SELECTED	Returns the number of selected messages.
MAIL$_MESSAGE_SENDER	Returns the name of the sender of the current message.
MAIL$_MESSAGE_SIZE	Returns the size in records of the current message.
MAIL$_MESSAGE_SUBJECT	Returns the text in the *Subject:* field of the specified message.
MAIL$_MESSAGE_TO	Returns the text in the *To:* field of the specified message.
Send Context	
MAIL$_SEND_COPY_FORWARD	Returns the value of the caller's copy forward flag.
MAIL$_SEND_COPY_REPLY	Returns the value of the caller's copy reply flag.
MAIL$_SEND_COPY_SEND	Returns the value of the caller's copy send flag.
MAIL$_SEND_RESULTSPEC	Returns the resultant file specification of the file to be sent.
MAIL$_SEND_USER	Returns the process owner's user name.
User Context	
MAIL$_USER_AUTO_PURGE	Returns the value of the automatic purge mail flag.
MAIL$_USER_CAPTIVE	Returns the value of the UAF captive flag.
MAIL$_USER_CC_PROMPT	Returns the value of the *CC* prompt flag.
MAIL$_USER_COPY_FORWARD	Returns the value of the copy self forward flag.
MAIL$_USER_COPY_REPLY	Returns the value of the copy self reply flag.
MAIL$_USER_COPY_SEND	Returns the value of the copy self send flag.
MAIL$_USER_EDITOR	Returns the name of the default editor.
MAIL$_USER_FORM	Returns the default print form string.
MAIL$_USER_FORWARDING	Returns the forwarding address string.
MAIL$_USER_FULL_DIRECTORY	Returns the complete directory path of the mail file subdirectory.
MAIL$_USER_NEW_MESSAGES	Returns the new message count.
MAIL$_USER_PERSONAL_NAME	Returns the personal name string.
MAIL$_USER_QUEUE	Returns the default queue name string.
MAIL$_USER_RETURN_USERNAME	Returns the user name string.
MAIL$_USER_SIGFILE	Returns the default signature file specification.
MAIL$_USER_SUB_DIRECTORY	Returns the subdirectory specification.

16.13. Using the MAIL Routines: Examples

This section provides examples of using the MAIL routines in various programming scenarios including the following:

- Example 16.2 is a C program that sends a Mail message to another user.
- Example 16.3 is a C program that displays a user's folders and returns how many messages are in each folder.
- Example 16.4 is a C program that displays fields in the user's Mail profile.

Example 16.2. Sending a File

```
/* send_message.c */

#include <stdio>
#include <descrip>
#include <ssdef>
#include <maildef>
#include <nam>
#include <string>
#include <stdlib>
#include <iledef>
#include <mail$routines>

# define __NEW_STARLET

typedef struct _ile3 ITMLST;

unsigned int *
  send_context = 0
  ;

ITMLST
  nulllist[] = { {0,0,0,0} };

int
  getline(char *line, int max)
{
  if (fgets(line, max, stdin) == NULL)
    return 0;
  else
    return strlen(line);
}

static int handler (void) {
    return SS$_CONTINUE;
}

int
  main (int argc, char *argv[])

{
  char
    to_user[NAM$C_MAXRSS],
    subject_line[NAM$C_MAXRSS],
    file[NAM$C_MAXRSS],
    resultspec[NAM$C_MAXRSS]
      ;
```

```
  unsigned int status = SS$_NORMAL;

  unsigned short
  to_user_len = 0,
  file_len = 0,
  resultspeclen,
  subject_line_len = 0
  ;

(void)lib$establish (&handler);

  ITMLST
    address_itmlst[] = {
      {sizeof(to_user), MAIL$_SEND_USERNAME, to_user, &to_user_len},
      {0,0,0,0}},
  bodypart_itmlst[] = {
    {sizeof(file), MAIL$_SEND_FILENAME, file, &file_len},
    {0,0,0,0}},
  out_bodypart_itmlst[] = {
    {sizeof(resultspec), MAIL$_SEND_RESULTSPEC, resultspec,
 &resultspeclen},
    {0,0,0,0}},
  attribute_itmlst[] = {
    {sizeof(to_user), MAIL$_SEND_TO_LINE, to_user, &to_user_len},
    {sizeof(subject_line), MAIL$_SEND_SUBJECT, subject_line,
 &subject_line_len},
    {0,0,0,0}}
  ;

  status = mail$send_begin(&send_context, &nulllist, &nulllist);
  if (status != SS$_NORMAL)
    exit(status);

  /* Get the destination and add it to the message */
  printf("To: ");
  to_user[getline(to_user, NAM$C_MAXRSS) - 1] = '\0';

  address_itmlst[0].ile3$w_length = strlen(to_user);
  address_itmlst[0].ile3$ps_bufaddr = to_user;

  status = mail$send_add_address(&send_context, address_itmlst, &nulllist);

  if (status != SS$_NORMAL)
      return(status);

  /* Get the subject line and add it to the message header */
  printf("Subject: ");
  subject_line[getline(subject_line, NAM$C_MAXRSS) - 1] = '\0';

  /*  Displayed TO: line */
  attribute_itmlst[0].ile3$w_length = strlen(to_user);
  attribute_itmlst[0].ile3$ps_bufaddr = to_user;

  /* Subject: line */
  attribute_itmlst[1].ile3$w_length = strlen(subject_line);
  attribute_itmlst[1].ile3$ps_bufaddr = subject_line;
```

```
  status = mail$send_add_attribute(&send_context, attribute_itmlst,
 &nulllist);
  if (status != SS$_NORMAL)
    return(status);

  /* Get the file to send and add it to the bodypart of the message */
  printf("File: ");
  file[getline(file, NAM$C_MAXRSS) - 1] = '\0';

  bodypart_itmlst[0].ile3$w_length = strlen(file);
  bodypart_itmlst[0].ile3$ps_bufaddr = file;

  status = mail$send_add_bodypart(&send_context, bodypart_itmlst,
 out_bodypart_itmlst);
  if (status != SS$_NORMAL)
    return(status);

  resultspec[resultspeclen] = '\0';
  printf("Full file spec actually sent: [%s]\n", resultspec);

  /* Send the message */
  status = mail$send_message(&send_context, nulllist, nulllist);
  if (status != SS$_NORMAL)
      return(status);

  /* Done processing witht the SEND context */
  status = mail$send_end(&send_context, nulllist, nulllist);
  if (status != SS$_NORMAL)
    return(status);

  return (status);
}
```

Example 16.3 shows a C program that displays folders.

Example 16.3. Displaying Folders

```
/* show_folders.c */

#include <stdio>
#include <descrip>
#include <ctype>
#include <ssdef>
#include <maildef>
#include <string>
#include <stdlib>
#include <mail$routines>

typedef struct itmlst
{
  short buffer_length;
  short item_code;
  long buffer_address;
  long return_length_address;
} ITMLST;

struct node
{
```

```
  struct node *next;            /* Next folder name node */
  char *folder_name;            /* Zero terminated folder name */
};
int
  folder_routine(struct node *list, struct dsc$descriptor *name)
{
  if (name->dsc$w_length)
    {
      while (list->next)
      list = list->next;

      list->next = malloc(sizeof(struct node));
      list = list->next;
      list->next = 0;
      list->folder_name = malloc(name->dsc$w_length + 1);
      strncpy(list->folder_name,name->dsc$a_pointer,name->dsc$w_length);
      list->folder_name[name->dsc$w_length] = '\0';

    }
  return(SS$_NORMAL);
}

main (int argc, char *argv[])
{
  struct node list = {0,0};

  int
    message_context = 0,
    file_context = 0,
    messages_selected = 0,
    total_folders = 0,
    total_messages = 0
      ;
  ITMLST
    nulllist[] = {{0,0,0,0}},
    message_in_itmlst[] = {
      {sizeof(file_context),MAIL$_MESSAGE_FILE_CTX,(long)&file_context,0},
      {0,0,0,0}},
    mailfile_info_itmlst[] = {
      {4,MAIL$_MAILFILE_FOLDER_ROUTINE,(long)folder_routine,0},
      {4,MAIL$_MAILFILE_USER_DATA,(long)&list,0},
      {0,0,0,0}},
    message_select_in_itmlst[] = {
      {0,MAIL$_MESSAGE_FOLDER,0,0},
      {0,0,0,0}},
    message_select_out_itmlst[] = {
      {sizeof(messages_selected),MAIL$_MESSAGE_SELECTED,
(long)&messages_selected,0},
      {0,0,0,0}};

  if (mail$mailfile_begin(&file_context, nulllist, nulllist) == SS$_NORMAL)
 {
    if (mail$mailfile_open(&file_context, nulllist, nulllist) == SS
$_NORMAL) {
      if (mail$mailfile_info_file(&file_context,
      mailfile_info_itmlst,
      nulllist) == SS$_NORMAL) {
 if (mail$message_begin(&message_context,
```

```
          message_in_itmlst,
          nulllist) == SS$_NORMAL) {
   struct node *tmp = &list;

   while(tmp->next) {
     tmp = tmp->next;
     message_select_in_itmlst[0].buffer_address = (long)tmp->folder_name;
     message_select_in_itmlst[0].buffer_length = strlen(tmp->folder_name);
     if (mail$message_select(&message_context,
        message_select_in_itmlst,
        message_select_out_itmlst) == SS$_NORMAL) {
       printf("Folder %s has %d messages\n",
       tmp->folder_name, messages_selected);
       total_messages += messages_selected;
       total_folders++;
     }
   }
   printf("Total of %d messages in %d folders\n",total_messages,
 total_folders);
 }
 mail$message_end(&message_context, nulllist, nulllist);
      }
      mail$mailfile_close(&file_context, nulllist, nulllist);
    }
    mail$mailfile_end(&file_context, nulllist, nulllist);
  }
}
```

Example 16.4 shows a C program that displays user profile information.

Example 16.4. Displaying User Profile Information

```
/* show_profile.c */

#include <stdio>
#include <ssdef>
#include <jpidef>
#include <maildef>
#include <stsdef>
#include <ctype>
#include <nam>
#include <string>
#include <stdlib>
#include <starlet>
#include <mail$routines>

struct itmlst
{
  short buffer_length;
  short item_code;
  long buffer_address;
  long return_length_address;
};

int
  user_context = 0
  ;
```

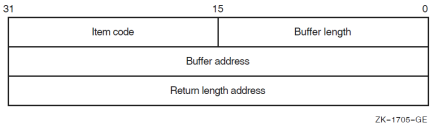
```
struct
  itmlst nulllist[] = { {0,0,0,0} };

int
  main (int argc, char *argv[])
{
  int

    userlen = 0,

    /* return length of strings */

    editor_len = 0,
    form_len = 0,
    forwarding_len = 0,
    full_directory_len = 0,
    personal_name_len = 0,
    queue_len = 0,

    /* Flags */

    auto_purge = 0,
    cc_prompt = 0,
    copy_forward = 0,
    copy_reply = 0,
    copy_send = 0
      ;

  char
    user[13],
    editor[NAM$C_MAXRSS],
    form[NAM$C_MAXRSS],
    forwarding[NAM$C_MAXRSS],
    full_directory[NAM$C_MAXRSS],
    personal_name[NAM$C_MAXRSS],
    queue[NAM$C_MAXRSS]
      ;

  short
    new_messages = 0
    ;

  struct itmlst
    jpi_list[]  = {
      {sizeof(user) - 1, JPI$_USERNAME, (long)user, (long)&userlen},
      {0,0,0,0}},
  user_itmlst[] = {
    {0, MAIL$_USER_USERNAME, 0, 0},
    {0,0,0,0}},
  out_itmlst[] = {
              /* Full directory spec */
    {sizeof(full_directory),MAIL$_USER_FULL_DIRECTORY,(long)full_directory,
(long)&full_directory_len},
              /* New message count */
    {sizeof(new_messages), MAIL$_USER_NEW_MESSAGES, (long)&new_messages,
 0},
              /* Forwarding field */
```

```
   {sizeof(forwarding), MAIL$_USER_FORWARDING, (long)forwarding,
(long)&forwarding_len},
            /* Personal name field */
   {sizeof(personal_name), MAIL$_USER_PERSONAL_NAME, (long)personal_name,
(long)&personal_name_len},
            /* Editor field */
   {sizeof(editor), MAIL$_USER_EDITOR, (long)editor, (long)&editor_len},
            /* CC prompting flag */
   {sizeof(cc_prompt), MAIL$_USER_CC_PROMPT, (long)&cc_prompt, 0},
            /* Copy send flag */
   {sizeof(copy_send), MAIL$_USER_COPY_SEND, (long)&copy_send, 0},
            /* Copy reply flag */
   {sizeof(copy_reply), MAIL$_USER_COPY_REPLY, (long)&copy_reply, 0},
            /* Copy forward flag */
   {sizeof(copy_forward), MAIL$_USER_COPY_FORWARD, (long)&copy_forward,
0},
            /* Auto purge flag */
   {sizeof(auto_purge), MAIL$_USER_AUTO_PURGE, (long)&auto_purge, 0},
            /* Queue field */
   {sizeof(queue), MAIL$_USER_QUEUE, (long)queue, (long)&queue_len},
            /* Form field */
   {sizeof(form), MAIL$_USER_FORM, (long)form, (long)&form_len},

   {0,0,0,0}};
 int
   status = SS$_NORMAL
     ;

 /* Get a mail user context */
 status = MAIL$USER_BEGIN(&user_context,
     &nulllist,
     &nulllist);
 if (status != SS$_NORMAL)
   return(status);

 if (argc > 1) {
   strcpy(user,argv[1]);
 }
 else
   {
     sys$getjpiw(0,0,0,jpi_list,0,0,0);
     user[userlen] = '\0';
   };

 while(isspace(user[--userlen]))
   user[userlen] = '\0';

 user_itmlst[0].buffer_length = strlen(user);
 user_itmlst[0].buffer_address = (long)user;

 status = MAIL$USER_GET_INFO(&user_context, user_itmlst, out_itmlst);
 if (status != SS$_NORMAL)
   return (status);

 /* Release the mail USER context */
 status = MAIL$USER_END(&user_context, &nulllist, &nulllist);
 if (status != SS$_NORMAL)
   return(status);
```

```
  /* display the information just gathered */

  full_directory[full_directory_len] = '\0';
  printf("Your mail file directory is %s.\n", full_directory);
  printf("You have %d new messages.\n", new_messages);

  forwarding[forwarding_len] = '\0';
  if (strlen(forwarding) == 0)
    printf("You have not set a forwarding address.\n");
  else
    printf("Your mail is being forwarded to %s.\n", forwarding);

  personal_name[personal_name_len] = '\0';
  printf("Your personal name is \"%s\"\n", personal_name);

  editor[editor_len] = '\0';
  if (strlen(editor) == 0)
    printf("You have not specified an editor.\n");
  else
    printf("Your editor is %s\n", editor);

  printf("CC prompting is %s.\n", (cc_prompt == TRUE) ? "disabled" :
 "enabled");

  printf("Automatic copy to yourself on");
  if (copy_send == TRUE)
    printf(" SEND");
  if (copy_reply == TRUE) {
    if (copy_send == TRUE)
      printf(",");
    printf(" REPLY");
  }
  if (copy_forward == TRUE) {
    if ((copy_reply == TRUE) || (copy_send == TRUE))
      printf(",");
    printf(" FORWARD");
  }
  if ((copy_reply == FALSE) && (copy_send == FALSE) && (copy_forward ==
 FALSE))
    printf(" Nothing");
  printf("\n");

  printf("Automatic deleted message purge is %s.\n", (auto_purge == TRUE) ?
 "disabled" : "enabled");

  queue[queue_len] = '\0';
  if (strlen(queue) == 0)
    printf("You have not specified a default queue.\n");
  else
    printf("Your default print queue is %s.\n", queue);
  form[form_len] = '\0';
  if (strlen(form) == 0)
    printf("You have not specified a default print form.\n");
  else
    printf("Your default print form is %s.\n", form);
}
```

16.14. MAIL Routines

This section describes the individual MAIL routines. Input and output item list arguments use item descriptor fields structured as shown in the following diagram:



Item Descriptor Fields

[buffer length]

For input item lists, this word specifies the length (in bytes) of the buffer that supplies the information needed by the routine to process the specified item code.

For output item lists, this word contains a user-supplied integer specifying the length (in bytes) of the buffer in which the routine is to write the information.

The required length of the buffer depends on the item code specified in the *item code* field of the item descriptor. If the value of *buffer length* is too small, the routine truncates the data.

[item code]

For input item lists, a word containing a user-supplied symbolic code that specifies an option for the Mail utility operation. For output item lists, a word containing a user-supplied symbolic code specifying the item of information that the routine is to return. Each programming language provides an appropriate mechanism for defining this information.

[buffer address]

For input item lists, a longword containing the address of the buffer that supplies information to the routine. For output item lists, a longword containing the user-supplied address of the buffer in which the routine is to write the information.

[return length address]

This field is not used for input item lists. For output item lists, this field contains a longword specifying the user-supplied address of a longword in which the routine writes the actual length in bytes of the information it returns.

MAIL$MAILFILE_BEGIN

Start Mail File Processing — Initiates mail file processing.

Format

```
MAIL$MAILFILE_BEGIN context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)

access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to other mail file routines. The *context* argument is the address of a longword that contains mail file context information.

You should specify the value of this argument as *0* in the first of a sequence of calls to mail file routines. In the following calls, you should specify the mail file context value returned by this routine.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by a longword value of *0*.

For this routine, there are no input item codes.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

The only output item code for this routine is the MAIL$_MAILFILE_MAIL_DIRECTORY item code. When you specify MAIL$_MAILFILE_MAIL_DIRECTORY, MAIL$MAILFILE_BEGIN returns the mail directory specification to the caller. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

MAIL$MAILFILE_BEGIN creates and initiates a mail file context for calls to other mail file routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$GET_VM, $GETJPIW, and $GETSYI.

MAIL$MAILFILE_CLOSE

Close the Current Mail File — Closes the currently open mail file.

Format

```
MAIL$MAILFILE_CLOSE context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify

mechanism:	by reference

Mail file context information to be passed to mail file routines. The *context* argument is the address of a longword that contains mail file context information returned by MAIL$MAILFILE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_FULL_CLOSE]

The Boolean item code MAIL$_MAILFILE_FULL_CLOSE specifies that MAIL$MAILFILE_CLOSE should purge the wastebasket folder when it closes the mail file. If the number of bytes deleted by the purge operation exceeds a system-defined threshold, the Mail utility reclaims the deleted space from the mail file.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

The system-defined threshold is reserved by VSI.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MAILFILE_DATA_RECLAIM]

When you specify MAIL$_MAILFILE_DATA_RECLAIM, MAIL$MAILFILE_CLOSE returns the number of data buckets reclaimed during the reclaim operation as a longword value.

[MAIL$_MAILFILE_DATA_SCAN]

When you specify MAIL$_MAILFILE_DATA_SCAN, MAIL$MAILFILE_CLOSE returns the number of data buckets scanned during the reclaim operation as a longword value.

[MAIL$_MAILFILE_INDEX_RECLAIM]

When you specify MAIL$_MAILFILE_INDEX_RECLAIM, MAIL$MAILFILE_CLOSE returns the number of index buckets reclaimed during a reclaim operation as a longword value.

[MAIL$_MAILFILE_MESSAGES_DELETED]

When you specify MAIL$_MAILFILE_MESSAGES_DELETED, MAIL$MAILFILE_CLOSE returns the number of messages deleted as a longword value.

[MAIL$_MAILFILE_TOTAL_RECLAIM]

When you specify MAIL$_MAILFILE_TOTAL_RECLAIM, MAIL$MAILFILE_CLOSE returns the number of bytes reclaimed during a reclaim operation as a longword value.

Description

If you specify the input item code MAIL$_MAILFILE_FULL_CLOSE, this procedure purges the wastebasket folder automatically before it closes the file. If the number of bytes deleted by this procedure exceeds the deleted byte threshold, the system performs a convert/reclaim operation on the file.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

No mail file is open.

SS$_ACCVIO

Access violation.

MAIL$MAILFILE_COMPRESS

Compress Mail File — Compresses a mail file.

Format

```
MAIL$MAILFILE_COMPRESS context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to various mail file routines. The *context* argument is the address of a longword that contains mail file context information returned by MAIL $MAILFILE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_DEFAULT_NAME]

MAIL$_MAILFILE_DEFAULT_NAME specifies the default file specification the Mail utility should use when opening a mail file. The *buffer address* field points to a character string 0 to 255 characters long that defines the default file specification.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you specify the value *0* in *buffer length* field of the item descriptor, MAIL $MAILFILE_COMPRESS uses the current default directory as the default mail file specification.

If you do not specify MAIL$_MAILFILE_DEFAULT_NAME, MAIL$MAILFILE_COMPRESS creates the default mail file specification from the following sources:

- Disk and directory defined in the caller's user authorization file (UAF)
- Subdirectory defined in the Mail user profile
- Default file type of .MAI

[MAIL$_MAILFILE_FULL_CLOSE]

The Boolean item code MAIL$_MAILFILE_FULL_CLOSE requests that the wastebasket folder be purged and that convert and reclaim operations be performed, if necessary.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_MAILFILE_NAME]

MAIL$_MAILFILE_NAME specifies the name of a mail file to be opened. The buffer that the *buffer address* field points to contains a character string of 0 to 255 characters.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you do not specify MAIL$_MAILFILE_NAME, the default mail file name is MAIL.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Code

[MAIL$_MAILFILE_RESULTSPEC]

When you specify MAIL$_MAILFILE_RESULTSPEC, the Mail utility returns the resultant mail file specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

If you do not specify an input file, the MAIL$MAILFILE_COMPRESS routine compresses the currently open Mail file. The MAIL$MAILFILE_COMPRESS routine signals informational messages concerning the phase of the compression.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOTISAM

The message file is not an indexed file.

RMS$_FNF

The specified file cannot be found.

RMS$_SHR

The specified file is not shareable.

SS$_ACCVIO

Access violation.

SS$_IVDEVNAM

The specified device name is invalid.

Any condition value returned by LIB$FIND_IMAGE_SYMBOL, LIB$RENAME_FILE, $CREATE, $OPEN, $PARSE, and $SEARCH.

MAIL$MAILFILE_END

End Mail File Processing — Terminates mail file processing.

Format

```
MAIL$MAILFILE_END context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to mail file routines. The *context* argument is the address of a longword that contains MAILFILE context information returned by MAIL$MAILFILE_BEGIN.

If mail file processing is terminated successfully, the Mail utility sets the value of the argument *context* to *0*.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_FULL_CLOSE]

The Boolean item code MAIL$_MAILFILE_FULL_CLOSE requests that the wastebasket folder be purged and that convert and reclaim operations be performed, if necessary.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

The MAIL$MAILFILE_END routine deallocates the mail file context created by MAIL $MAILFILE_BEGIN as well as any dynamic memory allocated by other mail file processing routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$FREE_VM.

MAIL$MAILFILE_INFO_FILE

Get Information About a Mail File — Obtains information about a specified mail file.

Format

```
MAIL$MAILFILE_INFO_FILE context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to mail file routines. The *context* argument is the address of a longword that contains mail file context information returned by MAIL$MAILFILE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_DEFAULT_NAME]

MAIL$_MAILFILE_DEFAULT_NAME specifies the default mail file specification MAIL $MAILFILE_INFO_FILE should use when opening a mail file. The *buffer address* field of the item descriptor points to a character string of 0 to 255 characters that defines the default mail file specification.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you specify the value *0* in *buffer length* field of the item descriptor, MAIL $MAILFILE_INFO_FILE uses the current default directory as the default mail file specification.

If you do not specify MAIL$_MAILFILE_DEFAULT_NAME, MAIL$MAILFILE_INFO_FILE creates the default mail file specification from the following sources:

- Disk and directory defined in the caller's user authorization file (UAF)
- Subdirectory defined in the Mail user profile
- Default file type of .MAI

[MAIL$_MAILFILE_FOLDER_ROUTINE]

MAIL$_MAILFILE_FOLDER_ROUTINE specifies an entry point longword address of a user-written routine that MAIL$MAILFILE_INFO_FILE should use to display folder names. MAIL $MAILFILE_INFO_FILE calls the user-written routine for each folder in the mail file.

[MAIL$_MAILFILE_NAME]

MAIL$_MAILFILE_NAME specifies the name of the mail file to be opened. The *buffer address* field points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you do not specify MAIL$_MAILFILE_NAME, the default mail file name is MAIL.

[MAIL$_MAILFILE_USER_DATA]

MAIL$_MAILFILE_USER_DATA specifies a longword that MAIL$MAILFILE_INFO_FILE should pass to the user-defined folder name action routine.

This item code is valid only when used with the item code MAIL$_MAILFILE_FOLDER_ROUTINE.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MAILFILE_DELETED_BYTES]

When you specify MAIL$_MAILFILE_DELETED_BYTES, MAIL$MAILFILE_INFO_FILE returns the number of deleted bytes in a specified mail file as longword value.

[MAIL$_MAILFILE_RESULTSPEC]

When you specify MAIL$_MAILFILE_RESULTSPEC, MAIL$MAILFILE_INFO_FILE returns the resultant mail file specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MAILFILE_WASTEBASKET]

When you specify MAIL$_MAILFILE_WASTEBASKET, MAIL$MAILFILE_INFO_FILE returns the name of the wastebasket folder of the specified mail file. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 39 characters long.

Specify a value from *0* to *39* in the *buffer length* field of the item descriptor.

Description

If you do not specify an input file, the MAIL$MAILFILE_INFO_FILE returns information about the currently open mail file.

Folder Action Routines

If you use the item code MAIL$_MAILFILE_FOLDER_ROUTINE to specify a folder name routine, MAIL$MAILFILE_INFO_FILE passes control to a user-specified routine. For example, the folder action routine could display folder names. The user routine must return a 32-bit integer code. If the return code indicates success, the interaction between the user's routine and the callable routine can continue.

The folder action routine passes a pointer to the descriptor of a folder name as well as the user data longword. A descriptor of zero length indicates that the MAIL$MAILFILE_INFO_FILE routine has displayed all folder names. If you do not specify the item code MAIL $_MAILFILE_FOLDER_ROUTINE, MAIL$MAILFILE_INFO_FILE does not call any folder action routines.

Condition Values Returned

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_NOTISAM

The message file is not an indexed file.

MAIL$_OPENIN

Mail cannot open the file as input.

SS$_ACCVIO

Access violation.

Any condition value returned by $CLOSE, $OPEN, $PARSE, and $SEARCH.

MAIL$MAILFILE_MODIFY

Modify Record of an Indexed File — Modifies the informational record of an indexed mail file, including the mail file name, the default mail file name, and the wastebasket name.

Format

```
MAIL$MAILFILE_MODIFY context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to mail file routines. The *context* argument is the address of a longword that contains mail file context information returned by MAIL$MAILFILE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_DEFAULT_NAME]

MAIL$_MAILFILE_DEFAULT_NAME specifies the default file specification that the Mail utility should use when opening a mail file. The *buffer address* field points to a buffer that contains a character string of 0 to 255 characters that defines the default mail file specification.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you specify the value *0* in the *buffer length* field of the item descriptor, MAIL $MAILFILE_MODIFY uses the current default directory as the default mail file specification.

If you do not specify MAIL$_MAILFILE_DEFAULT_NAME, MAIL$MAILFILE_MODIFY creates the default mail file specification from the following sources:

- Disk and directory defined in the caller's user authorization file (UAF)
- Subdirectory defined in the Mail user profile
- Default file type of .MAI

[MAIL$_MAILFILE_NAME]

MAIL$_MAILFILE_NAME specifies the name of the mail file that the Mail utility should open. The *buffer address* field points to a buffer that contains a character string of 0 to 255 characters.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you do not specify MAIL$_MAILFILE_NAME, the default mail file name is MAIL.

[MAIL$_MAILFILE_WASTEBASKET_NAME]

MAILFILE_WASTEBASKET_NAME specifies a new folder name for the wastebasket in the specified mail file. The *buffer address* field points to a buffer that contains a character string of 1 to 39 characters.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Code

[MAIL$_MAILFILE_RESULTSPEC]

When you specify MAIL$_MAILFILE_RESULTSPEC, the Mail utility returns the resultant mail file specification. The *buffer address* field points to a buffer that receives a character string from 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

If a mail file is not specified, the currently open mail file is used.

Condition Values Returned

MAIL$_ILLFOLNAM

The specified folder name is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOTISAM

The message file is not an indexed file.

MAIL$_OPENIN

Mail cannot open the file as input.

SS$_ACCVIO

Access violation.

Any condition value returned by $CLOSE, $FIND, $PUT, and $UPDATE.

MAIL$MAILFILE_OPEN

Open a Mail File for Processing — Opens a specified mail file for processing. You must use this routine to open a mail file before you can do either of the following: call any mail file routines to manipulate mail files; and call message routines to read messages from the specified mail file.

Format

```
MAIL$MAILFILE_OPEN context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to mail file routines. The *context* argument is the address of a longword that contains mail file context information returned by MAIL$MAILFILE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_DEFAULT_NAME]

MAIL$_MAILFILE_DEFAULT_NAME specifies the default file specification MAIL $MAILFILE_OPEN should use when opening a mail file. The *buffer address* field points to a character string of 0 to 255 characters that defines the default file specification.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you specify the value *0* in the *buffer length* field of the item descriptor, MAIL $MAILFILE_OPEN uses the current default directory as the default mail file specification.

If you do not specify MAIL$_MAILFILE_DEFAULT_NAME, MAIL$MAILFILE_OPEN creates the default mail file specification from the following sources:

- Disk and directory defined in the caller's user authorization file (UAF)
- Subdirectory defined in the Mail user profile
- Default file type of .MAI

[MAIL$_MAILFILE_NAME]

MAIL$_MAILFILE_NAME specifies the name of the mail file MAIL$MAILFILE_OPEN should open. The *buffer address* field points to a buffer that contains a character string of 0 to 255 characters.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

If you do not MAIL$_MAILFILE_NAME, the default mail file name is MAIL.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MAILFILE_DELETED_BYTES]

When you specify MAIL$_MAILFILE_DELETED_BYTES, MAIL$MAILFILE_OPEN returns the number of deleted bytes in the specified mail file as a longword value.

[MAIL$_MAILFILE_INDEXED]

When you specify MAIL$_MAILFILE_INDEXED, MAIL$MAILFILE_OPEN returns a Boolean TRUE when you open an indexed file. The *buffer length* field points to a longword that receives the Boolean value.

[MAIL$_MAILFILE_RESULTSPEC]

When you specify MAIL$_MAILFILE_RESULTSPEC, MAIL$MAILFILE_OPEN returns the resultant mail file specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MAILFILE_WASTEBASKET]

When you specify MAIL$_MAILFILE_WASTEBASKET, MAIL$MAILFILE_OPEN returns the name of the wastebasket for the specified mail file. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

The default mail file specification is MAIL.MAI in the MAIL subdirectory.

Condition Values Returned

MAIL$_FILEOPEN

The mail file is already open.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOMSGS

No messages are available.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$GET_VM, $CONNECT, and $OPEN.

MAIL$MAILFILE_PURGE_WASTE

Delete Wastebasket Messages — Deletes messages contained in the wastebasket folder of the currently open mail file.

Format

```
MAIL$MAILFILE_PURGE_WASTE context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Mail file context information to be passed to other mail file routines. The *context* argument is the address of a longword that contains mail file context information.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MAILFILE_RECLAIM]

The Boolean item code MAIL$_MAILFILE_RECLAIM specifies that MAIL $MAILFILE_PURGE_WASTE purge the wastebasket folder and reclaim deleted space in the mail file.

Specify the value *0* in the *buffer length* field of the item descriptor.

MAIL$_MAILFILE_RECLAIM explicitly requests a reclaim operation and overrides the deleted byte's threshold regardless of the number of bytes deleted during a mail file purge operation.

out_item_list

OpenVMS usage:	itmlst_3

type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MAILFILE_DATA_RECLAIM]

When you specify MAIL$_MAILFILE_DATA_RECLAIM, MAIL$MAILFILE_PURGE_WASTE returns the number of data buckets reclaimed during the reclaim operation as a longword value.

[MAIL$_MAILFILE_DATA_SCAN]

When you specify MAIL$_MAILFILE_DATA_SCAN, MAIL$MAILFILE_PURGE_WASTE returns the number of data buckets scanned during the reclaim operation as a longword value.

[MAIL$_MAILFILE_INDEX_RECLAIM]

When you specify MAIL$_MAILFILE_INDEX_RECLAIM, the Mail utility returns the number of index buckets reclaimed during a reclaim operation as a longword value.

[MAIL$_MAILFILE_DELETED_BYTES]

When you specify MAIL$_MAILFILE_DELETED_BYTES, MAIL$MAILFILE_PURGE_WASTE returns the number of bytes deleted from the mail file as a longword value.

[MAIL$_MAILFILE_MESSAGES_DELETED]

When you specify MAIL$_MAILFILE_MESSAGES_DELETED, MAIL $MAILFILE_PURGE_WASTE returns the number of deleted messages as a longword value.

[MAIL$_MAILFILE_TOTAL_RECLAIM]

When you specify MAIL$_MAILFILE_TOTAL_RECLAIM, MAIL$MAILFILE_PURGE_WASTE returns the number of bytes reclaimed due to a reclaim operation as a longword value.

Description

If you specify the MAIL$_MAILFILE_RECLAIM item descriptor, all the bytes deleted from the mail file by this routine are reclaimed.

Condition Values Returned

MAIL$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

No mail file is currently open.

MAIL$_NOTISAM

The message file is not an indexed file.

SS$_ACCVIO

Access violation.

MAIL$MESSAGE_BEGIN

Start Message Processing — Begins message processing. You must call this routine before calling any other message routines.

Format

```
MAIL$MESSAGE_BEGIN context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to various message routines. The *context* argument is the address of a longword that contains message context information.

You should specify the value of this argument as *0* in the first of a sequence of calls to message routines. In the following calls, you should specify the message context value returned by this routine.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_FILE_CTX]

MAIL$_MESSAGE_FILE_CTX specifies the mail file context received from MAIL $MAILFILE_BEGIN to be passed to the message routines. The *buffer address* field of the item descriptor points to a longword that contains mail file context information.

The item code MAIL$_MESSAGE_FILE_CTX is required.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Code

[MAIL$_MESSAGE_SELECTED]

When you specify MAIL$_MESSAGE_SELECTED, MAIL$MESSAGE_BEGIN returns the number of messages selected as a longword value.

Description

MAIL$MESSAGE_BEGIN creates and initializes a message context for subsequent calls to message routines.

Condition Values Returned

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_ACCVIO

Access violation.

Any condition value returned by $GET and LIB$GET_VM.

MAIL$MESSAGE_COPY

Copy Messages to Another File or Folder — Copies messages between files or folders.

Format

```
MAIL$MESSAGE_COPY context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)

access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information returned by MAIL$MESSAGE_BEGIN.

You should specify this argument as 0 in the first of a sequence of calls to message routines. In the following calls, you should specify the message context value returned by the previous routine.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_BACK]

When you specify the Boolean item code MAIL$_MESSAGE_BACK, MAIL$MESSAGE_COPY copies the message preceding the current message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_COPY.

[MAIL$_MESSAGE_DEFAULT_NAME]

MAIL$_MESSAGE_DEFAULT_NAME specifies the default file specification of a mail file to open in order to copy a message. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_DELETE]

When you specify the Boolean item code MAIL$_MESSAGE_DELETE, MAIL$MESSAGE_COPY deletes the message in the current folder after the message has been copied to a destination folder.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Specify MAIL$_MESSAGE_DELETE to emulate the operation of MAIL MOVE or FILE command.

[MAIL$_MESSAGE_FILE_ACTION]

MAIL$_MESSAGE_FILE_ACTION specifies the address of the mail file action routine called if a mail file is to be created. Two parameters are passed as follows:

- User data longword
- Address of the descriptor of the file name to be created

The *buffer address* field of the item descriptor points to a longword that denotes a procedure value.

[MAIL$_MESSAGE_FILENAME]

MAIL$_MESSAGE_FILENAME specifies the name of the mail file to which the current message will be moved. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_FOLDER]

MAIL$_MESSAGE_FOLDER specifies the name of the target folder for moving mail messages. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The item code MAIL$_MESSAGE_FOLDER is required.

[MAIL$_MESSAGE_FOLDER_ACTION]

MAIL$_MESSAGE_FOLDER_ACTION specifies the entry point address of the folder action routine called if a folder is to be created. Two parameters are passed as follows:

- User data longword
- Address of a descriptor of the folder name to be created.

The *buffer address* field of the item descriptor points to a longword that specifies a procedure value.

[MAIL$_MESSAGE_ID]

MAIL$_MESSAGE_ID specifies the message identification number of the message on which the operation is to be performed. The *buffer address* field of the item descriptor points to a longword that contains the message identification number.

Do not specify MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_COPY.

[MAIL$_MESSAGE_NEXT]

When you specify the Boolean item code MAIL$_MESSAGE_NEXT, the Mail utility copies the message following the current message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_COPY.

[MAIL$_MESSAGE_USER_DATA]

MAIL$_MESSAGE_USER_DATA specifies data passed to the folder action and mail file action routines. The *buffer address* field of the item descriptor points to a user data longword.

Specify MAIL$_MESSAGE_USER_DATA with the item codes MAIL$_MESSAGE_FILE_ACTION and MAIL$_MESSAGE_FOLDER_ACTION only.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MESSAGE_FILE_CREATED]

When you specify the Boolean item code MAIL$_MESSAGE_FILE_CREATED, MAIL $MESSAGE_COPY returns the value of the file created flag as longword value.

[MAIL$_MESSAGE_FOLDER_CREATED]

When you specify the Boolean item code MAIL$_MESSAGE_FOLDER_CREATED, MAIL $MESSAGE_COPY returns the value of the folder created flag as a longword value.

[MAIL$_MESSAGE_RESULTSPEC]

When you specify MAIL$_MESSAGE_RESULTSPEC, MAIL$MESSAGE_COPY returns the mail file resultant file specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

If you do not specify a file name, the routine copies the message to another folder in the currently open mail file. The target mail file must be an indexed file.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_BADVALUE

The specified keyword value is invalid.

MAIL$_CONITMCOD

The specified item codes define conflicting operations.

MAIL$_DATIMUSED

The date and time is currently used in the specified file.

MAIL$_DELMSG

The message is deleted.

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_MSGINFO

Informational records are successfully returned.

MAIL$_MSGTEXT

Text record is successfully returned.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_NOMOREREC

No more records can be found.

MAIL$_NOTREADIN

The operation is invalid; you are not reading a message.

MAIL$_RECTOBIG

The record is too large for the MAIL buffer.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_IVDEVNAM

The device name is invalid.

SS$_ACCVIO

Access violation.

Any condition value returned by $CONNECT, $CREATE, $OPEN, $WRITE, $READ, and $PUT.

MAIL$MESSAGE_DELETE

Delete Message From Current Folder — Deletes a specified message from the currently selected folder.

Format

```
MAIL$MESSAGE_DELETE context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_ID]

MAIL$_MESSAGE_ID specifies the message identification number of the message on which the operation is to be performed. The *buffer address* field points to a longword that contains the message identification number.

The item code MAIL$_MESSAGE_ID is required.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

When you delete a message from a selected folder, it is moved to the wastebasket folder. You cannot delete a message from the wastebasket folder. You must use the MAIL$MAILFILE_PURGE_WASTE routine to empty the wastebasket folder.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_ACCVIO

Access violation.

MAIL$MESSAGE_END

End Message Processing — Ends message processing.

Format

```
MAIL$MESSAGE_END context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information returned by MAIL$MESSAGE_BEGIN. If message processing ends successfully, the argument *context* is changed to *0*.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)

access:	read only
mechanism:	by reference

Item list specifying options for the routine. This routine does not use the *in_item_list* argument.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. This routine does not use the *out_item_list* argument.

Description

The MAIL$MESSAGE_END routine deallocates the message context created by MAIL $MESSAGE_BEGIN as well as any dynamic memory allocated by other message routines.

Condition Values Returned

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$FREE_VM.

MAIL$MESSAGE_GET

Get Message From a Set of Messages — Retrieves a message from the set of currently selected messages.

