This manual describes the OpenVMS utility routines, a set of routines that provides a programming interface to various OpenVMS utilities.

Revision/Update Information: This manual supersedes the OpenVMS Utility Routines Manual, OpenVMS Version 7.2.

Software Version: OpenVMS Alpha Version 7.3
OpenVMS VAX Version 7.3
## Contents

### Preface

**1 Introduction to Utility Routines**

**2 Access Control List (ACL) Editor Routine**

2.1 Introduction to the ACL Editor Routine .................................. ACL–1
2.2 Using the ACL Editor Routine: An Example ............................ ACL–1
2.3 ACL Editor Routine ................................................ ACL–2
ACLEDIT$EDIT ............................................ ACL–3

**3 Backup (BACKUP) Routine**

3.1 Introduction to the Backup API ..................................... BCK–1
3.2 Using the Backup API: An Example ................................. BCK–2
3.3 Backup API ................................................ BCK–3
BACKUP$START ............................................ BCK–4

**4 Command Language Interface (CLI) Routines**

4.1 Introduction to CLI Routines ...................................... CLI–1
4.2 Using the CLI Routines: An Example .............................. CLI–2
4.3 CLI Routines ............................................ CLI–4
CLI$DCL_PARSE ............................................ CLI–5
CLI$DISPATCH ............................................ CLI–8
CLI$GET_VALUE ............................................ CLI–9
CLI$PRESENT ............................................. CLI–12

**5 Common File Qualifier Routines**

5.1 Introduction to the Common File Qualifier Routines ............... CQUAL–1
5.2 Using the Common File Qualifier Routines ......................... CQUAL–1
5.2.1 Calling UTIL$CQUAL_FILE_PARSE .......................... CQUAL–2
5.2.1.1 Specifying Times ...................................... CQUAL–3
5.2.1.2 Specifying Exclude Pattern Strings ..................... CQUAL–3
5.2.2 Calling UTIL$CQUAL_FILE_MATCH ....................... CQUAL–3
5.2.2.1 Specifying Prompts .................................. CQUAL–4
5.2.2.2 Ignoring Qualifiers .................................. CQUAL–4
5.2.3 Calling UTIL$CQUAL_FILE_END ......................... CQUAL–5
5.2.4 Calling UTIL$CQUAL_CONFIRM_ACT ..................... CQUAL–5
5.2.5 Creating a Command Language Definition File ................ CQUAL–6
5.3 UTIL$CQUAL Routines .................................................. CQUAL–10
UTIL$CQUAL_FILE_PARSE ............................................. CQUAL–11
UTIL$CQUAL_FILE_MATCH ........................................... CQUAL–13
UTIL$CQUAL_FILE_END ............................................... CQUAL–17
UTIL$CQUAL_CONFIRM_ACT .......................................... CQUAL–18

6 Convert (CONVERT) Routines
6.1 Introduction to CONVERT Routines ................................. CONV–1
6.2 Using the CONVERT Routines: Examples ........................ CONV–1
6.3 CONVERT Routines .................................................. CONV–7
CONV$CONVERT ....................................................... CONV–8
CONV$PASS_FILES .................................................... CONV–12
CONV$PASS_OPTIONS ............................................... CONV–15
CONV$RECLAIM ....................................................... CONV–20

7 Data Compression/Expansion (DCX) Routines
7.1 Introduction to DCX Routines ....................................... DCX–1
7.1.1 Compression Routines ........................................... DCX–1
7.1.2 Expansion Routines ............................................ DCX–3
7.2 Using the DCX Routines: Examples ................................. DCX–3
7.3 DCX Routines ....................................................... DCX–11
DCX$ANALYZE_DATA ................................................ DCX–12
DCX$ANALYZE_DONE ............................................... DCX–14
DCX$ANALYZE_INIT ................................................ DCX–15
DCX$COMPRESS_DATA ............................................. DCX–18
DCX$COMPRESS_DONE ............................................. DCX–20
DCX$COMPRESS_INIT ............................................. DCX–21
DCX$EXPAND_DATA ................................................ DCX–23
DCX$EXPAND_DONE ................................................ DCX–25
DCX$EXPAND_INIT ................................................ DCX–26
DCX$MAKE_MAP ..................................................... DCX–28

8 DEC Text Processing Utility (DECTPU) Routines
8.1 Introduction to DECTPU Routines ................................. DECTPU–1
8.1.1 Interfaces to Callable DECTPU ................................ DECTPU–2
8.1.1.1 Simplified Callable Interface ................................. DECTPU–2
8.1.1.2 Full Callable Interface ....................................... DECTPU–2
8.1.2 The DECTPU Shareable Image ................................. DECTPU–3
8.1.3 Passing Parameters to Callable DECTPU Routines .......... DECTPU–3
8.1.4 Error Handling .................................................. DECTPU–3
8.1.5 Return Values ................................................... DECTPU–4
8.2 Simplified Callable Interface .................................... DECTPU–4
8.3 Full Callable Interface ............................................. DECTPU–5
8.3.1 Main Callable DECTPU Utility Routines ...................... DECTPU–6
8.3.2 Other DECTPU Utility Routines ............................... DECTPU–6
8.3.3 User-Written Routines ........................................ DECTPU–7
8.4 Using the DECTPU Routines: Examples ......................... DECTPU–7
8.5 Creating and Calling a USER Routine ........................... DECTPU–22
8.5.1 The CALL_USER Code ...................................... DECTPU–23
8.5.2 Linking the CALL_USER Image .............................. DECTPU–25
8.6 Accessing the USER Routine from DECTPU ...................... DECTPU–26
8.7 DECTPU Routines ................................................. DECTPU–27
TPU$CLEANUP .................................................... DECTPU–28
TPU$CLIPARSE ..................................................... DECTPU–32
TPU$CLOSE_TERMINAL ............................................ DECTPU–34
TPU$CONTROL ...................................................... DECTPU–35
TPU$EDIT .......................................................... DECTPU–37
TPU$EXECUTE_COMMAND ......................................... DECTPU–39
TPU$EXECUTE_INIFILE ........................................... DECTPU–40
TPU$FILEIO ....................................................... DECTPU–42
TPU$FILE_PARSE .................................................... DECTPU–46
TPU$FILE_SEARCH .................................................. DECTPU–49
TPU$HANDLER ....................................................... DECTPU–52
TPU$INITIALIZE ..................................................... DECTPU–54
TPU$MESSAGE ....................................................... DECTPU–61
TPU$PARSEINFO ..................................................... DECTPU–62
TPU$SIGNAL ........................................................ DECTPU–63
TPU$SPECIFY_ASYNC_ACTION .................................... DECTPU–64
TPU$TPU ............................................................. DECTPU–66
TPU$TRIGGER_ASYNC_ACTION .................................... DECTPU–67
FILEIO .............................................................. DECTPU–68
FILE_PARSE ........................................................... DECTPU–70
FILE_SEARCH ........................................................ DECTPU–72
HANDLER ............................................................. DECTPU–75
INITIALIZE ............................................................ DECTPU–76
USER ................................................................. DECTPU–77