Format

```
MAIL$MESSAGE_GET context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)

access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information returned by MAIL$MESSAGE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_AUTO_NEWMAIL]

When you specify the Boolean item code MAIL$_MESSAGE_AUTO_NEWMAIL, MAIL $MESSAGE_GET automatically places a new message in the mail folder as it is read. MAIL $_MESSAGE_AUTO_NEWMAIL is valid only when specified with the item code MAIL $_MESSAGE_CONTINUE.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_MESSAGE_BACK]

When you specify the Boolean item code MAIL$_MESSAGE_BACK, MAIL$MESSAGE_GET reads the message identification number of a specified message to return the first record of the preceding message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_CONTINUE, MAIL $_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_GET.

[MAIL$_MESSAGE_CONTINUE]

When you specify the Boolean item code MAIL$_MESSAGE_CONTINUE, MAIL$MESSAGE_GET reads the message identification number of a specified message to return the next text record of the current message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_CONTINUE, MAIL $_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_GET.

[MAIL$_MESSAGE_ID]

MAIL$_MESSAGE_ID specifies the message identification number of a message on which an operation is to be performed. The *buffer address* field of the item descriptor points to a longword that contains the message identification number.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_CONTINUE, MAIL $_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_GET.

[MAIL$_MESSAGE_NEXT]

When you specify the Boolean item code MAIL$_MESSAGE_NEXT, MAIL$MESSAGE_GET reads the message identification number of a specified message to return the first record of the message following the current message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_CONTINUE, MAIL $_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_GET.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MESSAGE_BINARY_DATE]

When you specify MAIL$_MESSAGE_BINARY_DATE, MAIL$MESSAGE_GET returns the message arrival date as a quadword binary value.

[MAIL$_MESSAGE_CC]

When you specify MAIL$_MESSAGE_CC, MAIL$MESSAGE_GET returns the *CC:* field of the current message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_CURRENT_ID]

When you specify MAIL$_MESSAGE_CURRENT_ID, MAIL$MESSAGE_GET returns the message identification number of the current message. The *buffer address* field of the item descriptor points to a longword that receives the message identifier number.

[MAIL$_MESSAGE_DATE]

When you specify MAIL$_MESSAGE_DATE, MAIL$MESSAGE_GET returns the message creation date string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_EXTID]

MAIL$_MESSAGE_EXTID specifies the external message identification number of the current message. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

[MAIL$_MESSAGE_FROM]

When you specify MAIL$_MESSAGE_FROM, MAIL$MESSAGE_GET returns the *From:* field of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_RECORD]

When you specify MAIL$_MESSAGE_RECORD, MAIL$MESSAGE_GET returns a record of the message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

MAIL$_MESSAGE_RECORD is valid only when specified with the item code MAIL $_MESSAGE_CONTINUE.

Do not specify MAIL$_MESSAGE_RECORD with the following item codes:

- MAIL$_MESSAGE_BACK
- MAIL$_MESSAGE_ID
- MAIL$_MESSAGE_NEXT

[MAIL$_MESSAGE_RECORD_TYPE]

When you specify MAIL$_MESSAGE_RECORD_TYPE, MAIL$MESSAGE_GET returns the record type. A record may be either header information (MAIL$_MESSAGE_HEADER) or text (MAIL $_MESSAGE_TEXT). The *buffer address* field of the item descriptor points to a word that receives the record type.

[MAIL$_MESSAGE_RETURN_FLAGS]

When you specify MAIL$_MESSAGE_RETURN_FLAGS, MAIL$MESSAGE_GET returns the Mail system flag for the current message as a 2-byte bit mask value.

[MAIL$_MESSAGE_SENDER]

When you specify MAIL$_MESSAGE_SENDER, MAIL$MESSAGE_GET returns the name of the sender of the current message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_SIZE]

When you specify MAIL$_MESSAGE_SIZE, MAIL$MESSAGE_GET returns the size in records of the current message as a longword value.

[MAIL$_MESSAGE_SUBJECT]

When you specify MAIL$_MESSAGE_SUBJECT, MAIL$MESSAGE_GET returns the *Subject:* field of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_TO]

When you specify MAIL$_MESSAGE_TO, MAIL$MESSAGE_GET returns the *To:* field of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

Description

The first time the MAIL$MESSAGE_GET routine is called, the message information is returned for the first requested message, and the status returned is MAIL$_MSGINFO. Subsequent calls to MAIL$MESSAGE_GET with the MAIL$_MESSAGE_CONTINUE item code return the message text records with the status MAIL$_MSGTEXT, until no more records are left, when MAIL $_NOMOREREC is returned.

Condition Values Returned

MAIL$_MSGINFO

Informational records are successfully returned.

MAIL$_MSGTEXT

Text record is successfully returned.

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_NOMOREREC

No more records can be found.

MAIL$_NOTREADIN

The operation is invalid; you are not reading a message.

MAIL$_RECTOBIG

The record is too large for the mail buffer.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_ACCVIO

Access violation.

Any condition value returned by $FIND and $UPDATE.

MAIL$MESSAGE_INFO

Get Information About a Message — Obtains information about a specified message contained in the set of currently selected messages.

Format

```
MAIL$MESSAGE_INFO context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)

access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information returned by MAIL$MESSAGE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_BACK]

When you specify Boolean item code MAIL$_MESSAGE_BACK, MAIL$MESSAGE_INFO reads the identification number of the current message and returns the preceding message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_INFO.

[MAIL$_MESSAGE_ID]

MAIL$_MESSAGE_ID specifies the message identification number of the message on which the operation is to be performed. The *buffer address* field of the item descriptor points to a longword that contains the message identification number.

Do not specify MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_INFO.

[MAIL$_MESSAGE_NEXT]

When you specify the Boolean item code MAIL$_MESSAGE_NEXT, MAIL$MESSAGE_INFO reads the message identification number of the current message and returns the message that follows it.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL$_MESSAGE_NEXT in the same call to MAIL$MESSAGE_INFO.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_MESSAGE_BINARY_DATE]

When you specify MAIL$_MESSAGE_BINARY_DATE, MAIL$MESSAGE_INFO returns the message arrival date as a quadword binary value.

[MAIL$_MESSAGE_CC]

When you specify MAIL$_MESSAGE_CC, MAIL$MESSAGE_INFO returns the *CC:* field of the current message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_CURRENT_ID]

When you specify MAIL$_MESSAGE_ID, MAIL$MESSAGE_INFO returns the message identification number of the current message. The *buffer address* field of the item descriptor points to a longword that receives the message identification number of the current message.

[MAIL$_MESSAGE_DATE]

When you specify MAIL$_MESSAGE_DATE, MAIL$MESSAGE_INFO returns the message creation date string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_EXTID]

When you specify MAIL$_MESSAGE_EXTID, MAIL$MESSAGE_INFO returns the external identification number of the current message as a string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_FROM]

When you specify MAIL$_MESSAGE_FROM, MAIL$MESSAGE_INFO returns the *From:* field of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_REPLY_PATH]

When you specify MAIL$_MESSAGE_REPLY_PATH, MAIL$MESSAGE_INFO returns the reply path of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_RETURN_FLAGS]

When you specify MAIL$_MESSAGE_RETURN_FLAGS, MAIL$MESSAGE_INFO returns the Mail system flag values for the current message as a 2-byte bit mask value.

[MAIL$_MESSAGE_SENDER]

When you specify MAIL$_MESSAGE_SENDER, MAIL$MESSAGE_INFO returns the name of the sender of the current message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_SIZE]

When you specify MAIL$_MESSAGE_SIZE, MAIL$MESSAGE_INFO returns the size of the current message in records as a longword value.

[MAIL$_MESSAGE_SUBJECT]

When you specify MAIL$_MESSAGE_SUBJECT, MAIL$MESSAGE_INFO returns the *Subject:* field of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_TO]

When you specify MAIL$_MESSAGE_TO, MAIL$MESSAGE_INFO returns the *To:* field of the specified message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

Description

MAIL$MESSAGE_INFO obtains information about a particular message. MAIL$MESSAGE_GET retrieves a message from the set of currently selected messages.

The first call to MAIL$MESSAGE_GET passes control to MAIL$MESSAGE_INFO. Subsequent calls that include the MAIL$_MESSAGE_CONTINUE item code return text records.

Condition Values Returned

MAIL$_CONITMCOD

The specified item codes define conflicting operations.

MAIL$_DELMSG

The message is deleted.

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_NOMOREMSG

No more messages.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$GET_VM.

MAIL$MESSAGE_MODIFY

Modify Header Information — Modifies information in the message header.

Format

```
MAIL$MESSAGE_MODIFY context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information returned by MAIL$MESSAGE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_BACK]

When you specify the Boolean item code MAIL$_MESSAGE_BACK, MAIL$MESSAGE_MODIFY reads the identification number of the specified message in order to return the first record in the preceding message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL $_MESSAGE_NEXT in the same call to MAIL$MESSAGE_MODIFY.

[MAIL$_MESSAGE_FLAGS]

MAIL$_MESSAGE_FLAGS specifies system flags for new mail. The *buffer address* field of the item descriptor points to a word that contains bit mask offsets. The following offsets can be used to modify the 2-byte bit mask:

- MAIL$V_replied
- MAIL$V_marked

[MAIL$_MESSAGE_ID]

MAIL$_MESSAGE_ID specifies the message identification number of the message on which an operation is to be performed. The *buffer address* field of the item descriptor points to a longword that contains the message identification number.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL $_MESSAGE_NEXT in the same call to MAIL$MESSAGE_MODIFY.

[MAIL$_MESSAGE_NEXT]

When you specify the Boolean item code MAIL$_MESSAGE_NEXT, MAIL$MESSAGE_MODIFY reads the message identification number of a message and returns the first record in the message following the current message.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify the item codes MAIL$_MESSAGE_BACK, MAIL$_MESSAGE_ID, and MAIL $_MESSAGE_NEXT in the same call to MAIL$MESSAGE_MODIFY.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Code

[MAIL$_MESSAGE_CURRENT_ID]

When you specify MAIL$_MESSAGE_CURRENT_ID, MAIL$MESSAGE_MODIFY returns the message identification number of the current message. The *buffer address* field of the item descriptor points to a longword that receives the message identification number.

Condition Values Returned

MAIL$_CONITMCOD

The specified item codes define conflicting operations.

MAIL$_DELMSG

The message is deleted.

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_NOMOREMSG

No more messages.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_ACCVIO

Access violation.

Any condition value returned by $FIND and $UPDATE.

MAIL$MESSAGE_SELECT

Select Message from Current Mail File — Selects a message or messages from the currently open mail file. Before you attempt to read a message, you must select it.

Format

```
MAIL$MESSAGE_SELECT context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)

access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Message context information to be passed to message routines. The *context* argument is the address of a longword that contains message context information returned by MAIL$MESSAGE_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_MESSAGE_BEFORE]

When you specify MAIL$_MESSAGE_BEFORE, MAIL$MESSAGE_SELECT selects a message received before a specified date and time. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long in absolute time.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_CC_SUBSTRING]

MAIL$_MESSAGE_CC_SUBSTRING specifies a character string that must match a substring contained in the *CC:* field of the specified message. If the strings match, the message is selected. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_MESSAGE_FLAGS]

MAIL$_MESSAGE_FLAGS specifies bit masks that must be initialized to *1*.

[MAIL$_MESSAGE_FLAGS_MBZ]

MAIL$_MESSAGE_FLAGS_MBZ specifies Mail system flags that must be set to *0*.

[MAIL$_MESSAGE_FOLDER]

MAIL$_MESSAGE_FOLDER specifies the name of the folder that contains messages to be selected.

The `buffer address` field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the `buffer length` field of the item descriptor.

This item code is required.

[MAIL$_MESSAGE_FROM_SUBSTRING]

MAIL$_MESSAGE_FROM_SUBSTRING specifies a user-specified character string that must match the substring contained in the *From:* field of a specified message. If the strings match, the message is selected.

The `buffer address` field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the `buffer length` field of the item descriptor.

[MAIL$_MESSAGE_SINCE]

When you specify MAIL$_MESSAGE_SINCE, the Mail utility selects a message received on or after a specified date and time.

The `buffer address` field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long in absolute time.

Specify a value from *0* to *255* in the `buffer length` field of the item descriptor.

[MAIL$_MESSAGE_TO_SUBSTRING]

MAIL$_MESSAGE_TO_SUBSTRING specifies a user-specified character string that must match a substring contained in the *To:* field of a specified message. If the strings match, the message is selected.

The `buffer address` field of the item descriptor points to a buffer that contains a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the `buffer length` field of the item descriptor.

[MAIL$_MESSAGE_SUBJ_SUBSTRING]

MAIL$_MESSAGE_SUBJ_SUBSTRING specifies a user-specified character string that must match a substring contained in the *Subject:* field of a specified message. If the strings match, the message is selected.

The `buffer address` field of the item descriptor points to a buffer that contains a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Code

[MAIL$_MESSAGE_SELECTED]

When you specify MAIL$_MESSAGE_SELECTED, MAIL$MESSAGE_SELECT returns the number of selected messages as a longword value.

Description

MAIL$MESSAGE_SELECT deselects previously selected messages whether or not you request a valid selection.

Condition Values Returned

MAIL$_ILLCTXADR

The context block address is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_INVQUAVAL

The specified qualifier is invalid

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOFILEOPEN

The mail file is not open.

MAIL$_NOTEXIST

The specified folder does not exist.

MAIL$_NOTISAM

The operation applies only to indexed files.

MAIL$_WRONGCTX

The context block is incorrect.

MAIL$_WRONGFILE

The specified file is incorrect in this context.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$GET_VM.

MAIL$SEND_ABORT

Cancel Send Operation — Cancels a currently executing send operation.

Format

```
MAIL$SEND_ABORT context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Value Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to send routines. The *context* argument is the address of a longword that contains send context information returned by MAIL$SEND_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3

type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. This routine does not use the *in_item_list* argument.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. This routine does not use the *out_item_list* argument.

Description

MAIL$SEND_ABORT is useful when, for example, the user presses Ctrl/C during the execution of MAIL$SEND_MESSSAGE.

Condition Value Returned

SS$_NORMAL

Normal successful completion.

MAIL$SEND_ADD_ADDRESS

Add Address to List — Adds an address to the address list. If an address list does not exist, MAIL $SEND_ADD_ADDRESS creates one.

Format

```
MAIL$SEND_ADD_ADDRESS context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to send routines. The *context* argument is the address of a longword that contains send context information returned by MAIL$SEND_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_SEND_USERNAME]

MAIL$_SEND_USERNAME specifies that the Mail utility add a specified user name to the address list. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The item code MAIL$_SEND_USERNAME is required.

[MAIL$_SEND_USERNAME_TYPE]

MAIL$_SEND_USERNAME_TYPE specifies the type of user name added to the address list. The *buffer address* field of the item descriptor points to a word that contains the user name type.

There are two types of user names, as follows:

- User name specified as a *To:* address (default)
- User name specified as a *CC:* address

Note

Currently, the symbols MAIL$_TO and MAIL$_CC define user name types.

out_item_list

OpenVMS usage:	itmlst_3

type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

If you do not specify a MAIL$_SEND_USERNAME_TYPE, MAIL$SEND_ADD_ADDRESS uses MAIL$_TO. You can specify only one user name per call to MAIL$SEND_ADD_ADDRESS.

Condition Values Returned

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition values returned by LIB$TPARSE.

MAIL$SEND_ADD_ATTRIBUTE

Add Attribute to the Current Message — Adds an attribute, such as *Subject* or *To*, to the message you are currently constructing.

Format

```
MAIL$SEND_ADD_ATTRIBUTE context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to send routines. The *context* argument is the address of a longword that contains send context information returned by MAIL$SEND_BEGIN.

You should specify this argument as 0 in the first of a sequence of calls to MAIL routines. In following calls, you should specify the Send context value returned by the previous routine.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_SEND_CC_LINE]

MAIL$_SEND_CC_LINE specifies a descriptor of the *CC:* field text. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_SEND_FROM_LINE]

MAIL$_SEND_FROM_LINE specifies a descriptor of the *From:* field text of the message to be sent. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

Calls to MAIL$SEND_ADD_ATTRIBUTE using this input item code must be made before any calls to MAIL$SEND_ADD_ADDRESS.

The SYSPRV privilege is required to alter the *From:* of a message.

[MAIL$_SEND_SUBJECT]

MAIL$_SEND_SUBJECT specifies a descriptor of the *Subject:* field text of a message to be sent. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

[MAIL$_SEND_TO_LINE]

MAIL$_SEND_TO_LINE specifies a descriptor of the *To:* field text of the message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

If you do not specify a *To:* line, the Mail utility supplies a *To:* line composed of user names on the *To:* address list. If you do not specify a *CC:* line, the Mail utility supplies a *CC:* line composed of user names on the *CC:* address list. In either of the above cases, commas separate the user names.

To add a message's *From:* field, you must have the SYSPRV privilege, and the Mail DECnet object must have the SYSPRV privilege on OUTGOING CONNECT (users can set the DECnet object privileges at their discretion).

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

MAIL$SEND_ADD_BODYPART

Build Message Body — Builds the body of a message.

Format

```
MAIL$SEND_ADD_BODYPART context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to send routines. The *context* argument is the address of a longword that contains send context information returned by MAIL$SEND_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

See MAIL$SEND_BEGIN for a description of an input item descriptor.

Input Item Codes

[MAIL$_SEND_DEFAULT_NAME]

MAIL$_SEND_DEFAULT_NAME specifies the default file specification of a text file to be opened. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_SEND_FID]

MAIL$_SEND_FID specifies the file identifier of the text file to be opened. The *buffer address* field of the item descriptor points to a buffer that contains the file identifier. To identify a file using a file identifier, you must also specify the device identifier for the file. Specify the device identifier using the MAIL$_SEND_DEFAULT_NAME item code. More information about using a file ID for specifying files can be found in *VSI OpenVMS Record Management Services Reference Manual*. Note that the MAIL $_SEND_FID item code and the MAIL$_SEND_FILENAME item code are mutually exclusive.

[MAIL$_SEND_FILENAME]

MAIL$_SEND_FILENAME specifies the input file specification of the text file to be opened. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long. Note that the MAIL$_SEND_FILENAME item code and the MAIL$_SEND_FID item code are mutually exclusive.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_SEND_RECORD]

MAIL$_SEND_RECORD specifies a descriptor of a text record to be added to the body of the message. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 998 characters long.

Specify a value from *0* to *998* in the *buffer length* field of the item descriptor.

When creating a message, do not specify MAIL$_SEND_RECORD in the same call (or series of calls) to MAIL$SEND_ADD_BODYPART with the following item codes:

- MAIL$_SEND_FID
- MAIL$_SEND_FILENAME

Note

Do not use the MAIL$_SEND_RECORD item code with the MAIL$SEND_ADD_BODYPART routine called from a detached process. The routine creates a temporary file in SYS$SCRATCH that is inaccessible to the detached process.

out_item_list

OpenVMS usage: itmlst_3

type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Code

[MAIL$_SEND_RESULTSPEC]

When you specify MAIL$_SEND_RESULTSPEC, MAIL$SEND_ADD_BODYPART returns the resultant file specification identified with MAIL$_SEND_FILENAME. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

You can use MAIL$SEND_ADD_BODYPART to specify a file that contains the entire message or to add a single record to a message. If the message is contained in a file, you call MAIL $SEND_ADD_BODYPART once, specifying the file name. If you want to add to the message record-by-record, you can call MAIL$SEND_ADD_BODYPART repeatedly, specifying a different record each time until you complete the message.

You cannot specify both a file name and a record for the same message. You can specify either MAIL $_SEND_FILENAME or MAIL$_SEND_FID once, or you can specify MAIL$_SEND_RECORD one or more times.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_CONITMCOD

The specified item codes define conflicting operations.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_OPENIN

The required file is missing.

SS$_ACCVIO

Access violation.

MAIL$SEND_BEGIN

Start Sending Message — Initiates processing to send a message to the users on the address list. You must call MAIL$SEND_BEGIN before you call any other send routine.

Format

```
MAIL$SEND_BEGIN context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to other send routines. The *context* argument is the address of a longword that contains send context information.

You should specify the value of this argument as *0* in the first of a sequence of calls to send routines. In subsequent calls, you should specify the send context value returned by this routine.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_SEND_PERS_NAME MAIL$_SEND_NO_PERS_NAME]

Note that you must specify only one of these item codes. An error is generated if you specify both item codes. MAIL$_SEND_PERS_NAME specifies the personal name text to be used in the message header. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 127 characters long.

Specify a value from *0* to *127* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_SEND_NO_PERS_NAME specifies that no personal name string be used during message construction.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_SEND_SIGFILE MAIL$_SEND_NO_SIGFILE]

Note that you must specify only one of these item codes. An error is generated if you specify both item codes. MAIL$_SEND_SIGFILE specifies the full OpenVMS file specification of the signature file to be used in the message. The default file specification used for a signature file is the user mail directory specification and .SIG as the file type. The buffer address field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_SEND_NO_SIGFILE specifies that no signature file be used during message construction.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_SEND_COPY_FORWARD]

When you specify the Boolean item code MAIL$_SEND_COPY_FORWARD, MAIL$SEND_BEGIN returns the value of the caller's copy forward flag as a longword value.

[MAIL$_SEND_COPY_SEND]

When you specify the Boolean item code MAIL$_SEND_COPY_SEND, MAIL$SEND_BEGIN returns the value of the caller's copy send flag as a longword value.

[MAIL$_SEND_COPY_REPLY]

When you specify the Boolean item code MAIL$_SEND_COPY_REPLY, MAIL$SEND_BEGIN returns the value of the caller's copy reply flag as a longword value.

[MAIL$_SEND_USER]

When you specify MAIL$_SEND_USER, MAIL$SEND_BEGIN returns the process owner's user name. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

MAIL$SEND_BEGIN creates and initializes a send context for subsequent calls to send routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_CODERR

Internal system error.

MAIL$_CONITMCOD

The specified item codes perform conflicting operations.

MAIL$_ILLPERNAME

The specified personal name string is illegal.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition values returned by $GETJPIW, LIB$FREE_VM, and LIB$GET_VM.

MAIL$SEND_END

End Sending Message — Terminates send processing.

Format

```
MAIL$SEND_END context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to send routines. The *context* argument is the address of a longword that contains send context information returned by MAIL$SEND_BEGIN.

If send processing is successfully terminated, the value of the *context* argument is changed to *0*.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. This routine does not use the *in_item_list* argument.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. This routine does not use the *out_item_list* argument.

Description

The MAIL$SEND_END routine deallocates the send context as well as any dynamic memory allocated by previous send routine calls.

Condition Values Returned

SS$_NORMAL

Normal successful completion

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$FREE_VM.

MAIL$SEND_MESSAGE

MAIL$SEND_MESSAGE — Begins the actual sending of the message after the message has been constructed.

Format

```
MAIL$SEND_MESSAGE context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Send context information to be passed to send routines. The *context* argument is the address of a longword that contains send context information returned by MAIL$SEND_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_SEND_ERROR_ENTRY]

MAIL$_SEND_ERROR_ENTRY specifies the longword address of an entry point to process errors during a send operation. The descriptor of the recipient that failed, the address of the signal array, and the user-specified data are passed as input to the routine. Refer to the *VSI OpenVMS Programming Concepts Manual* for more information about the signal array and its use by condition-handling routines.

[MAIL$_SEND_RECIP_FOLDER]

MAIL$_SEND_RECIP_FOLDER specifies the descriptor of a recipients folder name. If you do not specify the MAIL$_SEND_RECIP_FOLDER item code, the mail will be sent to the default NEWMAIL folder. A valid folder name can be 1 to 39 characters in length.

[MAIL$_SEND_SUCCESS_ENTRY]

MAIL$_SEND_SUCCESS_ENTRY specifies the longword address of an entry point to process successes during a send operation. The descriptor of the recipient that succeeded, the address of the signal array, and the user-specified data are passed as input to the routine. Refer to the *VSI OpenVMS Programming Concepts Manual* for more information about the signal array and its use by condition-handling routines.

[MAIL$_SEND_USER_DATA]

MAIL$_SEND_USER_DATA specifies a longword that MAIL$SEND_MESSAGE passes to the SEND action routines.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

The MAIL$SEND_MESSAGE routine sends a message built with the MAIL $SEND_ADD_BODYPART routine to every user on the address list. If you have not used MAIL $SEND_ADD_BODYPART to construct a message, MAIL$SEND_MESSAGE sends only a message header.

If MAIL$SEND_MESSAGE encounters errors sending to an addressee, it calls the routine specified by MAIL$_SEND_ERROR_ENTRY. Otherwise, it calls the routine specified by MAIL $_SEND_SUCCESS_ENTRY.

If either routine is not specified, MAIL$SEND_MESSAGE calls no other routines.

If you specify the MAIL$_SEND_RECIP_FOLDER item code, the mail is placed in the specified folder. Otherwise, the mail is sent to the default NEWMAIL folder.

Condition Values Returned

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition value returned by $CONNECT.

MAIL$USER_BEGIN

Access the User Profile Database — Initiates access to the Mail common user database. You must call MAIL$USER_BEGIN before you call any other user routines.

Format

```
MAIL$USER_BEGIN context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only

mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

User context information to be passed to other user routines. The *context* argument is the address of a longword that contains user context information.

You should specify the value of this argument as *0* in the first of a sequence of calls to MAIL routines. In following calls, you should specify the user context value returned by the previous routine.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. This routine does not use the *in_item_list* argument.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_USER_AUTO_PURGE]

When you specify the Boolean item code MAIL$_USER_AUTO_PURGE, MAIL$USER_BEGIN returns the value of the automatic purge mail flag as a longword value.

[MAIL$_USER_CAPTIVE]

When you specify the Boolean item code MAIL$_USER_CAPTIVE, MAIL$USER_BEGIN returns the value of the UAF CAPTIVE flag as a longword value.

[MAIL$_USER_CC_PROMPT]

When you specify the Boolean item code MAIL$_USER_CC_PROMPT, MAIL$USER_BEGIN returns the value of the cc prompt flag as a longword value.

[MAIL$_USER_COPY_FORWARD]

When you specify the Boolean item code MAIL$_USER_COPY_FORWARD, MAIL$USER_BEGIN returns the value of the copy self forward flag as a longword value.

[MAIL$_USER_COPY_REPLY]

When you specify the Boolean item code MAIL$_USER_COPY_REPLY, MAIL$USER_BEGIN returns the value of the copy self reply flag as a longword value.

[MAIL$_USER_COPY_SEND]

When you specify the Boolean item code MAIL$_USER_COPY_SEND, MAIL$USER_BEGIN returns the value of the copy self send flag as a longword value.

[MAIL$_USER_FORWARDING]

When you specify MAIL$_USER_FORWARDING, MAIL$USER_BEGIN returns the forwarding address string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_FORM]

When you specify MAIL$_USER_FORM, MAIL$USER_BEGIN returns the default print form string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_FULL_DIRECTORY]

When you specify MAIL$_USER_FULL_DIRECTORY, MAIL$USER_BEGIN returns complete directory path of the MAIL subdirectory. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_NEW_MESSAGES]

When you specify MAIL$_USER_NEW_MESSAGES, MAIL$USER_BEGIN returns the new message count. The *buffer address* field of the item descriptor points to a word that receives the new message count.

[MAIL$_USER_PERSONAL_NAME]

When you specify MAIL$_USER_PERSONAL_NAME, MAIL$USER_BEGIN returns the personal name string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 127 characters long.

Specify a value from *0* to *127* in the *buffer length* field of the item descriptor.

[MAIL$_USER_QUEUE]

When you specify MAIL$_USER_QUEUE, MAIL$USER_BEGIN returns the default print queue name. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_RETURN_USERNAME]

When you specify MAIL$_USER_RETURN_USERNAME, MAIL$USER_BEGIN returns the user name string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_SIGFILE]

When you specify MAIL$_USER_SIGFILE, MAIL$USER_BEGIN returns the default signature file specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_SUB_DIRECTORY]

When you specify MAIL$_USER_SUB_DIRECTORY, MAIL$USER_BEGIN returns the subdirectory specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

MAIL$USER_BEGIN creates and initializes a user database context for subsequent calls to other user routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

MAIL$USER_DELETE_INFO

Delete Database Record — Removes a record from the user profile database.

Format

```
MAIL$USER_DELETE_INFO context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

User context information to be passed to send routines. The *context* argument is the address of a longword that contains user context information returned by MAIL$USER_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list must include at least one device item descriptor. The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_USER_USERNAME]

MAIL$_USER_USERNAME specifies the record to be deleted from the user profile database. The *buffer address* field of the item descriptor points to a buffer that contains the user name string encoded in a character string 0 to 31 characters long.

Specify a value from *0* to *31* in the *buffer length* field of the item descriptor.

Setting bit 4 of DCL_CTLFLAGS, enables the user name string encoded in a character string 0 to 255 characters long.

Note

Once this bit is set, user name length is set to a maximum of 255 characters long. Even if this bit is cleared, the behavior remains unchanged, that is, supports user name length of 255 characters long, but there is no way to reset it to 31 characters long.

The item code MAIL$_USER_USERNAME is required.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

To delete a record from the user profile database, you must have SYSPRV privilege.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOSUCHUSR

The specified user name is not valid.

MAIL$_NOSYSPRV

The operation requires the SYSPRV privilege.

SS$_ACCVIO

Access violation.

MAIL$USER_END

End Access to the User Profile Database — Terminates access to the user profile database.

Format

```
MAIL$USER_END context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

User context information to be passed to user routines. The *context* argument is the address of a longword that contains user context information.

If the Mail utility terminates access to the user profile database successfully, the value of the argument *context* is changed to *0*.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. This routine does not use the *in_item_list* argument.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. This routine does not use the *out_item_list* argument.

Description

The MAIL$USER_END routine deallocates the user database context created by MAIL$USER_BEGIN as well as all dynamic memory allocated by previous user routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

SS$_ACCVIO

Access violation.

Any condition value returned by LIB$FREE_VM.

MAIL$USER_GET_INFO

Get User Profile Information — Obtains information about a user from the user profile database.

Format

```
MAIL$USER_GET_INFO context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

User context information to be passed to user routines. The *context* argument is the address of a longword that contains user context information returned by MAIL$USER_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list must include at least one device item descriptor. The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_USER_FIRST]

The Boolean item code MAIL$_USER_FIRST specifies that MAIL$USER_GET_INFO return information in the user profile about the first entry in the user profile database.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify MAIL$_USER_FIRST, MAIL$_USER_NEXT or MAIL$_USER_USERNAME in the same call to MAIL$USER_GET_INFO.

[MAIL$_USER_NEXT]

The Boolean item code MAIL$_USER_NEXT specifies that MAIL$USER_GET_INFO return information in the user profile about the next user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

Do not specify MAIL$_USER_FIRST, MAIL$_USER_NEXT or MAIL$_USER_USERNAME in the same call to MAIL$USER_GET_INFO.

[MAIL$_USER_USERNAME]

The item code MAIL$_USER_USERNAME points to the username string.

Specify the address of the username string in the *buffer address* field and specify the length of the username string in the *buffer length* field of the item descriptor.

Do not specify MAIL$_USER_FIRST, MAIL$_USER_NEXT and MAIL$_USER_USERNAME in the same call to MAIL$USER_GET_INFO.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

[MAIL$_USER_AUTO_PURGE]

When you specify the Boolean item code MAIL$_USER_AUTO_PURGE, MAIL$USER_GET_INFO returns the value of the automatic purge mail flag as a longword value.

[MAIL$_USER_CC_PROMPT]

When you specify the Boolean item code MAIL$_USER_CC_PROMPT, MAIL$USER_GET_INFO returns the value of the cc prompt flag as a longword value.

[MAIL$_USER_COPY_FORWARD]

When you specify the Boolean item code MAIL$_USER_COPY_FORWARD, MAIL $USER_GET_INFO returns the value of the copy self forward mail flag as a longword value.

[MAIL$_USER_COPY_REPLY]

When you specify the Boolean item code MAIL$_USER_COPY_REPLY, MAIL$USER_GET_INFO returns the value of the copy self reply mail flag as a longword value.

[MAIL$_USER_COPY_SEND]

When you specify the Boolean item code MAIL$_USER_COPY_SEND, MAIL$USER_GET_INFO returns the value of the copy self send mail flag as a longword value.

[MAIL$_USER_EDITOR]

When you specify MAIL$_USER_EDITOR, MAIL$USER_GET_INFO returns the name of the default editor. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_FORWARDING]

When you specify MAIL$_USER_FORWARDING, MAIL$USER_GET_INFO returns the forwarding address. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_FORM]

When you specify MAIL$_USER_FORM, MAIL$USER_GET_INFO returns the default print form string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_FULL_DIRECTORY]

When you specify MAIL$_USER_FULL_DIRECTORY, MAIL$USER_GET_INFO returns the complete directory path of the MAIL subdirectory string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_NEW_MESSAGES]

When you specify MAIL$_USER_NEW_MESSAGES, MAIL$USER_GET_INFO returns the new messages count. The *buffer address* field of the item descriptor points to a word that receives the new message count as a word value.

[MAIL$_USER_PERSONAL_NAME]

When you specify MAIL$_USER_PERSONAL_NAME, MAIL$USER_GET_INFO returns the personal name string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 127 characters long.

Specify a value from *0* to *127* in the *buffer length* field of the item descriptor.

[MAIL$_USER_QUEUE]

When you specify MAIL$_USER_QUEUE, MAIL$USER_GET_INFO returns the default print queue name string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_RETURN_USERNAME]

When you specify MAIL$_USER_RETURN_USERNAME, MAIL$USER_GET_INFO returns the user name. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

[MAIL$_USER_SIGFILE]

When you specify MAIL$_USER_SIGFILE, MAIL$USER_GET_INFO returns the default signature file specification. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the buffer length field of the item descriptor.

[MAIL$_USER_SUB_DIRECTORY]

When you specify MAIL$_USER_SUB_DIRECTORY, MAIL$USER_GET_INFO returns the MAIL subdirectory specification string. The *buffer address* field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

Description

The MAIL$USER_GET_INFO routine returns information about specified entries in the user profile database. If you do not specify a user name, MAIL$USER_GET_INFO returns information about the user name associated with the calling process. To obtain information about a user name other than that associated with the calling process, you need the SYSNAM privilege.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_CONITMCOD

The specified item codes perform conflicting operations.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NOSUCHUSR

The specified user name is invalid.

MAIL$_NOSYSPRV

The specified operation requires the SYSPRV privilege.

SS$_ACCVIO

Access violation.

MAIL$USER_SET_INFO

Add User Profile Information — Adds or modifies a specified user record in the user profile database.

Format

```
MAIL$USER_SET_INFO context ,in_item_list ,out_item_list
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All utility routines return a condition value in R0. Condition values that can be returned by this routine are listed under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

User context information to be passed to user routines. The *context* argument is the address of a longword that contains user context information returned by MAIL$USER_BEGIN.

in_item_list

OpenVMS usage:	itmlst_3
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Item list specifying options for the routine. The *in_item_list* argument is the address of a list of item descriptors, each of which specifies an option and provides the information needed to perform the operation.

The item list must include at least one device item descriptor. The item list is terminated by longword value of *0*.

Input Item Codes

[MAIL$_USER_CREATE_IF]

The Boolean item code MAIL$_USER_CREATE_IF specifies that MAIL$USER_SET_INFO should create the record for the specified user if it does not already exist.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_AUTO_PURGE]

The Boolean item codes MAIL$_USER_SET_AUTO_PURGE and MAIL $_USER_SET_NO_AUTO_PURGE set and clear the auto purge flag for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_CC_PROMPT]

The Boolean item codes MAIL$_USER_SET_CC_PROMPT and MAIL $_USER_SET_NO_CC_PROMPT set and clear the cc prompt flag for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_COPY_FORWARD]

The Boolean item codes MAIL$_USER_SET_COPY_FORWARD and MAIL $_USER_SET_NO_COPY_FORWARD set and clear the copy self forward flag for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_COPY_REPLY]

The Boolean item codes MAIL$_USER_SET_COPY_REPLY and MAIL $_USER_SET_NO_COPY_REPLY set and clear the copy self reply flag for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_COPY_SEND]

The Boolean item codes MAIL$_USER_SET_COPY_SEND and MAIL $_USER_SET_NO_COPY_SEND set and clear the copy self send flag for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_EDITOR]

MAIL$_USER_SET_EDITOR specifies the name of a default editor to be used by the specified user. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_EDITOR clears the default editor field for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_FORM]

MAIL$_USER_SET_FORM specifies the default print form string for the specified user. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_FORM clears the default print form field for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_FORWARDING]

MAIL$_USER_SET_FORWARDING specifies a forwarding address string for the specified user. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_FORWARDING clears the forwarding address field for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_NEW_MESSAGES]

MAIL$_USER_SET_NEW_MESSAGES specifies the new message count for the specified user. The *buffer address* field of the item descriptor points to a word that contains the new number of new messages.

[MAIL$_USER_SET_PERSONAL_NAME]

MAIL$_USER_SET_PERSONAL_NAME specifies a personal name string for the specified user. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 127 characters long.

Specify a value from *0* to *127* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_PERSONAL_NAME clears the personal field for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_QUEUE]

MAIL$_USER_SET_QUEUE specifies a default print queue name string for the specified user. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_QUEUE clears the default print queue field for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_SIGFILE]

MAIL$_USER_SET_SIGFILE specifies a signature file specification for the specified user. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_SIGFILE clears the signature file field for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_SET_SUB_DIRECTORY]

MAIL$_USER_SET_SUB_DIRECTORY specifies a MAIL subdirectory. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 255 characters long.

Specify a value from *0* to *255* in the *buffer length* field of the item descriptor.

The Boolean item code MAIL$_USER_SET_NO_SUB_DIRECTORY disables the use of a MAIL subdirectory for the specified user.

Specify the value *0* in the *buffer length* and *buffer address* fields of the item descriptor.

[MAIL$_USER_USERNAME]

MAIL$_USER_USERNAME specifies the record to be modified in the user profile database and points to the user name string. The *buffer address* field of the item descriptor points to a buffer that contains a character string 0 to 31 characters long.

Specify a value from *0* to *31* in the *buffer length* field of the item descriptor.

Setting bit 4 of DCL_CTLFLAGS, enables the user name string encoded in a character string 0 to 255 characters long.

Note

Once this bit is set, user name length is set to a maximum of 255 characters long. Even if this bit is cleared, the behavior remains unchanged, that is, supports user name length of 255 characters long, but there is no way to reset it to 31 characters long.

out_item_list

OpenVMS usage:	itmlst_3
type:	longword
access:	write only
mechanism:	by reference

Item list specifying the information you want the routine to return. The *out_item_list* argument is the address of a list of item descriptors, each of which describes an item of information. The list of item descriptors is terminated by longword value of *0*.

Output Item Codes

None.

Description

The MAIL$USER_SET_INFO routine modifies specified records in the user profile database. If you do not specify a user name, the routine modifies the user record associated with the calling process.

To modify any user record other than that associated with the calling process, you must have SYSPRV privilege. However, if you want to add or modify only the forwarding address of another user, SYSNAM privilege is sufficient.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

MAIL$_CONITMCOD

The specified item codes perform conflicting operations.

MAIL$_ILLCHAR

Unacceptable character in personal name. Utility returns three formatted ASCII output (FAO) arguments including the illegal character, the length of the string, and the string address.

MAIL$_ILLPERNAM

Personal name formatted improperly. Returns an FAO argument containing the improperly formatted personal name.

MAIL$_ILLSUBDIR

Illegal subdirectory specification. Returns an FAO argument containing the subdirectory string.

MAIL$_INVITMCOD

The specified item code is invalid.

MAIL$_INVITMLEN

The specified item length is invalid.

MAIL$_MISREQITEM

The required item is missing.

MAIL$_NAMTOOBIG

Specified name exceeds 255-character limit.

MAIL$_NOTSUBDIR

No such subdirectory. Returns an FAO argument containing the subdirectory string.

MAIL$_NOSUCHUSR

No such user. Returns the name of the unfound user.

MAIL$_NOSYSNAM

Caller needs SYSNAM privileges.

MAIL$_NOSYSPRV

Caller needs system privileges.

SS$_ACCVIO

Access violation.

Chapter 17. National Character Set (NCS) Utility Routines

This chapter describes the National character set (NCS) utility routines. The NCS utility provides a common facility for defining and accessing collating sequences and conversion functions. Collating sequences are used to compare strings for sorting purposes. Conversion functions are used to derive an altered form of an input string based on an appropriate conversion algorithm.

17.1. Introduction to NCS Routines

Using NCS, you can formulate collating sequences and conversion functions and register them in an NCS library. The NCS routines provide a programming interface to NCS that lets you access the collating sequences and conversion functions from an NCS library for doing string comparisons.

Typically, NCS collating sequences are selective subsets of the multinational character set. They are used extensively in programming applications involving various national character sets. For example, a program might use the Spanish collating sequence to assign appropriate collating weight to characters from the Spanish national character set. Another program might use the French collating sequence to assign appropriate collating weight to characters in the French national character set.

In addition to providing program access to collating sequences and conversion functions in an NCS library, the NCS routines provide a means for saving definitions in a local file for subsequent use by the comparison and conversion routines.

17.1.1. List of NCS Routines

Table 17.1 lists the individual NCS routines.

Table 17.1. NCS Routines

Routine	Description
NCS$COMPARE	Compares two strings using a specified collating sequence as comparison basis.
NCS$CONVERT	Converts a string using the specified conversion function.
NCS$END_CF	Terminates the use of a conversion function by the calling program.
NCS$END_CS	Terminates the use of a collating sequence by the calling program.
NCS$GET_CF	Retrieves the definition of the named conversion function from the NCS library.
NCS$GET_CS	Retrieves the definition of the named collating sequence from the NCS library.
NCS$RESTORE_CF	Permits the calling program to restore the definition of a “saved” conversion function from a database or an OpenVMS RMS file.

Routine	Description
NCS$RESTORE_CS	Permits the calling program to restore the definition of a “saved” collating sequence from a database or an RMS file.
NCS$SAVE_CF	Provides the calling program with information that permits the application to store the definition of a conversion function in a local database or an RMS file.
NCS$SAVE_CS	Provides the calling program with information that permits the application to store the definition of a collating sequence in a local database or an RMS file.

17.1.2. Sample Application Process

In a typical application, the program does the following:

1. Prepares a string for comparison.
2. Makes a call to the NCS$GET routine, specifying the appropriate collating sequence.
3. Makes one or more calls to the NCS$COMPARE routine, which does the actual comparison.
4. Terminates the comparison with a call to the NCS$END routine.

The program can also include the use of conversion functions in preparation for the comparison routines.

17.2. Using the NCS Utility Routines: Examples

This section includes two examples of how to use NCS utility routines in program applications:

Example 17.1 illustrates the use of NCS utility routines in a VSI Fortran for OpenVMS program.

Example 17.1. Using NCS Routines in a VSI Fortran for OpenVMS Program

```
        PROGRAM NCS_EXAMPLE

        CHARACTER*80 CSSTRING,STRING1,STRING2
        INTEGER*4    CSLENGTH,LENGTH1,LENGTH2,CSID,STATUS,RESULT
        INTEGER*4    NCS$GET_CS,NCS$COMPARE,NCS$END_CS

        CHARACTER*1  CMP(3)

        CMP(1) = '<'
        CMP(2) = '='
        CMP(3) = '>'
C
C       Read the name of the collating sequence..
C
        WRITE (6,30)
        READ (5,15,END=999) CSLENGTH,CSSTRING
30      FORMAT(' Collating Sequence: ')
C
C       Get the collating sequence from the NCS library
C
        CSID = 0
```

```
        STATUS = NCS$GET_CS (CSID, CSSTRING(1:CSLENGTH))
        IF ((STATUS .AND. 1) .NE. 1) THEN
            CALL LIB$SIGNAL (%VAL(STATUS))
            ENDIF
C
C       Read two strings to be compared according to the collating sequence
C
100     WRITE (6,10)
        READ (5,15,END=999) LENGTH1,STRING1
        WRITE (6,20)
        READ (5,15,END=999) LENGTH2,STRING2

        IF (LENGTH1 .EQ. 0 .AND. LENGTH2 .EQ. 0) THEN
            GOTO 200
            ENDIF

10      FORMAT(' String1: ')
20      FORMAT(' String2: ')
15      FORMAT (q,a80)
C
C       Compare the strings
C
        result = ncs$compare (csid, string1(1:length1), string2(1:length2))
C
C       Display the results of the comparison
C
        WRITE (6,40) STRING1(1:LENGTH1), CMP(RESULT+2), STRING2(1:LENGTH2)
40      FORMAT(' ',A,' ',A,' ',A)
        GOTO 100
C
C       Come here if both inputs are blank -- we are done.
C       Call NCS$END_CS to free any storage used to hold the CS.
C
200     STATUS = NCS$END_CS (CSID)
        IF ((STATUS .AND. 1) .NE. 1) THEN
            CALL LIB$SIGNAL (%VAL(STATUS))
            ENDIF
        CALL EXIT

999     CONTINUE
        END
```

Example 17.2 illustrates the use of NCS routines in a VSI C for OpenVMS VAX program.

Note

Each programming language provides an appropriate mechanism for defining symbols, status codes, completion codes, and other relevant information.

Example 17.2. Using NCS Routines in a VSI C for OpenVMS VAX Program

```
/*
** ============================================================================
**
** NCS_EXAMPLE.C
**
** NCS conversion function example using the VAX C programming language
**
** ============================================================================
*/

/*
** ----------------------------------------------------------------------------
```

```
** Header files
*/
# include "sys$library:descrip.h"     /* Descriptor macros     */
# include "sys$library:rms.h"      /* RMS structure definitions    */
# include "sys$library:rmsdef.h"     /* RMS completion codes      */
# include "sys$share:ssdef.h"      /* System service completion    */
         /* codes       */
# include "sys$library:stdio.h"      /* Standard I/O definitions     */
/*
** --------------------------------------------------------------------------
** Data definitions
*/
#define SIZE 1024       /* Maximum record size     */

unsigned long int
    cfid,        /* Address of conversion     */
         /* function       */
    expected_status,       /* Expected return status     */
    rms_status,        /* RMS return status     */
    status;        /* Function return status     */

unsigned short int
    return_length;       /* Length of returned string in */
         /* bytes       */

char
    file[NAM$C_MAXRSS],       /* File name      */
    inrec[SIZE],       /* Input record      */
    outrec[SIZE];       /* Output record      */

$DESCRIPTOR(cfname_d,"EDT_VT2xx");     /* Conversion function name     */
         /* descriptor      */
$DESCRIPTOR(prompt_d,"_File:    ");     /* Prompt string descriptor     */
$DESCRIPTOR(file_d,file);      /* File name descriptor     */
$DESCRIPTOR(inrec_d,inrec);      /* Input record descriptor     */
$DESCRIPTOR(outrec_d,outrec);      /* Output record descriptor     */

struct FAB infab;       /* Input file access block     */
struct RAB inrab;       /* Input record access block    */
/*
** --------------------------------------------------------------------------
** Function prototypes
*/
void status_check();
/*
** ==========================================================================
*/
main ()
{
    /*
    ** ----------------------------------------------------------------------
    ** Initialize RMS user structures for the file.
    */
    infab  = cc$rms_fab;     /* Initialize to default FAB    */
         /* values       */

    infab.fab$l_fna = file;      /* Now supply our specific     */
         /* values       */
    infab.fab$b_fns = NAM$C_MAXRSS;

    inrab  = cc$rms_rab;     /* Initialize to default RAB    */
         /* values       */

    inrab.rab$l_fab = &infab;     /* Now supply our specific     */
         /* values       */
    inrab.rab$l_ubf = inrec;
```

```
   inrab.rab$w_usz = SIZE;
   /*
   ** ------------------------------------------------------------------------
   ** Get the EDT_VT2xx conversion function from the default NCS library
   */
   cfid = 0;        /* Initialize ID      */
   status = ncs$get_cf(&cfid,&cfname_d,0);
   status_check(status,SS$_NORMAL);
   /*
   ** ------------------------------------------------------------------------
   ** Get the file to be converted and set the length of the returned file
   ** name
   */
   status = lib$get_input(&file_d,&prompt_d,&return_length);
   status_check(status,SS$_NORMAL);
   file_d.dsc$w_length = return_length;
   /*
   ** ------------------------------------------------------------------------
   ** Open the input file to be converted and connect to the RAB
   */
   rms_status = sys$open(&infab,0,0);
   status_check(rms_status,RMS$_NORMAL);

   rms_status = sys$connect(&inrab,0,0);
   status_check(rms_status,RMS$_NORMAL);
   /*
   ** ------------------------------------------------------------------------
   ** Read each record from the file, convert the input string to EDT
   ** fallback, and write the result to the output
   */
   while(TRUE)
   {
/*
** ------------------------------------------------------------------------
** Read each record
*/
rms_status = sys$get(&inrab,0,0);
if (rms_status == RMS$_EOF)      /* Reached end of file  */
    break;
else
    status_check(rms_status,RMS$_NORMAL);   /* Read a record     */
/*
** ------------------------------------------------------------------------
** Call NCS$CONVERT to convert the input string to EDT fallback
**
** e.g. Convert form feed to <FF>, escape to <ESC>, et cetera
*/
inrec_d.dsc$w_length = inrab.rab$w_rsz;
status = ncs$convert(&cfid,&inrec_d,&outrec_d,&return_length);
status_check(status,SS$_NORMAL);
outrec_d.dsc$w_length = return_length;
/*
** ------------------------------------------------------------------------
** Write the result to the output, SYS$OUTPUT in this case
*/
status = lib$put_output(&outrec_d);
status_check(status,SS$_NORMAL);
outrec_d.dsc$w_length = SIZE;
   }
   /*
   ** ------------------------------------------------------------------------
   ** Close the input file.
   */
   rms_status = sys$close(&infab,0,0);
   status_check(rms_status,RMS$_NORMAL);
   /*
```

```
    ** ------------------------------------------------------------------------
    ** Free any storage used to hold the conversion function.
    */
    status = ncs$end_cf(&cfid);
    status_check(status,SS$_NORMAL);

}

void status_check(status,expected_status)
/*
** ========================================================================
**
** Checks the function return status against the one expected, and exits upon
** error. Otherwise, return to the main program.
**
** ========================================================================
*/

{
    if (status != expected_status)
 sys$exit(status);
    else
 return;
}
```

17.3. NCS Routines

This section describes the NCS routines.

Note that several routines contain the heading Condition Value Signaled to indicate that the condition value originates in another utility.

NCS$COMPARE

Compare Strings — The NCS$COMPARE routine compares two strings using a specified collating sequence as a comparison basis.

Format

```
NCS$COMPARE cs_id ,string_1 ,string_2
```

Returns

OpenVMS usage:	integer
type:	longword integer (signed)
access:	write only
mechanism:	by value

Longword condition value. Most routines return a condition value in R0, but the NCS$COMPARE routine uses R0 to return the result of the comparison, as shown in the following table:

Returned Value	Comparison Result
-	*string_1* is less than *string_2*
0	*string_1* is equal to *string_2*
1	*string_1* is greater than *string_2*

The NCS$COMPARE routine uses the Signaling Mechanism to indicate completion status as described under Condition Value Signaled.

Arguments

cs_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	read only
mechanism:	by reference

Address of a longword that NCS uses to identify a collating sequence. The *cs_id* argument is required and can be obtained by a call to the NCS$GET_CS routine.

All calls to the NCS$COMPARE routine and the call to the NCS$END_CS routine that terminates the comparison must pass this longword identifier. Upon completion, the NCS$END_CS routine releases the memory used to store the collating sequence and sets the value of the longword identifier to 0.

string_1

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Descriptor (length and address) of the first string.

string_2

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Descriptor of the second string.

Description

The NCS$COMPARE routine compares two strings using the specified collating sequence as the comparison basis. The routine indicates whether the value of the first string is greater than, less than, or equal to the value of the second string.

Condition Value Signaled

STR$_ILLSTRCLA

Illegal string class. Severe error. The descriptor of *string_1* or *string_2*, or both, contains a class code not supported by the OpenVMS Calling Standard.

NCS$CONVERT

Convert String — The NCS$CONVERT routine converts a string using the specified conversion function.

Format

```
NCS$CONVERT cf_id ,source ,dest [,ret_length] [,not_cvt]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

cf_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	read only
mechanism:	byreference

Address of a longword that NCS uses to identify a conversion function. The *cf_id* argument is required and can be obtained by a call to the NCS$GET_CF routine.

All calls to the NCS$CONVERT routine and the call to the NCS$END_CF routine that terminates the conversion must pass this longword identifier. Upon completion, the NCS$END_CF routine releases the memory used to store the conversion function and sets the value of the longword identifier to 0.

source

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Descriptor of source string.

dest

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Descriptor of destination string.

ret_length

OpenVMS usage:	word unsigned
type:	word (unsigned)
access:	write only
mechanism:	by reference

Length of converted string.

not_cvt

OpenVMS usage:	word unsigned
type:	word (unsigned)
access:	write only
mechanism:	by reference

Number of characters in the source string that were not fully converted.

Description

Using the specified conversion function, the NCS$CONVERT routine converts the source string and stores the result in the specified destination. Optionally, the calling program can request that the routine return the length of the converted string as well as the number of characters that were not fully converted.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

NCS$_NOT_CF

Name of identifier does not refer to a conversion function.

STR$_TRU

Successful completion. However, the resultant string was truncated because the storage allocation for the destination string was inadequate.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

Any value signaled by STR$COPY_DX or STR$ANALYZE_SDESC.

NCS$END_CF

End Conversion Function — The NCS$END_CF routine terminates a conversion function.

Format

```
NCS$END_CF cf_id
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

cf_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	modify
mechanism:	by reference

Address of a longword that NCS uses to store a nonzero value identifying a conversion function.

The *cf_id* argument is required.

Description

The NCS$END_CF routine indicates to NCS that the calling program no longer needs the conversion function. NCS releases the memory space allocated for the conversion function and sets the value of the longword identifier to 0.

Condition Values Returned

NCS$_NORMAL

Normal successful completion. The longword identifier value is set to 0.

NCS$_NOT_CF

Name of identifier does not refer to a conversion function.

NCS$END_CS

End Collating Sequence — The NCS$END_CS routine terminates a collating sequence.

Format

```
NCS$END_CS cs_id
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

cs_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	modify
mechanism:	by reference

Address of a longword that NCS uses to store a nonzero value identifying a collating sequence.

The *cs_id* argument is required.

Description

The NCS$END_CS routine indicates to NCS that the calling program no longer needs the collating sequence. NCS releases the memory space allocated for the collating sequence and sets the value of the longword identifier to 0.

Condition Values Returned

NCS$_NORMAL

Normal successful completion. The longword identifier value is set to 0.

NCS$_NOT_CS

Name of identifier does not refer to a collating sequence.

NCS$GET_CF

Get Conversion Function — The NCS$GET_CF routine retrieves the definition of the named conversion function from the NCS library.

Format

```
NCS$GET_CF cf_id [,cfname] [,librar]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

cf_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	modify
mechanism:	by reference

Address of a longword used by NCS to identify a conversion function. The calling program must ensure that the longword contains 0 before invoking the NCS$GET_CF routine because the routine stores a nonzero value in the longword. The nonzero value identifies the conversion function. All subsequent calls to the NCS$CONVERT routine and the call to the NCS$END_CF routine to terminate the conversion function pass the longword identifier. When it completes the conversion, the NCS$END_CF routine releases the memory used to store the conversion function and sets the value of the longword identifier to 0.

The conversion function identifier enhances modular programming and permits concurrent use of multiple conversion functions within a program.

The calling program should not attempt to interpret the contents of the longword identifier.

The *cf_id* argument is required.

cfname

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Name of the conversion function being retrieved.

librar

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Name of the library where the conversion function is stored.

Description

The NCS$GET_CF routine extracts the named conversion function from the specified NCS library.

If the calling program omits the *cfname* argument, an "identity" conversion function padded with NUL characters (hex 0) is provided. The identity conversion function effectively leaves each character unchanged by converting each character to itself. For example, A becomes A, B becomes B, C becomes C, and so forth.

If the calling program omits the *librar* argument, NCS accesses the default NCS library.

Condition Values Returned

NCS$_DIAG

Operation completed with signaled diagnostics.

NCS$_NOT_CF

Name of identifier does not refer to a conversion function.

NCS$_NOT_FOUND

Name of identifier not found in the NCS library.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

NCS$GET_CS

Get Collating Sequence — The NCS$GET_CS routine retrieves the definition of the named collating sequence from the NCS library.

Format

```
NCS$GET_CS cs_id [,csname] [,librar]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

cs_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)

access:	modify
mechanism:	by reference

Address of a longword that NCS uses to store a nonzero value identifying a collating sequence. The calling program must ensure that the longword identifier contains 0 before invoking the NCS$GET_CS routine.

All subsequent calls to the NCS$COMPARE routine and the call to the NCS$END_CS routine that terminates the use of the collating sequence must pass this longword identifier. Upon completion of the comparisons, the NCS$END_CS routine releases the memory used to store the collating sequence and sets the value of the longword identifier to 0.

The collating sequence identifier enhances modular programming and permits concurrent use of multiple collating sequences within a program.

The calling program should not attempt to interpret the contents of the longword identifier.

The *cs_id* argument is required.

csname

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Name of the collating sequence being retrieved.

librar

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

File specification of the library where the collating sequence is stored.

Description

The NCS$GET_CS routine extracts the named collating sequence from the specified NCS library.
If the calling program omits the *csname* argument, NCS creates a collating sequence that uses the "native" collating sequence as a basis for the comparisons. This collating sequence is padded with NUL characters (hex 0).

If the calling program omits the *librar* argument, NCS accesses the default NCS library.

Condition Values Returned

NCS$_DIAG

Operation completed with signaled diagnostics.

NCS$_NOT_CS

Name of identifier does not refer to a collating sequence.

NCS$_NOT_FOUND

Name of identifier not found in the NCS library.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

NCS$RESTORE_CF

Restore Conversion Function — The NCS$RESTORE_CF routine permits the calling program to restore the definition of a saved conversion function from a database or a file.

Format

```
NCS$RESTORE_CF cf_id [,length] [,address]
```

Returns

OpenVMS 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 under Condition Value Returned.

Arguments

cf_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	write only
mechanism:	by reference

Address of a longword that NCS uses to identify a conversion function.

The *cf_id* argument is required.

length

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only

mechanism:	by reference

Longword that the calling program uses to indicate the length of the conversion function being restored.

address

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Longword that the calling program uses as a pointer to the conversion function being restored.

Description

The NCS$RESTORE_CF routine, used in conjunction with the NCS$SAVE_CF routine, permits the application program to keep a local copy of the conversion function. The NCS$SAVE_CF routine obtains the length and location of the conversion function and returns it to the application program. The application program subsequently provides this information to the NCS$RESTORE_CF routine, which uses it to access the conversion function.

This routine also does some integrity checking on the conversion function as it is being processed.

Condition Value Returned

NCS$_NOT_CF

Name of identifier does not refer to a conversion function.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

NCS$RESTORE_CS

Restore Collating Sequence — The NCS$RESTORE_CS routine permits the calling program to restore the definition of a “saved” collating sequence from a database or a file.

Format

```
NCS$RESTORE_CS cs_id [,length] [,address]
```

Returns

OpenVMS 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 under Condition Value Returned.

Arguments

cs_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	write only
mechanism:	by reference

Address of a longword that NCS uses to identify a collating sequence.

The *cs_id* argument is required.

length

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Longword that the calling program uses to indicate the length of the collating sequence being restored.

address

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Longword that the calling program uses as a pointer to the collating sequence being restored.

Description

The NCS$RESTORE_CS routine, used in conjunction with the NCS$SAVE_CS routine, permits the application program to keep a local copy of the collating sequence. The NCS$SAVE_CS routine obtains the length and location of the collating sequence and returns it to the application program. The application program subsequently provides this information to the NCS$RESTORE_CS routine, which uses it to access the collating sequence.

This routine also does some integrity checking on the collating sequence as it is being processed.

Condition Value Returned

NCS$_NOT_CS

Name of identifier does not refer to a collating sequence.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

NCS$SAVE_CF

Save Conversion Function — The NCS$SAVE_CF routine provides the calling program with information that permits the application to store the definition of a conversion function in a local database or a file rather than in the NCS library.

Format

```
NCS$SAVE_CF cf_id [,length] [,address]
```

Returns

OpenVMS 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 under Condition Value Returned.

Arguments

cf_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	read only
mechanism:	by reference

Address of a longword that NCS uses to identify a conversion function.

The *cf_id* argument is required.

length

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Longword used to store the length of the specified conversion function.

address

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Longword used to store the address of the specified conversion function.

Description

The NCS$SAVE_CF routine, used in conjunction with the NCS$RESTORE_CF routine, permits the application program to store a conversion function definition in a local file or in a database. When the calling program specifies the conversion function identifier, NCS returns the location of the definition and its length in bytes, permitting the calling program to store the definition locally, rather than in an NCS library. Subsequently, the application supplies this information to the NCS$RESTORE_CF routine, which restores the conversion function to a form that can be used by the NCS$CONVERT routine.

This routine also does some integrity checking on the conversion function as it is being processed.

Condition Value Returned

NCS$_NOT_CF

Name of identifier does not refer to a conversion function.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

NCS$SAVE_CS

Save Collating Sequence — The NCS$SAVE_CS routine provides the calling program with information that permits the application program to store the definition of a collating sequence in a database or a file rather than in the NCS library.

Format

```
NCS$SAVE_CS cs_id [,length] [,address]
```

Returns

OpenVMS 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 under Condition Value Returned.

Arguments

cs_id

OpenVMS usage:	identifier
type:	longword integer (unsigned)
access:	read only

mechanism:	by reference

Address of a longword that NCS uses to identify a collating sequence.

The *cs_id* argument is required.

length

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Longword that NCS uses to indicate the length of the specified collating sequence to the calling program.

address

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Longword that NCS uses to indicate the address of the specified collating sequence to the calling program.

Description

The NCS$SAVE_CS routine, used in conjunction with the NCS$RESTORE_CS routine, permits the application program to store a collating sequence definition in a local file or in a database. When the calling program specifies the collating sequence identifier, NCS returns the location of the definition sequence and its length in bytes, permitting the calling program to store the definition locally, rather than in a library. Subsequently, the application supplies this information to the NCS$RESTORE_CS routine, which restores the collating sequence to a form that can be used by the NCS$COMPARE routine.

This routine also does some integrity checking on the collating sequence as it is being processed.

Condition Value Returned

NCS$_NOT_CS

Name of identifier does not refer to a collating sequence.

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.

Chapter 18. Print Symbiont Modification (PSM) Routines

The print symbiont modification (PSM) routines allow you to modify the behavior of the print symbiont supplied with the operating system.

18.1. Introduction to PSM Routines

The print symbiont processes data for output to standard line printers and printing terminals by performing the following functions:

- Reading the data from disk
- Formatting the data
- Sending the data to the printing device
- Composing separation pages (flag, burst, and trailer pages) and inserting them into the data stream for printing

Some of the reasons for modifying the print symbiont include the following:

- To include additional information on the separation pages (flag, burst, and trailer) or to format them differently
- To filter and modify the data stream sent to the printer
- To change some of the ways that the symbiont controls the printing device

You might not always be able to modify the print symbiont to suit your needs. For example, you cannot modify the:

- Symbiont's control logic or the sequence in which the symbiont calls routines
- Interface between the symbiont and the job controller

If you cannot modify the print symbiont to suit your needs, you can write your own symbiont. However, VSI recommends that you modify the print symbiont rather than write your own.

The rest of this chapter contains the following information about PSM routines:

- Section 18.2 contains an overview of the print symbiont and of symbionts in general. It explains concepts such as “symbiont streams”; describes the relationship between a symbiont, a device driver, and the job controller; and gives an overview of the print symbiont's internal logic.

  This section is recommended for those who want to either modify the print symbiont or write a new symbiont.

- Section 18.3 details the procedure for modifying the print symbiont. It includes an overview of the entire procedure, followed by a detailed description of each step.

- Section 18.4 contains an example of a simple modification to the print symbiont.

- Section 18.5 describes each PSM routine and the interface used by the routines you substitute for the standard PSM routines.

18.2. Print Symbiont Overview

The operating system supplies two symbionts: a print symbiont, which is an *output* symbiont, and a card reader, which is an *input* symbiont. An output symbiont receives tasks from the job controller, whereas an input symbiont sends jobs to the job controller. The card reader symbiont cannot be modified. You can modify the print symbiont, described in this section, using PSM routines.

There are two types of output symbiont: device and server. A device symbiont processes data for output to a device, for example, a printer. A server symbiont also processes data but not necessarily for output to a device, for example, a symbiont that copies files across a network. The operating system supplies no server symbionts.

18.2.1. Components of the Print Symbiont

The print symbiont includes the following major components:

- PSM routines that are used to modify the print symbiont
- Routines that implement input, format, and output services in the print symbiont
- Routines that implement the internal logic of the print symbiont

The print symbiont is implemented using the Symbiont Services facility. This facility provides communication and control between the job controller and symbionts through a set of Symbiont/Job Controller Interface routines (SMB routines), which are documented in Chapter 19.

All of these routines are contained in a shareable image with the file specification SYS $SHARE:SMBSRVSHR.EXE.

18.2.2. Creation of the Print Symbiont Process

The print symbiont is a device symbiont, receiving tasks from the job controller and processing them for output to a printing device. In the operating system, the existence of a print symbiont process is linked to the existence of at least one print execution queue that is started.

The job controller creates the print symbiont process by calling the $CREPRC system service; it does this whenever either of the following conditions occurs:

- A print execution queue is started (from the stopped state) and no symbiont process is running the image specified with the START/QUEUE command.

  A print execution queue is started by means of the DCL command START/QUEUE. Use the /PROCESSOR qualifier with the START/QUEUE command to specify the name of the symbiont image that is to service an execution queue; if you omit /PROCESSOR, then the default symbiont image is PRTSMB.

- Currently existing symbiont processes suited to a print execution queue cannot accept additional devices; that is, the symbionts have no more available streams. In such a case, the job controller creates another print symbiont process. The next section discusses symbiont streams.

The print symbiont process runs as a detached process.

18.2.3. Symbiont Streams

A **stream** is a logical link between a print execution queue and a printing device. When the queue is started (by means of START/QUEUE), the job controller creates a stream linking the queue with a symbiont process. Because each print execution queue has a single associated printing device (specified with the /ON= *device* qualifier in the INITIALIZE/QUEUE or START/QUEUE command), each stream created by the job controller links a print execution queue, a symbiont process, and the queue's associated printer.

A symbiont that can support multiple streams simultaneously (that is, multiple print execution queues and multiple devices) is termed a multithreaded symbiont. The job controller enforces an upper limit of 16 on the number of streams that any symbiont can service simultaneously.

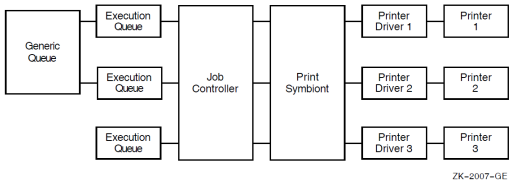
Therefore, in the operating system environment, only one print symbiont process is needed as long as the number of print execution queues (and associated printers) does not exceed 16. If there are more than 16 print execution queues, the job controller creates another print symbiont process.

The print symbiont is, therefore, a multithreaded symbiont that can service as many as 16 queues and devices, and you can modify it to service any number of queues and devices as long as the number is less than or equal to 16.

A symbiont stream is “active” when a queue is started on that stream. The print symbiont maintains a count of active streams. It increments this count each time a queue is started and decrements it when a queue is stopped with the DCL command STOP/QUEUE/NEXT or STOP/QUEUE/RESET. When the count falls to zero, the symbiont process exits. The symbiont does not decrement the count when the queue is paused by STOP/QUEUE.

Figure 18.1 shows the relationship of generic print queues, execution print queues, the job controller, the print symbiont, printer device drivers, and printers. The lines connecting the boxes denote streams.

Figure 18.1. Multithreaded Symbiont



18.2.4. Symbiont and Job Controller Functions

This section compares the roles of the symbiont and job controller in the execution of print requests. You issue print requests using the PRINT command.

The job controller uses the information specified on the PRINT command line to determine the following:

- Which queue to place the job in (/QUEUE, /REMOTE, /LOWERCASE, and /DEVICE)
- How many copies to print (/COPIES and /JOB_COUNT)
- Scheduling constraints for the job (/PRIORITY, /AFTER, /HOLD, /FORM, /CHARACTERISTICS, and /RESTART)

- How and whether to display the status of jobs and queues (/NOTIFY, /OPERATOR, and /IDENTIFY)

The print symbiont, on the other hand, interprets the information supplied with the qualifiers that specify this information:

- Whether to print file separation pages (/BURST, /FLAG, and /TRAILER)
- Information to include when printing the separation pages (/NAME and /NOTE)
- Which pages to print (/PAGES)
- How to format the print job (/FEED, /SPACE, and /PASSALL)
- How to set up the job (/SETUP)

The print symbiont, not the job controller, performs all necessary device-related functions. It communicates with the printing device driver. For example, when a print execution queue is started (by means of START/QUEUE/ON= *device*) and the stream is established between the queue and the symbiont, the symbiont parses the device name specified by the /ON qualifier in the START/QUEUE command, allocates the device, assigns a channel to it, obtains the device characteristics, and determines the device class. In versions of the operating system prior to Version 4.0, the job controller performed these functions.

The print symbiont's output routine returns an error to the job controller if the device class is neither printer nor terminal.

18.2.5. Print Symbiont Internal Logic

The job controller deals with units of work called jobs, while the print symbiont deals with units of work called tasks. A print job can consist of several print tasks. Thus, in the processing of a print job, the job controller's role is to divide a print job into one or more print tasks, which the symbiont can process. The symbiont reports the completion of each task to the job controller, but the symbiont contains no logic to determine that the print job as a whole is complete.

In the processing of a print task, the symbiont performs three basic functions: input, format, and output. The symbiont performs these functions by calling routines to perform each function.

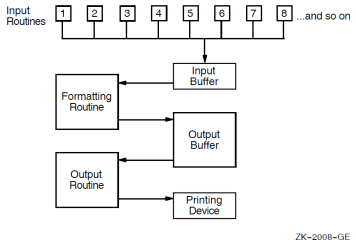
The following steps describe the action taken by the symbiont in processing a task:

1. The symbiont receives the print request from the job controller and stores it in a message buffer.
2. The symbiont searches its list of input routines and selects the first input routine that is applicable to the print task.
3. The input routine returns a data record to the symbiont's input buffer or in a buffer supplied by the input routine.
4. Data in the input buffer is moved to the symbiont's output buffer by the formatting routines, which format it in the process.
5. Data in the output buffer is sent to the printing device by the output routine.
6. When an input routine completes execution, that is, when it has no more input data to process, the symbiont selects another applicable input routine. Steps 3, 4, and 5 are repeated until all applicable input routines have executed.

7. The symbiont informs the job controller that the task is complete.

Figure 18.2 illustrates the steps taken by the symbiont in the processing of a print task.

Figure 18.2. Symbiont Execution Sequence or Flow of Control



As Figure 18.2 shows, most of the input routines execute in a specified sequence. This sequence is defined by the symbiont's main control routine. You cannot modify this main control routine; thus, you cannot modify the sequence in which symbiont routines are called.

The input routines that do not execute in sequence are called “demand input routines.” These routines are called whenever the service they provide is required and include the page header, page setup, and library module input routines.

The symbiont can perform input, formatting, and output functions asynchronously; that is, the order in which the symbiont calls the input, formatting, and output routines can vary. For example, the symbiont can call an input routine, which returns a record to the input buffer; it can then call the format routine, which moves that record to the output buffer; and then it can call the output routine to move that data to the printing device. This sequence results in the movement of a single data record from disk to printing device.

On the other hand, the symbiont can call the input and formatting routines several times before calling the output routine for a single buffer. The buffer can contain one or more formatted input records. In some cases an output buffer might contain only a portion of an input record.

In this way the symbiont can store input records; then call the format routine, which moves one of those records to the output buffer; and finally call the output routine, which moves that data to the printing device. Note, however, that the formatting routine must be called once for each input record.

Similarly, the symbiont can store several formatted records before calling the output routine to move them to the printing device.

The symbiont requires this flexibility in altering the sequence in which input, format, and output routines are called for reasons of efficiency (high rate of throughput) and adaptability to various system parameters and system events.

The value specified with the call to PSM$PRINT determines the maximum size of the symbiont's output buffer, which cannot be larger than the value of the system parameter MAXBUF. If the buffer is very small, the symbiont might need to call its output routine one or more times for each record formatted. If the buffer is large, the symbiont stores several formatted records before calling the output routine to move them to the printing device.

18.3. Symbiont Modification Procedure

To modify the print symbiont, perform the following steps. These steps are described in more detail in the sections that follow.

1. Determine the modification needed. The modification might involve changing the way the symbiont performs a certain function, or it might involve adding a new function.

2. Determine where to make the modification. This involves selecting a function and determining where that function is performed within the symbiont's execution sequence. You specify a function by calling the PSM$REPLACE routine and specifying the code that identifies the function.

   Some codes correspond to symbiont-supplied routines. When you specify one of these codes, you replace that routine with your routine. Other codes do not correspond to symbiont-supplied routines. When you specify one of these codes, you add your routine to the set of routines the symbiont executes. Table 18.1 lists these codes.

3. Write the routine. Because the symbiont calls your routine, your routine must have one of three call interfaces, depending on whether it is an input, format, or output routine. See the descriptions of the USER-INPUT-ROUTINE, USER-FORMAT-ROUTINE, and USER-OUTPUT-ROUTINE routines, which follow the descriptions of the PSM routines.

4. Write the symbiont-initialization routine. This routine executes when the symbiont is first activated by the job controller. It initializes the symbiont's internal database; specifies, by calling PSM $REPLACE, the routines you have supplied; activates the symbiont by calling PSM$PRINT; and performs any necessary cleanup operations when PSM$PRINT completes.

5. Construct the modified symbiont. This involves compiling your routines, then linking them.

6. Integrate the modified symbiont with the system. This involves placing the executable image in SYS $SYSTEM, identifying the symbiont image to the job controller, and debugging the symbiont.

As mentioned previously, you identify each routine you write for the symbiont by calling the PSM $REPLACE routine. The `code` argument for this routine specifies the point within the symbiont's execution sequence at which you want your routine to execute. You should know which code you will use to identify your routine before you begin to write the routine. Section 18.3.6 provides more information about these codes.

18.3.1. Guidelines and Restrictions

The following guidelines and restrictions apply to the writing of any symbiont routine:

- Do not use the process-permanent files identified by the logical names SYS$INPUT, SYS $OUTPUT, SYS$ERROR, and SYS$COMMAND.

- The symbiont code should be linked against SMBSRVSHR.EXE in order to define the following status codes:

  - PSM$_FLUSH

  - PSM$_FUNNOTSUP

  - PSM$_PENDING

  - PSM$_SUSPEND

  - PSM$_EOF

  - PSM$_BUFFEROVF

  - PSM$_NEWPAGE
  - PSM$_ESCAPE
  - PSM$_INVVMSOSC
  - PSM$_MODNOTFND
  - PSM$_NOFILEID
  - PSM$_OSCTOOLON
  - PSM$_TOOMANYLEV
  - PSM$_INVITMCOD
  - PSM$_LATSYM
- Do not use the system services $HIBER and $WAKE.
- The job completion (PSM$K_JOB_COMPLETION) and output (PSM$K_OUTPUT) routines are not replaceable when using the LAT protocol option.
- Use the following two OpenVMS Run-Time Library routines for allocation and deallocation of memory: LIB$GET_VM and LIB$FREE_VM.
- Minimize the amount of time that your routine spends executing at AST level. The job controller sends messages to the symbiont by means of user-mode ASTs; the symbiont cannot receive these ASTs while your user routine is executing at AST level.
- The symbiont can call your routines at either AST level or non-AST level.
- If your routine returns any error-condition value (low bit clear), the symbiont aborts the current task and notifies the job controller. Note that, by default, an error-condition value returned during the processing of a task causes the job controller to abort the entire job. However, this default behavior can be overridden. See the description of the /RETAIN qualifier of the DCL commands START/QUEUE, INITIALIZE/QUEUE, and SET QUEUE in the *VSI OpenVMS DCL Dictionary*.

  The symbiont stores the first error-condition value (low bit clear) returned during the processing of a task. The symbiont's file-errors routine, an input routine (code PSM$K_FILE_ERRORS), places the message text associated with this condition value in the symbiont's input stream. The symbiont prints this text at the end of the listing, immediately before the trailer pages.

  The symbiont sends this error-condition value to the job controller; the job controller then stores this condition value with the job record in the job controller's queue file. The job controller also writes this condition value in the accounting record for the job.

  If you choose to return a condition value when an error occurs, you should choose one from the system message file. This lets system programs access the message text associated with the condition value. Specifically, the Accounting and SHOW/QUEUE utilities and the job controller will be able to translate the condition value to its corresponding message text and to display this message text as appropriate.

  This guideline applies to input, input-filter, and output-filter routines, and to the symbiont's use of dynamic string descriptors in these routines.

The simplest way for an input routine to pass the data record to the symbiont is for it to use a Run-Time Library string-handling routine (for example, STR$COPY_R). These routines use dynamic string descriptors to point to the record they have handled and to copy that record from your input buffer to the symbiont-supplied buffer specified in the *funcdesc* argument.

By default, the symbiont initializes a dynamic string descriptor that your input routine can use to describe the data record it returns. Specifically, the symbiont initializes the DSC$B_DTYPE field of the string descriptor with the value DSC$K_DTYPE_T (which indicates that the data to which the descriptor points is a string of characters) and initializes the DSC$B_CLASS field with the value DSC$K_CLASS_D (which indicates that the descriptor is dynamic).

Alternatively, the input routine can pass a data record to the symbiont by providing its own buffer and passing a static string descriptor that describes the buffer. To do this, you must redefine the fields of the descriptor to which the *funcdesc* argument points, as follows:

1. Initialize the field DSC$B_CLASS with the value DSC$K_CLASS_S (which indicates that the descriptor points to a scalar value or a fixed-length string).

2. Initialize the field DSC$A_POINTER with the address of the buffer that contains the data record.

3. Initialize the field DSC$W_LENGTH with the length, in bytes, of the data record.

Each time the symbiont calls the routine to read some data, the symbiont reinitializes the descriptor to make it a dynamic descriptor. Consequently, if you want to use the descriptor as a static descriptor, your input routine must initialize the descriptor each time it is called to perform a reading operation.

Input-filter routines and output-filter routines return a data record to the symbiont by means of the *func_desc_2* argument. The symbiont initializes a descriptor for this argument the same way it does for descriptors used by the input routine. Thus, the guidelines described for the input routine apply to the input-filter routine and output-filter routine.

18.3.2. Writing an Input Routine

This section provides an overview of the logic used in the print symbiont's main input routine, and it discusses the way in which the print symbiont handles carriage-control effectors.

The print symbiont calls your input routine, supplying it with arguments. Your routine must return arguments and condition values to the print symbiont. For this reason, your input routine must use the interface described in the description of the USER-INPUT-ROUTINE.

When the print symbiont calls your routine, it specifies a particular request in the *func* argument. Each function has a corresponding code.

Your routine must provide the functions identified by the codes PSMK_OPEN, PSMK_READ, and PSM$K_CLOSE. Your routine need not respond to the other function codes, but it can if you want it to. If your routine does not provide a function that the symbiont requests, it must return the condition value PSM$_FUNNOTSUP to the symbiont.

The description of the *func* argument of the USER-INPUT-ROUTINE describes the codes that the symbiont can send to an input routine.

See Section 18.3.5 for additional information about other function codes used in the user-written input routine.

For each task that the symbiont processes, it calls some input routines only once, and some more than once; it always calls some routines and calls others only when needed.

Table 18.1 lists the codes that you can specify when you call the PSM$REPLACE routine to identify your input routine to the symbiont. The description of the PSM$REPLACE routine describes these routines.

18.3.2.1. Internal Logic of the Symbiont's Main Input Routine

The internal logic of the symbiont's main input routine, as described in this section, is subject to change without notice. This logic is summarized here. This summary is not intended as a tutorial on the writing of a symbiont's main input routine, although it does provide insight into such a task.

A main input routine is one that the symbiont calls to read data from the file that is to be printed. A main input routine must perform three sets of tasks: one set when the symbiont calls the routine with an OPEN request, one set when the symbiont calls with a READ request, and one set when the symbiont calls with a CLOSE request.

The following table lists the codes that identify each of these three requests and describes the tasks that the symbiont's main input routine performs for each request:

Code	**Action Taken by the Input Routine**
PSM$K_OPEN	An OPEN request. When the main input routine receives this request code, it does the following: 1. Opens the input file. 2. Stores information about the input file. 3. Returns the type of carriage control used in the input file. If this routine cannot open the file, it returns an error.
	Note that the print symbiont's main input routine performs these tasks when it receives the PSM$K_START_TASK function code, rather than the PSM$K_OPEN function code. This atypical behavior occurs because some of the information stored by the main input routine must be available for other input routines that execute before the main input routine. For example, information about file attributes and record formats is needed by the symbiont's separation-page routines, which print flag and burst pages. Consequently, if you supply your own main input routine, some of the information about the file being printed that appears on the standard separation pages is not available, and the symbiont prints a message on the separation page stating so. The symbiont receives the file-identification number from the job controller in the SMBMSG$K_FILE_IDENTIFICATION item of the requesting message and uses this value rather than the file specification to open the main input file.
PSM$K_READ	A READ request. When the main input routine receives this request, it returns the next record from the file. In addition, when the carriage

Code	Action Taken by the Input Routine
	control used by the data file is PSM$K_CC_PRINT, the main input routine returns the associated record header.
PSM$K_CLOSE	A CLOSE request. When the main input routine receives this request, it closes the input file.

18.3.2.2. Symbiont Processing of Carriage Control

Each input record can be thought of as consisting of three parts: leading carriage control, data, and trailing carriage control. Taken together, these three parts are called the composite data record.

Leading and trailing carriage control are determined by the type of carriage control used in the file and explicit carriage-control information returned with each record. For embedded carriage control, however, leading and trailing carriage control is always null.

The type of carriage control returned by the main input routine on the PSM$K_OPEN request code determines, for that invocation of the input routine, how the symbiont applies carriage control to each record that the main input routine returns on the PSM$K_READ request code.

Note that, for all four carriage control types, the first character returned on the first PSM$K_READ call to an input routine receives special processing. If that character is a line feed or a form feed and if the symbiont is currently at line 1, column 1 of the current page, then the symbiont discards that line feed or form feed.

The Four Types of Carriage Control

The following table briefly describes each type of carriage control and how the symbiont's main input routine processes it. For a detailed explanation of each type of carriage control, refer to the description of the FAB$B_RAT field of the FAB block in the *VSI OpenVMS Record Management Services Reference Manual*.

Type of Carriage Control	Symbiont Processing
Embedded	Leading and trailing carriage control are embedded in the data portion of the input record. Therefore, the symbiont supplies no special carriage control processing; it assumes that leading and trailing carriage control are null.
Fortran	The first byte of each data record contains a Fortran carriage-control character. This character specifies both the leading and trailing carriage control for the data record. The symbiont extracts the first byte of each data record and interprets that byte as a Fortran carriage-control character. If the data record is empty, the symbiont generates a leading carriage control of line feed and a trailing carriage control of carriage return.
PRN	Each data record contains a 2-byte header that contains the carriage-control specifier. The first byte specifies the carriage control to apply before printing the data portion of the record. The second byte specifies the carriage control to apply after printing the data portion. The abbreviation PRN stands for print-file format. Unlike other types of carriage control, PRN carriage control information is returned through the `funcarg` argument of the main input routine; this occurs with the PSM$K_READ request. The `funcarg` argument specifies a longword; your routine writes the

Type of Carriage Control	Symbiont Processing
	2-byte PRN carriage control specifier into the first two bytes of this longword.
Implied	The symbiont provides a leading line feed and a trailing carriage return. But if the data record consists of a single form feed, the symbiont sets to null the leading and trailing carriage control for that record, and the leading carriage control for the record that follows it.

18.3.3. Writing a Format Routine

To write a format routine, follow the modification procedure described in Section 18.3. Do not replace the symbiont's main format routine. Instead, modify its action by writing input and output filter routines. These execute immediately before and after the main format routine, respectively. The main formatting routine uses an undocumented and nonpublic interface; you cannot replace the main formatting routine. The DCL command PRINT/PASSALL bypasses the main format routine of the print symbiont.

See Section 18.3.5 for additional information about other function codes used in the user-written formatting routine.

18.3.3.1. Internal Logic of the Symbiont's Main Format Routine

The main format routine contains all the logic necessary to convert composite data records to a data stream for output. Actions taken by the format routine include the following:

- Tracking the current column and line
- Implementing the special processing of the first character of the first record
- Implementing the alignment data mask specified by the DCL command START/QUEUE/ALIGN=MASK
- Handling margins as specified by the forms definition
- Initiating processing of page headers when specified by the DCL command PRINT/HEADER
- Expanding leading and trailing carriage control
- Handling line overflow
- Handling page overflow
- Expanding tab characters to spaces for some devices
- Handling escape sequences
- Accumulating accounting information
- Implementing double-spacing when specified by the DCL command PRINT/SPACE
- Implementing automatic page ejection when specified by the DCL command PRINT/FEED

The symbiont's main format routine uses a special rule when processing the first character of the first composite data record returned by an input routine. (A composite data record is the input data record and a longword that contains carriage-control information for the input data record.) This rule is that if the first character is a vertical format effector (form feed or line feed) and if the symbiont has processed no printable characters on the current page (that is, the current position is column 1, line 1), then that vertical format effector is discarded.

18.3.4. Writing an Output Routine

To write an output routine, follow the modification procedure described in Section 18.3.

The print symbiont calls your output routine. Input arguments are supplied by the print symbiont; output arguments and status values are returned by your routine to the print symbiont. For this reason, your output routine must have the call interface that is described in the USER-OUTPUT-ROUTINE routine.

When the print symbiont calls your routine, it specifies in one of the input arguments—the *func* argument—the reason for the call. Each reason has a corresponding function code.

There are several function codes that the print symbiont can supply when it calls your output routine. Your routine must contain the logic to respond to the following function codes: PSM$K_OPEN, PSM K_WRITE, PSMK_WRITE_NOFORMAT, and PSM$K_CLOSE.

It is not required that your output routine contain the logic to respond to the other function codes, but you can provide this logic if you want to.

A complete list and description of all relevant function codes for output routines is provided in the description of the *func* argument of the USER-OUTPUT-ROUTINE routine.

See Section 18.3.5 for additional information about other function codes.

18.3.4.1. Internal Logic of the Symbiont's Main Output Routine

When the symbiont calls the main output routine with the PSM$K_OPEN function code, the main output routine takes the following steps:

1. Allocates the print device
2. Assigns a channel to the device
3. Obtains the device characteristics
4. Returns the device-status longword in the *funcarg* argument (for more information, see the description of the SMBMSG$K_DEVICE_STATUS message item in Chapter 19)
5. Returns an error if the device is not a terminal or a printer

When this routine receives a PSM$K_WRITE service request code, it sends the contents of the symbiont output buffer to the device for printing.

When this routine receives a PSM$K_WRITE_NOFORMAT service request code, it sends the contents of the symbiont output buffer to the device for printing and suppresses device drive formatting as appropriate for the device in use.

When this routine receives a PSM$K_CANCEL service request code, it requests the device driver to cancel any outstanding output operations.

When this routine receives a PSM$K_CLOSE service request code, it deassigns the channel to the device and deallocates the device.

18.3.5. Other Function Codes

A status PSM$_PENDING might not be returned whenever the symbiont notifies user-written input, output, and format routines using the following message function codes:

Function Code	Description
PSM$K_START_STREAM	Job controller sends a message to the symbiont to start a queue
PSM$K_START_TASK	Symbiont parses a message from job controller directing it to start a queue
PSM$K_PAUSE_TASK	Job controller sends a message to the symbiont to suspend processing of the current task
PSM$K_STOP_STREAM	Job controller sends a message to the symbiont to stop the queue
PSM$K_STOP_TASK	Job controller sends a message to the symbiont to stop the task
PSM$K_RESUME_TASK	Job controller sends a message to the symbiont to resume processing of the current task
PSM$K_RESET_STREAM	Same as PSM$K_STOP_STREAM

18.3.6. Writing a Symbiont Initialization Routine

Writing a symbiont initialization routine involves writing a program that calls the following:

1. PSM$REPLACE once for each routine (input, output, or format) that you have written. PSM $REPLACE identifies your routines to the symbiont.

2. PSM$PRINT exactly once after you have identified all your service routines using PSM$REPLACE.

Table 18.1 lists all routine codes that you can specify in the PSM$REPLACE routine. Choosing the correct routine code is important because the code specifies when the symbiont will call your routine. The functions of these routines are described further in the description of the PSM$REPLACE routine.

For those input routines that execute in a predefined sequence, the second column contains a number showing the order in which that input routine is called relative to the other input routines for a single file job. If the routine does not execute in a predefined sequence, the second column contains the character x.

Column three specifies whether the routine is an input, format, or output routine; this information directs you to the section describing how to write a routine of that type.

Column four specifies whether there is a symbiont-supplied routine corresponding to that routine code. The codes for the input-filter and output-filter routines, which have no corresponding routines in the symbiont, allow you to specify new routines for inclusion in the symbiont.

Table 18.1. Routine Codes for Specification to PSM$REPLACE

Routine Code	Sequence	Function	Supplied
PSM$K_JOB_SETUP	1	Input	Yes
PSM$K_FORM_SETUP	2	Input	Yes
PSM$K_JOB_FLAG	3	Input	Yes
PSM$K_JOB_BURST	4	Input	Yes
PSM$K_FILE_SETUP	5	Input	Yes
PSM$K_FILE_FLAG	6	Input	Yes
PSM$K_FILE_BURST	7	Input	Yes

Routine Code	Sequence	Function	Supplied
PSM$K_FILE_SETUP_2	8	Input	Yes
PSM$K_MAIN_INPUT	9	Input	Yes
PSM$K_FILE_INFORMATION	10	Input	Yes
PSM$K_FILE_ERRORS	11	Input	Yes
PSM$K_FILE_TRAILER	12	Input	Yes
PSM$K_JOB_RESET	13	Input	Yes
PSM$K_JOB_TRAILER	14	Input	Yes
PSM$K_JOB_COMPLETION [1]	15	Input	Yes
PSM$K_PAGE_SETUP	x	Input	Yes
PSM$K_PAGE_HEADER	x	Input	Yes
PSM$K_LIBRARY_INPUT	x	Input	Yes
PSM$K_INPUT_FILTER	x	Formatting	No
PSM$K_MAIN_FORMAT	x	Formatting	Yes
PSM$K_OUTPUT_FILTER	x	Formatting	No
PSM$K_OUTPUT [1]	x	Output	Yes

[1]The job completion (PSM$K_JOB_COMPLETION) and output (PSM$K_OUTPUT) routines are not replaceable when using the LAT protocol option.

18.3.7. Integrating a Modified Symbiont

To integrate your user routine and the symbiont initialization routine, perform the following steps; note that the sequence of steps described here assumes that you will be debugging the modified symbiont:

1. Compile or assemble the user routine and the symbiont initialization routine into an object module.
2. Enter the following DCL command:

   ```
   $ LINK/DEBUG your-symbiont
   ```

   The file name *your-symbiont* is the object module built in Step 1. Symbols necessary for this link operation are located in the shareable images SYS$SHARE:SMBSRVSHR.EXE and SYS $LIBRARY:IMAGELIB.EXE. The linker automatically searches these shareable images and extracts the necessary information.
3. Place the resulting executable symbiont image in SYS$SYSTEM.
4. Locate two unallocated terminals: one at which to issue DCL commands and one at which to debug the symbiont image.
5. Log in on one of the terminals under UIC [1,4], which is the system manager's account. This terminal is the one at which you enter DCL commands. Do not log in at the other terminal.
6. Enter the following DCL command:

   ```
   $ SET TERMINAL/NODISCONNECT/PERMANENT _TTcu:
   ```

   The variable `_TTcu:` is the physical terminal name of the terminal at which you want to debug (the terminal at which you are not logged in). You must specify the underscore (_) and colon (:) characters.

7. Enter the following DCL commands:

```
$ DEFINE/GROUP DBG$INPUT  _TTcu:
$ DEFINE/GROUP DBG$OUTPUT _TTcu:
```

   The variable _TTcu: specifies the physical terminal name of the terminal at which you will be debugging. Note that other users having a UIC with group number 1 should not use the debugger at the same time.

8. Initialize the queue by entering the following DCL command:

```
$ INITIALIZE/QUEUE/PROCESSOR= your-symbiont /ON= printer_name
```

   The symbiont image specified by the file name *your-symbiont* must reside in SYS$SYSTEM. Note too that the /PROCESSOR qualifier accepts only a file name; the device, directory, and file type default to SYS$SYSTEM:.EXE.

   The /ON qualifier specifies the device that will be served by the symbiont while you debug the symbiont.

9. Enter the following DCL command to execute the modified symbiont routine:

```
$ PRINT/HEADER/QUEUE=queue-id
```

   Enter the following DCL command to start the queue and invoke the debugger:

```
$ START/QUEUE queue-name
```

10. After you debug your symbiont, relink the symbiont by entering the following DCL command:

```
$ LINK/NOTRACEBACK/NODEBUG your-symbiont
```

11. Deassign the logical names DBG$INPUT and DBG$OUTPUT so that they will not interfere with other users in UIC group 1.

18.4. Using the PSM Routines: An Example

Example 18.1 shows how to use PSM routines to supply a page header routine in a VAX MACRO program.

Example 18.1. Using PSM Routines to Supply a Page Header Routine in a VAX MACRO Program

```
        .TITLE  EXAMPLE - Example user modified symbiont
        .IDENT  'V03-000'

;++
;       THIS PROGRAM SUPPLIES A USER WRITTEN PAGE HEADER
;       ROUTINE TO THE STANDARD SYMBIONT.  THE PAGE HEADER
;       INCLUDES THE SUBMITTER'S ACCOUNT NAME AND USER NAME,
;       THE FULL FILE SPECIFICATION, AND THE PAGE NUMBER.
;       THE HEADER LINE IS UNDERLINED BY A ROW OF DASHES
;       PRINTED ON A SECOND HEADER LINE.
;--
        .LIBRARY  /SYS$LIBRARY:LIB.MLB/
;
; System definitions
;
        $PSMDEF                                 ; Symbiont definitions
        $SMBDEF                                 ; Message item definitions
```

```
        $DSCDEF                                 ; Descriptor definitions

;
; Define argument offsets for user supplied services called by symbiont
;
        CONTEXT         = 04                    ; symbiont context
        WORK_AREA       = 08                    ; user context
        FUNC            = 12                    ; function code
        FUNC_DESC       = 16                    ; function dependent descriptor
        FUNC_ARG        = 20                    ; function dependent argument

;
; Macro to create dynamic descriptors
;
        .MACRO  D_DESC
                .WORD   0                       ; DSC$W_LENGTH = 0
                .BYTE   DSC$K_DTYPE_T           ; DSC$B_DTYPE = STRING
                .BYTE   DSC$K_CLASS_D           ; DSC$B_CLASS = DYNAMIC
                .LONG   0                       ; DSC$A_POINTER = 0
        .ENDM

;
; Storage for page header information
;
        FILE:           D_DESC                  ; file name descriptor
        USER:           D_DESC                  ; user name descriptor
        ACCOUNT:        D_DESC                  ; account name descriptor

        PAGE:           .LONG   0               ; page number
        LINE:           .LONG   0               ; line number

;
; FAO control string and work buffer.  Header format:
;       "[account,name] filename ........ Page 9999"
;
        FAO_Ctrl:       .ASCID  /!71<[!AS, !AS] !AS!>Page 9999/
        FAO_Ctrl_2:     .ASCID  /!4UL/
        FAO_DESC:       .LONG   80              ; work buffer descriptor
                        .ADDRESS FAO_BUFF
        FAO_BUFF:       .BLKB   80              ; work buffer
;
; Own storage for values passed by reference
;
        CODE:           .LONG   0               ; service or item code
        STREAMS:        .LONG   1               ; number of simultaneous streams
        BUFSIZ:         .LONG   2048            ; output buffer size
        LINSIZ:         .WORD   81              ; line size for underlines

;
; Main routine -- invoked at image startup
;
START:  .WORD   0       ; save nothing because this routine uses only R0 and R1


;
; Supply private page header routine
;
        MOVZBL  #PSM$K_PAGE_HEADER,CODE         ; set the service code
        PUSHAL  HEADER                          ; address of modified routine
        PUSHAL  CODE                            ; address of service code
        CALLS   #2,G^PSM$REPLACE                ; replace the routine
        BLBC    R0,10$                          ; exit if any errors

;
; Transfer control to the standard symbiont
;
```

```
        PUSHAL  BUFSIZ                          ; address of output buffer size
        PUSHAL  STREAMS                         ; address of number of streams
        CALLS   #2,G^PSM$PRINT                  ; invoke standard symbiont
10$:    RET



;
; Page header routine
;
HEADER: .WORD   0                               ; save nothing

;
; Check function code
;
        CMPL    #PSM$K_START_TASK,@FUNC(AP)     ; new task?
        BEQL    20$                             ; branch if so
        CMPL    #PSM$K_READ,@FUNC(AP)           ; READ function?
        BNEQ    15$
        BRW     50$                             ; branch if so
15$:    CMPL    #PSM$K_OPEN, @FUNC(AP)          ; OPEN function?
        BNEQ    16$
        BRW     66$                             ; branch if so
16$:    MOVL    #PSM$_FUNNOTSUP,R0              ; unsupported function
        RET                                     ; return to symbiont
;
; Starting a new file
;
20$:
        CLRL    PAGE                            ; reset the page number
        MOVZBL  #2,LINE                         ; and the line number

;
; Get the account name
;
        MOVZBL  #SMBMSG$K_ACCOUNT_NAME,CODE     ; set item code
        PUSHAL  ACCOUNT                         ; address of descriptor
        PUSHAL  CODE                            ; address of item code
        PUSHAL  @CONTEXT(AP)                    ; address of symbiont ctx value
        CALLS   #3,G^PSM$READ_ITEM_DX           ; read it
        BLBC    R0,40$                          ; branch if any errors

;
; Get the file name
;
        MOVZBL  #SMBMSG$K_FILE_SPECIFICATION,CODE       ; set item code
        PUSHAL  FILE                            ; address of descriptor
        PUSHAL  CODE                            ; address of item code
        PUSHAL  @CONTEXT(AP)                    ; address of symbiont ctx value
        CALLS   #3,G^PSM$READ_ITEM_DX           ; read it
        BLBC    R0,40$                          ; branch if any errors

;
; Get the user name
;
        MOVZBL  #SMBMSG$K_USER_NAME,CODE        ; set item code
        PUSHAL  USER                            ; address of descriptor
        PUSHAL  CODE                            ; address of item code
        PUSHAL  @CONTEXT(AP)                    ; address of symbiont ctx value
        CALLS   #3,G^PSM$READ_ITEM_DX           ; read it
        BLBC    R0,40$                          ; branch if any errors

;
; Set up the static header information that is constant for the task
;
        $FAO_S  CTRSTR  = FAO_Ctrl, -           ; FAO control string desc
```

```
                OUTBUF  = FAO_DESC, -           ; output buffer descriptor
                P1      = #ACCOUNT, -           ; account name descriptor
                P2      = #USER, -              ; user name descriptor
                P3      = #FILE                 ; file name descriptor
        BLBC    R0,40$                          ; branch if any errors
        MOVL    #PSM$_FUNNOTSUP,R0              ; unsupported function
40$:    RET                                     ; return usupported status or error
;
; Read a page header
;
50$:
        DECL    LINE                            ; decrement the line number
        BEQL    60$                             ; branch if second read
        BLSS    70$                             ; branch if third read

;
; Insert the page number into the header
;
        INCL    PAGE                            ; increment the page number
        MOVAB   FAO_BUFF+76,FAO_DESC+4          ; point to page number buffer
        $FAO_S  CTRSTR  = FAO_Ctrl_2, -         ; FAO control string desc
                OUTBUF  = FAO_DESC, -           ; output buffer descriptor
                P1      = PAGE                  ; page number
        MOVAB   FAO_BUFF,FAO_DESC+4             ; point to work buffer
        BLBC    R0,55$                          ; return if error

;
; Copy the line to the symbiont's buffer
;
        PUSHAB  FAO_DESC                        ; work buffer descriptor
        PUSHL   FUNC_DESC(AP)                   ; symbiont descriptor
        CALLS   #2,G^STR$COPY_DX                ; copy to symbiont buffer
55$:    RET                                     ; return success or any error

;
; Second line -- underline header
;
60$:
        PUSHL   FUNC_DESC(AP)                   ; symbiont descriptor
        PUSHAL  LINSIZ                          ; number of bytes to reserve
        CALLS   #2,G^STR$GET1_DX                ; reserve the space
        BLBC    R0,67$                          ; exit if error
        MOVL    FUNC_DESC(AP),R1                ; get address of descriptor
        MOVL    4(R1),R1                        ; get address of buffer
        MOVAB   80(R1),R0                       ; set up transfer limit
65$:    MOVB    #^A/-/,(R1)+                    ; fill with dashes
        CMPL    R0,R1                           ; reached limit?
        BGTRU   65$                             ; branch if not
        MOVB    #10,(R1)+                       ; extra line feed
66$:    MOVZBL  #SS$_NORMAL,R0                  ; set success
67$:    RET                                     ; return

;
; Done with this page header
;
70$:
        MOVL    #PSM$_EOF,R0                    ; return end of input
        MOVZBL  #2,LINE                         ; reset line counter
        RET                                     ; return

        .END    START
```

18.5. PSM Routines

This section describes the individual PSM routines.

PSM$PRINT

Invoke OpenVMS-Supplied Print Symbiont — The PSM$PRINT routine invokes the OpenVMS-supplied print symbiont. PSM$PRINT must be called exactly once after all user service routines have been specified using PSM$REPLACE.

Format

```
PSM$PRINT [streams] [,bufsiz] [,worksiz] [,maxqios] [,options]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

streams

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Maximum number of streams that the symbiont is to support. The *streams* argument is the address of a longword containing this number, which must be in the range of 1 to 16. If you do not specify *streams*, a default value of 1 is used. Thus, by default, a user-modified symbiont supports one stream, which is to say that it is a single-threaded symbiont.

A stream (or thread) is a logical link between a print execution queue and a printing device. When a symbiont process can accept simultaneous links to more than one queue, that is, when it can service multiple queues simultaneously, the symbiont is said to be multithreaded.

bufsiz

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Maximum buffer size in bytes that the print symbiont is to use for output operations. The *bufsiz* argument is the address of a longword containing the specified number of bytes.

The print symbiont actually uses a buffer size that is the smaller of: (1) the value specified by *bufsiz* or (2) the system parameter MAXBUF. If you do not specify *bufsiz*, the print symbiont uses the value of MAXBUF.

The print symbiont uses this size limit only for output operations. Output operations involve the placing of processed or formatted pages into a buffer that will be passed to the output routine.

The print symbiont uses the value specified by *bufsiz* only as an upper limit; most buffers that it writes will be smaller than this value.

worksiz

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Size in bytes of a work area to be allocated for the use of user routines. The *worksiz* argument is the address of a longword containing this size in bytes. If you do not specify *worksiz*, no work area is allocated.

A separate area of the specified size is allocated for each active symbiont stream.

maxqios

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Specifies the maximum number of outstanding $QIOs that a print symbiont stream using the LAT protocol may generate. Set symbiont process quotas large enough to handle the maximum number of QIOs multiplied by the number of streams, using a number between 2 and 32. For normal printing capabilities, the suggested quota is 10; for high-speed printing, use a larger number.

options

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Longword bit vector that specifies the LAT protocol option using the PSM$M_LAT_PROTOCOL symbolic value. Note that using the LAT_PROTOCOL option carries the following restrictions:

- Replacement of the output and job completion routines will be overridden
- Output device must be a LAT device

Description

The PSM$PRINT routine must be called exactly once after all user routines have been specified to the print symbiont. Each user routine is specified to the symbiont in a call to the PSM$REPLACE routine.

The PSM$PRINT routine allows you to specify whether the print symbiont is to be single-threaded or multithreaded, and if multithreaded, how many streams or threads it can have. In addition, this routine allows you to control the maximum size of the output buffer.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

This routine also returns any condition values returned by the $SETPRV, $GETSYI, $PURGWS, and $DCLAST system services, as well as any condition values returned by the SMB$INITIALIZE routine documented in Chapter 19.

PSM$READ_ITEM_DX

PSM$READ_ITEM_DX — The PSM$READ_ITEM_DX routine obtains the value of message items that are sent by the job controller and stored by the symbiont.

Format

PSM$READ_ITEM_DX request_id , item , buffer

Returns

OpenVMS 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 under Condition Values Returned.

Argument

request_id

OpenVMS usage:	**address**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Request identifier supplied by the symbiont to the user routine currently calling PSM $READ_ITEM_DX. The symbiont always supplies a request identifier when it calls a user routine with a service request. The **request_id** argument is the address of a longword containing this request identifier value.

Your user routine must copy the request identifier value that the symbiont supplies (in the **request_id** argument) when it calls your user routine. Then, when your user routine calls PSM$READ_ITEM_DX, it must supply (in the **request_id** argument) the address of the request identifier value that it copied.

item

OpenVMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Item code that identifies the message item that PSM$READ_ITEM_DX is to return. The **item** argument is the address of a longword that specifies the item's code.

For a complete list and description of each item code, refer to the documentation of the **item** argument in the SMB$READ_MESSAGE_ITEM routine in Chapter 19.

buffer

OpenVMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

Buffer into which PSM$READ_ITEM_DX returns the specified informational item. The **buffer** argument is the address of a descriptor pointing to this buffer.

The PSM$READ_ITEM_DX routine returns the specified informational item by copying that item to the buffer using one of the STR$COPY_xx routines documented in the *OpenVMS RTL String Manipulation (STR$) Manual*.

Description

The PSM$READ_ITEM_DX routine obtains the value of message items that are sent by the job controller and stored by the symbiont. Use PSM$READ_ITEM_DX to obtain information about the task currently being processed, for example, the name of the file being printed (SMBMSG $K_FILE_SPECIFICATION) or the name of the user who submitted the job (SMBMSG $K_USER_NAME).

Condition Values Returned

SS$_NORMAL

Normal successful completion.

PSM$_INVITMCOD

Invalid item code specified in the **item** argument.

This routine also returns any condition values returned by any of the STR$COPY_xx routines documented in the *OpenVMS RTL String Manipulation (STR$) Manual*.

PSM$REPLACE

Declare User Service Routine — The PSM$REPLACE routine substitutes a user service routine for a symbiont routine or adds a user service routine to the set of symbiont routines. You must call PSM $REPLACE once for each routine that you replace or add.

Format

```
PSM$REPLACE code ,routine
```

Returns

OpenVMS 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 under Condition Value Returned.

Arguments

code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Routine code that identifies the symbiont routine to be replaced by a user service routine. The *code* argument is the address of a longword containing the routine code.

Some routine codes identify routines that are supplied with the symbiont; when you specify such a routine code, you replace the symbiont-supplied routine with your service routine.

Two routine codes identify routines that are not supplied with the symbiont; when you specify such a routine code, your service routine is added to the set of symbiont routines.

Table 18.1 lists each routine code in the order in which it is called within the symbiont execution stream; this table also specifies whether a routine code identifies an input, formatting, or output routine and whether the routine is supplied with the symbiont.

Each programming language provides an appropriate mechanism for defining these routine codes. The following pages list each routine code in alphabetical order; the description of each code includes the following information about its corresponding routine:

- Whether the routine is supplied by the symbiont
- Whether the routine is an input, formatting, or output routine

- Under what conditions the routine is called
- What task the routine performs

Routine Codes

[PSM$K_FILE_BURST]

This code identifies a symbiont-supplied input routine; it is called whenever a file burst page is requested. This routine obtains information about the job, formats the file burst page, and returns the contents of the page to the input buffer. A file burst page follows a file flag page and precedes the contents of the file.

[PSM$K_FILE_ERRORS]

This code identifies a symbiont-supplied input routine; it is called when errors have occurred during the job. This routine places the error message text in the input buffer.

[PSM$K_FILE_FLAG]

This code identifies a symbiont-supplied input routine; it is called whenever a file flag page is requested. This routine obtains information about the job, formats the file flag page, and returns the contents of the page to the input buffer. A flag page follows the job burst page (if any) and precedes the file burst page (if any). It contains such information as the file specification of the file and the name of the user issuing the print request.

[PSM$K_FILE_INFORMATION]

This code identifies a symbiont-supplied input routine; it is called when the file information item has been specified by the job controller. This routine expands the file information item to text and returns it to the input buffer.

[PSM$K_FILE_SETUP]

This code identifies a symbiont-supplied input routine; it is always called. This routine queues any specified file-setup modules for insertion in the input stream when the PSM$K_FILE_SETUP routine closes.

[PSM$K_FILE_SETUP_2]

This code identifies a symbiont-supplied input routine; it is always called. This routine returns a form feed to ensure that printing of the file begins at the top of the page. This routine is called just before the main input routine.

[PSM$K_FILE_TRAILER]

This code identifies a symbiont-supplied input routine; it is called whenever a file trailer page is requested. This routine obtains information about the job, formats the file trailer page, and returns the contents of the page to the input buffer. A trailer page follows the last page of the file contents.

[PSM$K_MAIN_FORMAT]

This code identifies the symbiont-supplied formatting routine; it is always called. This routine performs numerous formatting functions. You cannot replace this routine.

[PSM$K_FORM_SETUP]

This code identifies a symbiont-supplied input routine; it is always called. This routine queues any specified form-setup modules for insertion in the input stream when the PSM$K_FORM_SETUP routine closes.

[PSM$K_INPUT_FILTER]

This code identifies a format routine that is not supplied by the symbiont. If the routine is supplied by the user, it is always called immediately prior to the symbiont-supplied formatting routine (routine code PSM$K_MAIN_FORMAT). An input-filter service routine is useful for modifying input data records and their carriage control before they are formatted by the symbiont.

[PSM$K_JOB_BURST]

This code identifies a symbiont-supplied input routine; it is called whenever a job burst page is requested. This routine obtains information about the job, formats the job burst page, and returns the contents of the page to the input buffer. A job burst page follows the job flag page and precedes the file flag page (if any) of the first file in the job. It is similar to a file burst page except that it appears only once per job and only at the beginning of the job.

[PSM$K_JOB_COMPLETION]

This code identifies a symbiont-supplied input routine that returns a form feed, which causes any output stored by the device to be printed. The routine is always called. It cannot be replaced when using the LAT protocol option.

[PSM$K_JOB_FLAG]

This code identifies a symbiont-supplied input routine; it is called whenever a job flag page is requested. This routine obtains information about the job, formats the job flag page, and returns the contents of the page to the input buffer. A job flag page is similar to a file flag page except that it appears only once per job, preceding the job burst page (if any).

[PSM$K_JOB_RESET]

This code identifies a symbiont-supplied input routine; it is always called. This routine queues any specified job-reset modules for insertion in the input stream when the PSM$K_JOB_RESET routine closes.

[PSM$K_JOB_SETUP]

This code identifies a symbiont-supplied input routine; it is always called. This routine checks to see if this is the first job to be printed on the device, and if so, it issues a form feed and then performs a job reset. See the description of the PSM$K_JOB_RESET routine for information about job reset.

[PSM$K_JOB_TRAILER]

This code identifies a symbiont-supplied input routine; it is called whenever a job trailer page is requested. This routine obtains information about the job, formats the job trailer page, and returns the contents of the page to the input buffer. A job trailer page is similar to a file trailer page except that it appears only once per job, as the last page in the job.

[PSM$K_MAIN_INPUT]

This code identifies a symbiont-supplied input routine; it is always called. This routine opens the file to be printed, returns input records to the input buffer, and closes the file.

[PSM$K_LIBRARY_INPUT]

This code identifies a symbiont-supplied input routine; it is called when an input routine closes and when modules have been requested for insertion in the input stream. This routine returns the contents of the specified modules, one record per call. You cannot replace this routine.

[PSM$K_OUTPUT_FILTER]

This code identifies a formatting routine that is not supplied by the symbiont. If the routine is supplied by the user, it is always called. This routine executes prior to the symbiont output routine (routine code PSM$K_OUTPUT). An output-filter service routine is useful for modifying output data buffers before they are passed to the output routine.

At the point where the output-filter routine executes within the symbiont execution stream, the input data is no longer in record format; instead, the data exists as a stream of characters. The carriage control, for example, is embedded in the data stream. Thus, the output buffer might contain what was once a complete record, part of a record, or several records.

[PSM$K_PAGE_HEADER]

This code identifies a symbiont-supplied input routine; it is called once at the beginning of each page if page headers are requested. This routine returns to the input buffer one or more lines containing information about the file being printed and the current page number. This routine is called only while the main input routine is open.

[PSM$K_PAGE_SETUP]

This code identifies a symbiont-supplied routine; it is called at the beginning of each page if page-setup modules were specified. This routine queues any specified page-setup modules for insertion in the input stream when the PSM$K_PAGE_SETUP routine closes. This routine is called only while the main input routine is open.

[PSM$K_OUTPUT]

This code identifies the symbiont-supplied output routine that writes the contents of the output buffer to the printing device, together with many other functions. This routine is always called. It cannot be replaced when using the LAT protocol option.

Routine

OpenVMS usage:	procedure
type:	procedure value
access:	read only
mechanism:	by reference

User service routine that is to replace a symbiont routine or to be included. The *routine* argument is the address of the user routine entry point.

Description

The PSM$REPLACE routine must be called each time a user service routine replaces a symbiont routine or is added to a set of symbiont routines.

The code argument specifies the symbiont routine to be replaced. The routine codes that can be specified in the *code* argument are of two types: those that identify existing print symbiont routines and those that do not. All the routine codes are similar, however, in the sense that each supplies a location within the print symbiont execution stream where your routine can execute.

By selecting a routine code that identifies an existing symbiont routine, you effectively disable that symbiont routine. The service routine that you specify might or might not perform the function that the disabled symbiont routine performs. If it does not, the net effect of the replacement is to eliminate that function from the list of functions performed by the print symbiont. Exactly what your service routine does is up to you.

By selecting a routine code that does not identify an existing symbiont routine (those that identify the input-filter and output-filter routines), your service routine has a chance to execute at the location signified by the routine code. Because the service routine you specify to execute at this location does not replace another symbiont routine, your service routine is an addition to the set of symbiont routines.

As mentioned, each routine code identifies a location in the symbiont execution stream, whether or not it identifies a symbiont routine. Table 18.1 lists each routine code in the order in which the location it identifies is reached within the symbiont execution stream.

Condition Value Returned

SS$_NORMAL

Normal successful completion.

PSM$REPORT

Report Completion Status — The PSM$REPORT routine reports to the print symbiont the completion status of an asynchronous operation initiated by a user routine. Such a user routine must return the completion status PSM$_PENDING. PSM$REPORT must be called exactly once for each time a user routine returns the status PSM$_PENDING.

Format

```
PSM$REPORT request_id [,status]
```

Returns

OpenVMS 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 under Condition Value Returned.

Arguments

request_id

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Request identifier supplied by the symbiont to the user routine at the time the symbiont called the user routine with the service request. The user routine must return the completion status PSM$_PENDING on the call for this service request. The *request_id* argument is the address of a longword containing the request identifier value.

The symbiont calls the user routine with a request code that specifies the function that the symbiont expects the user routine to perform. In the call, the symbiont also supplies a request identifier, which serves to identify the request. If the user routine initiates an asynchronous operation, a mechanism is required for notifying the symbiont that the asynchronous operation has completed and for providing the completion status of the operation.

The PSM$REPORT routine conveys the above two pieces of information. In addition, PSM$REPORT returns to the symbiont (in the *request_id* argument) the same request identifier value as that supplied by the symbiont to the user routine that initiated the operation. In this way, the symbiont synchronizes the completion status of an asynchronous operation with that invocation of the user routine that initiated the operation.

Any user routine that initiates an asynchronous operation must, therefore, copy the request identifier value that the symbiont supplies (in the *request_id* argument) when it calls the user routine. The user routine will later need to supply this value to PSM$REPORT.

In addition, when the user routine returns, which it does before the asynchronous operation has completed, the user routine must return the status PSM$_PENDING.

status

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Completion status of the asynchronous operation that has completed. The *status* argument is the address of a longword containing this completion status. The *status* argument is optional; if it is not specified, the symbiont assumes the completion status SS$_NORMAL.

The user routine that initiates the asynchronous operation must test for the completion of the operation and must supply the operation's completion status as the *status* argument to the PSM$REPORT routine. The Description section describes this procedure in greater detail.

If the completion status specified by *status* has the low bit clear, the symbiont aborts the task.

Description

An asynchronous operation is an operation that, once initiated, executes “off to the side” and need not be completed before other operations can begin to execute. Asynchronous operations are common in symbiont applications because a symbiont, if it is multithreaded, must handle concurrent I/O operations.

One example of a user routine that performs an asynchronous operation is an output routine that calls the $QIO system service to write a record to the printing device. When the user output routine completes execution, the I/O request queued by $QIO might not have completed. In order to synchronize this I/O request, that is, to associate the I/O request with the service request that initiated it, you should use the following mechanism:

1. In making the call to $QIO, specify the *astadr* and *iosb* arguments. The *astadr* argument specifies an AST routine to execute when the queued output request has completed, and the *iosb* argument specifies an I/O status block to receive the completion status of the I/O operation. Step 3 describes some functions that your AST routine will need to do.

2. Have the user output routine return the status PSM$_PENDING.

3. Write the AST routine to perform the following functions:

   a. Copy the completion status word from the I/O status block to a longword location that you will specify as the *status* argument in the call to PSM$REPORT.

   b. Call PSM$REPORT. Specify as the *request_id* argument the request identifier that was supplied by the print symbiont in the original call to the user output routine.

Condition Value Returned

SS$_NORMAL

Normal successful completion.

USER-FORMAT-ROUTINE

Invoke User-Written Format Routine — The user-written USER-FORMAT-ROUTINE performs format operations. The symbiont's control logic routine calls your format routine at one of two possible points within the symbiont's execution stream. You select this point by specifying one of two routine codes when you call the PSM$REPLACE routine. A user format routine can be an input filter routine (routine code PSM$K_INPUT_FILTER) or an output filter routine (routine code PSM$K_OUTPUT_FILTER). The main format routine (routine code PSM$K_MAIN_FORMAT) cannot be replaced. A user format routine must use the call interface described here.

Format

```
USER-FORMAT-ROUTINE request_id ,work_area ,func ,func_desc_1 ,func_arg_1
                    ,func_desc_2 ,func_arg_2
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

request_id

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Request identifier supplied by the symbiont when it calls your format routine. The *request_id* argument is the address of a longword containing this request identifier value.

work_area

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Work area supplied by the symbiont for the use of your format routine. The symbiont supplies the address of this area when it calls your routine. The *work_area* argument is a longword containing the address of the work area. The work area is a section of memory that your format routine can use for buffering and other internal operations.

The size of the work area allocated is specified by the *work_size* argument in the PSM$PRINT routine. If you do not specify *work_size* in the call to PSM$PRINT, no work area is allocated.

In a multithreaded symbiont, a separate work area is allocated for each thread. This work area is shared by all user routines. The work area is initialized to zero when the symbiont is first started.

func

OpenVMS usage:	function_code
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Function code specifying the service that the symbiont expects your format routine to perform. The *func* argument is the address of a longword into which the symbiont writes this function code.

The function code specifies the reason the symbiont is calling your format routine or, in other words, the service that the symbiont expects your routine to perform at this time.

The PSM$K_FORMAT function code is the only one to which your format routine must respond. When the symbiont calls your format routine with this function code, your routine must move a record from the input buffer to the output buffer.

The symbiont can call your format routine with other function codes. Your routine should return the status PSM$_FUNNOTSUP (function not supported) when it is called with any of the following function codes or with any undocumented function code. When the status PSM$_FUNNOTSUP is

returned, the symbiont performs its normal action as if no format routine were supplied. To suppress the symbiont's normal action, you should return SS$_NORMAL.

PSM$K_START_STREAM	PSM$K_STOP_STREAM
PSM$K_START_TASK	PSM$K_PAUSE_TASK
PSM$K_RESUME_TASK	PSM$K_STOP_TASK
PSM$K_RESET_STREAM	

These function codes correspond to message items , which are discussed in more detail in Section 18.3.5, sent by the job controller to the symbiont.

Other function codes correspond to internal symbiont mechanisms that are not part of the public interface to the print symbiont.

Your format routine should return the status PSM$_FUNNOTSUP or SS$_NORMAL when it is called with a message function code or with a private function code.

func_desc_1

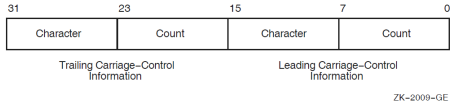
OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Descriptor supplying an input record to be processed by the format routine. The *func_desc_1* argument is the address of a string descriptor. By using this argument, the symbiont supplies the input record that your format routine is to process. Because this descriptor can be of any valid string type, your format routine should use the Run-Time Library string routines to analyze this descriptor and to manipulate the input record.

func_arg_1

OpenVMS usage:	vector_byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Carriage control for the input record supplied by *func_desc_1*. The *func_arg_1* argument is the address of a 4-byte vector that specifies the carriage control for the input record. The following diagram depicts the format of this 4-byte vector:



Bytes 0 and 1 describe the leading carriage control to apply to the input data record; bytes 2 and 3 describe the trailing carriage control.

Byte 0 is a number specifying the number of times the carriage control specifier in byte 1 is to be repeated preceding the input data record. Byte 2 is a number specifying the number of times the carriage control specifier in byte 3 is to be repeated following the input data record.

For values of the carriage control specifier from 1 to 255, the specifier is the ASCII character to be used as carriage control. Value *0* represents the ASCII “newline” sequence. Newline consists of a carriage return followed by a linefeed.

The *func_arg_1* argument is not used if your format routine is an output filter routine (routine code PSM$K_OUTPUT_FILTER). See the Description section for more information.

func_desc_2

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by reference

Descriptor of a buffer to which your format routine writes the formatted output record. The *func_desc_2* argument is the address of a string descriptor.

Your format routine must return the formatted data record by using the *func_desc_2* argument.

Your format routine should use the Run-Time Library string routines to write into the buffer specified by this descriptor.

func_arg_2

OpenVMS usage:	vector_byte_unsigned
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Carriage control for the output record returned in *func_desc_2*. The *func_arg_2* argument is the address of a 4-byte vector that specifies the carriage control for the output record. See the description of *func_arg_1* for the contents and format of this 4-byte vector.

If you do not process the carriage-control information supplied in *func_arg_1*, then you should copy that value into *func_arg_2*. Otherwise, the carriage-control information will be lost.

The *func_arg_2* argument is not used if your format routine is an output filter routine (routine code PSM$K_OUTPUT_FILTER). See the Description section help topic for more information.

Description

When used, the *func_arg_1* argument describes carriage-control information for the input data record, and the *func_arg_2* argument describes carriage-control information for the output data record.

The input data record is passed to the format routine (input filter or output filter) for processing, and the output data record is returned by the format routine (input filter or output filter).

One of the tasks performed by the main format routine (routine code PSM$K_MAIN_FORMAT) is that of embedding the carriage-control information (specified by *func_arg_1*) into the data record (specified by *func_desc_1*). Thus, the output data (specified by *func_desc_2*) contains

embedded carriage control and is thus no longer in record format; it is, therefore, properly referred to as an output data stream rather than an output data record.

Similarly, the output filter routine (routine code PSM$K_OUTPUT_FILTER), which executes after the main format routine, uses neither the *func_arg_1* nor *func_arg_2* argument; the data it receives (via *func_desc_1*) and the data it returns (via *func_desc_2*) are data streams, not data records.

However, the input filter routine (routine code PSM$K_INPUT_FILTER), which executes before the main format routine, uses both *func_arg_1* and *func_arg_2*. This is so because the main format routine has not yet executed, and so the carriage control information has not yet been embedded in the data record.

Condition Values Returned

SS$_NORMAL

Successful completion. The user format routine has completed the function that the symbiont requested.

PSM$_FUNNOTSUP

Function not supported. The user format routine does not support or does not recognize the function code supplied by the symbiont. To ensure future compatibility, your format routine should return this status for any unrecognized status codes.

This routine also returns any error condition values that you have coded your format routine to return. Refer to Section 18.3.1 for more information about error condition values.

USER-INPUT-ROUTINE

Invoke User-Written Input Routine — The user-written USER-INPUT-ROUTINE performs input operations. The symbiont calls your routine at a specified point in its execution stream; you specify this point using the PSM$REPLACE routine.

Format

```
USER-INPUT-ROUTINE request_id ,work_area ,func ,funcdesc ,funcarg
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

request_id

OpenVMS usage:	address

type:	longword (unsigned)
access:	read only
mechanism:	by reference

Request identifier value supplied by the symbiont when it calls your input routine. The *request_id* argument is the address of a longword containing this request identifier value.

If your input routine initiates an asynchronous operation (for example, a call to the $QIO system service), your input routine must copy the request identifier value specified by *request_id* because this value must later be passed to the PSM$REPORT routine. See the description of the PSM$REPORT routine for more information.

work_area

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Work area supplied by the symbiont for the use of your input routine. The symbiont supplies the address of this area when it calls your routine. The *work_area* argument is a longword into which the symbiont writes the address of the work area. The work area is a section of memory that your input routine can use for buffering and for other internal operations.

The size of the work area allocated is specified by the *work_size* argument in the PSM$PRINT routine. If you do not specify *work_size* in the call to PSM$PRINT, no work area is allocated.

In a multithreaded symbiont, a separate work area is allocated for each thread. This work area is shared by all user routines. The work area is initialized to zero when the symbiont is first started.

func

OpenVMS usage:	function_code
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Function code supplied by the symbiont when it calls your input routine. The *func* argument is the address of a longword containing this code.

The function code specifies the reason the symbiont is calling your input routine or, in other words, the function that the symbiont expects your routine to perform at this time.

Most function codes require or allow additional information to be passed in the call by means of the *funcdesc* and *funcarg* arguments. The description of each input function code, therefore, includes a description of how these two arguments are used with that function code.

Following is a list of all the function codes that the symbiont can specify when it calls your input routine (function codes applicable only to format and output routines are explained in the descriptions of the USER-FORMAT-ROUTINE and USER-OUTPUT-ROUTINE, respectively); all function codes are defined by the $PSMDEF macro.

Function Codes for Input Routines

[PSM$K_CLOSE]

When the symbiont calls your routine with this function code, your routine must terminate processing by releasing any resources it might have allocated.

The symbiont calls your routine with PSM$K_CLOSE when (1) your routine returns from a PSM $K_READ function call with the status PSM$_EOF (end of input) or with any error condition, or (2) the symbiont receives a task-abortion request from the job controller.

In any event, the symbiont always calls your input routine with PSM$K_CLOSE if your routine returns successfully from a PSM$K_OPEN function call. This guaranteed behavior ensures that any resources your routine might have allocated on the OPEN will be released on the CLOSE.

[PSM$K_GET_KEY]

Typically, the use of both the PSM$K_GET_KEY and PSMK$K_POSITION_TO_KEY function codes is appropriate only for a main input routine (routine code PSM$K_MAIN_INPUT).

When the symbiont calls your routine with this function code, your routine can do one of two things: (1) return PSM$_FUNNOTSUP (function not supported) or (2) return an input marker string to the symbiont.

If your routine returns PSM$_FUNNOTSUP to this function code, then your routine must also return PSM$_FUNNOTSUP if the symbiont subsequently calls your routine with the PSM $K_POSITION_TO_KEY function code. By returning PSM$_FUNNOTSUP, your routine is choosing not to respond to the symbiont request.

If your routine chooses to respond to the PSM$K_GET_KEY function code, your routine must return an input marker string to the symbiont; this input marker string identifies the input record that your input routine most recently returned to the symbiont. Subsequently, when the symbiont calls your input routine with the PSM$K_POSITION_TO_KEY function code, the symbiont passes your input routine one of the input marker strings that your input routine has returned on a previous PSM$K_GET_KEY function call. Using this marker string, your input routine must position itself so that, on the next PSM $K_READ call from the symbiont, your input routine will return (or reread) the input record identified by the marker string.

Coding your input routine to respond to PSM$K_GET_KEY and PSM$K_POSITION_TO_KEY allows the modified symbiont to perform the file-positioning functions specified by the DCL commands START/QUEUE/FORWARD, START/QUEUE/ALIGN, START/QUEUE/TOP_OF_FILE, START/ QUEUE/SEARCH, and START/QUEUE/BACKWARD. These file positioning functions also depend on the job controller's checkpointing capability for print jobs.

Note that your input routine might be called with a marker string that was originally returned in a different process context from the current one. This can occur because marker strings are sometimes stored in the queue-data file across system shutdowns or different invocations of your symbiont.

The *funcdesc* argument specifies the address of a string descriptor. Your routine must return the marker string by way of this argument. VSI recommends that you use one of the Run-Time Library string routines to copy the marker string to the descriptor.

The symbiont periodically calls your input routine with the PSM$K_GET_KEY function code when the symbiont wants to save a marker to a particular input record.

[PSM$K_OPEN]

When the symbiont calls your routine with this function code, your routine should prepare for input operations by performing such tasks as allocating necessary resources, initializing storage areas, opening an input file, and so on. Typically, the next time the symbiont calls your input routine, the symbiont will specify the PSM$K_READ function code. Note, however, that under some circumstances the symbiont might follow an OPEN call immediately with a CLOSE call.

The *funcdesc* argument points to the name of the file to be opened. Your routine can use this file specification or the file identification to open the file.

The *funcarg* argument specifies the address of a longword. Your input routine must return, in this longword, the carriage control type that is to be applied to the input records that your input routine will provide.

The symbiont formatting routine requires this information to determine where to apply leading and trailing carriage control characters to the input records that your input routine will provide.