9 EDT Routines
9.1 Introduction to EDT Routines ..................................... EDT–1
9.2 Using the EDT Routines: An Example ............................. EDT–1
9.3 EDT Routines ....................................................... EDT–2
EDT$EDIT ........................................................... EDT–3
FILEIO .............................................................. EDT–7
WORKIO ............................................................. EDT–11
XIMATE ............................................................... EDT–13

10 File Definition Language (FDL) Routines
10.1 Introduction to FDL Routines ..................................... FDL–1
10.2 Using the FDL Routines: Examples ............................... FDL–2
10.3 FDL Routines ....................................................... FDL–6
FDL$CREATE ....................................................... FDL–7
FDL$GENERATE ..................................................... FDL–13
FDL$PARSE ......................................................... FDL–16
FDL$RELEASE ...................................................... FDL–19
11 Librarian (LBR) Routines

11.1 Introduction to LBR Routines ........................................ LBR–1
   11.1.1 Types of Libraries ............................................ LBR–1
   11.1.2 Structure of Libraries ....................................... LBR–2
      11.1.2.1 Library Headers ...................................... LBR–2
      11.1.2.2 Modules .............................................. LBR–2
      11.1.2.3 Indexes and Keys .................................... LBR–2
   11.1.3 Summary of LBR Routines .................................... LBR–5

11.2 Using the LBR Routines: Examples .................................. LBR–6
   11.2.1 Creating, Opening, and Closing a Text Library .............. LBR–8
   11.2.2 Inserting a Module .......................................... LBR–10
   11.2.3 Extracting a Module ........................................ LBR–14
   11.2.4 Deleting a Module .......................................... LBR–16
   11.2.5 Using Multiple Keys and Multiple Indexes .................... LBR–19
   11.2.6 Accessing Module Headers .................................. LBR–22
   11.2.7 Reading Library Headers .................................... LBR–23
   11.2.8 Displaying Help Text ....................................... LBR–25
   11.2.9 Listing and Processing Index Entries ........................ LBR–26

11.3 LBR Routines ....................................................... LBR–27
   LBR$CLOSE .................................................. LBR–28
   LBR$DELETE_DATA ............................................. LBR–29
   LBR$DELETE_KEY ............................................... LBR–31
   LBR$FIND ................................................... LBR–33
   LBR$FLUSH .................................................. LBR–35
   LBR$GET_HEADER ............................................. LBR–37
   LBR$GET_HELP ............................................... LBR–39
   LBR$GET_HISTORY ............................................ LBR–42
   LBR$GET_INDEX ............................................... LBR–44
   LBR$GET_RECORD ............................................. LBR–46
   LBR$INI_CONTROL ............................................. LBR–48
   LBR$INSERT_KEY ............................................... LBR–50
   LBR$LOOKUP_KEY ............................................. LBR–52
   LBR$OPEN .................................................... LBR–54
   LBR$OUTPUT_HELP ............................................. LBR–58
   LBR$PUT_END .................................................. LBR–62
   LBR$PUT_HISTORY ............................................. LBR–63
   LBR$PUT_RECORD ............................................... LBR–65
   LBR$REPLACE_KEY ............................................. LBR–67
   LBR$RET_RMSSTV .............................................. LBR–69
   LBR$SEARCH .................................................. LBR–70
   LBR$SET_INDEX ............................................... LBR–72
   LBR$SET_LOCATE ............................................... LBR–74
   LBR$SET_MODULE .............................................. LBR–75
   LBR$SET_MOVE ................................................. LBR–77
12 Lightweight Directory Access Protocol (LDAP) Routines

12.1 Introduction .................................................. LDAP–1
12.1.1 Overview of the LDAP Model ................................ LDAP–1
12.1.2 Overview of LDAP API Use ................................ LDAP–2
12.1.3 LDAP API Use on OpenVMS Systems ....................... LDAP–2
12.1.4 64-bit Addressing Support ................................ LDAP–3
12.1.4.1 Background ........................................ LDAP–3
12.1.4.2 Implementation .................................... LDAP–4
12.1.4.2.1 Library Symbol Names ...................... LDAP–4
12.1.4.2.2 LDAP Data Structures ....................... LDAP–4
12.1.4.3 Mixing Pointer Sizes ................................ LDAP–6
12.1.5 Multithreading Support .................................... LDAP–6
12.2 Common Data Structures and Memory Handling ............... LDAP–7
12.3 LDAP Error Codes ........................................ LDAP–8
12.4 Initializing an LDAP Session ................................ LDAP–9
12.5 LDAP Session Handle Options ................................ LDAP–10
12.6 Working with Controls ...................................... LDAP–13
12.7 Authenticating to the Directory ............................. LDAP–14
12.8 Closing the Session ...................................... LDAP–15
12.9 Searching ................................................ LDAP–16
12.9.1 Reading and Listing the Children of an Entry ........... LDAP–18
12.10 Comparing a Value Against an Entry ....................... LDAP–19
12.11 Modifying an Entry ................................... LDAP–20
12.12 Modifying the Name of an Entry .......................... LDAP–22
12.13 Adding an Entry ..................................... LDAP–23
12.14 Deleting an Entry ...................................... LDAP–24
12.15 Extended Operations .................................. LDAP–25
12.16 Abandoning an Operation ................................ LDAP–27
12.17 Obtaining Results and Looking Inside LDAP Messages ... LDAP–27
12.18 Handling Errors and Parsing Results ....................... LDAP–29
12.18.1 Stepping Through a List of Results .................. LDAP–31
12.19 Parsing Search Results ................................ LDAP–31
12.19.1 Stepping Through a List of Entries .................. LDAP–31
12.19.2 Stepping Through the Attributes of an Entry .......... LDAP–32
12.19.3 Retrieving the Values of an Attribute ............... LDAP–33
12.19.4 Retrieving the Name of an Entry .................. LDAP–34
12.19.5 Retrieving Controls from an Entry ................. LDAP–34
12.19.6 Parsing References ............................... LDAP–35
12.20 Encoded ASN.1 Value Manipulation ......................... LDAP–35
12.20.1 Encoding ......................................... LDAP–37
12.20.1.1 Encoding Example ............................ LDAP–38
12.20.2 Decoding ......................................... LDAP–39
12.20.2.1 Decoding Example ............................ LDAP–41
12.21 Sample LDAP API Code ................................ LDAP–43