The $PSMDEF macro defines the following four carriage control types:

Carriage Control Type	Description
PSM$K_CC_IMPLIED	Implied carriage control. For this type, the symbiont inserts a leading line feed (LF) and trailing carriage return (CR) in each input record. This is the default carriage control type; it is used if your routine does not supply a carriage control type in the *funcarg* argument in response to the PSM $K_OPEN function call.
PSM$K_CC_FORTRAN	Fortran carriage control. For this type, the symbiont extracts the first byte of each input record and interprets the byte as a Fortran carriage control character, which it then applies to the input record.
PSM$K_CC_PRINT	PRN carriage control. For this type, the symbiont generates carriage control from a 2-byte record header that your input routine supplies, with each READ call, in the *funcarg* argument. The *funcarg* argument specifies the address of a longword to receive this 2-byte header record, which appears only in PRN print files.
PSM$K_CC_INTERNAL	Embedded carriage control. For this type, the symbiont supplies no carriage control to input records. Carriage control is assumed to be embedded in the input records.

[PSM$K_POSITION_TO_KEY]

When the symbiont calls your routine with this function code, your routine must locate the point in the input stream designated by the marker string that your routine returned to the symbiont on the PSM $K_GET_KEY function call.

The next time the symbiont calls your routine, the symbiont specifies the PSM$K_READ function call, expecting to receive the next sequential input record. After rereading this record, subsequent READ calls proceed from this new position of the file. This is not a one-time rereading of a single record but a repositioning of the file. The symbiont calls your routine with this function code when the job controller receives a request to resume printing at a particular page.

Refer to the description of the PSM$K_GET_KEY for more information.

[PSM$K_READ]

When the symbiont calls your routine with this function code, your routine must return an input record. The symbiont repeatedly calls your input routine with the PSM$K_READ function code until: (1) your routine indicates end of input by returning the status PSM$_EOF, (2) your routine or another routine returns an error status, or (3) the symbiont receives an asynchronous task-abortion request from the job controller.

The *funcdesc* argument specifies the address of a string descriptor. Your routine must return the input record by using this argument. VSI recommends that you use one of the Run-Time Library string routines to copy the input record to the descriptor.

The *funcarg* argument specifies the address of a longword. This argument is used only if the carriage control type returned by your input routine on the PSM$K_OPEN function call was PSM $K_CC_PRINT. In this case, your input routine must supply, in the *funcarg* argument, the 2-byte record header found at the beginning of each input record.

[PSM$K_REWIND]

When the symbiont calls your routine with this function code, your routine must do one of two things: (1) return PSM$_FUNNOTSUP (function not supported) or (2) locate the point in the input stream designated as the beginning of the file.

If your routine returns PSM$_FUNNOTSUP to this function code, then the symbiont subsequently calls your input routine with a PSM$K_CLOSE function call followed by a PSM$K_OPEN function call. By returning PSM$_FUNNOTSUP, your routine is choosing not to support the repositioning of the input service to the beginning of the file. The symbiont, therefore, performs the desired function by closing and then reopening the input routine.

You cannot use the *funcdesc* and the *funcarg* arguments with this function code.

This function call allows the modified symbiont to perform the file-positioning functions specified by the DCL commands START/QUEUE/TOP_OF_FILE, START/QUEUE/FORWARD, START/QUEUE/BACKWARD, START/QUEUE/SEARCH, and START/QUEUE/ALIGN. This is a required repositioning of the file.

[Other Input Function Codes]

The symbiont can call your input routine with other function codes. Your routine *must* return the status PSM$_FUNNOTSUP (function not supported) when it is called with any of the following function codes or with any undocumented function code. When the status PSM$_FUNNOTSUP is returned, the symbiont performs its normal action as if no input routine were supplied. To suppress the symbiont's normal action, you should return SS$_NORMAL.

PSM$K_START_STREAM	PSM$K_STOP_STREAM
PSM$K_START_TASK	PSM$K_PAUSE_TASK
PSM$K_RESUME_TASK	PSM$K_STOP_TASK
PSM$K_RESET_STREAM	

These function codes correspond to message items, which are discussed in detail in Section 18.3.5, sent by the job controller to the symbiont.

Other function codes correspond to internal symbiont mechanisms that are not part of the public interface to the print symbiont.

Your input routine should return the status PSM$_FUNNOTSUP or SS$_NORMAL when it is called with a message function code or with a private function code.

Routines

funcdesc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Function descriptor supplying information related to the function specified by the *func* argument. The *funcdesc* argument is the address of this descriptor.

The contents of the function descriptor can vary for each function. Refer to the description of each function code to determine the contents of the function descriptor. In some cases, the function descriptor is not used at all.

funcarg

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Function argument supplying information related to the function specified by the *func* argument. The *funcarg* argument is the address of a longword containing this function argument. This argument can be an input or an output argument, depending on the function request, but is usually used as an output argument.

Condition Values Returned

SS$_NORMAL

Successful completion. The user input routine has completed the function that the symbiont requested.

PSM$_FLUSH

Flush output stream. The user input routine can return this status only when called with the PSM $K_READ function code. When this status is returned to the symbiont, the symbiont stops calling the input routine with the PSM$K_READ function code until all outstanding format and output operations have completed.

PSM$_FUNNOTSUP

Function not supported. The user input routine does not support or does not recognize the function code supplied by the symbiont. To ensure future compatibility, your input routine should return this status for any unrecognized status codes.

PSM$_PENDING

Requested function accepted but not completed. Your input routine can return this status only with the PSM$K_READ function call. Further, if your routine returns PSM$_PENDING, your routine must eventually signal completion via the PSM$REPORT routine. Refer to the description of the PSM$REPORT routine for more information about asynchronous operations and the PSM $_PENDING condition value.

This routine also returns any error condition values that you have coded your format routine to return. Refer to Section 18.3.1 for more information about error condition values.

USER-OUTPUT-ROUTINE

Invoke User-Written Output Routine — The user-written USER-OUTPUT-ROUTINE performs output operations. You supply a user output routine by calling the PSM$REPLACE routine with the routine code PSM$K_OUTPUT.

Format

```
USER-OUTPUT-ROUTINE request_id ,work_area ,func ,funcdesc ,funcarg
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

request_id

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Request identifier value supplied by the symbiont when it calls your output routine. The *request_id* argument is the address of a longword containing this value.

If your output routine initiates an asynchronous operation (for example, a call to the $QIO system service), you must save the *request_id* argument because you will need to store the request identifier value for later use with the PSM$REPORT routine. See the description of the PSM$REPORT routine for more information.

work_area

OpenVMS usage:	address

type:	longword (unsigned)
access:	write only
mechanism:	by reference

Work area supplied by the symbiont for the use of your format routine. The symbiont supplies the address of this area when it calls your routine. The *work_area* argument is a longword containing the address of the work area. The work area is a section of memory that your format routine can use for buffering and other internal operations.

The size of the work area allocated is specified by the *work_size* argument in the PSM$PRINT routine. If you do not specify *work_size* in the call to PSM$PRINT, no work area is allocated.

In a multithreaded symbiont, a separate work area is allocated for each thread. This work area is shared by all user routines. The work area is initialized to zero when the symbiont is first started.

func

OpenVMS usage:	function_code
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Function code supplied by the symbiont when it calls your output routine. The *func* argument is the address of a longword containing this code.

The function code specifies the reason the symbiont is calling your output routine or, in other words, the function that the symbiont expects your routine to perform at this time.

Most function codes require or allow additional information to be passed in the call via the *funcdesc* and *funcarg* arguments. The description of each output function code, therefore, includes a description of how these two arguments are used for that function code.

The following list describes all the function codes that the symbiont might supply when it calls your output routine (function codes applicable only to input and formatting routines are explained in the descriptions of the user input routine and user formatting routine, respectively). Each programming language provides an appropriate mechanism for defining these function codes.

Function Codes for Output Routines

[PSM$K_OPEN]

When the symbiont calls your output routine with this function code, your routine should prepare to move data to the device by performing such tasks as allocating the device, assigning a channel to the device, and so on. The next time the symbiont calls your output routine, the symbiont specifies one of the WRITE function codes (PSM$K_WRITE or PSM$K_WRITE_NOFORMAT).

The symbiont calls your output routine with the PSM$K_OPEN function code when the symbiont receives the SMBMSG$K_START_STREAM message from the job controller.

If your output routine returns an error condition value (low bit clear) to the PSM$K_OPEN function call, the job controller stops processing on the stream and reports the error to whomever entered the DCL command START/QUEUE.

The *funcdesc* argument is the address of a descriptor that identifies the name of the device to which the output routine is to write. This device name is established by the DCL command INITIALIZE/QUEUE/ON= *device*.

The *funcarg* argument is the address of a longword into which the user output routine returns the device status longword. Your output routine sets bits in the device status longword to indicate to the job controller whether the device falls into one of the following categories:

- Can print lowercase letters
- Is a terminal
- Is connected to the CPU by means of a modem (remote)

If your output routine does not set any of these bits in the device status longword, the job controller assumes, by default, that the device is a line printer that prints only uppercase letters.

[PSM$K_WRITE]

When the symbiont calls your routine with this function code, your routine must write data to the device. The symbiont supplies the data to be written in the *funcdesc* argument. VSI recommends that you use one of the Run-Time Library string routines to access the data in the buffer described by the *funcdesc* argument.

[PSM$K_WRITE_NOFORMAT]

When the symbiont calls your routine with this function code, your routine must write data to the device and must indicate to the device driver that the data is not to be formatted.

The symbiont calls your routine with this function code when: (1) the print request specifies the PASSALL option or (2) data is introduced by the ANSI DCS (device control string) escape sequence.

The symbiont supplies the data to be written in the *funcdesc* argument. VSI recommends that you use one of the Run-Time Library string routines to move the data from the descriptor to the device.

The output routine of the symbiont informs the device driver not to format the data in the following way:

- When the device is a line printer, the symbiont's output routine specifies the IO$_WRITEPBLK function code when it calls the $QIO system service.
- When the device is a terminal, the symbiont's output routine specifies the IO$M_NOFORMAT function modifier when it calls the $QIO system serivce.

[PSM$K_CANCEL]

When the symbiont calls your routine with this function code, your routine must abort any outstanding asynchronous I/O requests.

The output routine supplied by the symbiont aborts outstanding I/O requests by calling the $CANCEL system service with the IO$_CANCEL function code.

If your output routine returned the condition value PSM$_PENDING to one or more previous write requests that are still outstanding (that is, PSM$REPORT has not yet been called to report completion), then your output routine must call PSM$REPORT one time for each outstanding write request that is canceled with this call. That is, canceling an asynchronous write request does not relieve the user output routine of the requirement to call PSM$REPORT once for each asynchronous write request.

You cannot use the *funcdesc* and *funcarg* arguments with this function code.

[PSM$K_CLOSE]

When the symbiont calls your routine with this function code, your output routine must terminate processing and release any resources it allocated (for example, channels assigned to the device).

You cannot use the *funcdesc* and *funcarg* arguments with this function code.

[Other Output Function Codes]

The symbiont can call your output routine with other function codes. Your routine should return the status PSM$_FUNNOTSUP (function not supported) when it is called with any of the following function codes or with any undocumented function code. When the status PSM$_FUNNOTSUP is returned, the symbiont performs its normal action as if no output routine were supplied. To suppress the symbiont's normal action, you should return SS$_NORMAL.

PSM$K_START_STREAM	PSM$K_STOP_STREAM
PSM$K_START_TASK	PSM$K_PAUSE_TASK
PSM$K_RESUME_TASK	PSM$K_STOP_TASK
PSM$K_RESET_STREAM	

These function codes correspond to message items, which are discussed in more detail in Section 19.1.6, sent by the job controller to the symbiont.

Other function codes correspond to internal symbiont mechanisms that are not part of the public interface to the print symbiont.

Your output routine should return the status PSM$_FUNNOTSUP or SS$_NORMAL when it is called with a message function code or with a private function code.

Routines

funcdesc

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Function descriptor supplying information related to the function specified by the *func* argument. The *funcdesc* argument is the address of this descriptor.

The contents of the function descriptor can vary for each function. Refer to the description of each function code to determine the contents of the function descriptor. In some cases, the function descriptor is not used at all.

funcarg

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only

mechanism: by reference

Function argument supplying information related to the function specified by the *func* argument. The *funcarg* argument is the address of a longword containing this function argument.

The contents of the function argument can vary for each function. Refer to the description of each function code to determine the contents of the function argument. In some cases, the function argument is not used.

Condition Values Returned

SS$_NORMAL

Normal successful completion. The user output routine has completed the function that the symbiont requested.

PSM$_FUNNOTSUP

Function not supported. The user output routine does not support or does not recognize the function code supplied by the symbiont. To ensure future compatibility, your output routine should return this status for any unrecognized status codes.

PSM$_PENDING

Requested function accepted but not completed. Your output routine can return this status only with PSM$K_WRITE and PSM$K_WRITE_NOFORMAT function calls. Further, if your routine returns PSM$_PENDING, your routine must eventually signal completion by way of the PSM $REPORT routine. Refer to the description of the PSM$REPORT routine for more information about asynchronous write operations and the PSM$_PENDING condition value.

This routine also returns any error condition values that you have coded your output routine to return. Refer to Section 18.3.1 for more information about error condition values.

Chapter 19. Symbiont/Job Controller Interface (SMB) Routines

The Symbiont/Job Controller Interface (SMB) routines provide the interface between the job controller and symbiont processes. A user-written symbiont must use these routines to communicate with the job controller.

19.1. Introduction to SMB Routines

Always use the SMB interface routines or the $SNDJBC or $GETQUI system services to communicate with the job controller. You need not and should not attempt to communicate directly with the job controller.

To write your own symbiont, you need to understand how symbionts work and, in particular, how the standard print symbiont behaves.

19.1.1. Types of Symbiont

There are two types of symbiont:

- Device symbiont, either an input symbiont or an output symbiont. An input symbiont is one that transfers data from a slow device to a fast device, for example, from a card reader to a disk. A card-reader symbiont is an input symbiont. An output symbiont is one that transfers data from a fast device to a slow device, for example, from a disk to a printer or terminal. A print symbiont is an output symbiont.

- Server symbiont, a symbiont that processes or transfers data but is not associated with a particular device; one example is a symbiont that transfers files across a network.

The operating system does not supply any server symbionts.

19.1.2. Symbionts Supplied with the Operating System

The operating system supplies two symbionts:

- SYS$SYSTEM:PRTSMB.EXE (PRTSMB for short), an output symbiont for use with printers and printing terminals

  PRTSMB performs such functions as inserting flag, burst, and trailer pages into the output stream; reading and formatting input files; and writing formatted pages to the printing device.

  You can modify PRTSMB using the Print Symbiont Modification (PSM) routines.

- SYS$SYSTEM:INPSMB.EXE (INPSMB for short), an input symbiont for use with card readers

  This symbiont handles the transferring of data from a card reader to a disk file. You cannot modify INPSMB, nor can you write an input symbiont using the SMB routines.

19.1.3. Symbiont Behavior in the OpenVMS Environment

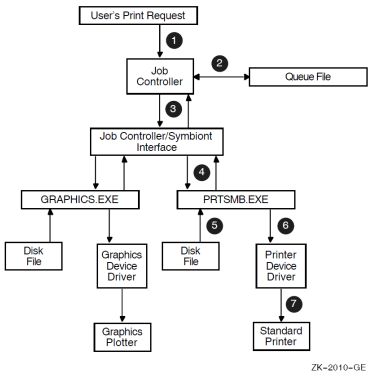
In the OpenVMS environment, a symbiont is a process under the control of the job controller that transfers or processes data.

Figure 19.1 depicts the components that take part in the handling of user requests that involve symbionts. This figure shows two symbionts: (1) the print symbiont supplied by the operating system, PRTSMB, and (2) a user-written symbiont, GRAPHICS.EXE, which services a graphics plotter. The numbers in the figure correspond to the numbers in the list that follows.

This list does not reflect the activities that must be performed by the hypothetical, user-written symbiont, GRAPHICS.EXE. This symbiont is represented in the figure to illustrate the correspondence between a user-written symbiont and the print symbiont supplied by the operating system.

Although SMB routines can be used for a different kind of symbiont, many of their arguments and associated symbols have names related to the print symbiont. The print symbiont is presented here as an example of a typical symbiont and illustrates points that are generally true for symbionts.

Figure 19.1. Symbionts in the OpenVMS Environment



1

You request a printing job with the DCL command PRINT. DCL calls the $SNDJBC system service, passing the name of the file to be printed to the job controller, along with any other information specified by qualifiers for the PRINT command.

2

The job controller places the print request in the appropriate queue and assigns the request a job number.

3

The job controller breaks the print job into a number of tasks (for example, printing three copies of the same file is three separate tasks). The job controller makes a separate request to the symbiont for each task.

Each request that the job controller makes consists of a message. Each message consists of a code that indicates what the symbiont is to do and a number of items of information that the symbiont needs to carry out the task (the name of the file, the name of the user, and so on).

4

PRTSMB interprets the information it receives from the job controller.

5

PRTSMB locates and opens the file it is to print by using the file-identification number the job controller specified in the start-task message.

6

PRTSMB sends the data from the file to the printer's driver.

7

The device driver sends the data to the printer.

19.1.4. Writing a Symbiont

Writing your own symbiont permits you to use the queuing mechanisms and control functions of the job controller. You might want to do this if you need a symbiont for a device that cannot be served by PRTSMB (or a modified form of PRTSMB) or if you need a server symbiont. The interface between the job controller and the symbiont permits the symbiont you write to use the many features of the job controller.

For example, when you use the DCL command PRINT, the job controller sends a message to the print symbiont telling it to print the file. However, when a user-written symbiont receives the same message (caused by entering a PRINT command), it might interpret it to mean something quite different. A robot symbiont, for example, might interpret the message as a command for movement and the file specification (specified with the PRINT command) might be a file describing the directions in which the robot is to move.

Note

Modifying PRTSMB is easier than writing your own symbiont; choose this option if possible. The Print Symbiont Modification (PSM) routines describe how to modify PRTSMB to suit your needs.

19.1.5. Guidelines for Writing a Symbiont

Although you can write a symbiont to use the queuing mechanisms and other features of the job controller in whatever way you want, you must follow these guidelines to ensure that your symbiont works correctly:

- The symbiont must not use any of the process-permanent channels, which are assigned to the following logical names:
  - SYS$INPUT
  - SYS$OUTPUT
  - SYS$ERROR
  - SYS$COMMAND

- The symbiont must allocate and deallocate memory using the Run-Time Library (RTL) routines LIB $GET_VM and LIB$FREE_VM.
- To be compatible with future releases of the operating system, you should write the symbiont to ignore unknown message-item codes and unknown message-request codes. (See the SMB $READ_ITEM_MESSAGE routine.)
- The symbiont must communicate with the job controller by using the SMB routines, the $SNDJBC system service, and the $GETQUI system service.
- The symbiont should not perform lengthy operations within the context of an AST routine. The symbiont can only receive messages from the job controller when it is not executing within the context of an AST routine.
- The symbiont code should be linked against SMBSRVSHR.EXE in order to define the SMB routine address and the following status codes:
  - SMB$_INVSTMNBR
  - SMB$_INVSTRLEV
  - SMB$_NOMOREITEMS
- To assign a symbiont to a queue after it is compiled and linked, the executable image of the symbiont must reside in SYS$SYSTEM, and you must enter either of the following commands:

  ```
  INITIALIZE/QUEUE/PROCESSOR=symbiont_filename
  START/QUEUE/PROCESSOR=symbiont_filename
  ```

  You should specify only the file name in the command. The disk and directory default to SYS $SYSTEM, and all fields except the file name are ignored.
- To help debug symbionts, you should define the logical names DBG$INPUT and DBG$OUTPUT in the LNM$GROUP_000001 logical name table to point to your debugging terminal.

19.1.6. The Symbiont/Job Controller Interface Routines

The five SMB routines form a public interface to the job controller. The job controller delivers requests to symbionts by means of this interface, and the symbionts communicate their responses to those requests through this interface. A user-written symbiont uses the following routines to exchange messages with the job controller:

Routine	Description
SMB$INITIALIZE	Initializes the SMB facility's internal database, establishes the interface to the job controller, and defines whether: • Messages from the job controller are to be delivered to the symbiont synchronously or asynchronously with respect to execution of the symbiont. • The symbiont is to be single-threaded or multithreaded; these concepts are described in the sections that follow.

Routine	Description
SMB$CHECK_FOR_MESSAGE	Checks to see if a message from the job controller to the symbiont has arrived (used with synchronous symbionts)
SMB$READ_MESSAGE	Reads the job controller's message into a buffer
SMB$READ_MESSAGE_ITEM	Returns one item of information from the job controller's message (which can have several informational items)
SMB$SEND_TO_JOBCTL	Sends a message from the symbiont to the job controller

The following sections discuss how to use the SMB routines when writing your symbiont.

19.1.7. Choosing the Symbiont Environment

The first SMB routine that a symbiont must call is the SMB$INITIALIZE routine. In addition to allocating and initializing the SMB facility's internal database, it offers you two options for your symbiont environment: (1) synchronous or asynchronous delivery of messages from the job controller, and (2) single streaming or multistreaming the symbiont.

19.1.7.1. Synchronous Versus Asynchronous Delivery of Requests

When you initialize your symbiont/job controller interface, the symbiont has the option of accepting requests from the job controller synchronously or asynchronously.

Synchronous Environment

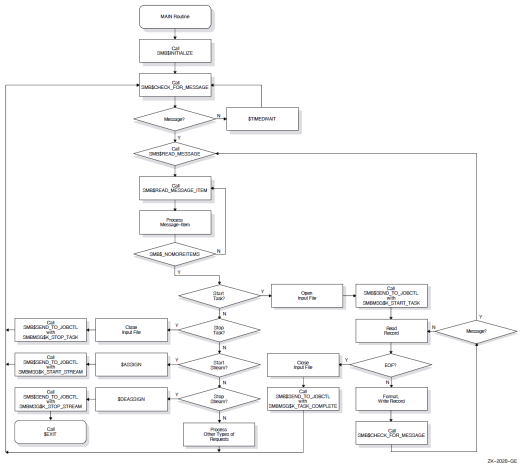
The address of an AST routine is an optional argument to the SMB$INITIALIZE routine; if it is not specified, the symbiont receives messages from the job controller synchronously. A symbiont that receives messages synchronously must call SMB$CHECK_FOR_MESSAGE periodically during the processing of tasks in order to ensure the timely delivery of STOP_TASK, PAUSE_TASK, and RESET_STREAM requests.

SMB$CHECK_FOR_MESSAGE checks to see if a message from the job controller is waiting. If a message is waiting, SMB$CHECK_FOR_MESSAGE returns a success code. The caller of SMB $CHECK_FOR_MESSAGE can then call SMB$READ_MESSAGE to read the message and take the appropriate action.

If no message is waiting, SMB$CHECK_FOR_MESSAGE returns a zero in R0. The caller of SMB $CHECK_FOR_MESSAGE can continue to process the task at hand.

Figure 19.2 is a flowchart for a synchronous, single-threaded symbiont. The flowchart does not show all the details of the logic the symbiont needs and does not show how the symbiont handles PAUSE_TASK, RESUME_TASK, or RESET_STREAM requests.

Figure 19.2. Flowchart for a Single-Threaded, Synchronous Symbiont



Asynchronous Environment

To receive messages asynchronously, a symbiont specifies a message-handling AST routine as the second argument to the SMB$INITIALIZE routine. In this scheme, whenever the job controller sends messages to the symbiont, the AST routine is called.

The AST routine is called with no arguments and returns no value. You have the option of having the AST routine read the message within the context of its execution or of having the AST routine wake a suspended process to read the message outside the context of the execution of the AST routine.

Be aware that an AST can be delivered only while the symbiont is not executing within the context of an AST routine. Thus, in order to ensure delivery of messages from the job controller, the symbiont should not perform lengthy operations at the AST level.

This is particularly important to the execution of STOP_TASK, PAUSE_TASK, and RESET_STREAM requests. If a STOP_TASK request cannot be delivered during the processing of a task, for example, it is useless.

One technique that ensures delivery of STOP and PAUSE requests in an asynchronous environment is to have the AST routine set a flag if it reads a PAUSE_TASK, STOP_TASK, or a RESET_STREAM request and to have the symbiont's main routine periodically check the flag.

Figure 19.3 and Figure 19.4 show flowcharts for a single-threaded, asynchronous symbiont. The figures do not show many details that your symbiont might include, such as a call to the $QIO system service.

Note that the broken lines in Figure 19.3 that connect the calls to $HIBER with the AST routine's calls to $WAKE show that the next action to take place is the call to $WAKE. They do not accurately represent the flow of control within the symbiont but represent the action of the job controller in causing the AST routine to execute.

Figure 19.3. Flowchart for a Single-Threaded, Asynchronous Symbiont (MAIN Routine)

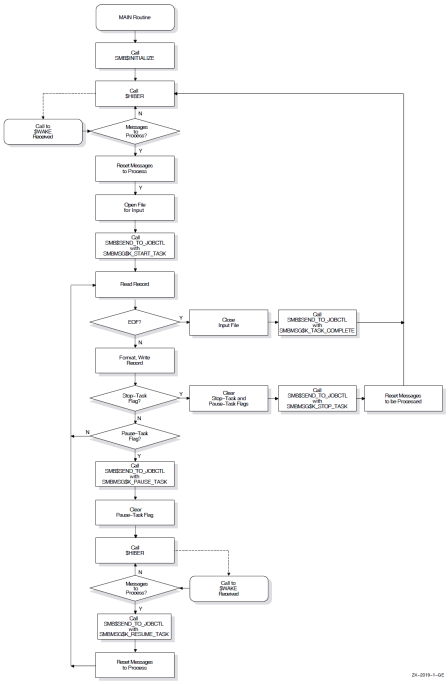
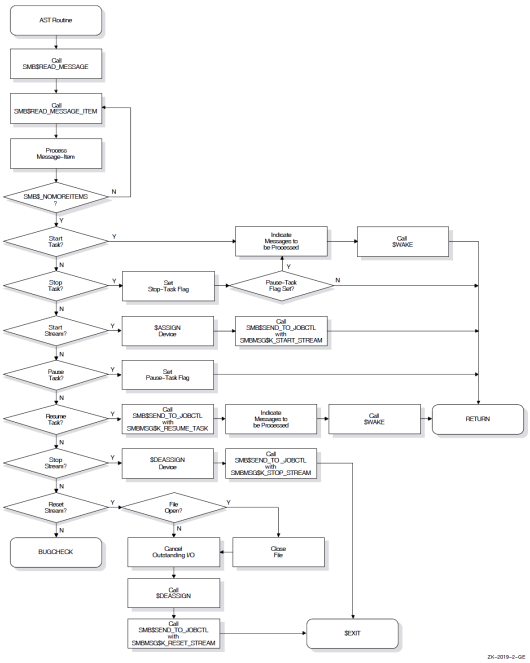


Figure 19.4. Flowchart for a Single-Threaded, Asynchronous Symbiont (AST Routine)



19.1.7.2. Single-Streaming Versus Multistreaming

A single-stream (or thread) is a logical link between a queue and a symbiont process. When a symbiont process is linked to more than one queue and serves those queues simultaneously, it is called a **multithreaded** symbiont.

The argument to the SMB$READ_MESSAGE routine provides a way for a multithreaded symbiont to keep track of the stream referred to by a request. Writing your own multithreaded symbiont, however, can be a complex undertaking.

19.1.8. Reading Job Controller Requests

The seven general functions that the job controller can request of the symbiont are as follows:

- SMBMSG$K_START_STREAM
- SMBMSG$K_STOP_STREAM
- SMBMSG$K_START_TASK
- SMBMSG$K_PAUSE_TASK
- SMBMSG$K_RESUME_TASK
- SMBMSG$K_STOP_TASK
- SMBMSG$K_RESET_STREAM

The job controller passes these requests to the symbiont in a structure that contains: (1) a code that identifies the requested function and (2) optional items of information that the symbiont might need to perform the requested function.

By calling SMB$READ_MESSAGE, the symbiont reads the function code and writes the associated items of information, if any, into a buffer. The symbiont then parses the message items stored in the buffer by calling the SMB$READ_MESSAGE_ITEM routine. SMB$READ_MESSAGE_ITEM reads one message item each time it is called.

Each message item consists of a code that identifies the type of information the item contains, and the information itself. For example, the SMBMSG$K_JOB_NAME code tells the symbiont that the item contains a string, which is the name of a job.

The number of message items in a request message varies with each type of request. Therefore, to ensure that all message items are read, SMB$READ_MESSAGE_ITEM must be called repeatedly for each request. SMB$READ_MESSAGE_ITEM returns status SMB$_NOMOREITEMS after it has read the last message item in a given request.

Typically, a symbiont checks the code of a message item against a case table and stores the message string in an appropriate variable until all the message items are read and the processing of the request can begin.

See the description of the SMB$READ_MESSAGE_ITEM routine for a table that shows the message items that make up each type of request.

19.1.9. Processing Job Controller Requests

After a request is read, it must be processed. The way a request is processed depends on the type of request. The following section lists, for each request that the job controller sends to the print symbiont, the actions that the standard symbiont (PRTSMB) takes when the message is received. These actions are oriented toward print symbionts in particular but can serve as a guideline for other kinds of symbionts as well.

The symbiont you write can respond to requests in a similar way or in a different way appropriate to the function of your symbiont. VSI suggests that your routines follow the guidelines described in this document. (Note that the behavior of the standard symbiont is subject to change without notice in future versions of the operating system.)

SMBMSG$K_START_STREAM

- Reset all stream-specific information that might have been altered by previous START_STREAM requests on this stream (for multithreaded symbionts).
- Read and store the message items associated with the request.
- Allocate the device specified by the SMBMSG$K_DEVICE_NAME item.
- Assign a channel to the device.
- Obtain the device characteristics.
- If the device is neither a terminal nor a printer, then abort processing and return an error to the job controller by means of the SMB$SEND_TO_JOBCTL routine. Note that, even though an error has occurred, the stream is still considered started. The job controller detects the error and sends a STOP_STREAM request to the symbiont.
- Set temporary device characteristics suited to the way the symbiont will use the device.
- For remote devices (devices connected to the system by means of a modem), establish an AST to report loss of the carrier signal.
- Report to the job controller that the request has been completed and that the stream is started, by specifying SMBMSG$K_START_STREAM in the call to SMB$SEND_TO_JOBCTL.

SMBMSG$K_START_TASK

- Reset all task-specific information that might have been altered by previous START_TASK requests on this stream number.
- Read and store the message items associated with the request.
- Open the main input file.
- Report to the job controller that the task has been started by specifying SMBMSG $K_START_TASK in the call to the SMB$SEND_TO_JOBCTL routine.
- Begin processing the task.
- When the task is complete, notify the job controller by specifying SMBMSG $K_TASK_COMPLETE in the call to the SMB$SEND_TO_JOBCTL routine.

SMBMSG$K_PAUSE_TASK

- Read and store the message items associated with the request.
- Set a flag that will cause the main processing routine to pause at the beginning of the next output page.
- When the main routine pauses, notify the job controller by specifying SMBMSG$K_PAUSE_TASK in the call to the SMB$SEND_TO_JOBCTL routine.

SMBMSG$K_RESUME_TASK

- Read and store the message items associated with the request.

- Perform any positioning functions specified by the message items.
- Clear the flag that causes the main input routine to pause, and resume processing the task.
- Notify the job controller that the task has been resumed by specifying SMBMSG $K_RESUME_TASK in the call to the SMB$SEND_TO_JOBCTL routine.

SMBMSG$K_STOP_TASK

- Read and store the message items associated with the request.
- If processing of the current task has paused, then resume it.
- Cancel any outstanding I/O operations.
- Close the input file.
- If the job controller specified, in the START_TASK message, that a trailer page should be printed when the task is stopped or if it specified that the device should be reset when the task is stopped, then perform those functions.
- Notify the job controller that the task has been stopped abnormally by specifying SMBMSG $K_STOP_TASK and by specifying an error vector in the call to SMB$SEND_TO_JOBCTL. PRTSMB specifies the value passed by the job controller in the SMBMSG$K_STOP_CONDITION item as the error condition in the error vector.

SMBMSG$K_STOP_STREAM

- Read and store the message items associated with the request.
- Release any stream-specific resources: (1) deassign the channel to the device, and (2) deallocate the device.
- Notify the job controller that the stream has been stopped by specifying SMBMSG $K_STOP_STREAM in the call to SMB$SEND_TO_JOBCTL.
- If this is a single-threaded symbiont or if this is a multithreaded symbiont but all other streams are currently stopped, then call the $EXIT system service with the condition code SS$_NORMAL.

SMBMSG$K_RESET_STREAM

- Read and store the message items associated with the request.
- Abort any task in progress – you do not need to notify the job controller that the task has been aborted, but you may do so if you want.
- If the job controller specified, in the START_TASK message, that a trailer page should be printed when the task is stopped or if it specified that the device should be reset when the task is stopped, then suppress those functions.

  The job controller sends the symbiont a RESET_STREAM request to regain control of a queue or a device that has failed to respond to a STOP_TASK request. The RESET_STREAM request should avoid any further I/O activity if possible. The printer might be disabled, for example, and requests for output on that device will never be completed.
- Continue as if this were a STOP_STREAM request.

Note

A STOP_STREAM request and a RESET_STREAM request each stop the queue; but a RESET_STREAM request is an emergency stop and is used, for example, when the device has failed. A RESET_STREAM request should prevent any further I/O activity because the printer might not be able to complete it.

19.1.10. Responding to Job Controller Requests

The symbiont uses the SMB$SEND_TO_JOBCTL routine to send messages to the job controller.

Most messages that the symbiont sends to the job controller are responses to requests made by the job controller. Such messages inform the job controller that the request has been completed successfully or unsuccessfully. The function code that the symbiont returns to the controller in the call to SMB $SEND_TO_JOBCTL indicates what request has been completed.

For example, if the job controller sends a START_TASK request using the SMBMSG$K_START_TASK code, the symbiont responds by calling SMB$SEND_TO_JOBCTL using SMBMSG$K_START_TASK as the *request* argument to indicate that task processing has begun. Until the symbiont responds, the DCL command SHOW QUEUE indicates that the queue is starting.

The responses to some requests use additional arguments to send more information than just the request code. See the SMB$SEND_TO_JOBCTL routine for a table showing the additional arguments allowed in response to each request.

In addition to sending messages in response to requests, the symbiont can send other messages to the job controller. In these messages the symbiont sends either the SMBMSG$K_TASK_COMPLETE code, indicating that it has completed a task, or SMBMSG$K_TASK_STATUS, indicating that the message contains information on the status of a task.

Note that, when a START_TASK request is delivered, the symbiont responds with a SMB $SEND_TO_JOBCTL message with the SMBMSG$K_START_TASK code. This response means the task has been started. It does not mean the task has been completed. When the symbiont completes the task, it calls SMB$SEND_TO_JOBCTL with the SMBMSG$K_TASK_COMPLETE code.

19.2. SMB Routines

This section describes the individual SMB routines.

SMB$CHECK_FOR_MESSAGE

Check for Message from Job Controller — The SMB$CHECK_FOR_MESSAGE routine determines whether a message sent from the job controller to the symbiont is waiting to be read.

Format

```
SMB$CHECK_FOR_MESSAGE
```

Returns

OpenVMS 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 under Condition Values Returned.

Description

When your symbiont calls the SMB$INITIALIZE routine to initialize the interface between the symbiont and the job controller, you can choose to have requests from the job controller delivered by means of an AST. If you choose not to use ASTs, your symbiont must call SMB $CHECK_FOR_MESSAGE during the processing of tasks in order to see if a message from the job controller is waiting to be read. If a message is waiting, SMB$CHECK_FOR_MESSAGE returns a success code; if not, it returns a zero.

If a message is waiting, the symbiont should call SMB$READ_MESSAGE to read it to determine if immediate action should be taken (as in the case of STOP_TASK, RESET_STREAM or PAUSE_TASK).

If a message is not waiting, SMB$CHECK_MESSAGE returns a zero. If this condition is detected, the symbiont should continue processing the request at hand.

Condition Values Returned

SS$_NORMAL

One or more messages waiting.

0

No messages waiting.

SMB$INITIALIZE

Initialize User-Written Symbiont — The SMB$INITIALIZE routine initializes the user-written symbiont and the interface between the symbiont and the job controller. It allocates and initializes the internal databases of the interface and sets up the mechanism that is to wake up the symbiont when a message is received.

Format

```
SMB$INITIALIZE structure_level [,ast_routine] [,streams]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

structure_level

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Version of the symbiont/job controller interface. The *structure_level* argument is the address of a longword containing the version of the symbiont/job controller interface used when the symbiont was compiled. Always place the value of the symbol SMBMSG$K_STRUCTURE_LEVEL in the longword addressed by this argument. Each programming language provides an appropriate mechanism for defining symbols.

ast_routine

OpenVMS usage:	ast_procedure
type:	procedure value
access:	read only
mechanism:	by reference

Message-handling routine called at AST level. The *ast_routine* argument is the address of the entry point of the message-handling routine to be called at AST level when the symbiont receives a message from the job controller. The AST routine is called with no parameters and returns no value. If an AST routine is specified, the routine is called once each time the symbiont receives a message from the job controller.

The AST routine typically reads the message and determines if immediate action must be taken. Be aware that an AST can be delivered only while the symbiont is operating at non-AST level. Thus, to ensure delivery of messages from the job controller, the symbiont should not perform lengthy operations at AST level.

If you do not specify the *ast_routine* argument, the symbiont must call the SMB $CHECK_FOR_MESSAGE routine to check for waiting messages.

streams

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Maximum number of streams the symbiont is to support. The *streams* argument is the address of a longword containing the number of streams that the symbiont is to support. The number must be in the range of *1* to *32*.

If you do not specify this argument, a default value of *1* is used. Thus, by default, a symbiont supports one stream. Such a symbiont is called a single-threaded symbiont.

A stream (or thread) is a logical link between a queue and a symbiont. When a symbiont is linked to more than one queue, and serves those queues simultaneously, it is called a multithreaded symbiont.

Description

Your symbiont must call SMB$INITIALIZE before calling any other SMB routines. It calls SMB $INITIALIZE in order to do the following:

- Allocate and initialize the SMB facility's internal database.
- Establish the interface between the job controller and the symbiont.
- Determine the threading scheme of the symbiont.
- Set up the mechanism to wake your symbiont when a message is received.

After the symbiont calls SMB$INITIALIZE, it can communicate with the job controller using the other SMB routines.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SMB$_INVSTRLEV

Invalid structure level.

This routine also returns any codes returned by $ASSIGN and LIB$GET_VM.

SMB$READ_MESSAGE

Obtain Message Sent by Job Controller — The SMB$READ_MESSAGE routine copies a message that the job controller has sent into the caller's specified buffer.

Format

```
SMB$READ_MESSAGE stream ,buffer ,request
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

stream

OpenVMS usage:	longword_unsigned

type:	longword (unsigned)
access:	write only
mechanism:	by reference

Stream number specifying the stream to which the message refers. The *stream* argument is the address of a longword into which the job controller writes the number of the stream referred to by the message. In single-threaded symbionts, the stream number is always 0.

buffer

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Address of the descriptor that points to the buffer into which the job controller writes the message. SMB $READ_MESSAGE uses the Run-Time Library string-handling (STR$) routines to copy the message into the buffer you supply. The buffer should be specified by a dynamic string descriptor.

request

OpenVMS usage:	identifier
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Code that identifies the request. The *request* argument is the address of a longword into which SMB $READ_MESSAGE writes the code that identifies the request.

There are seven request codes. Each code is interpreted as a message by the symbiont. The codes and their descriptions follow:

SMBMSG$K_START_STREAM	Initiates processing on an inactive symbiont stream. The job controller sends this message when a START/QUEUE or an INITIALIZE/QUEUE/START command is issued on a stopped queue.
SMBMSG$K_STOP_STREAM	Stops processing on a started queue. The job controller sends this message when a STOP/QUEUE/NEXT command is issued, after the symbiont completes any currently active task.
SMBMSG$K_RESET_STREAM	Aborts all processing on a started stream and requeues the current job. The job controller sends this message when a STOP/QUEUE/RESET command is issued.
SMBMSG$K_START_TASK	Requests that the symbiont begin processing a task. The job controller sends this message when a file is pending on an idle, started queue.
SMBMSG$K_STOP_TASK	Requests that the symbiont abort the processing of a task. The job controller sends this message

	when a STOP/QUEUE/ABORT or STOP/QUEUE/REQUEUE command is issued. The item SMBMSG$K_STOP_CONDITION identifies whether this is an abort or a requeue request.
SMBMSG$K_PAUSE_TASK	Requests that the symbiont pause in the processing of a task but retain the resources necessary to continue. The job controller sends this message when a STOP/QUEUE command is issued without the /ABORT, /ENTRY, /REQUEUE, or /NEXT qualifier for a queue that is currently printing a job.
SMBMSG$K_RESUME_TASK	Requests that the symbiont continue processing a task that has been stopped with a PAUSE_TASK request. This message is sent when a START/QUEUE command is issued for a queue served by a symbiont that has paused in processing the current task.

Description

Your symbiont calls SMB$READ_MESSAGE to read a message that the job controller has sent to the symbiont.

Each message from the job controller consists of a code identifying the function the symbiont is to perform and a number of message items. There are seven codes. Message items are pieces of information that the symbiont needs to carry out the requested function.

For example, when you enter the DCL command PRINT, the job controller sends a message containing a START_TASK code and a message item containing the specification of the file to be printed.

SMB$READ_MESSAGE writes the code into a longword (specified by the *request* argument) and writes the accompanying message items, if any, into a buffer (specified by the *buffer* argument).

See the description of the SMB$READ_MESSAGE_ITEM routine for information about processing the individual message items.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

LIB$_INVARG

Routine completed unsuccessfully because of an invalid argument.

This routine also returns any of the condition codes returned by the Run-Time Library string-handling (STR$) routines.

SMB$READ_MESSAGE_ITEM

Parse Next Item from Message Buffer — The SMB$READ_MESSAGE_ITEM routine reads a buffer that was filled in by the SMB$READ_MESSAGE routine, parses one message item from the buffer, writes the item's code into a longword, and writes the item into a buffer.

Format

```
SMB$READ_MESSAGE_ITEM message ,context ,item_code ,buffer [,size]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

message

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Message items that SMB$READ_MESSAGE_ITEM is to read. The *message* argument is the address of a descriptor of a buffer. The buffer is the one that contains the message items that SMB $READ_MESSAGE_ITEM is to read. The buffer specified here must be the same as that specified with the call to the SMB$READ_MESSAGE routine, which fills the buffer with the contents of the message.

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Value initialized to *0* specifying the first message item in the buffer to be read. The *context* argument is the address of a longword that the SMB$READ_MESSAGE_ITEM routine uses to determine the next message item to be returned. When this value is *0*, it indicates that SMB $READ_MESSAGE_ITEM is to return the first message item.