13 LOGINOUT (LGI) Routines

13.1 Introduction to LOGINOUT ................................ LGI–1
13.1.1 The LOGINOUT Process ................................ LGI–1
13.1.2 Using LOGINOUT with External Authentication .... LGI–2
13.1.3 The LOGINOUT Data Flow ............................ LGI–2
13.2 LOGINOUT Callouts ........................................ LGI–3
13.2.1 LOGINOUT Callout Routines ......................... LGI–3
13.2.2 LOGINOUT Callback Routines ......................... LGI–3
13.3 Using Callout Routines ........................................ LGI–4
13.3.1 Calling Environment ....................................... LGI–4
13.3.2 Callout Organization ....................................... LGI–5
13.3.3 Activating the Callout Routines ........................... LGI–6
13.3.4 Callout Interface ........................................ LGI–7
13.3.5 Sample Program ........................................ LGI–10
13.4 LOGI$OUTH Callout Routines ................................ LGI–14
LG$I$ICR_AUTHENTICATE ....................................... LGI–15
LG$I$ICR_CHKRESTRICT ......................................... LGI–18
LG$I$ICR_DECWINIT .............................................. LGI–20
LG$I$ICR_FINISH ................................................. LGI–22
LG$I$ICR_IACT_START ........................................... LGI–24
LG$I$ICR_IDENTIFY .............................................. LGI–26
LG$I$ICR_INIT ..................................................... LGI–28
LG$I$ICR_J OBSTEP ................................................ LGI–30
LG$I$ICR_LOGOUT ................................................ LGI–32
13.5 LOGI$OUTH Callback Routines .............................. LGI–34
LG$I$ICB_ACCTEXPIRED .......................................... LGI–35
LG$I$ICB_AUTOLOGIN ............................................ LGI–36
LG$I$ICB_CHECK_PASS ............................................ LGI–37
LG$I$ICB_DISUSER ................................................ LGI–38
LG$I$ICB_GET_INPUT .............................................. LGI–39
LG$I$ICB_GET_SYSPGW ........................................... LGI–40
LG$I$ICB_MODALHOURS .......................................... LGI–41
LG$I$ICB_PASSWORD .............................................. LGI–42
LG$I$ICB_PWDEXPIRED ........................................... LGI–44
LG$I$ICB_USERPARSE ............................................. LGI–45
LG$I$ICB_USERPROMPT ............................................ LGI–46
LG$I$ICBVALIDATE ............................................... LGI–47

14 Mail Utility (MAIL) Routines
14.1 Messages .................................................... MAIL–1
14.2 Folders ........................................................ MAIL–2
14.3 Mail Files .................................................... MAIL–3
14.4 User Profile Database ........................................ MAIL–3
14.5 Mail Utility Processing Contexts .............................. MAIL–3
14.5.1 Callable Mail Utility Routines .............................. MAIL–4
14.5.2 Single and Multiple Threads .............................. MAIL–5
14.6 Programming Considerations ................................ MAIL–5
14.6.1 Condition Handling ....................................... MAIL–6
14.6.2 Item Lists and Item Descriptors .......................... MAIL–6
14.6.2.1 Structure of an Item Descriptor ........................ MAIL–6
14.6.2.2 Null Item Lists ........................................... MAIL–7
14.6.2.3 Declaring Item Lists and Item Descriptors .......... MAIL–7
14.6.2.4 Terminating an Item List ............................... MAIL–7
14.6.3 Action Routines ........................................... MAIL–7
14.7 Managing Mail Files ......................................... MAIL–8
14.7.1 Opening and Closing Mail Files ........................... MAIL–9
14.7.1.1 Using the Default Specification for Mail Files .... MAIL–9
14.7.1.2 Specifying an Alternate Mail File Specification ... MAIL–10
14.7.2 Displaying Folder Names ........................................ MAIL–11
14.7.3 Purging Mail Files Using the Wastebasket Folder ............ MAIL–11
14.7.3.1 Reclaiming Disk Space ................................... MAIL–11
14.7.3.2 Compressing Mail Files .................................. MAIL–11
14.8 Message Context .................................................. MAIL–11
14.8.1 Selecting Messages ........................................ MAIL–12
14.8.2 Reading and Printing Messages ............................... MAIL–13
14.8.3 Modifying Messages .......................................... MAIL–13
14.8.4 Copying and Moving Messages ............................... MAIL–13
14.8.4.1 Creating Folders .......................................... MAIL–14
14.8.4.2 Deleting Folders .......................................... MAIL–14
14.8.4.3 Creating Mail Files ....................................... MAIL–14
14.8.5 Deleting Messages ........................................... MAIL–14
14.9 Send Context ..................................................... MAIL–14
14.9.1 Sending New Messages ........................................ MAIL–15
14.9.1.1 Creating a Message ........................................ MAIL–15
14.9.1.1.1 Constructing the Message Header .................... MAIL–15
14.9.1.1.2 Constructing the Body of the Message ............. MAIL–15
14.9.1.2 Creating an Address List ................................ MAIL–16
14.9.2 Sending Existing Messages .................................. MAIL–16
14.9.3 Send Action Routines ......................................... MAIL–16
14.9.3.1 Success Action Routines ................................. MAIL–16
14.9.3.2 Error Handling Routines ................................. MAIL–16
14.9.3.3 Aborting a Send Operation ............................... MAIL–17
14.10 User Profile Context ............................................. MAIL–17
14.10.1 User Profile Entries ......................................... MAIL–17
14.10.1.1 Adding Entries to the User Profile Database ............ MAIL–18
14.10.1.2 Modifying or Deleting User Profile Entries ........... MAIL–18
14.11 Input Item Codes ................................................ MAIL–18
14.12 Output Item Codes .............................................. MAIL–21
14.13 Using the MAIL Routines: Examples .......................... MAIL–23
14.14 MAIL Routines ................................................... MAIL–31
     MAIL$MAILFILE_BEGIN ........................................ MAIL–32
     MAIL$MAILFILE_CLOSE ....................................... MAIL–34
     MAIL$MAILFILE_COMPRESS ................................... MAIL–36
     MAIL$MAILFILE_END .......................................... MAIL–39
     MAIL$MAILFILE_INFO_FILE .................................. MAIL–41
     MAIL$MAILFILE_MODIFY ..................................... MAIL–44
     MAIL$MAILFILE_OPEN ......................................... MAIL–47
     MAIL$MAILFILE_PURGE_WASTE ................................ MAIL–50
     MAIL$MESSAGE_BEGIN ......................................... MAIL–53
     MAIL$MESSAGE_COPY .......................................... MAIL–55
     MAIL$MESSAGE_DELETE ....................................... MAIL–59
     MAIL$MESSAGE_END ............................................ MAIL–61
     MAIL$MESSAGE_GET ............................................ MAIL–63
     MAIL$MESSAGE_INFO ........................................... MAIL–68
     MAIL$MESSAGE_MODIFY ........................................ MAIL–72
     MAIL$MESSAGE_SELECT ........................................ MAIL–75
     MAIL$SEND_ABORT ............................................. MAIL–78
     MAIL$SEND_ADD_ADDRESS ...................................... MAIL–80
     MAIL$SEND_ADD_ATTRIBUTE ................................... MAIL–82
15 National Character Set (NCS) Utility Routines

15.1 Introduction to NCS Routines ....................................................... NCS–1
15.1.1 List of NCS Routines ................................................................. NCS–1
15.1.2 Sample Application Process ......................................................... NCS–2
15.2 Using the NCS Utility Routines: Examples ........................................ NCS–2
15.3 NCS Routines ................................................................................. NCS–7
   NCS$COMPARE ................................................................. NCS–8
   NCS$CONVERT ............................................................... NCS–10
   NCS$END_CF ................................................................. NCS–12
   NCS$END_CS ................................................................. NCS–13
   NCS$GET_CF ................................................................. NCS–14
   NCS$GET_CS ................................................................. NCS–16
   NCS$RESTORE_CF ........................................................... NCS–18
   NCS$RESTORE_CS ........................................................... NCS–20
   NCS$SAVE_CF ............................................................... NCS–22
   NCS$SAVE_CS ............................................................... NCS–24

16 Print Symbiont Modification (PSM) Routines

16.1 Introduction to PSM Routines .......................................................... PSM–1
16.2 Print Symbiont Overview ................................................................. PSM–2
16.2.1 Components of the Print Symbiont ............................................... PSM–2
16.2.2 Creation of the Print Symbiont Process ......................................... PSM–2
16.2.3 Symbiont Streams ............................................................... PSM–3
16.2.4 Symbiont and Job Controller Functions ..................................... PSM–3
16.2.5 Print Symbiont Internal Logic ....................................................... PSM–4
16.3 Symbiont Modification Procedure ................................................... PSM–6
16.3.1 Guidelines and Restrictions ......................................................... PSM–7
16.3.2 Writing an Input Routine ............................................................. PSM–9
16.3.2.1 Internal Logic of the Symbiont’s Main Input Routine .................. PSM–9
16.3.2.2 Symbiont Processing of Carriage Control .................................. PSM–10
16.3.3 Writing a Format Routine ............................................................. PSM–11
16.3.3.1 Internal Logic of the Symbiont’s Main Format Routine ............... PSM–12
16.3.4 Writing an Output Routine ............................................................. PSM–12
16.3.4.1 Internal Logic of the Symbiont’s Main Output Routine ............... PSM–13
16.3.5 Other Function Codes ................................................................. PSM–13
16.3.6 Writing a Symbiont Initialization Routine ..................................... PSM–14
16.3.7 Integrating a Modified Symbiont ................................................. PSM–15
16.4 Using the PSM Routines: An Example ............................................... PSM–16
17 Symbiont/Job Controller Interface (SMB) Routines

17.1 Introduction to SMB Routines ................................... SMB–1
  17.1.1 Types of Symbiont ......................................... SMB–1
  17.1.2 Symbionts Supplied with the Operating System ........ SMB–1
  17.1.3 Symbiont Behavior in the OpenVMS Environment ...... SMB–2
  17.1.4 Writing a Symbiont ......................................... SMB–3
  17.1.5 Guidelines for Writing a Symbiont ....................... SMB–3
  17.1.6 The Symbiont/Job Controller Interface Routines ...... SMB–4
  17.1.7 Choosing the Symbiont Environment ....................... SMB–5
    17.1.7.1 Synchronous Versus Asynchronous Delivery of Requests .... SMB–5
    17.1.7.2 Single-Streaming Versus Multistreaming .............. SMB–9
  17.1.8 Reading Job Controller Requests ......................... SMB–10
  17.1.9 Processing Job Controller Requests ...................... SMB–10
  17.1.10 Responding to Job Controller Requests ................. SMB–13
  17.2 SMB Routines ............................................. SMB–13
    SMB$CHECK_FOR_MESSAGE .................................. SMB–14
    SMB$INITIALIZE ............................................ SMB–15
    SMB$READ_MESSAGE ......................................... SMB–17
    SMB$READ_MESSAGE_ITEM .................................. SMB–20
    SMB$SEND_TO_JOBCTL ........................................ SMB–30

18 Sort/Merge (SOR) Routines

18.1 High-Performance Sort/Merge (Alpha Only) .................... SOR–1
  18.1.1 High-Performance SOR Routine Behavior .................. SOR–2
  18.1.2 Using Threads with High-Performance Sort/Merge ....... SOR–3
  18.2 Introduction to SOR Routines ................................ SOR–4
    18.2.1 Arguments to SOR Routines .............................. SOR–4
    18.2.2 Interfaces to SOR Routines ................................ SOR–5
      18.2.2.1 Sort Operation Using File Interface ................ SOR–5
      18.2.2.2 Sort Operation Using Record Interface ............. SOR–6
      18.2.2.3 Merge Operation Using File Interface ............... SOR–6
      18.2.2.4 Merge Operation Using Record Interface ........... SOR–6
    18.2.3 Reentrancy ............................................ SOR–6
  18.3 Using the SOR Routines: Examples ............................. SOR–8
  18.4 SOR Routines ................................................ SOR–28
    SOR$BEGIN_MERGE ........................................... SOR–29
    SOR$BEGIN_SORT ............................................ SOR–36
    SOR$DTYPE ................................................ SOR–42
    SOR$END_SORT ............................................... SOR–45
    SOR$PASS_FILES ............................................. SOR–47
Index

Examples

2–1 Calling the ACL Editor with a VAX BLISS Program .................. ACL–2
3–1 Calling the Backup API with a VAX C Program .................. BCK–2
4–1 Using the CLI Routines to Retrieve Information About Command Lines in a Fortran Program .................. CLI–2
5–1 Using UTIL$QUAL Routines to Process Files .................. CQUAL–7
6–1 Using the CONVERT Routines in a Fortran Program .................. CONV–1
6–2 Using the CONVERT Routines in a C Program .................. CONV–3
6–3 Using the CONV$RECLAIM Routine in a Fortran Program .................. CONV–5
6–4 Using the CONV$RECLAIM Routine in a C Program .................. CONV–6
7–1 Compressing a File in a Compaq Fortran Program .................. DCX–4
7–2 Expanding a Compressed File in a Compaq Fortran Program .................. DCX–8
8–1 Sample VAX BLISS Template for Callable DECTPU .................. DECTPU–1
8–2 Normal DECTPU Setup in Compaq Fortran .................. DECTPU–12
8–3 Building a Callback Item List with Compaq Fortran .................. DECTPU–14
8–4 Specifying a User-Written File I/O Routine in VAX C .................. DECTPU–17
9–1 Using the EDT Routines in a VAX BASIC Program .................. EDT–1
10–1 Using FDL$CREATE in a Fortran Program .................. FDL–3
10–2 Using FDL$PARSE and FDL$RELEASE in a C Program .................. FDL–3
10–3 Using FDL$PARSE and FDL$GENERATE in a Compaq Pascal Program .................. FDL–5
11–1 Creating a New Library Using Compaq Pascal .................. LBR–8
11–2 Inserting a Module into a Library Using Compaq Pascal .................. LBR–11
11–3 Extracting a Module from a Library Using Compaq Pascal .................. LBR–14
11–4 Deleting a Module from a Library Using Compaq Pascal .................. LBR–17
11–5 Associating Keys with Modules .................. LBR–19
11–6 Listing Keys Associated with a Module .................. LBR–21
11–7 Displaying the Module Header .................. LBR–22
11–8 Reading Library Headers .................. LBR–24
11–9 Displaying Text from a Help Library .................. LBR–25
11–10 Displaying Index Entries .................. LBR–27
14–1 Sending a File .................. MAIL–24
14–2 Displaying Folders .................. MAIL–26
14–3 Displaying User Profile Information .................. MAIL–28
15–1 Using NCS Routines in a Compaq Fortran for OpenVMS Program ........ NCS–3
15–2 Using NCS Routines in a Compaq C for OpenVMS VAX Program ........ NCS–5
16–1 Using PSM Routines to Supply a Page Header Routine in a VAX MACRO Program .................. PSM–16
18–1 Using SOR Routines to Perform a Merge Using Record Interface in a Compaq Fortran Program ......................................... SOR–9
18–2 Using SOR Routines to Sort Using Mixed Interface in a Compaq Fortran Program ..................................................... SOR–13
18–3 Using SOR Routines to Merge Three Input Files in a Compaq Pascal Program ............................................................. SOR–16
18–4 Using SOR Routines to Sort Records from Two Input Files in a Compaq Pascal Program .................................................. SOR–20
18–5 Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a Compaq C Program ......................... SOR–24