The SMB$READ_MESSAGE_ITEM routine updates this value each time it reads a message item. SMB $READ_MESSAGE_ITEM sets the value to *0* when it has returned all the message items in the buffer.

item_code

OpenVMS usage:	smb_item
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Item code specified in the message item that identifies its type. The *item_code* argument is the address of a longword into which SMB$READ_MESSAGE_ITEM writes the code that identifies which item it is returning.

The codes that identify message items are defined at the end of the Description section for this routine.

buffer

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Message item. The *buffer* argument is the address of a descriptor of a buffer. The buffer is the one in which the SMB$READ_MESSAGE_ITEM routine is to place the message item data. SMB $READ_MESSAGE_ITEM uses the Run-Time Library string-handling (STR$) routines to copy the message item data into the buffer.

size

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	write only
mechanism:	by reference

Size of the message item. The *size* argument is the address of a word in which the SMB $READ_MESSAGE_ITEM is to place the size, in bytes, of the item's data.

Description

The job controller can request seven functions from the symbiont. They are identified by the following codes:

SMBMSG$K_START_STREAM	SMBMSG$K_STOP_STREAM
SMBMSG$K_START_TASK	SMBMSG$K_PAUSE_TASK
SMBMSG$K_RESUME_TASK	SMBMSG$K_STOP_TASK
SMBMSG$K_RESET_STREAM	

The job controller passes the symbiont a request containing a code and, optionally, a number of message items containing information the symbiont might need to perform the function. The code specifies what function the request is for, and the message items contain information that the symbiont needs to carry out the function.

By calling SMB$READ_MESSAGE, the symbiont reads the request and writes the message items into the specified buffer. The symbiont then obtains the individual message items by calling the SMB $READ_MESSAGE_ITEM routine.

Each message item consists of a code that identifies the information the item represents, and the item itself. For example, the SMB$K_JOB_NAME code tells the symbiont that the item specifies a job's name.

The number of items in a request varies with each type of request. Therefore, you must call SMB $READ_MESSAGE_ITEM repeatedly for each request to ensure that all message items are read. Each time SMB$READ_MESSAGE_ITEM reads a message item, it updates the value in the longword specified by the *context* argument. SMB$READ_MESSAGE_ITEM returns the code SMB $_NOMOREITEMS after it has read the last message item.

The following table shows the message items that can be delivered with each request:

Request	**Message Item**
SMBMSG$K_START_TASK	SMBMSG$K_ACCOUNT_NAME
	SMBMSG$K_AFTER_TIME
	SMBMSG$K_BOTTOM_MARGIN
	SMBMSG$K_CHARACTERISTICS
	SMBMSG$K_CHECKPOINT_DATA
	SMBMSG$K_ENTRY_NUMBER
	SMBMSG$K_FILE_COPIES
	SMBMSG$K_FILE_COUNT
	SMBMSG$K_FILE_IDENTIFICATION
	SMBMSG$K_FILE_SETUP_MODULES
	SMBMSG$K_FILE_SPECIFICATION
	SMBMSG$K_FIRST_PAGE
	SMBMSG$K_FORM_LENGTH
	SMBMSG$K_FORM_NAME
	SMBMSG$K_FORM_SETUP_MODULES
	SMBMSG$K_FORM_WIDTH
	SMBMSG$K_JOB_COPIES
	SMBMSG$K_JOB_COUNT
	SMBMSG$K_JOB_NAME
	SMBMSG$K_JOB_RESET_MODULES
	SMBMSG$K_LAST_PAGE
	SMBMSG$K_LEFT_MARGIN
	SMBMSG$K_MESSAGE_VECTOR
	SMBMSG$K_NOTE
	SMBMSG$K_PAGE_SETUP_MODULES
	SMBMSG$K_PARAMETER_1 ⋮
	SMBMSG$K_PARAMETER_8
	SMBMSG$K_PRINT_CONTROL
	SMBMSG$K_SEPARATION_CONTROL
	SMBMSG$K_REQUEST_CONTROL
	SMBMSG$K_PRIORITY

Request	Message Item
	SMBMSG$K_QUEUE
	SMBMSG$K_RIGHT_MARGIN
	SMBMSG$K_TIME_QUEUED
	SMBMSG$K_TOP_MARGIN
	SMBMSG$K_UIC
	SMBMSG$K_USER_NAME
SMBMSG$K_STOP_TASK	SMBMSG$K_STOP_CONDITION
SMBMSG$K_PAUSE_TASK	None
SMBMSG$K_RESUME_TASK	SMBMSG$K_ALIGNMENT_PAGES
	SMBMSG$K_RELATIVE_PAGE
	SMBMSG$K_REQUEST_CONTROL
	SMBMSG$K_SEARCH_STRING
SMBMSG$K_START_STREAM	SMBMSG$K_DEVICE_NAME
	SMBMSG$K_EXECUTOR_QUEUE
	SMBMSG$K_JOB_RESET_MODULES
	SMBMSG$K_LIBRARY_SPECIFICATION
SMBMSG$K_STOP_STREAM	None
SMBMSG$K_RESET_STREAM	None

The following list describes each item code. For each code, the list describes the contents of the message item identified by the code and whether the code identifies an item sent from the job controller to the symbiont or from the symbiont to the job controller.

Many of the codes described are specifically oriented toward print symbionts. The symbiont you implement, which might not print files or serve an output device, need not recognize all these codes. In addition, it need not respond in the same way as the print symbiont to the codes it recognizes. The descriptions in the list describe how the standard print symbiont (PRTSMB.EXE) processes these items.

Note

Because new codes might be added in the future, you should write your symbiont so that it ignores codes it does not recognize.

Codes for Message Items

[SMBMSG$K_ACCOUNT_NAME]

This code identifies a string containing the name of the account to be charged for the job, that is, the account of the process that submitted the print job.

[SMBMSG$K_AFTER_TIME]

This code identifies a 64-bit, absolute-time value specifying the system time after which the job controller can process this job.

[SMBMSG$K_ALIGNMENT_PAGES]

This code identifies a longword specifying the number of alignment pages that the symbiont is to print.

[SMBMSG$K_BOTTOM_MARGIN]

This code identifies a longword containing the number of lines to be left blank at the bottom of a page.

The symbiont inserts a form feed character into the output stream if it determines that *all* of the following conditions are true:

- The number of lines left at the bottom of the page is equal to the value in SMBMSG $K_BOTTOM_MARGIN.
- Sending more data to the printer to be output on this page would cause characters to be printed within this bottom margin of the page.
- The /FEED qualifier was specified with the PRINT command that caused the symbiont to perform this task.

(Line feed, form feed, carriage-return, and vertical-tab characters in the output stream are collectively known as embedded carriage control.)

[SMBMSG$K_CHARACTERISTICS]

This code identifies a 16-byte structure specifying characteristics of the job. A detailed description of the format of this structure is contained in the description of the QUI$_CHARACTERISTICS code in the $GETQUI system service in the *VSI OpenVMS System Services Reference Manual*.

[SMBMSG$K_DEVICE_NAME]

This code identifies a string that is the name of the device to which the symbiont is to send data. The symbiont interprets this information. The name need not be the name of a physical device, and the symbiont can interpret this string as something other than the name of a device.

[SMBMSG$K_ENTRY_NUMBER]

This code identifies a longword containing the number that the job controller assigned to the job.

[SMBMSG$K_EXECUTOR_QUEUE]

This code identifies a string that is the name of the queue on which the symbiont stream is to be started.

[SMBMSG$K_FILE_COPIES]

This code identifies a longword containing the number of copies of the file that were requested.

[SMBMSG$K_FILE_COUNT]

This code identifies a longword that specifies, out of the number of copies requested for this job (SMBMSG$K_FILE_COPIES), the number of the copy of the file currently printing.

[SMBMSG$K_FILE_IDENTIFICATION]

This code identifies a 28-byte structure identifying the file to be processed. This structure consists of the following three file-identification fields in the OpenVMS RMS NAM block:

1. The 16-byte NAM$T_DVI field

2. The 6-byte NAM$W_FID field

3. The 6-byte NAM$W_DID field

These fields occur consecutively in the NAM block in the order listed.

[SMBMSG$K_FILE_SETUP_MODULES]

This code identifies a string specifying the names (separated by commas) of one or more text modules that the symbiont should copy from the library into the output stream before processing the file.

[SMBMSG$K_FILE_SPECIFICATION]

This code identifies a string specifying the name of the file that the symbiont is to process. This file name is formatted as a standard RMS file specification.

[SMBMSG$K_FIRST_PAGE]

This code identifies a longword containing the number of the page at which the symbiont should begin printing. The job controller sends this item to the symbiont. When not specified, the symbiont begins processing at page 1.

[SMBMSG$K_FORM_LENGTH]

This code identifies a longword value specifying the length (in lines) of the physical form (the paper).

[SMBMSG$K_FORM_NAME]

This code identifies a string specifying the name of the form.

[SMBMSG$K_FORM_SETUP_MODULES]

This code identifies a string consisting of the names (separated by commas) of one or more modules that the symbiont should copy from the device-control library before processing the file.

[SMBMSG$K_FORM_WIDTH]

This code identifies a longword specifying the width (in characters) of the print area on the physical form (the paper).

[SMBMSG$K_JOB_COPIES]

This code identifies a longword specifying the requested number of copies of the job.

[SMBMSG$K_JOB_COUNT]

This code identifies a longword specifying, out of the number of copies requested (SMBMSG $K_JOB_COPIES), the number of the copy of the job currently printing.

[SMBMSG$K_JOB_NAME]

This code identifies a string specifying the name of the job.

[SMBMSG$K_JOB_RESET_MODULES]

This code identifies a string specifying a list of one or more module names (separated by commas) that the symbiont should copy from the device-control library after processing the task. These modules can be used to reset programmable devices to a known state.

[SMBMSG$K_LAST_PAGE]

This code identifies a longword specifying the number of the last page that the symbiont is to print. When not specified, the symbiont attempts to print all the pages in the file.

[SMBMSG$K_LEFT_MARGIN]

This code identifies a longword specifying the number of spaces to be inserted at the beginning of each line.

[SMBMSG$K_LIBRARY_SPECIFICATION]

This code identifies a string specifying the name of the device-control library.

[SMBMSG$K_MESSAGE_VECTOR]

This code identifies a vector of longword condition codes, each of which contains information about the job to be printed.

When LOGINOUT cannot open a log file for a batch job, a code in the message vector specifies the reason for the failure. The job controller does not send the SMBMSG$K_FILE_IDENTIFICATION item if it has detected such a failure but instead sends the message vector, which the symbiont prints, along with a message stating that there is no file to print.

[SMBMSG$K_NOTE]

This code identifies a user-supplied string that the symbiont is to print on the job flag page and on the file flag page.

[SMBMSG$K_PAGE_SETUP_MODULES]

This code identifies a string consisting of the names (separated by commas) of one or more modules that the symbiont should copy from the device-control library before printing each page.

[SMBMSG$K_PARAMETER_1 through SMBMSG$K_PARAMETER_8]

Each of these eight codes identifies a user-supplied string. Both the semantics and syntax of each string are determined by the user-defined symbiont. The OpenVMS-supplied symbiont makes no use of these eight items.

[SMBMSG$K_PRINT_CONTROL]

This code identifies a longword bit vector, each bit of which supplies information that the symbiont is to use in controlling the printing of the file.

Symbol	**Description**
SMBMSG$V_DOUBLE_SPACE	The symbiont uses a double-spaced format; it skips a line after each line it prints.
SMBMSG$V_NO_INITIAL_FF	The symbiont suppresses the initial form feed if this bit is turned on.
SMBMSG$V_NORECORD_BLOCKING	The symbiont performs single record output, issuing a single output record for each input record.
SMBMSG$V_PAGE_HEADER	The symbiont prints a page header at the top of each page.

Symbol	Description
SMBMSG$V_PAGINATE	The symbiont inserts a form feed character when it detects an attempt to print in the bottom margin of the current form.
SMBMSG$V_PASSALL	The symbiont prints the file without formatting and bypasses all formatting normally performed. Furthermore, the symbiont outputs the file without formatting, by causing the output QIO to suppress formatting by the driver.
SMBMSG$V_RECORD_BLOCKING	The symbiont performs record blocking, buffering output to the device.
SMBMSG$V_SEQUENCED	This bit is reserved by VSI.
SMBMSG$V_SHEET_FEED	The symbiont pauses the queue after each page it prints.
SMBMSG$V_TRUNCATE	The symbiont truncates input lines that exceed the right margin of the current form.
SMBMSG$V_WRAP	The symbiont wraps input lines that exceed the right margin, printing the additional characters on a new line.

[SMBMSG$K_PRIORITY]

This code identifies a longword specifying the priority this job has in the queue in which it is entered.

[SMBMSG$K_QUEUE]

This code identifies a string specifying the name of the queue in which this job is entered. When generic queues are used, this item specifies the name of the generic queue, and the SMBMSG$K_EXECUTOR item specifies the name of the device queue or the server queue.

[SMBMSG$K_RELATIVE_PAGE]

This code identifies a signed, longword value specifying the number of pages that the symbiont is to move forward (positive value) or backward (negative value) from the current position in the file.

[SMBMSG$K_REQUEST_CONTROL]

This code identifies a longword bit vector, each bit of which specifies information that the symbiont is to use in processing the request that the job controller is making.

Symbol	Description
SMBMSG$V_ALIGNMENT_MASK	The symbiont is to replace all alphabetic characters with the letter *X*, and all numeric characters with the number *9*. Other characters (punctuation, carriage control, and so on) are left unchanged. This bit is ordinarily specified in connection with the SMBMSG$K_ALIGNMENT_PAGES item.
SMBMSG$V_PAUSE_COMPLETE	The symbiont is to pause when it completes the current request.
SMBMSG$V_RESTARTING	Indicates that this job was previously interrupted and requeued, and is now restarting.

Symbol	Description
SMBMSG$V_TOP_OF_FILE	The symbiont is to rewind the input file before it resumes printing.

[SMBMSG$K_RIGHT_MARGIN]

This code identifies a longword specifying the number of character positions to be left empty at the end of each line. When the right margin is exceeded, the symbiont truncates the line, wraps the line, or continues processing, depending on the settings of the WRAP and TRUNCATE bits in the SMBMSG $K_PRINT_CONTROL item.

[SMBMSG$K_SEARCH_STRING]

This code identifies a string containing the value specified in the START/QUEUE/SEARCH command. This string identifies the page at which to restart the current printing task on a paused queue.

[SMBMSG$K_SEPARATION_CONTROL]

This code identifies a longword bit vector, each bit of which specifies an operation that the symbiont is to perform between jobs or between files within a job. The $SMBDEF macro defines the following symbols for each bit:

Symbol	Description
SMBMSG$V_FILE_BURST	The symbiont is to print a file burst page.
SMBMSG$V_FILE_FLAG	The symbiont is to print a file flag page.
SMBMSG$V_FILE_TRAILER	The symbiont is to print a file trailer page.
SMBMSG$V_FILE_TRAILER_ABORT	The symbiont is to print a file trailer page when a task completes abnormally.
SMBMSG$V_FIRST_FILE_OF_JOB	The current file is the first file of the job. When specified with SMBMSG $V_LAST_FILE_OF_JOB, the current job contains a single file.
SMBMSG$V_JOB_FLAG	The symbiont is to print a job flag page.
SMBMSG$V_JOB_BURST	The symbiont is to print a job burst page.
SMBMSG$V_JOB_RESET	The symbiont is to execute a job reset sequence when the task completes.
SMBMSG$V_JOB_RESET_ABORT	The symbiont is to execute a job reset sequence when a task completes abnormally.
SMBMSG$V_JOB_TRAILER	The symbiont is to print a job trailer page.
SMBMSG$V_JOB_TRAILER_ABORT	The symbiont is to print a job trailer page when a task completes abnormally.
SMBMSG$V_LAST_FILE_OF_JOB	The current file is the last file of the job. When specified with SMBMSG $V_FIRST_FILE_OF_JOB, the current job contains a single job.

[SMBMSG$K_STOP_CONDITION]

This code identifies a longword containing a condition specifying the reason the job controller issued a STOP_TASK request.

[SMBMSG$K_TIME_QUEUED]

This code identifies a quadword specifying the time the file was entered into the queue. The time is expressed as 64-bit, absolute time.

[SMBMSG$K_TOP_MARGIN]

This code identifies a longword specifying the number of lines that the symbiont is to leave blank at the top of each page. PRTSMB inserts line feeds into the output stream after every form feed until the margin is cleared.

[SMBMSG$K_UIC]

This code identifies a longword specifying the user identification code (UIC) of the user who submitted the job.

[SMBMSG$K_USER_NAME]

This code identifies a string specifying the name of the user who submitted the job.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SMB$_NOMOREITEMS

End of item list reached.

This routine also returns any condition code returned by the Run-Time Library string-handling (STR$) routines.

SMB$SEND_TO_JOBCTL

Send Message to Job Controller — The SMB$SEND_TO_JOBCTL routine is used by your symbiont to send messages to the job controller. Three types of messages can be sent: request-completion messages, task-completion messages, and task-status messages.

Format

```
SMB$SEND_TO_JOBCTL stream [,request] [,accounting] [,checkpoint]
 [,device_status] [,error]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

stream

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Stream number specifying the stream to which the message refers. The *stream* argument is the address of a longword containing the number of the stream to which the message refers.

request

OpenVMS usage:	identifier
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Request code identifying the request being completed. The *request* argument is the address of a longword containing the code that identifies the request that has been completed.

The code usually corresponds to the code the job controller passed to the symbiont by means of a call to SMB$READ_MESSAGE. But the symbiont can also initiate task-completion and task-status messages that are not in response to a request. (See the Description section.)

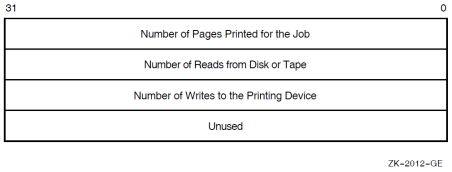
accounting

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Accounting information about a task. The *accounting* argument is the address of a descriptor pointing to the accounting information about a task. Note that this structure is passed by descriptor and not by reference.

The job controller accumulates task statistics into a job-accounting record, which it writes to the accounting file when the job is completed.

The following diagram depicts the contents of the 16-byte structure:



checkpoint

OpenVMS usage:	char_string

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

Checkpoint data about the currently executing task. The *checkpoint* argument is the address of the descriptor that points to checkpointing information that relates to the status of a task. When the symbiont sends this information to the job controller, the job controller saves it in the queue database. When a restart-from-checkpoint request is executed for the queue, the job controller retrieves the checkpointing information from the queue database and sends it to the symbiont in the SMBMSG $K_CHECKPOINT_DATA item that accompanies a SMBMSG$K_START_TASK request.

Print symbionts can use the checkpointing information to reposition the input file to the point corresponding to the page being output when the last checkpoint was taken. Other symbionts might use checkpoint information to specify restart information for partially completed tasks.

Note

Because each checkpoint causes information to be written into the job controller's queue database, taking a checkpoint incurs significant overhead. Use caution in regard to the size and frequency of checkpoints. When determining how often to checkpoint, weigh processor and file-system overhead against the convenience of restarting.

device_status

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Status of the device served by the symbiont. The *device_status* argument is the address of a longword passed to the job controller, which contains the status of the device to which the symbiont is connected.

This longword contains a longword bit vector, each bit of which specifies device-status information. Each programming language provides an appropriate mechanism for defining these device-status bits. The following table describes each bit:

Device Status Bit	Description
SMBMSG$V_LOWERCASE	The device to which the symbiont is connected supports lowercase characters.
SMBMSG$V_PAUSE_TASK	The symbiont sends this message to inform the job controller that the symbiont has paused on its own initiative.
SMBMSG$V_REMOTE	The device is connected to the symbiont by means of a modem.
SMBMSG$V_SERVER	The symbiont is not connected to a device.
SMBMSG$V_STALLED	Symbiont processing is temporarily stalled.
SMBMSG$V_STOP_STREAM	The symbiont requests that the job controller stop the queue.

Device Status Bit	Description
SMBMSG$V_TERMINAL	The symbiont is connected to a terminal.
SMBMSG$V_UNAVAILABLE	The device to which the symbiont is connected is not available.

error

OpenVMS usage:	vector_longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Condition codes returned by the requested task. The *error* argument is the address of a vector of longword condition codes. The first longword contains the number of longwords following it.

If the low bit of the first condition code is clear, the job controller aborts further processing of the job. Output of any remaining files, copies of files, or copies of the job is canceled. In addition, the job controller saves up to three condition values in the queue database. The first condition value is included in the job-accounting record that is written to the system's accounting file (SYS $MANAGER:ACCOUNTNG.DAT).

Description

The symbiont uses the SMB$SEND_TO_JOBCTL routine to send messages to the job controller.

Most messages the symbiont sends to the job controller are responses to requests made by the job controller. These responses inform the job controller that the request has been completed, either successfully or with an error. When the symbiont sends the message, it usually indicates that the request has been completed.

In such messages, the *request* argument corresponds to the function code of the request that has been completed. Thus, if the job controller sends a request using the SMBMSG$K_START_TASK code, the symbiont responds by sending a SMB$SEND_TO_JOBCTL message using SMBMSG $K_START_TASK as the *request* argument.

The responses to some requests use additional arguments to send more information in addition to the request code. The following table shows which additional arguments are allowed in response to each different request:

* - (This is usually the value specified in the SMBMSG$K_STOP_CONDITION item * – (This is usually the value specified in the SMBMSG$K_STOP_CONDITION item * – This is usually the value specified in the SMBMSG$K_STOP_CONDITION item that was sent by the job controller with the SMBMSG$K_STOP_TASK request.

Request	Arguments
SMBMSG$K_START_STREAM	request
	device_status
	error
SMBMSG$K_STOP_STREAM	request

Request	Arguments
SMBMSG$K_RESET_STREAM	request
SMBMSG$K_START_TASK	request
SMBMSG$K_PAUSE_TASK	request
SMBMSG$K_RESUME_TASK	request
SMBMSG$K_STOP_TASK	request
	error [1]

[1]This is usually the value specified in the SMBMSG$K_STOP_CONDITION item that was sent by the job controller with the SMBMSG $K_STOP_TASK request.

In addition to responding to requests from the job controller, the symbiont can send other messages to the job controller. If the symbiont sends a message that is not a response to a request, it uses either the SMBMSG$K_TASK_COMPLETE or SMBMSG$K_TASK_STATUS code. Following are the additional arguments that you can use with the messages identified by these codes:

Code	Arguments
SMBMSG$K_TASK_COMPLETE	request
	accounting
	error
SMBMSG$K_TASK_STATUS	request
	checkpoint
	device_status

The symbiont uses the SMBMSG$K_TASK_STATUS message to update the job controller on the status of a task during the processing of that task. The checkpoint information passed to the job controller with this message permits the job controller to restart an interrupted task from an appropriate point. The device-status information permits the symbiont to report changes in device's status (device stalled, for example).

The symbiont can use the SMBMSG$K_TASK_STATUS message to request that the job controller send a stop-stream request. It does this by setting the stop-stream bit in the *device-status* argument.

The symbiont can also use the SMBMSG$K_TASK_STATUS message to notify the job controller that the symbiont has paused in processing a task. It does so by setting the pause-task bit in the *device-status* argument.

The symbiont uses the SMBMSG$K_TASK_COMPLETE message to signal the completion of a task. Note that, when the symbiont receives a START_TASK request, it responds by sending a SMB $SEND_TO_JOBCTL message with SMBSMG$K_START_TASK as the *request* argument. This response means that the symbiont has started the task; it does not mean the task has been completed. When the symbiont has completed a task, it sends a SMB$SEND_TO_JOBCTL message with SMBMSG$K_TASK_COMPLETE as the *request* argument.

Optionally, the symbiont can specify accounting information when sending a task-completion message. The accounting statistics accumulate to give a total for the job when the job is completed.

Also, if the symbiont is aborting the task because of a symbiont-detected error, you can specify up to three condition values in the *error* argument. Aborting a task causes the remainder of the job to be aborted.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

This routine also returns any condition value returned by the $QIO system service and the LIB $GET_VM routine.

Chapter 20. Sort/Merge (SOR) Routines

The Sort/Merge (SOR) routines allow you to integrate a sort or merge operation into a program application. Using these callable routines, you can process records, sort or merge them, and then process them again.

20.1. High-Performance Sort/Merge (Alpha Only)

You can also choose the high-performance Sort/Merge utility. This utility takes advantage of the Alpha architecture to provide better performance for most sort and merge operations.

In addition, the high-performance Sort/Merge utility can increase performance by using threads to take advantage of multiple processors on an SMP configured system. Refer to Section 20.1.2 for further information about using threads.

The high-performance Sort/Merge utility supports a subset of the SOR routines. Any differences between the high-performance Sort/Merge utility and Sort/Merge utility (SORT/MERGE) are noted within this chapter.

Note

Memory allocation differences may limit the high-performance Sort/Merge utility's ability to perform the same number of concurrent sort operations as the Sort/Merge utility can perform in the same amount of virtual memory.

If this situation occurs, you can either increase the amount of virtual memory that is available to the process, or reduce the working set extent. For information on using system parameters to change the amount of virtual memory or reduce the working set extent, refer to the *OpenVMS System Management Utilities Reference Manual*.

Use the SORTSHR logical name to select the high-performance Sort/Merge utility. Define SORTSHR to point to the high-performance sort executable in SYS$LIBRARY as follows:

```
$ define sortshr sys$library:hypersort.exe
```

To return to SORT/MERGE, deassign SORTSHR. The Sort/Merge utility is the default if SORTSHR is not defined.

20.1.1. High-Performance SOR Routine Behavior

The behavior of the SOR routines for the high-performance Sort/Merge utility is the same as for SORT/MERGE except as shown in Table 20.1.

If you attempt to use an unsupported capability, the high-performance Sort/Merge utility generates an error. The high-performance Sort/Merge utility adds the following condition value to those listed for SORT/MERGE:

SOR$_NYI	Attempt to use a feature that is not yet implemented.

Table 20.1. High-Performance Sort/Merge: Differences in SOR$ Routine Behavior

Feature	High-Performance Sort/Merge Behavior
Work files	Permissible values of the SOR$BEGIN_SORT *work_files* argument range from 1 through 255. By default, the high-performance Sort/Merge utility creates two temporary work files.
Input file size	If you do not specify an input file size in the SOR$BEGIN_SORT *file_alloc* argument, the high-performance Sort/Merge utility determines a default based on the size of the input file, or if input is not from files, on available memory.
Specification files	The SOR$SPEC_FILE routine is not supported. (Implementation of this feature is deferred to a future OpenVMS Alpha release.)
Key data types	DSCK_DTYPE_O, DSCK_DTYPE_OU, DSC$K_DTYPE_H, and DSC$K_DTYPE_NZ are not valid key data types in the SOR$BEGIN_MERGE or SOR$BEGIN_SORT *key_buffer* argument. (Implementation of this feature is deferred to a future OpenVMS Alpha release.)
Key data types not normally supported by SORT/MERGE	The SOR$DTYPE routine is not supported. (Implementation of this feature is deferred to a future OpenVMS Alpha release.) Data types that would otherwise be specified using SOR$DTYPE include extended data types and the National Character Set (NCS) collating sequences.
Internal sorting processes	Only the record sort process is supported. You can specify the SOR$BEGIN_SORT routine *sort_process* argument as SOR $GK_RECORD or omit the argument. The SOR$GK_TAG, SOR $GK_ADDRESS, and SOR$GK_INDEX values are not supported for the *sort_process* argument. (Implementation of this feature is deferred to a future OpenVMS Alpha release.)
Statistical summary information	The following statistics are currently supported: Records read/input (SOR$K_REC_INP) Records sorted (SOR$K_REC_SOR) Records output (SOR$K_REC_OUT) Input record length (SOR$K_LRL_INP) The following statistics are currently unavailable: Internal length Output record length Sort tree size Number of initial runs Maximum merge order Number of merge passes Work file allocation Full implementation of this feature is deferred to a future OpenVMS Alpha release.
User-supplied action routines	The following user-supplied action routines are not supported for either SOR$BEGIN_MERGE or SOR$BEGIN_SORT. (Implementation of this feature is deferred to a future OpenVMS Alpha release.) You must provide a placeholder comma (,) in the

Feature	High-Performance Sort/Merge Behavior	
	argument list if other arguments follow the customary position of the *user_compare* or *user_equal* argument.	
	user_compare	Compares records to determine their sort or merge order.
	user_equal	Resolves the sort or merge order when records have duplicate keys.

20.1.2. Using Threads with High-Performance Sort/Merge

The high-performance Sort/Merge utility can take advantage of multiple processors on an SMP configured system by using threads to gain additional performance. Threads use is optimized under the following conditions:

- the SYSGEN parameter MULTITHREAD is set to the number of CPUs on the system
- the base image of the application using the high-performance Sort/Merge utility is linked with the /THREADS_ENABLE qualifier

When linking an executable image that uses the high-performance Sort/Merge utility, the executable should be linked with the /THREADS_ENABLE linker qualifier. Either /THREADS_ENABLE or /THREADS_ENABLE=(MULTIPLE_KERNEL_THREADS,UPCALLS) qualifiers may be used. (Refer to the *Guide to DECthreads* manual in the OpenVMS documentation set for more information on this linker qualifier.)

The high-performance Sort/Merge utility will not utilize multiple processors, and therefore won't run at peak performance, if the /THREADS_ENABLE linker qualifier is omitted, explicitly disabled (by the /NOTHREADS_ENABLED), or partially enabled (by the /THREADS_ENABLE=UPCALLS or /THREADS_ENABLE=MULTIPLE_KERNEL_THREADS). However, the high-performance Sort/Merge utility will still run and produce correct results.

20.2. Introduction to SOR Routines

The following SOR routines are available for use in a sort or merge operation:

Routine	Description
SOR$BEGIN_MERGE	Sets up key arguments and performs the merge. This is the only routine unique to MERGE.
SOR$BEGIN_SORT	Initializes the sort operation by passing key information and sort options. This is the only routine unique to SORT.
SOR$DTYPE	Defines a key data-type that is not normally supported by SORT/MERGE. (This feature is not currently supported by the high-performance Sort/Merge utility.)
SOR$END_SORT	Performs cleanup functions, such as closing files and releasing memory.
SOR$PASS_FILES	Passes names of input and output files to SORT or MERGE must be repeated for each input file.

Routine	Description
SOR$RELEASE_REC	Passes one input record to SORT or MERGE must be called once for each record.
SOR$RETURN_REC	Returns one sorted or merged record to a program must be called once for each record.
SOR$SORT_MERGE	Sorts the records.
SOR$SPEC_FILE	Passes a specification file or specification text. A call to this routine must precede all other calls to the SOR routines. (This feature is not currently supported by the high-performance Sort/Merge utility.)
SOR$STAT	Returns a statistic about the sort or merge operation. (This feature is partially supported by the high-performance Sort/Merge utility.)

You can call these SOR routines from any language that supports the OpenVMS calling standard. Note that the application program should declare referenced constants and return status symbols as external symbols these symbols will be resolved upon linking with the utility shareable image.

After being called, each of these routines performs its function and returns control to a program. It also returns a 32-bit condition code value indicating success or error, which a program can test to determine success or failure conditions.

20.2.1. Arguments to SOR Routines

For a sort operation, the arguments to the SOR routines provide SORT with file specifications, key information, and instructions about the sorting process. For a merge operation, the arguments to the SOR routines provide MERGE with the number of input files, input and output file specifications, record information, key information, and input routine information. To perform sort or merge operations, you must pass key information (*key_buffer* argument) to either the SOR$BEGIN_SORT or SOR $BEGIN_MERGE routine. The *key_buffer* argument is passed as an array of words. The first word of the array contains the number of keys to be used in the sort or merge. Each block of four words that follows describes one key (multiple keys are listed in order of their priority):

- The first word of each block describes the key data type.
- The second word determines the sort or merge order (0 for ascending, 1 for descending).
- The third word describes the relative offset of the key (beginning at position 0).
- The fourth word describes the length of the key in bytes.

There are both mandatory and optional arguments. The mandatory arguments appear first in the argument list. You must specify all arguments in the order in which they are positioned in the argument list, separating each with a comma. Pass a zero by value to specify any optional arguments that you are omitting from within the list. You can end the argument list any time after specifying all the mandatory and desired optional arguments.

20.2.2. Interfaces to SOR Routines

You can submit data to the SOR routines as complete files or as single records. When your program submits one or more files to SORT or MERGE, which then creates one sorted or merged output file,

you are using the file interface. When your program submits records one at a time and then receives the ordered records one at a time, you are using the record interface.

You can combine the file interface with the record interface by submitting files on input and receiving the ordered records on output or by releasing records on input and writing the ordered records to a file on output. Combining the two interfaces provides greater flexibility. If you use the record interface on input, you can process the records before they are sorted; if you use the record interface on output, you can process the records after they are sorted.

The SOR routines used and the order in which they are called depend on the type of interface used in a sorting or merging operation. The following sections detail the calling sequence for each of the interfaces.

20.2.2.1. Sort Operation Using File Interface

For a sort operation using the file interface, pass the input and output file specifications to SORT by calling SOR$PASS_FILES. You must call SOR$PASS_FILES for each input file specification. Pass the output file specification in the first call. If no input files are specified before the call to SOR $BEGIN_SORT, the record interface is used for input; if no output file is specified, the record interface is used for output.

Next, call SOR$BEGIN_SORT to pass instructions about keys and sort options. At this point, you must indicate whether you want to use your own key comparison routine. (This feature is not currently supported by the high-performance Sort/Merge utility.) SORT automatically generates a key comparison routine that is efficient for key data types; however, you might want to provide your own comparison routine to handle special sorting requirements. (For example, you might want names beginning with “Mc” and “Mac” to be placed together.) If you use your own key comparison routine, you must pass its address with the *user_compare* argument.

Call SOR$SORT_MERGE to execute the sort and direct the sorted records to the output file. Finally, call SOR$END_SORT to end the sort and release resources. The SOR$END_SORT routine can be called at any time to abort a sort or to merge and release all resources allocated to the sort or merge process.

20.2.2.2. Sort Operation Using Record Interface

For a sort operation using the record interface, first call SOR$BEGIN_SORT. As in the file interface, this routine sets up work areas and passes arguments that define keys and sort options. Note that, if you use the record interface, you must use a record-sorting process (not a tag, address, or index process).

Next, call SOR$RELEASE_REC to release a record to SORT. Call SOR$RELEASE_REC once for each record to be released. After all records have been passed to SORT, call SOR$SORT_MERGE to perform the sorting.

After the sort has been performed, call SOR$RETURN_REC to return a record from the sort operation. Call this routine once for each record to be returned. Finally, call the last routine, SOR$END_SORT, to complete the sort operation and release resources.

20.2.2.3. Merge Operation Using File Interface

For a merge operation using the file interface, pass the input and output file specifications to MERGE by calling SOR$PASS_FILES. You can merge up to 10 input files. (The high-performance Sort/Merge utility allows you to merge up to 12 input files.) by calling SOR$PASS_FILES once for each file. Pass

the file specification for the merged output file in the first call. If no input files are specified before the call to SOR$BEGIN_MERGE, the record interface is used for input; if no output file is specified, the record interface is used for output.

Next, to execute the merge, call SOR$BEGIN_MERGE to pass key information and merge options. At this point, you must indicate whether you want to use your own key comparison routine tailored to your data. (This feature is not currently supported by the high-performance Sort/Merge utility.) Finally, call SOR$END_SORT to release resources.

20.2.2.4. Merge Operation Using Record Interface

For a merge operation using the record interface, first call SOR$BEGIN_MERGE. As in the file interface, this routine passes arguments that define keys and merge options. It also issues the first call to the input routine, which you must create, to begin releasing records to the merge.

Next, call SOR$RETURN_REC to return the merged records to your program. You must call this routine once for each record to be returned. SOR$RETURN_REC continues to call the input routine. MERGE, unlike SORT, does not need to hold all the records before it can begin returning them in the desired order. Releasing, merging, and returning records all take place in this phase of the merge.

Finally, after all the records have been returned, call the last routine, SOR$END_SORT, to clean up and release resources.

20.2.3. Reentrancy

The SOR routines are reentrant; that is, a number of sort or merge operations can be active at the same time. Thus, a program does not need to finish one sort or merge operation before beginning another. For example, reentrancy lets you perform multiple sorts on a file such as a mailing list and to create several output files, one with the records sorted by name, another sorted by state, another sorted by zip code, and so on.

The *context* argument, which can optionally be passed with any of the SOR routines, distinguishes among multiple sort or merge operations. When using multiple sort or merge operations, the *context* argument is required. On the first call, the context longword must be zero. It is then set (by SORT/MERGE) to a value identifying the sort or merge operation. Additional calls to the same sort or merge operation must pass the same context longword. The SOR$END_SORT routine clears the context longword.

20.3. Using the SOR Routines: Examples

This section provides examples of using the SOR routines for various operations including the following:

- Example 20.1 is a VSI Fortran program that demonstrates a merge operation using a record interface.
- Example 20.2 is a VSI Fortran program that demonstrates a sort operation using a file interface on input and a record interface on output.
- Example 20.3 is a VSI Pascal program that demonstrates a merge operation using a file interface.
- Example 20.4 is a VSI Pascal program that demonstrates a sort operation using a record interface.
- Example 20.5 is a VSI C program that demonstrates a sort operation using the STABLE option and two text keys.

Example 20.1. Using SOR Routines to Perform a Merge Using Record Interface in a VSI Fortran Program

```
        Fortran Program
C...
C...    This program demonstrates the Fortran calling sequences
C...    for the merge record interface.
C...
C
C       THE INPUT FILES ARE LISTED BELOW.
C
C               INFILE1.DAT
C
C 1 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF RECORD
C
C               INFILE2.DAT
C
C 1 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C
C               INFILE3.DAT
C
C 1 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C
C               FOROUT.DAT
C
C 1 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF RECORD
C 1 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF RECORD
C
C
C.....................................................................
C
C


        IMPLICIT INTEGER (A-Z)

        CHARACTER*80 REC                  ! A record.

        EXTERNAL READ_REC                 ! Routine to read a record.
        EXTERNAL KOMPAR                   ! Routine to compare records.
        EXTERNAL SS$_ENDOFFILE            ! System end-of-file value

        INTEGER*4 SOR$BEGIN_MERGE         ! SORT/MERGE function names
        INTEGER*4 SOR$RETURN_REC
        INTEGER*4 SOR$END_SORT
        INTEGER*4 ISTAT                   ! storage for SORT/MERGE function value
        INTEGER*4 LENGTH                  ! length of the returned record
        INTEGER*2 LRL                     ! Longest Record Length (LRL)

        LOGICAL*1 ORDER                   ! #files to merge (merge order)

        DATA ORDER,LRL/3,80/              ! Order of the merge=3,LRL=80
C...
C...    First open all the input files.
C...
        OPEN (UNIT=10, FILE='INFILE1.DAT',TYPE='OLD',READONLY,
     *  FORM='FORMATTED')
        OPEN (UNIT=11, FILE='INFILE2.DAT',TYPE='OLD',READONLY,
     *  FORM='FORMATTED')
```

```
        OPEN (UNIT=12, FILE='INFILE3.DAT',TYPE='OLD',READONLY,
     *  FORM='FORMATTED')
C
C...    Open the output file.
C
        OPEN (UNIT=8, FILE='TEMP.TMP', TYPE='NEW')
C...
C...    Initialize the merge.  Pass the merge order, the largest
C...    record length, the compare routine address, and the
C...    input routine address.
C...
        ISTAT = SOR$BEGIN_MERGE (,LRL,,ORDER,
     *  KOMPAR,,READ_REC)
        IF (.NOT. ISTAT) GOTO 10        ! Check for error.

C...
C...    Now loop getting merged records.  SOR$RETURN_REC will
C...    call READ_REC when it needs input.
C...
5       ISTAT = SOR$RETURN_REC (REC, LENGTH)
        IF (ISTAT .EQ. %LOC(SS$_ENDOFFILE)) GO TO 30    ! Check for end of file.
        IF (.NOT. ISTAT) GO TO 10       ! Check for error.

        WRITE(8,200) REC                        ! Output the record.
200     FORMAT(' ',A)
        GOTO 5                                  ! And loop back.
C...
C...    Now tell SORT that we are all done.
C...

30      ISTAT = SOR$END_SORT()
        IF (.NOT. ISTAT) GOTO 10        ! Check for error.
        CALL EXIT

C...
C...    Here if an error occurred.  Write out the error status
C...    and exit.
C...
10      WRITE(8,201)ISTAT
201     FORMAT(' ?ERROR CODE', I20)
        CALL EXIT
        END


        FUNCTION READ_REC (RECX, FILE, SIZE)
C...
C...    This routine reads a record from one of the input files
C...    for merging.  It will be called by SOR$BEGIN_MERGE and by
C...    SOR$RETURN_REC.
C...    Parameters:
C...
C...            RECX.wcp.ds     character buffer to hold the record after
C...                            it is read in.
C...
C...            FILE.rl.r       indicates which file the record is
C...                            to be read from.  1 specifies the
C...                            first file, 2 specifies the second
C...                            etc.
C...
C...            LENGTH.wl.r     is the actual number of bytes in
C...                            the record.  This is set by READ_REC.
C...
        IMPLICIT INTEGER (A-Z)

        PARAMETER MAXFIL=10                     ! Max number of files.
```

```
        EXTERNAL SS$_ENDOFFILE                  ! End of file status code.
        EXTERNAL SS$_NORMAL                     ! Success status code.

        LOGICAL*1 FILTAB(MAXFIL)
        CHARACTER*(80) RECX                     ! MAX LRL =80

        DATA FILTAB/10,11,12,13,14,15,16,17,18,19/ ! Table of I/O unit numbers.

        READ_REC = %LOC(SS$_ENDOFFILE)          ! Give end of file return
        IF (FILE .LT. 1 .OR. FILE .GT. MAXFIL) RETURN   !   if illegal call.