Figures
11–1 Structure of a Macro, Text, or Help Library ................................ LBR–3
11–2 Structure of an Object or Shareable Image Library ..................... LBR–4
11–3 Structure of a User-Developed Library .................................. LBR–5
13–1 LOGINOUT Callout Routines Data Flow .................................. LGI–3
13–2 Callout Organization ..................................................... LGI–6
14–1 Standard Message Format ................................................ MAIL–1
14–2 Item Descriptor ............................................................. MAIL–6
16–1 Multithreaded Symbiont .................................................... PSM–3
16–2 Symbiont Execution Sequence or Flow of Control ....................... PSM–5
17–1 Symbions in the OpenVMS Environment .................................. SMB–2
17–2 Flowchart for a Single-Threaded, Synchronous Symbiont ............. SMB–6
17–3 Flowchart for a Single-Threaded, Asynchronous Symbiont (MAIN Routine) ................................................................. SMB–8
17–4 Flowchart for a Single-Threaded, Asynchronous Symbiont (AST Routine) ................................................................. SMB–9

Tables
3–1 Backup API Language Definition Files ..................................... BCK–2
3–2 BACKUP Option Structure Types ......................................... BCK–4
3–3 bckEvent Format ............................................................. BCK–19
3–4 Event Callback Buffer Formats ........................................ BCK–19
3–5 Control Event Subtypes .................................................... BCK–19
3–6 bckControl Format ........................................................ BCK–23
5–1 UTIL$CQUAL Routines .................................................. CQUAL–1
5–2 UTIL$CQUAL_FILE_PARSE Command Line Qualifiers .............. CQUAL–2
5–3 UTIL$CQUAL_FILE_PARSE Flags and Masks ................................ CQUAL–11
5–4 Prompting Form Values .................................................. CQUAL–19
5–5 Prompt Responses ......................................................... CQUAL–20
6–1 Conversion Statistics Array ............................................. CONV–9
6–2 CONVERT Qualifiers ..................................................... CONV–16
6–3 Bucket Recamation Statistics Array .................................... CONV–21
8–1 Valid Masks for the TPU$K_OPTIONS Item Code .......................... DECTPU–58
11–1 LBR Routines .............................................................. LBR–5
13–1 LOGINOUT Callouts ...................................................... LGI–3
<table>
<thead>
<tr>
<th>Page</th>
<th>Section Title</th>
<th>Page Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>13–2</td>
<td>LOGINOUT Callback Routines</td>
<td>LGI–4</td>
</tr>
<tr>
<td>13–3</td>
<td>Useful LOGINOUT Internal Variables</td>
<td>LGI–8</td>
</tr>
<tr>
<td>14–1</td>
<td>Default Mail Folders</td>
<td>MAIL–2</td>
</tr>
<tr>
<td>14–2</td>
<td>User Profile Information</td>
<td>MAIL–3</td>
</tr>
<tr>
<td>14–3</td>
<td>Levels of Mail Utility Processing</td>
<td>MAIL–4</td>
</tr>
<tr>
<td>14–4</td>
<td>Callable Mail Utility Routines</td>
<td>MAIL–4</td>
</tr>
<tr>
<td>14–5</td>
<td>Types of Action Routines</td>
<td>MAIL–8</td>
</tr>
<tr>
<td>14–6</td>
<td>Mail File Routines</td>
<td>MAIL–8</td>
</tr>
<tr>
<td>14–7</td>
<td>Message Routines</td>
<td>MAIL–11</td>
</tr>
<tr>
<td>14–8</td>
<td>Send Routines</td>
<td>MAIL–15</td>
</tr>
<tr>
<td>14–9</td>
<td>User Profile Context Routines</td>
<td>MAIL–17</td>
</tr>
<tr>
<td>14–10</td>
<td>Input Item Codes</td>
<td>MAIL–19</td>
</tr>
<tr>
<td>14–11</td>
<td>Output Item Codes</td>
<td>MAIL–22</td>
</tr>
<tr>
<td>15–1</td>
<td>NCS Routines</td>
<td>NCS–1</td>
</tr>
<tr>
<td>16–1</td>
<td>Routine Codes for Specification to PSM$REPLACE</td>
<td>PSM–14</td>
</tr>
<tr>
<td>17–1</td>
<td>Job Controller Functions</td>
<td>SMB–10</td>
</tr>
<tr>
<td>18–1</td>
<td>High-Performance Sort/Merge: Differences in SOR$ Routine Behavior</td>
<td>SOR–2</td>
</tr>
</tbody>
</table>
Preface

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

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 utility routine sets as follows:
- Access Control List (ACL) editor routine
- Backup (BACKUP) routine
- Command Language Interface (CLI) routines
- Common File Qualifier routines
- Convert (CONVERT) routines
- Data Compression/Expansion (DCX) routines
- DEC Text Processing Utility (DECTPU) routines
- EDT routines
- File Definition Language (FDL) routines
- Librarian (LBR) routines
- Lightweight Directory Access Protocol (LDAP) routines

Note
Because the LDAP routines are a C language interface, this chapter is formatted differently than the other chapters.

- 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

Related Documents

For additional information about OpenVMS products and services, access the following World Wide Web address:

http://www.openvms.compaq.com/

Reader’s Comments

Compaq welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet  openvmsdoc@compaq.com
Mail  Compaq Computer Corporation
OSSG Documentation Group, ZKO3-4/U08
110 Spit Brook Rd.
Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following World Wide Web address to order additional documentation:

http://www.openvms.compaq.com/

If you need help deciding which documentation best meets your needs, call 800-282-6672.

Conventions

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

The following conventions are also used in this manual:

Ctrl/x  A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.

PF1 x  A sequence such as PF1 x indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.

Return  In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)

...  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, if you choose more than one option, you must enclose the choices in parentheses.

[] In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)

{} In command format descriptions, braces indicate a required choice of options; you must choose one of the options listed.

**bold text** Bold text represents the introduction of a new term or 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=), and in command parameters in text (where device-name contains up to five alphanumeric characters).

**UPPERCASE TEXT** Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.

**Monospace text** Monospace type indicates code examples and interactive screen displays.

In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.

- A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.

**numbers** All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radices—binary, octal, or hexadecimal—are explicitly indicated.
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 utilities and their corresponding routines:

<table>
<thead>
<tr>
<th>Utility or Editor</th>
<th>Utility Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access control list editor</td>
<td>ACL editor routine</td>
</tr>
<tr>
<td>Backup application programming interface (API)</td>
<td>Backup API routine</td>
</tr>
<tr>
<td>Command Definition Utility</td>
<td>CLI routines</td>
</tr>
<tr>
<td>Common File Qualifier routines</td>
<td>UTIL$CQUAL routines</td>
</tr>
<tr>
<td>Convert and Convert/Reclaim utilities</td>
<td>CONVERT routines</td>
</tr>
<tr>
<td>EDT editor</td>
<td>EDT routines</td>
</tr>
<tr>
<td>DEC Text Processing Utility</td>
<td>DECTPU routines</td>
</tr>
<tr>
<td>File Definition Language facility</td>
<td>FDL routines</td>
</tr>
<tr>
<td>LOGINOUT callouts</td>
<td>LGI routines</td>
</tr>
<tr>
<td>Librarian utility</td>
<td>LBR routines</td>
</tr>
<tr>
<td>Mail utility</td>
<td>MAIL routines</td>
</tr>
<tr>
<td>National character set utility</td>
<td>NCS routines</td>
</tr>
<tr>
<td>Sort/Merge utility</td>
<td>SOR routines</td>
</tr>
</tbody>
</table>

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:
Like all other system routines in the OpenVMS environment, the utility routines described in this manual conform to the 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.
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
- Capabilities (VAX only)
- 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 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

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:

<table>
<thead>
<tr>
<th>Item code</th>
<th>Buffer length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer address</td>
<td></td>
</tr>
<tr>
<td>Return length address</td>
<td></td>
</tr>
</tbody>
</table>
The following table provides a detailed description of each item list entry:

<table>
<thead>
<tr>
<th>Item Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLEDIT$C_OBJNAM</td>
<td>Specifies the name of the object whose ACL is being edited.</td>
</tr>
<tr>
<td>ACLEDIT$C_OBJTYP</td>
<td>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).</td>
</tr>
<tr>
<td>ACLEDIT$OPTIONS</td>
<td>Represents a longword bit mask of the various options available to control the editing session.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLEDIT$V_JOURNAL</td>
<td>Indicates that the editing session is to be journaled.</td>
</tr>
<tr>
<td>ACLEDIT$V_RECOVER</td>
<td>Indicates that the editing session is to be recovered from an existing journal file.</td>
</tr>
<tr>
<td>ACLEDIT$V_KEEP_RECOVER</td>
<td>Indicates that the journal file used to recover the editing session is not to be deleted when the recovery is complete.</td>
</tr>
<tr>
<td>ACLEDIT$V_KEEP_JOURNAL</td>
<td>Indicates that the journal file used for the editing session is not to be deleted when the session ends.</td>
</tr>
<tr>
<td>ACLEDIT$V_PROMPT_MODE</td>
<td>Indicates that the session is to use automatic text insertion (prompting) to build new access control list entries (ACEs).</td>
</tr>
</tbody>
</table>

| 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. |
### Item Identifier Description

<table>
<thead>
<tr>
<th>Item Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLEDIT$C_CLSNAM</td>
<td>A string descriptor that points to the class name of the object whose ACL is being modified. The following are valid class names:</td>
</tr>
<tr>
<td></td>
<td>• CAPABILITY†</td>
</tr>
<tr>
<td></td>
<td>• COMMON_EVENT_FLAG_CLUSTER</td>
</tr>
<tr>
<td></td>
<td>• DEVICE</td>
</tr>
<tr>
<td></td>
<td>• FILE</td>
</tr>
<tr>
<td></td>
<td>• GROUP_GLOBAL_SECTION</td>
</tr>
<tr>
<td></td>
<td>• LOGICAL_NAME_TABLE</td>
</tr>
<tr>
<td></td>
<td>• QUEUE</td>
</tr>
<tr>
<td></td>
<td>• RESOURCE_DOMAIN</td>
</tr>
<tr>
<td></td>
<td>• SECURITY_CLASS</td>
</tr>
<tr>
<td></td>
<td>• SYSTEM_GLOBAL_SECTION</td>
</tr>
<tr>
<td></td>
<td>• VOLUME</td>
</tr>
<tr>
<td></td>
<td>If both OBJTYP and CLSNAM are omitted, the object is assumed to belong to the FILE class.</td>
</tr>
</tbody>
</table>

†VAX specific.

### 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 ACLEDIT$SECTION should be defined. See the OpenVMS System Management Utilities Reference Manual for more information.

### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>RMS$_xxx</td>
<td>See the OpenVMS Record Management Services Reference Manual for a description of OpenVMS RMS status codes.</td>
</tr>
<tr>
<td>TPU$_xxx</td>
<td>See Chapter 8 for a description of the TPU-specific condition values that may be returned by ACLEDIT$EDIT.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>bck_opt_struct_adr</td>
<td>32-bit address</td>
</tr>
<tr>
<td>bck_opt_struct_dsc</td>
<td>Static string descriptor</td>
</tr>
<tr>
<td>bck_opt_struct_dsc64</td>
<td>Reserved for use by Compaq</td>
</tr>
<tr>
<td>bck_opt_struct_dt</td>
<td>Date/Time quadword (ADT)</td>
</tr>
<tr>
<td>bck_opt_struct_flag</td>
<td>Logical bit flags</td>
</tr>
<tr>
<td>bck_opt_struct_int</td>
<td>32-bit integer</td>
</tr>
</tbody>
</table>

The option structure types are defined in the language definition files. Table 3–1 lists the language definition files.
3.1 Introduction to the Backup API

Table 3–1 Backup API Language Definition Files

<table>
<thead>
<tr>
<th>Language</th>
<th>API Definitions</th>
<th>Media Format (Save Set) Definitions</th>
<th>Backup Utility Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>BAPIDEF.BAS</td>
<td>BACKDEF.BAS</td>
<td>BACKSTRUC.BAS</td>
</tr>
<tr>
<td>BLISS</td>
<td>BAPIDEF.R32</td>
<td>BACKDEF.R32</td>
<td>BACKSTRUC.R32</td>
</tr>
<tr>
<td>C</td>
<td>BAPIDEF.H</td>
<td>BACKDEF.H</td>
<td>BACKSTRUC.H</td>
</tr>
<tr>
<td>Fortran</td>
<td>BAPIDEF.FOR</td>
<td>BACKDEF.FOR</td>
<td>BACKSTRUC.FOR</td>
</tr>
<tr>
<td>MACRO</td>
<td>BAPIDEF.MAR</td>
<td>BACKDEF.MAR</td>
<td>BACKSTRUC.MAR</td>
</tr>
</tbody>
</table>