        READ (FILTAB(FILE), 100, ERR=75, END=50) RECX   ! Read the record.
100     FORMAT(A)

        READ_REC = %LOC(SS$_NORMAL)             ! Return success code.
        SIZE = LEN (RECX)                       ! Return size of record.
        RETURN

C...    Here if end of file.
50      READ_REC = %LOC(SS$_ENDOFFILE)          ! Return "end of file" code.
        RETURN

C...    Here if error while reading
75      READ_REC = 0
        SIZE = 0
        RETURN
        END


        FUNCTION KOMPAR (REC1,REC2)
C...
C...    This routine compares two records.  It returns -1
C...    if the first record is smaller than the second,
C...    0 if the records are equal, and 1 if the first record
C...    is larger than the second.
C...
        PARAMETER KEYSIZ=10

        IMPLICIT INTEGER (A-Z)

        LOGICAL*1 REC1(KEYSIZ),REC2(KEYSIZ)

        DO 20 I=1,KEYSIZ
        KOMPAR = REC1(I) - REC2(I)
        IF (KOMPAR .NE. 0) GOTO 50
20      CONTINUE

        RETURN

50      KOMPAR = ISIGN (1, KOMPAR)
        RETURN
        END
```

Example 20.2 is a VSI Fortran program that demonstrates a sort operation using a file interface on input and a record interface on output.

Example 20.2. Using SOR Routines to Sort Using Mixed Interface in a VSI Fortran Program

```
Program

         PROGRAM CALLSORT
C
C
```

```
C       This is a sample Fortran program that calls the SOR
C       routines using the file interface for input and the
C       record interface for output.  This program requests
C       a record sort of the file 'R010SQ.DAT'  and  writes
C       the records to SYS$OUTPUT.  The key is  an  80-byte
C       character ascending  key starting in  position 1 of
C       each record.
C
C       A short version of the input and output files follows:
C
C                       Input file R010SQ.DAT
C 1 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF
 RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF
 RECORD
C 1 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF
 RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF
 RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF
 RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF
 RECORD
C 1 QQQQQQQQQQ REST OF DATA IN RECORD.........................END OF
 RECORD
C 2 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF
 RECORD
C 1 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF
 RECORD
C 2 QQQQQQQQQQ REST OF DATA IN RECORD.........................END OF
 RECORD
C
C                       Output file SYS$OUTPUT
C
C 1 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF RECORD
C 1 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 1 QQQQQQQQQQ REST OF DATA IN RECORD.........................END OF RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 1 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF RECORD
C 2 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 2 QQQQQQQQQQ REST OF DATA IN RECORD.........................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF RECORD
C
C------------------------------------------------------------------------
C
C       Define external functions and data.
C
        CHARACTER*80 RECBUF
        CHARACTER*10 INPUTNAME          !Input file name
        INTEGER*2 KEYBUF(5)             !Key definition buffer
        INTEGER*4 SOR$PASS_FILES        !SORT function names
        INTEGER*4 SOR$BEGIN_SORT
        INTEGER*4 SOR$SORT_MERGE
        INTEGER*4 SOR$RETURN_REC
        INTEGER*4 SOR$END_SORT
        INTEGER*4 ISTATUS               !Storage for SORT function value
```

```
        EXTERNAL SS$_ENDOFFILE
        EXTERNAL DSC$K_DTYPE_T
        EXTERNAL SOR$GK_RECORD
        INTEGER*4 SRTTYPE

C
C       Initialize data -- first the file names, then the key buffer for
C       one 80-byte character key starting in position 1, 3 work files,
C       and a record sort process.
C
        DATA INPUTNAME/'R010SQ.DAT'/
        KEYBUF(1) = 1
        KEYBUF(2) = %LOC(DSC$K_DTYPE_T)
        KEYBUF(3) = 0
        KEYBUF(4) = 0
        KEYBUF(5) = 80
        SRTTYPE = %LOC(SOR$GK_RECORD)

C
C       Call the SORT -- each call is a function.
C
C
C       Pass SORT the file names.
C
        ISTATUS = SOR$PASS_FILES(INPUTNAME)
        IF (.NOT. ISTATUS) GOTO 10
C
C       Initialize the work areas and keys.
C
        ISTATUS = SOR$BEGIN_SORT(KEYBUF,,,,,,SRTTYPE,%REF(3))
        IF (.NOT. ISTATUS) GOTO 10
C
C       Sort the records.
C
        ISTATUS = SOR$SORT_MERGE( )
        IF (.NOT. ISTATUS) GOTO 10
C
C       Now retrieve the individual records and display them.
C
5       ISTATUS = SOR$RETURN_REC(RECBUF)
        IF (.NOT. ISTATUS) GOTO 6
        ISTATUS = LIB$PUT_OUTPUT(RECBUF)
        GOTO 5
6       IF (ISTATUS .EQ. %LOC(SS$_ENDOFFILE)) GOTO 7
        GOTO 10
C
C       Clean up the work areas and files.
C
7       ISTATUS = SOR$END_SORT()
        IF (.NOT. ISTATUS) GOTO 10
        STOP 'SORT SUCCESSFUL'
10      STOP 'SORT UNSUCCESSFUL'
        END
```

Example 20.3 is a VSI Pascal program that demonstrates a merge operation using a file interface.

Example 20.3. Using SOR Routines to Merge Three Input Files in a VSI Pascal Program

```
Program


(* This program merges three input files, (IN_FILE.DAT,
 IN_FILE2.DAT IN_FILE3.DAT), and creates one merged output file.  *)


program mergerecs( output, in_file1, in_file2, in_file3, out_file );

CONST
   SS$_NORMAL = 1;
   SS$_ENDOFFILE = %X870;
   SOR$GK_RECORD = 1;
   SOR$M_STABLE = 1;
   SOR$M_SEQ_CHECK = 4;
   SOR$M_SIGNAL = 8;
   DSC$K_DTYPE_T = 14;

TYPE
   $UBYTE = [BYTE] 0..255;
   $UWORD = [WORD] 0..65535;

const
    num_of_keys = 1;
    merge_order = 3;
    lrl         = 131;

    ascending   = 0;
    descending  = 1;


type
    key_buffer_block=
        packed record
        key_type:       $uword;
        key_order:      $uword;
        key_offset:     $uword;
        key_length:     $uword;
        end;

   key_buffer_type=
        packed record
        key_count:      $uword;
        blocks:         packed array[1..num_of_keys] of key_buffer_block;
        end;

   record_buffer =      packed array[1..lrl] of char;

   record_buffer_descr =
        packed record
        length: $uword;
        dummy:  $uword;
        addr:   ^record_buffer;
        end;

var
```

```
   in_file1,
   in_file2,
   in_file3,
   out_file:    text;
   key_buffer:  key_buffer_type;
   rec_buffer:  record_buffer;
   rec_length:  $uword;
   status:      integer;
   i:           integer;

function sor$begin_merge(
        var buffer:     key_buffer_type;
        lrl:            $uword;
        mrg_options:    integer;
        merge_order:    $ubyte;
        %immed cmp_rtn: integer := 0;
        %immed eql_rtn: integer := 0;
        %immed [unbound] function
            read_record(
                var rec:        record_buffer_descr;
                var filenumber: integer;
                var recordsize: $uword): integer
        ): integer; extern;

function sor$return_rec(
        %stdescr rec:   record_buffer;
        var rec_size:   $uword
        ): integer; extern;

function sor$end_sort: integer; extern;

procedure sys$exit( %immed status : integer ); extern;

function read_record(
        var rec:        record_buffer_descr;
        var filenumber: integer;
        var recordsize: $uword
        ): integer;

procedure readone( var filename: text );
begin
recordsize := 0;
if eof(filename)
then
    read_record := ss$_endoffile
else
    begin
    while not eoln(filename) and (recordsize < rec.length) do
        begin
        recordsize := recordsize + 1;
        read(filename,rec.addr^[recordsize]);
        end;
    readln(filename);
    end;
end;

begin
read_record := ss$_normal;
```

```
case filenumber of
    1: readone(in_file1);
    2: readone(in_file2);
    3: readone(in_file3);
    otherwise
        read_record := ss$_endoffile;
    end;
end;

procedure initfiles;
begin
open( in_file1, 'infile1.dat', old );
open( in_file2, 'infile2.dat', old );
open( in_file3, 'infile3.dat', old );
open( out_file, 'temp.tmp' );
reset( in_file1 );
reset( in_file2 );
reset( in_file3 );
rewrite( out_file );
end;

procedure error( status : integer );
begin
writeln( 'merge unsuccessful.  status=%x', status:8 hex );
sys$exit(status);
end;
begin

with key_buffer do
    begin
    key_count := 1;
    with blocks[1] do
        begin
        key_type := dsc$k_dtype_t;
        key_order := ascending;
        key_offset := 0;
        key_length := 5;
        end;
    end;

initfiles;

status := sor$begin_merge( key_buffer, lrl,
        sor$m_seq_check + sor$m_signal,
        merge_order, 0, 0, read_record );

repeat
    begin
    rec_length := 0;
    status := sor$return_rec( rec_buffer, rec_length );
    if odd(status)
    then
        begin
        for i := 1 to rec_length do write(out_file, rec_buffer[i]);
        writeln(out_file);
        end;
    end
until not odd(status);
```

```
if status <> ss$_endoffile then error(status);

status := sor$end_sort;
if not odd(status) then error(status);

writeln( 'merge successful.' );

end.
```

Example 20.4 is a VSI Pascal program that demonstrates a sort operation using a record interface.

Example 20.4. Using SOR Routines to Sort Records from Two Input Files in a VSI Pascal Program

```
Pascal Program

PROGRAM FILETORECORDSORT (OUTPUT,SORTOUT);

(*      This program calls SOR routines to read and sort records from
        two input files, (PASINPUT1.DAT and PASINPUT2.DAT) and to return
        sorted records to this program to be written to the output file,
        (TEMP.TMP).  *)

(*      Declarations for external status codes, and data structures, such
        as the types $UBYTE (an unsigned byte) and $UWORD (an unsigned
 word). *)

CONST
   SS$_NORMAL = 1;
   SS$_ENDOFFILE = %X870;
   SOR$GK_RECORD = 1;
   SOR$M_STABLE = 1;
   SOR$M_SEQ_CHECK = 4;
   SOR$M_SIGNAL = 8;
   DSC$K_DTYPE_T = 14;

TYPE
   $UBYTE = [BYTE] 0..255;
   $UWORD = [WORD] 0..65535;

CONST
   Numberofkeys = 1 ;   (* Number of keys for this sort *)
   LRL = 131 ;          (* Longest Record Length for output records *)

(* Key orders *)

   Ascending = 0 ;
   Descending = 1 ;


TYPE
   Keybufferblock= packed record
                   Keytype : $UWORD ;
                   Keyorder : $UWORD ;
                   Keyoffset : $UWORD ;
                   Keylength : $UWORD
                   end ;
```

```
(* The keybuffer. Note that the field buffer is a one-component array in
   this program. This type definition would allow a multikeyed sort. *)

   Keybuffer= packed record
              Numkeys : $UWORD ;
              Blocks : packed array[1..Numberofkeys] OF Keybufferblock
              end ;

(* The record buffer. This buffer will be used to hold the returned
        records from SORT. *)

   Recordbuffer = packed array[1..LRL] of char ;

(* Name type for input and output files. A necessary fudge for %stdescr
   mechanism.  *)

   nametype= packed array[1..13] of char ;



VAR
   Sortout : text ;              (* the output file *)
   Buffer : Keybuffer ;          (* the actual keybuffer *)
   Sortoptions : integer ;       (* flag for sorting options *)
   Sorttype : $UBYTE ;           (* sorting process *)
   Numworkfiles : $UBYTE ;       (* number of work files *)
   Status : integer ;            (* function return status code *)
   Rec : Recordbuffer ;          (* a record buffer *)
   Recordlength : $UWORD ;       (* the returned record length *)
   Inputname:  nametype ;        (* input file name *)
   i : integer ;                 (* loop control variable *)



(* function and procedure declarations *)

(* Declarations of SORT functions *)
(* Note that the following SORT routine declarations
        do not use all of the possible routine parameters. *)
(* The parameters used MUST have all preceding parameters specified,
        however. *)

FUNCTION SOR$PASS_FILES
   (%STDESCR Inname : nametype )
   : INTEGER ; EXTERN ;

FUNCTION SOR$BEGIN_SORT(
    VAR Buffer : Keybuffer ;
    Lrlen : $UWORD ;
    VAR Sortoptions : INTEGER ;
    %IMMED Filesize : INTEGER ;
    %IMMED Usercompare : INTEGER ;
    %IMMED Userequal : INTEGER ;
    VAR Sorttype : $UBYTE ;
    VAR Numworkfiles : $UBYTE )
    : INTEGER ; EXTERN ;
```

```
FUNCTION SOR$SORT_MERGE
   : INTEGER ; EXTERN ;

FUNCTION SOR$RETURN_REC(
   %STDESCR  Rec : Recordbuffer ;
   VAR Recordsize : $UWORD )
   : INTEGER ; EXTERN ;

FUNCTION SOR$END_SORT
   : INTEGER ; EXTERN ;


(* End of the SORT function declarations *)

(* The CHECKSTATUS routine checks the return status for errors. *)
(* If there is an error, write an error message and exit via sys$exit *)
PROCEDURE CHECKSTATUS( var status : integer ) ;

        procedure sys$exit( status : integer ) ; extern ;

begin           (* begin checkstatus *)
   if odd(status) then
        begin
        writeln( ' SORT unsuccessful. Error status = ', status:8 hex ) ;
        SYS$EXIT( status ) ;
        end ;
end ;           (* end checkstatus *)

(* end function and routine declarations *)


BEGIN   (* begin the main routine *)

(* Initialize data for one 8-byte character key, starting at record
   offset 0, 3 work files, and the record sorting process *)

Inputname := 'PASINPUT1.DAT' ;
WITH Buffer DO
   BEGIN
   Numkeys := 1;
   WITH Blocks[1] DO
      BEGIN
      Keytype := DSC$K_DTYPE_T ;        (* Use OpenVMS descriptor data
 types to
                                                define SORT data types. *)
      Keyorder := Ascending ;
      Keyoffset := 0 ;
      Keylength := 8 ;
      END;
   END;

Sorttype := SOR$GK_RECORD ;             (* Use the global SORT constant to
                                             define the sort process. *)
Sortoptions := SOR$M_STABLE ;           (* Use the global SORT constant to
                                             define the stable sort option.
 *)
Numworkfiles := 3 ;
```

```
(* call the sort routines as a series of functions *)

(* pass the first filename to SORT *)
Status := SOR$PASS_FILES( Inputname ) ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* pass the second filename to SORT *)
Inputname := 'PASINPUT2.DAT' ;

Status := SOR$PASS_FILES( Inputname ) ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* initialize work areas and keys *)
Status := SOR$BEGIN_SORT( Buffer, 0, Sortoptions, 0, 0, 0,
                                Sorttype, Numworkfiles ) ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* sort the records *)
Status := SOR$SORT_MERGE ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* Ready output file for writing returned records from SORT. *)
OPEN( SORTOUT, 'TEMP.TMP' ) ;
REWRITE( SORTOUT ) ;

(* Now get the sorted records from SORT. *)
Recordlength := 0 ;
REPEAT
   Status := SOR$RETURN_REC( Rec, Recordlength ) ;

   if odd( Status )
   then                 (* if successful, write record to output file. *)
        begin
        for i := 1 to Recordlength do
           write( sortout, Rec[i] ) ;   (* write each character *)
        writeln (sortout) ;                     (* end output line *)
        end;
UNTIL not odd( Status ) ;

(* If there was just no more data to be returned (eof) continue, otherwise
        exit with an error. *)
if Status <> SS$_ENDOFFILE then
   CHECKSTATUS( Status ) ;

(* The sort has been successful to this point. *)

(* Close the output file *)
CLOSE( sortout ) ;

(* clean up work areas and files *)
```

```
Status := SOR$END_SORT ;

(* Check status for error. *)
CHECKSTATUS( Status );

WRITELN ('SORT SUCCESSFUL') ;

END.
```

Example 20.5 is a VSI C program that demonstrates a sort operation using the STABLE option and two test keys.

Example 20.5. Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a VSI C Program

```
/*

C Program Example

This program demonstrates the use of the STABLE option
with 2 ascending text keys to sort a file of names.
The names are sorted by the first 6 characters of the last
name and the first 6 characters of the first name.
The contents of the input file and resulting output file
are listed below.  The associated C program code listing follows.

.............................................................

Input file: example.in

JONES  DAVID
WARNER LIZZY
SMITTS JAMES
SMITH  RANDY
BROWN  TONY
GRANT  JOSEPH
BROWN  JAMES
JONES  DAVID
BAKER  PAMELA
SMART  SHERYL
RUSSO  JOSEPH
JONES  DONALD
BROWN  GORDON

.............................................................

Output file: example.out

BAKER  PAMELA
BROWN  GORDON
BROWN  JAMES
BROWN  TONY
GRANT  JOSEPH
JONES  DAVID
JONES  DAVID
JONES  DONALD
RUSSO  JOSEPH
SMART  SHERYL
```

```
SMITH  RANDY
SMITTS JAMES
WARNER LIZZY

.....................................................................
*/
/*
**=====================================================================
**
**  EXAMPLE.C code:
**
**  Abstract:     Example of using sort with the STABLE option and
**                2 text keys (both ascending).
**
**
**  Input file:   example.in
**  Output file:  example.out
**
**=====================================================================
*/
/* ---------------------------------------------------------------------
** Include files:
*/
# include <stdlib.h>
# include <stdio.h>
# include <string.h>
# include <descrip.h>
# include <ssdef.h>
# include <sor$routines.h>

/* ---------------------------------------------------------------------
** Local macro definitions:
*/
# define MAX_REC_LEN   150
# define MAX_NUM_KEYS   10

/* ---------------------------------------------------------------------
** Local structure definitions.
*/

/* Define the description for each key. */
typedef struct {
    unsigned short    type;   /* Data type of key */
    unsigned short    order;  /* Order of key */
    unsigned short    offset; /* Offset of key */
    unsigned short    len;    /* Length of key */
  } key_info;


struct {
    unsigned short    num;    /* number of keys */
    key_info          key[MAX_NUM_KEYS];
} key_buffer;

/* ---------------------------------------------------------------------
** External literals.
*/
globalvalue
```

```
    int
        SOR$M_STABLE;

/* ------------------------------------------------------------------------
** Main entry point.
*/
main (int argc, char *argv[])
{
  int               i;
  unsigned int      options;               /* Sort options */
  unsigned int      num_records_in;
  unsigned int      num_records_out;
  unsigned int      lrl;                   /* longest record length */
  unsigned short    size;                  /* record size from return_rec
 */
  unsigned int      status;
  unsigned long int return_status;
  FILE              *infile;               /* input file */
  FILE              *outfile;              /* output file */
  char              record [MAX_REC_LEN];
  $DESCRIPTOR       (record_desc, record);

    lrl = sizeof(record);
    key_buffer.num              = 2;
    key_buffer.key[0].type      = DSC$K_DTYPE_T;
    key_buffer.key[0].order     = 0;            /* ascending */
    key_buffer.key[0].offset    = 0;
    key_buffer.key[0].len       = 6;

    key_buffer.key[1].type      = DSC$K_DTYPE_T;
    key_buffer.key[1].order     = 0;            /* ascending */
    key_buffer.key[1].offset    = 7;
    key_buffer.key[1].len       = 6;

    /* Open input and output files. */

    if (argc != 3)
    {
        printf("Usage: example inputfile outputfile\n");
        exit(-1);
    }

    infile = fopen(argv[1], "r");
    if (infile == (FILE *) NULL)
    {
        printf("Can't open input file %s\n",argv[1]);
        exit(-1);
    }

    outfile = fopen(argv[2], "w");
    if (outfile == (FILE *) NULL)
    {
        printf("Can't create output file %s\n",argv[2]);
        exit(-1);
    }

    /* Specify options.  Initialize the sort and check for errors. */
```

```
   options = SOR$M_STABLE;
   return_status = SOR$BEGIN_SORT(&key_buffer, &lrl, &options,
0,0,0,0,0,0);
   if (return_status != SS$_NORMAL)
   {
       printf ("Status from SOR$BEGIN_SORT:    0x%x\n", return_status);
       exit(return_status);
   }


   /* Within a loop, get all the records from the input file. */
   /* Exit if an error occurs. */

   num_records_in = 0;
   while (fgets( record, lrl, infile) != NULL)
   {
       record_desc.dsc$w_length = strlen(record)-1;
       num_records_in++;
       return_status = SOR$RELEASE_REC(&record_desc,0);
       if (return_status != SS$_NORMAL)
       {
           printf ("Status from SOR$RELEASE_REC:    0x%x\n",
return_status);
           exit(return_status);
       }
   }

   /* Sort all of the input records. */
   /* Exit if an error occurs. */

   return_status = SOR$SORT_MERGE(0);
   if (return_status != SS$_NORMAL)
   {
       printf ("Status from SOR$SORT_MERGE:    0x%x\n", return_status);
       exit(return_status);
   }

   /* Within a loop, write the sorted records to the output file. */
   /* Exit if an error occurs, other than end-of-file. */

   record_desc.dsc$w_length = lrl;
   num_records_out = 0;
   do
   {
       return_status = SOR$RETURN_REC(&record_desc,&size,0);
       if (return_status == SS$_NORMAL)
       {
           num_records_out++;
           status = fprintf (outfile,"%.*s\n", size, record);
           if (status < 0 )
           {
               printf ("Error writing to output file, status = %d\n",
status);
               exit(status);
           }
       }
       else
           if (return_status != SS$_ENDOFFILE)
```

```
            {
                printf ("Status from SOR$RETURN_REC:   0x%x\n",
 return_status);
                exit(return_status);
            };

    } while (return_status != SS$_ENDOFFILE);

    /* Sanity check - assure number of input and output records match. */

    if (num_records_out != num_records_in)
    {
        printf("Number of records out is not correct. # in = %d, # out = %d
\n",
                num_records_out, num_records_in);
        exit(status);
    }


    /* Successful completion.  Close input and output files. End program.
 */

    return_status = SOR$END_SORT(0);
    if (return_status != SS$_NORMAL)
    {
        printf ("Status from SOR$END_SORT:    0x%x\n", return_status);
        exit(return_status);
    }

    fclose (infile);
    fclose (outfile);
    }
```

20.4. SOR Routines

This section describes the individual SOR routines.

SOR$BEGIN_MERGE

Initialize a Merge Operation — The SOR$BEGIN_MERGE routine initializes the merge operation by opening the input and output files and by providing the number of input files, the key specifications, and the merge options.

Format

```
SOR$BEGIN_MERGE
[key-buffer] [,lrl] [,options] [,merge_order] [,user_compare] [,user_equal]
 [,user_input] [,context]
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)

access:	write only
mechanism:	by value

Longword condition value. Most Sort/Merge utility routines return a condition value in R0. Condition values that this routine can return are listed under Condition Values Returned.

Arguments

key_buffer

OpenVMS usage:	vector_word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Array of words describing the keys on which you plan to merge. The *key_buffer* argument is the address of an array containing the key descriptions.

The first word of this array contains the number of keys described (up to 255). Following the first word, each key is described (in order of priority) in blocks of four words. The four words specify the key's data type, order, offset, and length, respectively.

The first word of the block specifies the key's data type. The following data types are accepted:

DSC$K_DTYPE_Z	Unspecified (uninfluenced by collating sequence)
DSC$K_DTYPE_B	Byte integer (signed)
DSC$K_DTYPE_BU	Byte (unsigned)
DSC$K_DTYPE_W	Word integer (signed)
DSC$K_DTYPE_WU	Word (unsigned)
DSC$K_DTYPE_L	Longword integer (signed)
DSC$K_DTYPE_LU	Longword (unsigned)
DSC$K_DTYPE_Q	Quadword integer (signed)
DSC$K_DTYPE_QU	Quadword (unsigned)
DSC$K_DTYPE_O [dag]	Octaword integer (signed)
DSC$K_DTYPE_OU [dag]	Octaword (unsigned)
DSC$K_DTYPE_F	Single-precision floating
DSC$K_DTYPE_D	Double-precision floating
DSC$K_DTYPE_G	G-format floating
DSC$K_DTYPE_H [dag]	H-format floating
DSC$K_DTYPE_FS [DDAG]	IEEE single-precision S floating
DSC$K_DTYPE_FT [DDAG]	IEEE double-precision T floating
DSC$K_DTYPE_T	Text (may be influenced by collating sequence)
DSC$K_DTYPE_NU	Numeric string, unsigned
DSC$K_DTYPE_NL	Numeric string, left separate sign
DSC$K_DTYPE_NLO	Numeric string, left overpunched sign

DSC$K_DTYPE_NR	Numeric string, right separate sign
DSC$K_DTYPE_NRO	Numeric string, right overpunched sign
DSC$K_DTYPE_NZ [dag]	Numeric string, zoned sign
DSC$K_DTYPE_P	Packed decimal string

[dag]Data type is not currently supported by the high-performance Sort/Merge utility.
[DDAG]Data type is Alpha specific.

The *VSI OpenVMS Programming Concepts Manual* manual describes each of these data types.

The second word of the block specifies the key order: *0* for ascending order, *1* for descending order. The third word of the block specifies the relative offset of the key in the record. (Note that the first byte in the record is at position *0*.) The fourth word of the block specifies the key length in bytes (in digits for packed decimal—DSC$K_DTYPE_P).

If you do not specify the `key_buffer` argument, you must pass either a key comparison routine or use a specification file to define the key.

lrl

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Length of the longest record that will be released for merging. The `lrl` (longest record length) argument is the address of a word containing the length. If the input file is on a disk, this argument is not required. It is required when you use the record interface. For Vertical Format Control (VFC) records, this length must include the length of the fixed-length portion of the record.

options

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags that identify merge options. The `options` argument is the address of a longword bit mask whose settings determine the merge options selected.

The following table lists and describes the bit mask values available:

Flag	Description
SOR$M_STABLE	Keeps records with equal keys in the same order as they appeared on input.
SOR$M_EBCDIC	Orders ASCII character keys according to EBCDIC collating sequence. No translation takes place.
SOR$M_MULTI	Orders character keys according to the multinational collating sequence, which collates the international character set.
SOR$M_NOSIGNAL	Returns a status code instead of signaling errors.

Flag	Description
SOR$M_NODUPS	Omits records with duplicate keys. You cannot use this option if you specify your own equal-key routine.
SOR$M_SEQ_CHECK	Requests an "out of order" error return if an input file is not already in sequence. By default, this check is not done. You must request sequence checking if you specify an *equal-key* routine.

All other bits in the longword are reserved and must be zero.

merge_order

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Number of input streams to be merged. The *merge_order* argument is the address of a byte containing the number of files (1 through 10) to be merged. (The high-performance Sort/Merge utility allows you to specify 1 through 12 files.) When you use the record interface on input, this argument is required.

user_compare

OpenVMS usage:	procedure
type:	procedure value
access:	function call
mechanism:	by reference

Routine that compares records to determine their merge order. (This routine is not currently supported by the high-performance Sort/Merge utility.) The *user_compare* argument is the address of the procedure value for this user-written routine. If you do not specify the *key_buffer* argument or if you define key information in a specification file, this argument is required.

MERGE calls the comparison routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records to be compared, the lengths of these two records, and the context longword.

The comparison routine must return a 32-bit integer value:

- –1 if the first record collates before the second
- 0 if the records collate as equal
- 1 if the first record collates after the second

user_equal

OpenVMS usage:	procedure
type:	procedure value
access:	function call

mechanism:	by reference

Routine that resolves the merge order when records have duplicate keys. (This routine is not currently supported by the high-performance Sort/Merge utility.) The *user_equal* argument is the address of the procedure value for this user-written routine. If you specify SOR$M_STABLE or SOR $M_NODUPS in the *options* argument, do not use this argument.

MERGE calls the duplicate key routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records that compare equally, the lengths of the two records that compare equally, and the context longword.

The routine must return one of the following 32-bit condition codes:

Code	Description
SOR$_DELETE1	Delete the first record from the merge.
SOR$_DELETE2	Delete the second record from the merge.
SOR$_DELBOTH	Delete both records from the merge.
SS$_NORMAL	Keep both records in the merge.

Any other failure value causes the error to be signaled or returned. Any other success value causes an undefined result.

user_input

OpenVMS usage:	procedure
type:	procedure value
access:	function call
mechanism:	by reference

Routine that releases records to the merge operation. The *user_input* argument is the address of the procedure value for this user-written routine. SOR$BEGIN_MERGE and SOR$RETURN_REC call this routine until all records have been passed.

This input routine must read (or construct) a record, place it in a record buffer, store its length in an output argument, and then return control to MERGE.

The input routine must accept the following four arguments:

- A descriptor of the buffer where the routine must place the record
- A longword, passed by reference, containing the stream number from which to input a record (the first file is 1, the second 2, and so on)
- A word, passed by reference, where the routine must return the actual length of the record
- The context longword, passed by reference

The input routine must also return one of the following status values:

- SS$_NORMAL or any other success status causes the merge operation to continue.
- SS$_ENDOFFILE indicates that no more records are in the file. The contents of the buffer are ignored.

- Any other error status terminates the merge operation and passes the status value back to the caller of SOR$BEGIN_MERGE or SOR$RETURN_REC.

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value that was supplied by SORT/MERGE.

Description

The SOR$BEGIN_MERGE routine initializes the merge process by passing arguments that provide the number of input streams, the key specifications, and any merge options.

You must define the key by passing either the key buffer address argument or your own comparison routine address. (You can also define the key in a specification file and call the SOR$SPEC_FILE routine.)

The SOR$BEGIN_MERGE routine initializes the merge process in the file, record, and mixed interfaces. For record interface on input, you must also pass the merge order, the input routine address, and the longest record length. For files not on disk, you must pass the longest record length.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, you should use LIB$MATCH_COND if you want to check for a specific status.

Condition Values Returned

SS$_NORMAL

Success.

SOR$_BADDTYPE

Invalid or unsupported CDD data type.

SOR$_BADLENOFF

Length and offset must be multiples of 8 bits.

SOR$_BADLOGIC

Internal logic error detected.

SOR$_BADOCCURS

Invalid OCCURS clause.

SOR$_BADOVRLAY

Invalid overlay structure.

SOR$_BADPROTCL

Node is an invalid CDD object.

SOR$_BAD_KEY

Invalid key specification.

SOR$_BAD_LRL

Record length `n` greater than specified longest record length.

SOR$_BAD_MERGE

Number of input files must be between 0 and 10. (For the high-performance Sort/Merge utility, the maximum number is 12.)

SOR$_BAD_ORDER

Merge input is out of order.

SOR$_BAD_SRL

Record length `n` is too short to contain keys.

SOR$_BAD_TYPE

Invalid sort process specified.

SOR$_CDDERROR

CDD error at node *name*.

SOR$_CLOSEIN

Error closing *file* as input.

SOR$_CLOSEOUT

Error closing *file*.

SOR$_COL_CHAR

Invalid character definition.

SOR$_COL_CMPLX

Collating sequence is too complex.

SOR$_COL_PAD

Invalid pad character.

SOR$_COL_THREE

Cannot define 3-byte collating values.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_ILLBASE

Nondecimal base is invalid.

SOR$_ILLLITERL

Record containing symbolic literals is unsupported.

SOR$_ILLSCALE

Nonzero scale invalid for floating-point data item.

SOR$_INCDIGITS

Number of digits is not consistent with the type or length of item.

SOR$_INCNODATA

Include specification references no data, at line `n`.

SOR$_INCNOKEY

Include specification references no keys, at line `n`.

SOR$_IND_OVR

Indexed output file must already exist.

SOR$_KEYAMBINC

Key specification is ambiguous or inconsistent.

SOR$_KEYED

Mismatch between SORT/MERGE keys and primary file key.

SOR$_KEY_LEN

Invalid key length, key number `n`, length `n`.

SOR$_LRL_MISS

Longest record length must be specified.

SOR$_MISLENOFF

Length and offset required.

SOR$_MISS_PARAM

A required subroutine argument is missing.

SOR$_MULTIDIM

Invalid multidimensional OCCURS.

SOR$_NODUPEXC

Equal-key routine and no-duplicates option cannot both be specified.

SOR$_NOTRECORD

Node *name* is a name, not a record definition.

SOR$_NUM_KEY

Too many keys specified.

SOR$_NYI

Not yet implemented.

SOR$_OPENIN

Error opening *file* as input.

SOR$_OPENOUT

Error opening *file* as output.

SOR$_READERR

Error reading *file*.

SOR$_RTNERROR

Unexpected error status from user-written routine.

SOR$_SIGNCOMPQ

Absolute Date and Time data type represented in 1-second units.

SOR$_SORT_ON

Sort or merge routines called in incorrect order.

SOR$_SPCIVC

Invalid collating sequence specification at line `n`.

SOR$_SPCIVD

Invalid data type at line `n`.

SOR$_SPCIVF

Invalid field specification at line `n`.

SOR$_SPCIVI

Invalid include or omit specification at line `n`.

SOR$_SPCIVK

Invalid key or data specification at line `n`.

SOR$_SPCIVP

Invalid sort process at line `n`.

SOR$_SPCIVS

Invalid specification at line `n`.

SOR$_SPCIVX

Invalid condition specification at line `n`.

SOR$_SPCMIS

Invalid merge specification at line `n`.

SOR$_SPCOVR

Overridden specification at line `n`.

SOR$_SPCSIS

Invalid sort specification at line `n`.

SOR$_SRTIWA

Insufficient space. The specification file is too complex.

SOR$_STABLEEX

Equal-key routine and stable option cannot both be specified.

SOR$_SYSERROR

System service error.

SOR$_UNDOPTION

Undefined option flag was set.

SOR$_UNSUPLEVL

Unsupported core level for record *name*.

SOR$_WRITEERR

Error writing *file*.

SOR$BEGIN_SORT

Begin a Sort Operation — The SOR$BEGIN_SORT routine initializes a sort operation by opening input and output files and by passing the key information and any sort options.

Format

```
SOR$BEGIN_SORT [key_buffer] [,lrl] [,options] [,file_alloc] [,user_compare]
 [,user_equal] [,sort_process] [,work_files] [,context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

key_buffer

OpenVMS usage:	vector_word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Array of words describing the keys on which you plan to sort. The *key_buffer* argument is the address of an array containing the key descriptions.

The first word of this array contains the number of keys described (up to 255). Following the first word, each key is described (in order of priority) in blocks of four words. The four words specify the key's data type, order, offset, and length, respectively.

The first word of the block specifies the data type of the key. The following data types are accepted:

DSC$K_DTYPE_Z	Unspecified (uninfluenced by collating sequence)
DSC$K_DTYPE_B	Byte integer (signed)
DSC$K_DTYPE_BU	Byte (unsigned)
DSC$K_DTYPE_W	Word integer (signed)
DSC$K_DTYPE_WU	Word (unsigned)
DSC$K_DTYPE_L	Longword integer (signed)
DSC$K_DTYPE_LU	Longword (unsigned)
DSC$K_DTYPE_Q	Quadword integer (signed)
DSC$K_DTYPE_QU	Quadword (unsigned)
DSC$K_DTYPE_O [dag]	Octaword integer (signed)
DSC$K_DTYPE_OU [dag]	Octaword (unsigned)
DSC$K_DTYPE_F	Single-precision floating
DSC$K_DTYPE_D	Double-precision floating

DSC$K_DTYPE_G	G-format floating
DSC$K_DTYPE_H [dag]	H-format floating
DSC$K_DTYPE_FS [DDAG]	IEEE single-precision S floating
DSC$K_DTYPE_FT [DDAG]	IEEE double-precision T floating
DSC$K_DTYPE_T	Text (may be influenced by collating sequence)
DSC$K_DTYPE_NU	Numeric string, unsigned
DSC$K_DTYPE_NL	Numeric string, left separate sign
DSC$K_DTYPE_NLO	Numeric string, left overpunched sign
DSC$K_DTYPE_NR	Numeric string, right separate sign
DSC$K_DTYPE_NRO	Numeric string, right overpunched sign
DSC$K_DTYPE_NZ [dag]	Numeric string, zoned sign
DSC$K_DTYPE_P	Packed decimal string

[dag]Data type is not currently supported by the high-performance Sort/Merge utility.
[DDAG]Data type is Alpha specific.

The *VSI OpenVMS Programming Concepts Manual* describes each of these data types.

The second word of the block specifies the key order: *0* for ascending order, *1* for descending order. The third word of the block specifies the relative offset of the key in the record. Note that the first byte in the record is at position *0*. The fourth word of the block specifies the key length in bytes (in digits for packed decimal—DSC$K_DTYPE_P).

The `key_buffer` argument specifies the address of the key buffer in the data area. If you do not specify this argument, you must either pass a key comparison routine or use a specification file to define the key.

lrl

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Length of the longest record that will be released for sorting. The `lrl` argument is the address of a word containing the length. This argument is not required if the input files are on disk but is required when you use the record interface. For VFC records, this length must include the length of the fixed-length portion of the record.

options

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Flags that identify sort options. The `options` argument is the address of a longword bit mask whose settings determine the merge options selected. The following table lists and describes the bit mask values available.

Flags	Description
SOR$M_STABLE	Keeps records with equal keys in the same order in which they appeared on input. With multiple input files that have records that collate as equal, records from the first input file are placed before the records from the second input file, and so on.
SOR$M_EBCDIC	Orders ASCII character keys according to EBCDIC collating sequence. No translation takes place.
SOR$M_MULTI	Orders character keys according to the multinational collating sequence, which collates the international character set.
SOR$M_NOSIGNAL	Returns a status code instead of signaling errors.
SOR$M_NODUPS	Omits records with duplicate keys. You cannot use this option if you specify your own equal-key routine.

All other bits in the longword are reserved and must be zero.

file_alloc

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Input file size in blocks. The *file_alloc* argument is the address of a longword containing the size of the input file. This argument is optional because, by default, SORT uses the allocation of the input files. If you are using the record interface, or if the input files are not on disk, the default is 1000 blocks. (The high-performance Sort/Merge utility determines the default based on the size of the input file, or if input is not from files, on available memory.) When you specify the input size with this argument, it overrides the default size.

This optional argument is useful when you are using the record interface and you have a good idea of the total input size. You can use this argument to improve the efficiency of the sort by adjusting the amount of resources the sort process allocates to match the input size.

user_compare

OpenVMS usage:	procedure
type:	procedure value
access:	function call
mechanism:	by reference

User-written routine that compares records to determine their sort order. (This argument is not currently supported by the high-performance Sort/Merge utility.) The *user_compare* argument is the address of the procedure value for this user-written routine. If you do not specify the *key_buffer* argument or if you define key information in a specification file, this argument is required.

SORT/MERGE calls the comparison routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records to be compared, the lengths of these

two records, and the context longword. The LENG1 and LENG2 arguments are addresses that point to 16-bit word structures that contain the length information.

The comparison routine must return a 32-bit integer value:

- –1 if the first record collates before the second
- 0 if the records collate as equal
- 1 if the first record collates after the second

user_equal

OpenVMS usage:	procedure
type:	procedure value
access:	function call
mechanism:	by reference

User-written routine that resolves the sort order when records have duplicate keys. (This argument is not currently supported by the high-performance Sort/Merge utility.) The *user_equal* argument is the address of the procedure value for this user-written routine. If you specify SOR$M_STABLE or SOR $M_NODUPS in the *options* argument, do not use this argument.

SORT/MERGE calls the duplicate key routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records that compare equally, the lengths of the two records that compare equally, and the context longword. The LENG1 and LENG2 arguments are addresses that point to 16-bit word structures that contain the length information.

The routine must return one of the following 32-bit integer condition codes:

Code	Description
SOR$_DELETE1	Delete the first record from the sort.
SOR$_DELETE2	Delete the second record from the sort.
SOR$_DELBOTH	Delete both records from the sort.
SS$_NORMAL	Keep both records in the sort.

Any other failure value causes the error to be signaled or returned. Any other success value causes an undefined result.

sort_process

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Code indicating the type of sort process. The *sort_process* argument is the address of a byte whose value indicates whether the sort type is record, tag, index, or address. (The high-performance Sort/Merge utility supports only the record process. Implementation of the tag, address, and index processes is deferred to a future OpenVMS Alpha release.) The default is record. If you select the record interface on input, you can use only a record sort process.

To specify a byte containing the value for the type of sort process you want, enter one of the following:

- SOR$GK_RECORD (record sort)
- SOR$GK_TAG (tag sort)
- SOR$GK_ADDRESS (address sort)
- SOR$GK_INDEX (index sort)

work_files

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Number of work files to be used in the sorting process. The *work_files* argument is the address of a byte containing the number of work files; permissible values for SORT range from 0 through 10. (For the high-performance Sort/Merge utility, you can specify from 1 through 255 work files. The default is 2.)

By default, SORT creates two temporary work files when it needs them and determines their size from the size of your input files. By increasing the number of work files, you can reduce their individual size so that each fits into less disk space. You can also assign each of them to different disk-structured devices (highly recommended).

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

The SOR$BEGIN_SORT routine initializes the sort process by setting up sort work areas and provides key specification and sort options.

Specify the key information with the *key_buffer* argument, with the *user_compare* argument, or in a specification file. If no key information is specified, the default (character for the entire record) is used.

You must use the SOR$BEGIN_SORT routine to initialize the sort process for the file, record, and mixed interfaces. For record interface on input, you must use the *lrl* (longest record length) argument.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_BADLOGIC

Internal logic error detected.

SOR$_BAD_KEY

Invalid key specification.

SOR$_BAD_LRL

Record length `n` greater than specified longest record length.

SOR$_BAD_MERGE

Number of work files must be between 0 and 10. (For the high-performance Sort/Merge utility, the maximum number is 255.)

SOR$_BAD_TYPE

Invalid sort process specified.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_INSVIRMEM

Insufficient virtual memory.

SOR$_KEYAMBINC

Key specification is ambiguous or inconsistent.

SOR$_KEY_LEN

Invalid key length, key number `n`, length `n`.

SOR$_LRL_MISS

Longest record length must be specified.

SOR$_NODUPEXC

Equal-key routine and no-duplicates option cannot both be specified.

SOR$_NUM_KEY

Too many keys specified.

SOR$_NYI

Not yet implemented.

SOR$_RTNERROR

Unexpected error status from user-written routine.

SOR$_SORT_ON

Sort or merge routine called in incorrect order.

SOR$_STABLEEXC

Equal-key routine and stable option cannot both be specified.

SOR$_SYSERROR

System service error.

SOR$_UNDOPTION

Undefined option flag was set.

SOR$DTYPE

Define Data Type — The SOR$DTYPE routine defines a key data type that is not normally supported by SORT/MERGE. (This routine is not currently supported by the high-performance Sort/Merge utility.) This routine returns a key data type code that can be used in the *key_buffer* argument to SOR $BEGIN_SORT or SOR$BEGIN_MERGE to describe special key data types (such as extended data types and National character set (NCS) collating sequences).

Format

```
SOR$DTYPE [context] ,dtype_code ,usage ,p1
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

context

OpenVMS usage:	context
type:	longword (unsigned)

access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the context longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

dtype_code

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	write only
mechanism:	by reference

Returned key data type code. The *dtype_code* argument is the address of a word into which SORT/MERGE writes the key data type code that can be used in the *key_buffer* argument to SOR $BEGIN_SORT or SOR$BEGIN_MERGE.

usage

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of a longword containing a code that indicates the interpretation of the *p1* argument. The following table lists and describes the valid usage codes:

Flag	Description
SOR$K_ROUTINE	The *p1* argument should be interpreted as the address of the procedure value of a routine that SORT/MERGE will call to compare keys described by the *dtype_code* returned by the call to SOR$DTYPE.
SOR$K_NCS_TABLE	The *p1* argument should be interpreted as the address of a collating sequence identification returned by a call to NCS$GET_CS. SORT/MERGE will use this collating sequence to compare keys described by the *dtype_code* returned by the call to SOR$DTYPE.

If SOR$K_ROUTINE is returned, SORT/MERGE will call this routine with five reference arguments —ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two keys to be compared, the lengths of the two keys, and the context longword.

The comparison routine must return a 32-bit integer value:

- –1 if the first key collates before the second

- 0 if the keys collate as equal
- +1 if the first key collates after the second

p1

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the procedure value of a routine or the address of a collating sequence identification, depending on the *usage* argument.

Description

Call SOR$DTYPE to define a key data type not normally supported by SORT/MERGE.

If your SORT/MERGE application needs to compare dates (for example) that are stored in text form *and* that is the only key in the records, then use the *user_compare* argument to SOR$BEGIN_SORT or SOR$BEGIN_MERGE. However, if the records contain several keys besides the dates in text form, it may be easier to call SOR$DTYPE to allocate a key data type code that can then be used in the *key_buffer* argument to SOR$BEGIN_SORT or SOR$BEGIN_MERGE.

If your SORT/MERGE application has a string key that should be collated by a collating sequence defined by the NCS utility, the NCS$GET_CS routine can be used to fetch the collating sequence definition, and SOR$DTYPE can be called to allocate a key data type code for the collating sequence. This key data type code can then be used to describe keys that should be compared by this collating sequence.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_NYI

Not yet implemented.

SOR$_SORT_ON

Sort or merge routine called in incorrect order.

SOR$END_SORT

End a Sort Operation — The SOR$END_SORT routine performs cleanup functions, such as closing files and releasing memory.

Format

```
SOR$END_SORT [context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

context

OpenVMS usage:	context
type:	longword
access:	write only
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

The SOR$END_SORT routine ends a sort or merge operation, either at the end of a successful process or between calls because of an error. If an error status is returned, you must call SOR$END_SORT to release all allocated resources. In addition, this routine can be called at any time to close files and release memory.

The value of the optional context argument is cleared when the SOR$END_SORT routine completes its operation.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_CLOSEIN

Error closing *file* as input.

SOR$_CLOSEOUT

Error closing *file* as output.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_END_SORT

SORT/MERGE terminated, context = *context*.

SOR$_SYSERROR

System service error.

SOR$PASS_FILES

Pass File Name — The SOR$PASS_FILES routine passes the names of input and output files and output file characteristics to SORT or MERGE.

Format

```
SOR$PASS_FILES [inp_desc] [,out_desc] [,org] [,rfm] [,bks] [,bls] [,mrs]
 [,alq] [,fop] [,fsz] [,context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

inp_desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Input file specification. The *inp_desc* argument is the address of a descriptor pointing to the file specification. In the file interface, you must call SOR$PASS_FILES to pass SORT the input file specifications. For multiple input files, call SOR$PASS_FILES once for each input file, passing one input file specification descriptor each time.

In the mixed interface, if you are using the record interface on input, pass only the output file specification; do not pass any input file specifications. If you are using the record interface on output, pass only the input file specifications; do not pass an output file specification or any of the optional output file arguments.

out_desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Output file specification. The *out_desc* argument is the address of a descriptor pointing to the file specification. In the file interface, when you call SOR$PASS_FILES, you must pass the output file specification. Specify the output file specification and characteristics only once, as part of the first call, as in the following:

```
Call SOR$PASS_FILES(Input1,Output)
Call SOR$PASS_FILES(Input2)
Call SOR$PASS_FILES(Input3)
```

In the mixed interface, if you are using the record interface on input, pass only the output file specification; do not pass any input file specifications. If you are using the record interface on output, pass only the input file specifications; do not pass an output file specification or any of the optional output file arguments.

org

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

File organization of the output file, if different from the input file. The *org* argument is the address of a byte whose value specifies the organization of the output file; permissible values include the following:

```
FAB$C_SEQ
FAB$C_REL
FAB$C_IDX
```

For the record interface on input, the default value is sequential. For the file interface, the default value is the file organization of the first input file for record or tag sort and sequential for address and index sort.

For more information about OpenVMS RMS file organizations, see the *VSI OpenVMS Record Management Services Reference Manual*.

rfm

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Record format of the output file, if different from the input file. The *rfm* argument is the address of a byte whose value specifies the record format of the output file; permissible values include the following:

```
FAB$C_FIX
```

FAB$C_VAR
FAB$C_VFC

For the record interface on input, the default value is variable. For the file interface, the default value is the record format of the first input file for record or tag sort and fixed format for address or index sort. For the mixed interface with record interface on input, the default value is variable format.

For more information about OpenVMS RMS record formats, see the *VSI OpenVMS Record Management Services Reference Manual*.

bks

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Bucket size of the output file, if different from the first input file. The *bks* argument is the address of a byte containing this size. Use this argument with relative and indexed-sequential files only. If the bucket size of the output file is to differ from that of the first input file, specify a byte to indicate the bucket size. Acceptable values are from 1 to 32. If you do not pass this argument—and the output file organization is the same as that of the first input file—the bucket size defaults to the value of the first input file. If the file organizations differ or if the record interface is used on input, the default value is 1 block.

bls

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Block size of a magnetic tape output file. The *bls* argument is the address of a word containing this size. Use this argument with magnetic tapes only. Permissible values range from 20 to 65,532. However, to ensure compatibility with non-VSI systems, ANSI standards require that the block size be less than or equal to 2048.

The block size defaults to the block size of the input file magnetic tape. If the input file is not on magnetic tape, the output file block size defaults to the size used when the magnetic tape was mounted.

mrs

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Maximum record size for the output file. The *mrs* argument is the address of a word specifying this size. Following are acceptable values for each type of file:

File Organization	Acceptable Value
Sequential	0 to 32,767

File Organization	Acceptable Value
Relative	0 to 16,383
Indexed sequential	0 to 16,362

If you omit this argument or if you specify a value of 0, SORT does not check maximum record size.

If you do not specify this argument, the default is based on the output file organization and format, unless the organization is relative or the format is fixed. The longest output record length is based on the longest calculated input record length, the type of sort, and the record format.

alq

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Number of preallocated output file blocks. The *alq* argument is the address of a longword specifying the number of blocks you want to preallocate to the output file. Acceptable values range from 1 to 4,294,967,295.

Pass this argument if you know your output file allocation will be larger or smaller than that of your input files. The default value is the total allocation of all the input files. If the allocation cannot be obtained for any of the input files or if the record interface is used on input, the file allocation defaults to 1000 blocks.

fop

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

File-handling options. The *fop* argument is the address of a longword whose bit settings determine the options selected. For a list of valid file-handling options, see the description of the FAB$L_FOP field in the *VSI OpenVMS Record Management Services Reference Manual*. By default, only the DFW (deferred write) option is set. If your output file is indexed, you should set the CIF (create if) option.

fsz

OpenVMS usage:	byte_unsigned
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Size of the fixed portion of VFC records. The *fsz* argument is the address of a byte containing this size. If you do not pass this argument, the default is the size of the fixed portion of the first input file. If you specify the VFC size as 0, RMS defaults the value to 2 bytes.

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Value that distinguishes between multiple concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

The SOR$PASS_FILES routine passes input and output file specifications to SORT. The SOR $PASS_FILES routine must be repeated for multiple input files. The output file name string and characteristics should be specified only in the first call to SOR$PASS_FILES.

This routine also accepts optional arguments that specify characteristics for the output file. By default, the output file characteristics are the same as the first input file; specified output file characteristics are used to change these defaults.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_DUP_OUTPUT

Output file has already been specified.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_INP_FILES

Too many input files specified.

SOR$_NYI

Not yet implemented.

SOR$_SORT_ON

Sort or merge routine called in incorrect order.

SOR$_SYSERROR

System service error.

SOR$RELEASE_REC

Pass One Record to Sort — The SOR$RELEASE_REC routine is used with the record interface to pass one input record to SORT or MERGE.

Format

```
SOR$RELEASE_REC desc [,context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Input record buffer. The *desc* argument is the address of a descriptor pointing to the buffer containing the record to be sorted. If you use the record interface, this argument is required.

context

OpenVMS usage:	context
type:	longword
access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

Call SOR$RELEASE_REC to pass records to SORT or MERGE with the record interface. SOR $RELEASE_REC must be called once for each record to be sorted.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_BADLOGIC

Internal logic error detected.

SOR$_BAD_LRL

Record length `n` greater than longest specified record length.

SOR$_BAD_SRL

Record length `n` too short to contain keys.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_EXTEND

Unable to extend work file for needed space.

SOR$_MISS_PARAM

The *desc* argument is missing.

SOR$_NO_WRK

Work files required; cannot do sort in memory as requested.

SOR$_OPENOUT

Error opening *file* as output.

SOR$_OPERFAIL

Error requesting operator service.

SOR$_READERR

Error reading *file*.

SOR$_REQ_ALT

Specify alternate *name* file (or nothing to try again).

SOR$_RTNERROR

Unexpected error status from user-written routine.

SOR$_SORT_ON

Sort or merge routines called in incorrect order.

SOR$_SYSERROR

System service error.

SOR$_USE_ALT

Using alternate file *name*.

SOR$_WORK_DEV

Work file *name* must be on random access local device.

SOR$RETURN_REC

Return One Sorted Record — The SOR$RETURN_REC routine is used with the record interface to return one sorted or merged record to a program.

Format

```
SOR$RETURN_REC desc [,length] [,context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

desc

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Output record buffer. The *desc* argument is the address of a descriptor pointing to the buffer that receives the sorted or merged record.

length

OpenVMS usage:	word_unsigned
type:	word (unsigned)

access:	write only
mechanism:	by reference

Length of the output record. The *length* argument is the address of a word receiving the length of the record returned from SORT/MERGE.

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

Call the SOR$RETURN_REC routine to release the sorted or merged records to a program. Call this routine once for each record to be returned.

SOR$RETURN_REC places the record into a record buffer that you set up in the program's data area. After SORT has successfully returned all the records to the program, it returns the status code SS $_ENDOFFILE, which indicates that there are no more records to return.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_BADLOGIC

Internal logic error detected.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_EXTEND

Unable to extend work file for needed space.

SOR$_MISS_PARAM

A required subroutine argument is missing.

SOR$_OPERFAIL

Error requesting operator service.

SOR$_READERR

Error reading *file*.

SOR$_REQ_ALT

Specify alternate *name* file (or specify nothing to simply try again).

SOR$_RTNERROR

Unexpected error status from user-written routine.

SOR$_SORT_ON

Sort or merge routines called in incorrect order.

SOR$_SYSERROR

System service error.

SOR$_USE_ALT

Using alternate file *name*.

SOR$_WORK_DEV

Work file *name* must be on random access local device.

SOR$SORT_MERGE

Sort — The SOR$SORT_MERGE routine sorts the input records.

Format

```
SOR$SORT_MERGE [context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Argument

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

After you have passed either the file names or the records to SORT, call the SOR$SORT_MERGE routine to sort the records. For file interface on input, the input files are opened and the records are released to the sort. For the record interface on input, the record must have already been released (by calls to SOR$RELEASE_REC). For file interface on output, the output records are reformatted and directed to the output file. For the record interface on output, SOR$RETURN_REC must be called to get the sorted records.

Some of the return values are used with different severities depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

Condition Values Returned

SS$_NORMAL

Normal successful completion.

SOR$_BADDTYPE

Invalid or unsupported CDD data type.

SOR$_BADLENOFF

Length and offset must be multiples of 8 bits.

SOR$_BADLOGIC

Internal logic error detected.

SOR$_BADOCCURS

Invalid OCCURS clause.

SOR$_BADOVRLAY

Invalid overlay structure.

SOR$_BADPROTCL

Node is an invalid CDD object.

SOR$_BAD_LRL

Record length `n` greater than longest specified record length.

SOR$_BAD_TYPE

Invalid sort process specified.

SOR$_CDDERROR

CDD error at node *name*.

SOR$_CLOSEIN

Error closing *file* as input.

SOR$_CLOSEOUT

Error closing *file* as output.

SOR$_COL_CHAR

Invalid character definition.

SOR$_COL_CMPLX

Collating sequence is too complex.

SOR$_COL_PAD

Invalid pad character.

SOR$_COL_THREE

Cannot define 3-byte collating values.

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_EXTEND

Unable to extend work file for needed space.

SOR$_ILLBASE

Nondecimal base is invalid.

SOR$_ILLLITERL

Record containing symbolic literals is unsupported.

SOR$_ILLSCALE

Nonzero scale invalid for floating-point data item.

SOR$_INCDIGITS

Number of digits is inconsistent with the type or length of item.

SOR$_INCNODATA

Include specification references no *data* keyword, at line `n`.

SOR$_INCNOKEY

Include specification references no *keys* keyword, at line `n`.

SOR$_IND_OVR

Indexed output file must already exist.

SOR$_KEYED

Mismatch between SORT/MERGE keys and primary file key.

SOR$_LRL_MISS

Longest record length must be specified.

SOR$_MISLENOFF

Length and offset required.

SOR$_MULTIDIM

Invalid multidimensional OCCURS.

SOR$_NOTRECORD

Node *name* is a name, not a record definition.

SOR$_NO_WRK

Work files required, cannot do sort in memory as requested.

SOR$_OPENIN

Error opening *file* as input.

SOR$_OPENOUT

Error opening *file* as output.

SOR$_OPERFAIL

Error requesting operator service.

SOR$_READERR

Error reading *file*.

SOR$_REQ_ALT

Specify alternate *name* file (or nothing to try again).

SOR$_RTNERROR

Unexpected error status from user-written routine.

SOR$_SIGNCOMPQ

Absolute Date and Time data type represented in 1-second units.

SOR$_SORT_ON

Sort or merge routines called in incorrect order.

SOR$_SPCIVC

Invalid collating sequence specification, at line `n`.

SOR$_SPCIVD

Invalid data type, at line `n`.

SOR$_SPCIVF

Invalid field specification, at line `n`.

SOR$_SPCIVI

Invalid include or omit specification, at line `n`.

SOR$_SPCIVK

Invalid key or data specification, at line `n`.

SOR$_SPCIVP

Invalid sort process, at line `n`.

SOR$_SPCIVS

Invalid specification, at line `n`.

SOR$_SPCIVX

Invalid condition specification, at line `n`.

SOR$_SPCMIS

Invalid merge specification, at line `n`.

SOR$_SPCOVR

Overridden specification, at line `n`.

SOR$_SPCSIS

Invalid sort specification, at line `n`.

SOR$_SRTIWA

Insufficient space. Specification file is too complex.

SOR$_SYSERROR

System service error.

SOR$_UNSUPLEVL

Unsupported core level for record *name*.

SOR$_USE_ALT

Using alternate file *name*.

SOR$_WORK_DEV

Work file *name* must be on random access local device.

SOR$_WRITEERR

Error writing *file*.

SOR$SPEC_FILE

Pass a Specification File Name — The SOR$SPEC_FILE routine is used to pass a specification file or specification text to a sort or merge operation. (This routine is not currently supported by the high-performance Sort/Merge utility.)

Format

```
SOR$SPEC_FILE [spec_file] [,spec_buffer] [,context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

spec_file

OpenVMS usage:	char_string
type:	character-coded text string
access:	read-only
mechanism:	by descriptor

Specification file name. The *spec_file* argument is the address of a descriptor pointing to the name of a file that contains the text of the options requested for the sort or merge. The specification file name string and the specification file buffer arguments are mutually exclusive.

spec_buffer

OpenVMS usage:	char_string

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

Specification text buffer. The *spec_buffer* argument is the address of a descriptor pointing to a buffer containing specification text. This text has the same format as the text within the specification file. The specification file name string and the specification file buffer arguments are mutually exclusive.

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

Call SOR$SPEC_FILE to pass a specification file name or a buffer with specification text to a sort or merge operation. Through the use of a specification file, you can selectively omit or include particular records from the sort or merge operation and specify the reformatting of the output records. (See the Sort Utility in the *VSI OpenVMS User's Manual* for a complete description of specification files.)

If you call the SOR$SPEC_FILE routine, you must do so before you call any other routines. You must pass either the *spec_file* or *spec_buffer* argument, but not both.

Some of the return condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_NYI

Not yet implemented.

SOR$_SORT_ON

Sort or merge routine called in incorrect order.

SOR$_SYSERROR

System service error.

SOR$STAT

Obtain a Statistic — The SOR$STAT routine returns one statistic about the sort or merge operation to the user program.

Format

```
SOR$STAT code ,result [,context]
```

Returns

OpenVMS 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 under Condition Values Returned.

Arguments

code

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

SORT/MERGE statistic code. The *code* argument is the address of a longword containing the code that identifies the statistic you want returned in the *result* argument. The following table describes the values that are accepted.

Note: The high-performance Sort/Merge utility currently supports only the following subset of these values: SORK_REC_INP, SORK_REC_SOR, SORK_REC_OUT, SORK_LRL_INP.

Code	Description
SOR$K_IDENT	Address of ASCII string for version number
SOR$K_REC_INP	Number of records input
SOR$K_REC_SOR	Records sorted
SOR$K_REC_OUT	Records output
SOR$K_LRL_INP	Longest record length (LRL) for input
SOR$K_LRL_INT	Internal LRL
SOR$K_LRL_OUT	LRL for output
SOR$K_NODES	Nodes in sort tree
SOR$K_INI_RUNS	Initial dispersion runs
SOR$K_MRG_ORDER	Maximum merge order

Code	Description
SOR$K_MRG_PASSES	Number of merge passes
SOR$K_WRK_ALQ	Work file allocation
SOR$K_MBC_INP	Multiblock count for input
SOR$K_MBC_OUT	Multiblock count for output
SOR$K_MBF_INP	Multibuffer count for input
SOR$K_MBF_OUT	Multibuffer count for output

Note that performance statistics (such as direct I/O, buffered I/O, and elapsed and CPU times) are not available because user-written routines may affect those values. However, they are available if you call LIB$GETJPI.

result

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

SORT/MERGE statistic value. The *result* argument is the address of a longword into which SORT/MERGE writes the value of the statistic identified by the *code* argument.

context

OpenVMS usage:	context
type:	longword (unsigned)
access:	modify
mechanism:	by reference

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The *context* argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the *context* longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

Description

The SOR$STAT routine returns one statistic about the sort or merge operation to your program. You can call the SOR$STAT routine at any time while the sort or merge is active.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB $MATCH_COND.

Condition Values Returned

SOR$_ENDDIAGS

Completed with diagnostics.

SOR$_MISS_PARAM

A required subroutine argument is missing.

SOR$_NYI

Functionality is not yet implemented.

SOR$_SYSERROR

System service error.

Chapter 21. Traceback Facility (TBK) Routines

The Traceback facility for VSI OpenVMS Integrity server and Alpha systems is a debugging tool that provides information (symbolizations) about call stack PCs. In normal operation, when a process suffers a fatal unhandled exception, the operating system launches Traceback which sends to SYS$OUTPUT the complete call stack at the time of the exception. Applications can also directly use the Traceback facility to sequentially generate information for an individual call stack PC. In this case, the Traceback simply returns information to the caller, not to SYS$OUTPUT. This chapter describes this direct Traceback interface.

21.1. Introduction to TBK Routines

On Integrity server systems, the Traceback facility can be invoked at any time by using the TBK $I64_SYMBOLIZE routine. This routine uses a single data structure for its inputs and outputs. It can be called from User, Supervisor, or Executive mode.

Similarly, on Alpha systems, the Traceback facility can be invoked at any time using the TBK $ALPHA_SYMBOLIZE routine. This routine uses a single data structure for its inputs and outputs and it can be called from USER, SUPERVISOR, or EXCECUTIVE mode.

Section 21.2 provides sample programs showing how to use the TBK routines. Section 21.3 is a reference section that provides details about the TBK routines.

21.2. Using TBK Routines—Example

This section provides an example program containing three small subroutines to illustrate using the TBK $I64_SYMBOLIZE routine. The example program runs a test to exercise the Integrity servers `librtl` call stack walking routines, the TRACE API, `sys$unwind`, and `sys$unwind_goto_64`. It is presented in three parts with callout information that describes the processing:

- Part 1 of the example defines the necessary call stack walking headers, TRACE API headers, local subroutines, and a subroutine exception handler (see Section 21.2.1).
- Part 2 issues `librtl` call stack walking calls for each of three subroutines, defines a pointer to a call stack walk invocation context block, defines storage for the return TRACE symbolizations and information, and defines storage and initializes the TRACE API parameter block (see Section 21.2.2).

  Part 3 allocates and initializes the invocation context block and obtains the context handler's current context. Subroutine `subc` signals into a frame-based handler, `subc_handler` which walks the stack, calls TBK$I64_SYMBOLIZE to symbolize each frame's PC, and then prints out the symbolizations (see Section 21.2.3).

21.2.1. TBK$I64_SYMBOLIZE Example—Part 1

The first part of the example defines the necessary call stack walking headers, TRACE API headers, local subroutines, and a subroutine exception handler.

Example 21.1. TBK$I64_SYMBOLIZE Example—Part 1

```
$ run/nodebug unwind4
```

```
In subc_handler, ch_cnt = 1
Call stack:
image      module          routine                line       PC
UNWIND4    UNWIND4         subc_handler           27271
 0000000000030650
DECC$SHR   C$SHELL_HANDLER decc$$shell_handler   5566
 FFFFF80208613E50
DECC$SHR   C$SHELL_HANDLER decc$$shell_handler   0
 FFFFFFFF803EC680
DECC$SHR   C$SHELL_HANDLER decc$$shell_handler   0
 FFFFFFFF803E00B0
UNWIND4    UNWIND4         subc                   27409
 00000000000310A0
UNWIND4    UNWIND4         subb                   27200
 0000000000030300
UNWIND4    UNWIND4         suba                   27187
 0000000000030200
UNWIND4    UNWIND4         main                   27175
 0000000000030140
UNWIND4    UNWIND4         __main                 27171
 00000000000300E0
UNWIND4    UNWIND4         __main                 0
 FFFFFFFF80B72C80

Continue (versus exit)? [Y/N]:
```

```
 /*
  * NOTE: to compile include "/define=(__NEW_STARLET)".
  */


#include <stdio.h> (1)
#include <stdarg.h>
#include <starlet.h>
#include <stddef.h>
#include <ssdef.h>
#include <descrip.h>


/* librtl headers for call stack walking
 */

#include <lib$routines.h> (2)
#include <libicb.h>

/* trace headers for trace api
 */

#include <tbkdef.h> (3)
#include <tbk$routines.h>

/* some local subroutines
 */

void suba (void); (4)
void subb (void);
void subc (void);
```

```
/* a subroutine exception handler
 */

int subc_handler (unsigned long int *sigarg, unsigned long int *mecharg);
 (5)
unsigned long int a_cnt, b_cnt, c_cnt, ch_cnt;
unsigned __int64 a_invo_handle, b_invo_handle, c_invo_handle;
int status;

int main ()
    {
    suba ();

    return 1;
    }

void suba ()
    {
```

1. This program runs a test to exercise the Integrity server `librtl` call stack walking routines, the TRACE API, `sys$unwind`, and `sys$unwind_goto_64`.
2. The necessary `librtl` call stack walking headers. LIBICB defines the invocation context block. LIB$ROUTINES defines the call stack walk function prototypes.
3. The necessary TRACE API header files. TBKDEF defines the TRACE_API call parameter. TBK $ROUTINES defines the TRACE API function prototype.
4. This code defines the local subroutines `suba`, `subb`, and `subc`.
5. This code defines a subroutine exception handler.

21.2.2. TBK$I64_SYMBOLIZE Example—Part 2

The second part of the example issues `librtl` call stack walking calls for each of three subroutines, defines a pointer to a call stack walk invocation context block, defines storage for the return TRACE symbolizations and information, and defines storage and initializes the TRACE API parameter block.

Example 21.2. TBK$I64_SYMBOLIZE Example—Part 2

```
    /* Get routine a's invocation context handle, used in subc_handler
     */
    status = lib$i64_get_curr_invo_handle (&a_invo_handle); ❶

    a_cnt++;
    subb ();

    a_cnt++;
    subb ();
    }

    void subb ()
    {

    /* Get routine b's invocation context handle, used in subc_handler
     */
    status = lib$i64_get_curr_invo_handle (&b_invo_handle); ❷
```

```
    b_cnt++;
    subc ();

    b_cnt++;
    subc ();

    b_cnt++;
    subc ();
    }

void subc ()
    {
    lib$establish (subc_handler);

    /* Get routine c's invocation context handle, used in subc_handler
     */
    status = lib$i64_get_curr_invo_handle (&c_invo_handle); ❸

    /* Signal into subc_handler
     */
    c_cnt++;
    lib$signal (c_cnt);

    c_cnt++;
    lib$signal (c_cnt);
    }

int subc_handler (unsigned long int *sigarg, unsigned long int *mecharg)
    {
    int status, tbk_status=0, callstack_depth = 0;
    unsigned int depth;
    /* local pointer for the call stack walk invocation context block
     */
    INVO_CONTEXT_BLK *myICB; ❹

    /* local storage for image, module, routine names, line number, and
 image
     * and module base addresses returned by the trace api
     */
    static char image [128], module [128], routine [128], inquire_continue
 [128];
    static struct dsc$descriptor_vs image_dsc = {125, DSC$K_DTYPE_VT, DSC
$K_CLASS_VS, &image[0]};
    static struct dsc$descriptor_vs module_dsc = {125, DSC$K_DTYPE_VT, DSC
$K_CLASS_VS, &module[0]};
    static struct dsc$descriptor_vs routine_dsc = {125, DSC$K_DTYPE_VT, DSC
$K_CLASS_VS, &routine[0]};
    unsigned int     list_line;
    unsigned __int64 image_base_addr;          ❺
    unsigned __int64 module_base_addr;

    /* Local storage and setup for the trace api parameter block
     */
    unsigned __int64 symbolize_flags={0};   ❻
    TBK_API_PARAM params = {
                TBK$K_LENGTH,   /* trace api parameter block length */
                0,              /* trace api parameter block type, MBZ */
```

```
                TBK$K_VERSION,  /* trace api parameter block length, MBZ */
                0,      /* reserved, MBZ */
                0,      /* pc, input */
                0,      /* fp, input, not used for Integrity servers */
                0,      /* filename desc, output, not used here */
                0,      /* library module desc, output, not used here */
                0,      /* record number, output, not used here  */
                (struct _descriptor *)&image_dsc,       /* image
descriptor, output */
                (struct _descriptor *)&module_dsc,      /* module
descriptor, output */
                (struct _descriptor *)&routine_dsc,     /*
routine_descriptor, output */
                &list_line,     /* compiler listing line number, output */
                0,      /* relative pc, output, not used here */
                &image_base_addr,       /* image base address, output */
                &module_base_addr,      /* module base address, output */
                0,      /* malloc routine, input */
                0,      /* free routine, input */
                &symbolize_flags,       /* symbolize flags, input */
                0,      /* reserved */
                0,      /* reserved */
                0};     /* reserved */

    if (*(sigarg+1) == SS$_UNWIND)
        return SS$_CONTINUE;
    else
        ch_cnt++;

    printf ("\nIn subc_handler, ch_cnt = %d\n", ch_cnt);
    printf ("Call stack: \n");

    status = 1;
```

❶ A `librtl` call stack walk call to get the `suba` subroutine's invocation context handle used in `subc_handler`.
❷ A `librtl` call stack walk call to get the `subb` subroutine's invocation context handle used in `subc_handler`.
❸ A `librtl` call stack walk call to get the `subc` subroutine's invocation context handle used in `subc_handler`.
❹ A pointer is defined to a call stack walk invocation context block.
❺ Storage is defined for the return TRACE symbolizations and information, which includes local storage for image, module, routine names, line number, and image and module base addresses returned by the TRACE API.
❻ Local storage is defined for the TRACE API parameter block, which is initialized.

21.2.3. TBK$I64_SYMBOLIZE Example—Part 3

The third part of the example allocates and initializes the invocation context block and obtains the context handler's current context. Subroutine `subc` signals into a frame-based handler (`subc_handler`), which walks the stack, calls TBK$I64_SYMBOLIZE to symbolize each frame's PC, and prints out the symbolizations.

Example 21.3. TBK$I64_SYMBOLIZE Example—Part 3

```
    /* Walk the call stack top to bottom, symbolize each frame's PC, and
     * print out the symbolizations.
```

```
     *
     * First, create the invocation context block and get my
 (subc_handler's)
     * current context.
     */
    myICB = (INVO_CONTEXT_BLK *) lib$i64_create_invo_context ();    ❶
    lib$i64_get_curr_invo_context (myICB);

    printf ("image       module     routine          line           PC\n");

    while (!(myICB->libicb$v_bottom_of_stack) &&    ❷
           ((status & 1) != 0))
        {
        /* Use the PC from the call stack invocation context block.
         */
        params.tbk$q_faulting_pc = (unsigned __int64) myICB->libicb
$ih_pc; ❸

        /* Call trace to do the symbolizations.
         */
        tbk_status = tbk$i64_symbolize (&params); ❹

        /* And print out results
         */
        image [*((short *) image) + 2] = 0;
        module [*((short *) module) + 2] = 0;
        routine [*((short *) routine) + 2] = 0;
        /* Print out the tbk$i64_symbolize info (with formating
         * to align columns).
         */
        if (*((short *) module) > 8)
            {
            if (*((short *) routine) > 8)
                {
                printf ("%s   %s    %s    %ld       %16.16LX\n",
                        &image [2],
                        &module [2],
                        &routine [2],
                        list_line,
                        (unsigned __int64) myICB->libicb$ih_pc);
                }
            else
                {
                printf ("%s     %s      %s    %ld      %16.16LX\n",
                        &image [2],
                        &module [2],
                        &routine [2],
                        list_line,
                        (unsigned __int64) myICB->libicb$ih_pc);
                }
            }
        else
            {
            if (*((short *) routine) > 8)
                {
                printf ("%s       %s       %s      %ld      %16.16LX\n",
                        &image [2],
                        &module [2],
```

```
                        &routine [2],
                        list_line,
                        (unsigned __int64) myICB->libicb$ih_pc);
                }
            else
                {
                printf ("%s       %s         %s         %ld      %16.16LX
\n",
                        &image [2],
                        &module [2],
                        &routine [2],
                        list_line,
                        (unsigned __int64) myICB->libicb$ih_pc);
                }
            }
        /* Get the previous call frame.
         */
        status = lib$i64_get_prev_invo_context (myICB);  ❺
        callstack_depth++;
        }

    /* Terminate the call stack walk and free up the memory that it used.
     */
    lib$i64_prev_invo_end (myICB);  ❻
    lib$i64_free_invo_context (myICB);

    /* Set up to unwind if we continue execution.
     */
    switch (ch_cnt)
        {
        /* first, some sys$unwinds    ❼
         */
        case 1 :
            status = sys$unwind (0, 0);
            break;
        case 2 :
            depth = 0;
            status = sys$unwind (&depth, 0);
            break;
        case 3 :
            depth = 1;
            status = sys$unwind (&depth, 0);
            break;
        case 4 :
            depth = 2;
            status = sys$unwind (&depth, 0);
            break;

        /* now, some sys$goto_unwinds
         */
        case 5 :
            status = sys$goto_unwind_64 (&c_invo_handle, 0, 0, 0);
            break;
        case 6 :
            status = sys$goto_unwind_64 (&b_invo_handle, 0, 0, 0);
            break;
        case 7 :
            status = sys$goto_unwind_64 (&a_invo_handle, 0, 0, 0);
```

```
            break;

        default :
            break;
        }

    /* Continue (after unwinding) or exit?  Let the user decide.
     */
    printf ("\nContinue (versus exit)? [Y/N]: ");
    gets (inquire_continue);

    if ((inquire_continue [0] == 'Y') || (inquire_continue [0] == 'y'))
        return SS$_CONTINUE;
    else
        sys$exit (1);
    }
```

❶ To prepare for walking the call stack, the invocation context block is allocated and initialized, and the context handler's (`subc_handler`) current context is obtained.
❷ The call stack is walked from the current context, `subc_handler`, to the bottom of the stack.
❸ The essential call stack PC value is provided. On Integrity server systems, all the symbolization is based on a call stack frame's PC value.
❹ A call is made to the TRACE symbolize routine, TBK$I64_SYMBOLIZE, for the call frame's PC. This is the call to the TRACE API. The information is then printed out.
❺ The previous call frame context is obtained.
❻ Cleanup is performed and memory used by the call stack walk is deallocated.
❼ Unwind is set up to continue program execution.

21.3. TBK Routines

This section describes the TBK routines. The TBK$I64_SYMBOLIZE routine is for use on Integrity server systems and the TBK$ALPHA_SYMBOLIZE routine is for use on Alpha systems.

TBK$I64_SYMBOLIZE

TBK$I64_SYMBOLIZE — The TBK$I64_SYMBOLIZE routine attempts to symbolize a PC, returning as much symbolic representation for that location as was requested. For information about the TBK symbolize routine for Alpha systems, see the information for TBK$ALPHA_SYMBOLIZE later in TBK $ALPHA_SYMBOLIZE.

Format

TBK$I64_SYMBOLIZE parameter_block

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Argument

parmeter_block

OpenVMS usage:	**TBK_API_PARAM**
type:	**structure**
access:	**modify**
mechanism:	**by reference**

Table 21.1 shows the values for TBK_API_PARAM (defined in TBKDEF).

Table 21.1. Values for TBK_API_PARAM

Field	Size	Description
TBK$W_LENGTH	Word	Input by value, structure length, must be TBK$K_LENGTH
TBK$B_TYPE	Byte	Input, MBZ
TBK$B_VERSION	Byte	Input by value, must be TBK $K_VERSION
TBK$L_RESERVEDA	Longword	Reserved for future use, MBZ
TBK$Q_FAULTING_PC	Quadword	Input by value, call stack frame PC
TBK$PQ_FILENAME_DESC	64-bit pointer	Optional output by reference (Integrity servers only), pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with preallocated adequate buffer space. The descriptor is filled with the image file name. This can be a dynamic descriptor (rather than fixed-length), but only if the caller is in user mode.
TBK $PQ_LIBRARY_MODULE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with pre-allocated adequate buffer space. The descriptor is filled in with library module name *if* the image filename (see previous field) is a text library file. This can be a dynamic descriptor (rather than fixed length) but only if the caller is in user mode.
TBK$PQ_RECORD_NUMBER	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a longword to be filled with the relevant image file record number.
TBK$PQ_IMAGE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with preallocated adequate buffer space. The descriptor is filled in with

Field	Size	Description
		the image name. This can be a dynamic descriptor (rather than fixed length), but only if the caller is in user mode.
TBK$PQ_MODULE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with preallocated adequate buffer space. The descriptor is filled in with the module name.
TBK$PQ_ROUTINE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with preallocated adequate buffer space. The descriptor is filled in with the routine name.
TBK$PQ_LISTING_LINENO	64-bit pointer	Optional output, pointer (if not requested, MBZ) to longword to be filled in with the line number (as show in the modules LIS file).
TBK$PQ_REL_PC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to quadword to be filled in with the relative PC. This can be an image or module relative PC.
TBK$PQ_ MALLOC_RTN	64-bit pointer	Optional input, pointer (if not supplied, MBZ) address to a user-supplied malloc routine. Must be supplied when called from supervisor or executive mode (kernel mode is not supported).
TBK$PQ_ FREE_RTN	64-bit pointer	Optional input, pointer (if not supplied, MBZ) address to a user-supplied free routine. Must be supplied when called from supervisor or executive mode (kernel mode not supported).
TBK$PQ_SYMBOLIZE_FLAGS	64-bit pointer	Optional input and output, pointer (if not supplied, MBZ) to TBK_SYMBOLIZE_FLAGS (quadword, see below). Used to control symbolization options and to return additional status.
TBK$Q_RESERVED0	Quadword	Reserved for future use, MBZ.
TBK$Q_RESERVED1	Quadword	Reserved for future use, MBZ.
TBK$Q_RESERVED2	Quadword	Reserved for future use, MBZ.
TBK$V_EXCEPTION_IS_FAULT	0	Adjusts the PC value used for symbolization for target frames that suffered a fault exception.
	All remaining bits	Reserved, Must be initialized to zero.

Description

The TBK$I64_SYMBOLIZE routine attempts to symbolize a PC, that is, given a PC, this routine returns as much of the symbolic representation for that location that has been requested: image name, file name, module name, routine name, listing line number, file record number, and so on.

The degree of symbolization depends upon the images symbolic information. For best results, compile the images source modules with either traceback (the default) or debug information (/DEBUG) and link the image with either traceback (/TRACE) or debug (/DEBUG) information. If no symbolic information records exists within the image for the PC, then only partial symbolization is possible.

TBK$I64_SYMBOLIZE can be called by programs in user, supervisor, or executive mode. Calls from kernel mode are not allowed; calls when IPL is nonzero are not allowed.

Callers in supervisor or executive mode must supply routines that perform the equivalent of malloc and free operations that are legal for the given mode. (The C Run Time Library malloc and free routines are only supported in user mode.) Pointers to these user-written replacement routines are specified in the TBK$PQ_MALLOC_RTN and TBK$PQ_FREE_RTN fields.

Condition Values Returned

SS$_KERNELINV

This API does not support kernel mode calls.

SS$_BADPARAM

Incorrect TBK_API_PARAM length, type, or version.

SS$_INSFMEM

Unable to allocate needed memory.

SS$_NORMAL

Successful completion.

SS$_ACCVIO

Unable to read from the TBK_API_PARAM block.

Other conditions indicate TRACE failures such as failure status from `sys$crmpsc_file_64` on an Integrity servers system.

TBK$ALPHA_SYMBOLIZE

TBK$ALPHA_SYMBOLIZE — The TBK$ALPHA_SYMBOLIZE routine attempts to symbolize a call stack PC, returning as much symbolic representation for that location as was requested. For information about the TBK symbolize routine for Integrity server systems, see the information for TBK $I64_SYMBOLIZE earlier in TBK$I64_SYMBOLIZE.

Format

`TBK$ALPHA_SYMBOLIZE` parameter_block

Returns

OpenVMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. Most utility routines return a condition value. Condition values that this routine can return are listed under Condition Values Returned.

Argument

parmeter_block

OpenVMS usage:	**TBK_API_PARAM**
type:	**structure**
access:	**modify**
mechanism:	**by reference**

Table 21.2 shows the values for TBK_API_PARAM (defined in TBKDEF).

Table 21.2. Values for TBK_API_PARAM on Alpha

Field	Size	Description
TBK$W_LENGTH	Word	Input by value, structure length, must be TBK$K_LENGTH
TBK$B_TYPE	Byte	Input, MBZ
TBK$B_VERSION	Byte	Input by value, must be TBK $K_VERSION
TBK$L_RESERVEDA	Longword	Reserved for future use, MBZ
TBK$Q_FAULTING_PC	Quadword	Input by value, call stack frame PC
TBK$Q_FAULTING_FP	Quadword	Input by value, call stack Frame Pointer
TBK$PQ_IMAGE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with preallocated adequate buffer space. The descriptor is filled in with the image name. This can be a dynamic descriptor (rather than fixed length) but only if the caller is in user mode.
TBK$PQ_MODULE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be set up with preallocated adequate buffer space. The descriptor is filled in with the module name.
TBK$PQ_ROUTINE_DESC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to a fixed-length string text descriptor. The descriptor must be

Field	Size	Description
		set up with preallocated adequate buffer space. The descriptor is filled in with the routine name.
TBK$PQ_LISTING_LINENO	64-bit pointer	Optional output, pointer (if not requested, MBZ) to longword to be filled in with the line number (as show in the modules LIS file).
TBK$PQ_REL_PC	64-bit pointer	Optional output, pointer (if not requested, MBZ) to quadword to be filled in with the relative PC. This may be an image or module relative PC.
TBK$PQ_IMAGE_BASE_ADDR	64-bit pointer	Optional output, pointer (if not requested, MBZ) to quadword to be filled in with the image base address.
TBK$PQ_MODULE_BASE_ADDR	64-bit pointer	Optional output pointer (if not requested, MBZ) to quadword to be filled in with the module base address.
TBK$PQ_MALLOC_RTN	64-bit pointer	Optional input, pointer (if not supplied, MBZ) address to a user-supplied malloc routine. Must be supplied when called from supervisor or executive mode (kernel mode is not supported).
TBK$PQ_FREE_RTN	64-bit pointer	Optional input, pointer (if not supplied, MBZ) address to a user-supplied free routine. Must be supplied when called from supervisor or executive mode (kernel mode not supported).
TBK$PQ_SYMBOLIZE_FLAGS	64-bit pointer	Optional input and output, pointer (if not supplied, MBZ) to TBK_SYMBOLIZE_FLAGS (quadword, see below). Used to control symbolization options and to return additional status.
TBK$Q_RESERVED0	Quadword	Reserved for future use, MBZ.
TBK$Q_RESERVED1	Quadword	Reserved for future use, MBZ.
TBK$Q_RESERVED2	Quadword	Reserved for future use, MBZ.
TBK$V_EXCEPTION_IS_FAULT	0	Adjusts the PC value used for symbolization for target frames that suffered a fault exception.
	All remaining bits	Reserved. Must be initialized to zero.

Description

The TBK$ALPHA_SYMBOLIZE routine attempts to symbolize a PC. That is, given a PC and a frame pointer, this routine returns as much of the symbolic representation for that location that has been requested: image name, file name, module name, routine name, listing line number, file record number, and so on. This must be a PC in an active call stack frame.

The degree of symbolization depends on the images symbolic information. For best results, compile the image source modules with either traceback (the default) or debug information (/DEBUG), and link the image with either traceback (/TRACE) or debug (/DEBUG) information. If no symbolic information records exists within the image for the PC, then only partial symbolization is possible.

The TBK$ALPHA_SYMBOLIZE routine can be called by programs in user, supervisor, or executive mode. Calls from kernel mode are not allowed; calls when IPL is nonzero are not allowed.

Callers in supervisor or executive mode must supply routines that perform the equivalent of malloc and free operations that are legal for the given mode. (The C Run Time Library malloc and free routines are only supported in user mode.) Pointers to these user-written replacement routines are specified in the TBK$PQ_MALLOC_RTN and TBK$PQ_FREE_RTN fields.

Condition Values Returned

SS$_KERNELINV

This API does not support kernel mode calls.

SS$_BADPARAM

Incorrect TBK_API_PARAM length, type, or version.

SS$_INSFMEM

Unable to allocate needed memory.

SS$_NORMAL

Successful completion.

SS$_ACCVIO

Unable to read from the TBK_API_PARAM block.