See the OpenVMS System Management Utilities Reference Manual: A–L for detailed definitions of the DCL command BACKUP qualifiers. See the OpenVMS System Manager’s Manual, Volume 1: Essentials 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:

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

Example 3–1 Calling the Backup API with a VAX C Program

```c
#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";
```

(continued on next page)
Example 3–1 (Cont.) Calling the Backup API with a VAX C Program

```c
main()
{
    input_dsc.dsc$b_dtype = DSC$K_DTYPE_T;
    output_dsc.dsc$b_dtype = DSC$K_DTYPE_T;
    input_dsc.dsc$b_class = DSC$K_CLASS_S;
    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 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

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_END_OPT</td>
<td>Flag that contains all zeros to denote the end of <code>argument-buffer</code>. 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.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 3–2 (Cont.)  BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_ALIAS</td>
<td>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.)</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use of BCK_OPT_K_ALIAS and BCK_OPT_K_PHYSICAL in the same call results in a fatal error.</td>
</tr>
<tr>
<td>BCK_OPT_K_ASSIST</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_BACKUP</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_BEFORE_TYPE</td>
<td>Logical bit flags that qualify the date specified in the BCK_OPT_K_BEFORE_VALUE option structure. Type can be one of the following:</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_BEFORE_K_BACKUP</strong></td>
</tr>
<tr>
<td></td>
<td>Selects files last saved or copied by BACKUP before the date specified. Also selects files with no BACKUP date.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_BEFORE_K_CREATED</strong></td>
</tr>
<tr>
<td></td>
<td>Selects files created before the date specified.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_BEFORE_K_EXPIRED</strong></td>
</tr>
<tr>
<td></td>
<td>Selects files that have expired as of the date specified.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_BEFORE_K_MODIFIED</strong> (Default) Selects files last modified before the date specified.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_BEFORE_K_SPECIFIED</strong></td>
</tr>
<tr>
<td></td>
<td>Reserved for use by Compaq.</td>
</tr>
</tbody>
</table>

(See the BACKUP qualifiers /BEFORE, /BACKUP, /CREATED, /EXPIRED, and /MODIFIED.)

(continued on next page)
Backup API
BACKUP$START

Table 3–2 (Cont.) BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BCK_OPT_K_BEFORE_VALUE</strong></td>
<td>Date-Time Quadword that specifies the date qualified by BCK_OPT_K_BEFORE_TYPE. You cannot use delta time.</td>
</tr>
<tr>
<td></td>
<td>(See the BACKUP qualifier /BEFORE.)</td>
</tr>
<tr>
<td><strong>BCK_OPT_K_BLOCK</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>The default block size for disk is 32,256 bytes.</td>
</tr>
<tr>
<td></td>
<td>(See the BACKUP qualifier /BLOCK_SIZE.)</td>
</tr>
<tr>
<td><strong>BCK_OPT_K_CARTRIDGE_MEDIA_IN</strong></td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> 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.</td>
</tr>
<tr>
<td><strong>BCK_OPT_K_CARTRIDGE_NAME_IN</strong></td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> 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.</td>
</tr>
<tr>
<td><strong>BCK_OPT_K_CARTRIDGE_SIDE_IN</strong></td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use of BCK_OPT_K_CARTRIDGE_SIDE_IN without BCK_OPT_K_CARTRIDGE_NAME_IN in the same call results in a fatal error.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> 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.</td>
</tr>
<tr>
<td><strong>BCK_OPT_K_CARTRIDGE_MEDIA_OUT</strong></td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> 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.</td>
</tr>
<tr>
<td><strong>BCK_OPT_K_CARTRIDGE_NAME_OUT</strong></td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> 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.</td>
</tr>
</tbody>
</table>

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

(continued on next page)
Table 3–2 (Cont.)  BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_CARTRIDGE_SIDE_OUT(^1)</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td><strong>Note:</strong> Use of BCK_OPT_K_CARTRIDGE_SIDE_OUT without BCK_OPT_K_CARTRIDGE_NAME_OUT in the same call results in a fatal error.</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_COMMAND</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_COMMENT</td>
<td>32-bit descriptor that specifies a comment string to be placed in the output save set.</td>
</tr>
<tr>
<td>(See the BACKUP qualifier /COMMENT.)</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_COMPARE</td>
<td>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).</td>
</tr>
<tr>
<td>(See the BACKUP qualifier /COMPARE.)</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_CONFIRM</td>
<td>Flag that specifies whether to prompt for confirmation before processing each file. Values are TRUE and FALSE (default).</td>
</tr>
<tr>
<td>(See the BACKUP qualifier /CONFIRM.)</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_CRC</td>
<td>Flag that specifies whether the software cyclic redundancy check (CRC) is to be performed. Values are TRUE (default) and FALSE.</td>
</tr>
<tr>
<td>(See the BACKUP qualifier /CRC.)</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_CREATED</td>
<td>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.</td>
</tr>
<tr>
<td>(See the BACKUP qualifiers /BEFORE, /SINCE, and /CREATED.)</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_DCL_INTERFACE</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_DELETE</td>
<td>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).</td>
</tr>
<tr>
<td>(See the BACKUP qualifier /DELETE.)</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\)Reserved for use by Media Management Extension (MME) layered products.
Table 3–2 (Cont.) BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_DENSITY</td>
<td>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.)</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use of BCK_OPT_K_DENSITY and BCK_OPT_K_MEDIA_FORMAT in the same call results in a fatal error.</td>
</tr>
<tr>
<td>BCK_OPT_K_DISMOUNT</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_DISPOSITION1</td>
<td>Logical bit flags. Values are the following:</td>
</tr>
<tr>
<td></td>
<td>BCK_OPTYP_DISP_K_KEEP</td>
</tr>
<tr>
<td></td>
<td>BCK_OPTYP_DISP_K_RELEASE</td>
</tr>
<tr>
<td>BCK_OPT_K_DRIVE_CLASS_IN1</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td>BCK_OPT_K_DRIVE_CLASS_OUT1</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td>BCK_OPT_K_ENCRYPT2</td>
<td>Flag.</td>
</tr>
<tr>
<td>BCK_OPT_K_ENCRYPT_USERALG2</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td>BCK_OPT_K_ENCRYPT_USERKEY2</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use of BCK_OPT_K_ENCRYPT_USERKEY and BCK_OPT_K_ENCRYPT_KEY_VALUE in the same call results in a fatal error.</td>
</tr>
<tr>
<td>BCK_OPT_K_ENCRYPT_KEY_VALUE2</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use of BCK_OPT_K_ENCRYPT_KEY_VALUE and BCK_OPT_K_ENCRYPT_USERKEY in the same call results in a fatal error.</td>
</tr>
<tr>
<td>BCK_OPT_K_EVENT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>BCK_OPT_K_EXACT_ORDER</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Values are TRUE (default) and FALSE.</td>
</tr>
<tr>
<td></td>
<td>(See the BACKUP qualifier /EXACT_ORDER.)</td>
</tr>
</tbody>
</table>

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

2Reserved for future use by a security utility or layered product.

(continued on next page)
### Table 3-2 (Cont.)  BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_EXCLUDE</td>
<td>32-bit descriptor that specifies the name of an input file to be excluded from the current BACKUP save or copy operation. 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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_EXPIRED</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_FAST</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_FILE_CALLBACK</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_FILEMERGE</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_PULL</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_GROUP</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_HANDLE</td>
<td>Reserved for use by Compaq.</td>
</tr>
</tbody>
</table>
### Backup Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BCK_OPT_K_IGNORE_TYPES</strong></td>
<td>Logical bit flags that override tape labeling checks or restrictions placed on files. Values are one of the following:</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_IGNORE_K_ACCESS</strong></td>
</tr>
<tr>
<td></td>
<td>Processes files on a tape that is protected by a volume accessibility character, or a tape created by HSC Backup. Applies to all tapes in the save set.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_IGNORE_K_INTERLOCK</strong></td>
</tr>
<tr>
<td></td>
<td>Processes files otherwise inaccessible because of file access conflicts.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_IGNORE_K_LABELS</strong></td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>BCK_OPTYP_IGNORE_K_NOBACKUP</strong></td>
</tr>
<tr>
<td></td>
<td>Processes both the file header and the contents of files marked with the NOBACKUP option.</td>
</tr>
</tbody>
</table>

(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.)                                                                   |

(continued on next page)
### Table 3–2 (Cont.) BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_INTERCHANGE</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_JOURNAL</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_JOURNAL_FILE</td>
<td>32-bit descriptor that specifies the name of a BACKUP journal file to be processed. (See the BACKUP qualifier /JOURNAL.)</td>
</tr>
<tr>
<td>BCK_OPT_K_LABEL</td>
<td>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.)</td>
</tr>
<tr>
<td><strong>Note:</strong> Use of BCK_OPT_K_LABEL with any BCK_OPT_K_SCRATCH_* option structure in the same call results in a fatal error.</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_LIST</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_LIST_FILE</td>
<td>32-bit descriptor that specifies the name of a file of a BACKUP journal file to be processed. (See the BACKUP qualifier /LIST.)</td>
</tr>
<tr>
<td>BCK_OPT_K_LOG</td>
<td>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.)</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 3–2 (Cont.) BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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)  
  (See the BACKUP qualifier /MEDIA_FORMAT.) |
| 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. Compaq 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 |

(continued on next page)
<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_OUTPUT</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_OVERLAY</td>
<td>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. Values are TRUE and FALSE (default). (See the BACKUP qualifier /OVERLAY.)</td>
</tr>
<tr>
<td>BCK_OPT_K_OWNER_IN_VALUE</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_OWNER_OUT_TYPE</td>
<td>Logical bit flags to specify the user identification code (UIC) of restored files. Values are one of the following:</td>
</tr>
<tr>
<td></td>
<td>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).</td>
</tr>
<tr>
<td></td>
<td>BCK_OPTYP_OWN_OUT_KORIGINAL Retains the owner UIC of the file being restored (default if BCK_OPT_K_IMAGE or BCK_OPT_K_INCREMENTAL is TRUE).</td>
</tr>
<tr>
<td></td>
<td>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 SYSPRIV user privilege, or be the owner of the output volume, or must have the parent UIC.</td>
</tr>
</tbody>
</table>

(See the BACKUP qualifier /BY_OWNER.)

(continued on next page)
Backup API
BACKUP$START

Table 3–2 (Cont.)  BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_OWNER_OUT_VALUE</td>
<td>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. (See the BACKUP qualifier /BY_OWNER.)</td>
</tr>
<tr>
<td>BCK_OPT_K_PHYSICAL</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_PROTECTION</td>
<td>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 OpenVMS Record Management Services Reference Manual for information about the format of this field. (Also see BACKUP utility qualifier /PROTECTION.)</td>
</tr>
<tr>
<td>BCK_OPT_K_RECORD</td>
<td>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.)</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_RELEASE_TAPE</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_REPLACE</td>
<td>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). (See the BACKUP qualifier /REPLACE.)</td>
</tr>
<tr>
<td>BCK_OPT_K_REWIND</td>
<td>Flag. Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_REWIND_IN</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_REWIND_OUT</td>
<td>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.</td>
</tr>
<tr>
<td>BCK_OPT_K_SAVE_SET_IN</td>
<td>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.)</td>
</tr>
<tr>
<td>BCK_OPT_K_SAVE_SET_OUT</td>
<td>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.)</td>
</tr>
</tbody>
</table>
Table 3–2 (Cont.)  BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_SCRATCH_ASGN_TYPE ¹</td>
<td>Logical bit flags.</td>
</tr>
<tr>
<td><strong>Note:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_SCRATCH_COLLECTION ¹</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td><strong>Note:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_SCRATCH_LOCATION ¹</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td><strong>Note:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_SCRATCH_MEDIA_NAME ¹</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td><strong>Note:</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>BCK_OPT_K_SELECT</td>
<td>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.)</td>
</tr>
</tbody>
</table>

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

(continued on next page)
### Table 3–2 (Cont.) BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_SINCE_TYPE</td>
<td>Logical bit flags that qualify the date specified in the BCK_OPT_K_SINCE_VALUE option structure.</td>
</tr>
<tr>
<td></td>
<td>Type can be one of the following:</td>
</tr>
<tr>
<td>BCK_OPTYP_SINCE_K_BACKUP</td>
<td>Selects files last saved or copied by BACKUP on or after the date specified. Also selects files with no BACKUP date.</td>
</tr>
<tr>
<td>BCK_OPTYP_SINCE_K_CREATED</td>
<td>Selects files created on or after the date specified.</td>
</tr>
<tr>
<td>BCK_OPTYP_SINCE_K_EXPIRED</td>
<td>Selects files that have expired since the date specified.</td>
</tr>
<tr>
<td>BCK_OPTYP_SINCE_K_MODIFIED</td>
<td>Selects files last modified on or after the date specified (default).</td>
</tr>
<tr>
<td>BCK_OPTYP_SINCE_K_SPECIFIED</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_SINCE_VALUE</td>
<td>Date-Time Quadword that specifies the date qualified by BCK_OPTYP_K_SINCE_TYPE. You cannot use delta time.</td>
</tr>
<tr>
<td></td>
<td>(See the BACKUP qualifier /SINCE.)</td>
</tr>
<tr>
<td>BCK_OPT_K_STORAGE_MANAGEMENT ¹</td>
<td>32-bit descriptor.</td>
</tr>
<tr>
<td>BCK_OPT_K_TAPE_EXPIRATION</td>
<td>ADT (Date-Time) that specifies when the tape expires.</td>
</tr>
<tr>
<td></td>
<td>(See the BACKUP qualifier /TAPE_EXPIRATION.)</td>
</tr>
<tr>
<td>BCK_OPT_K_TRUNCATE</td>
<td>Flag that specifies whether a copy or restore operation truncates a sequential output file at the end-of-file (EOF) when creating it.</td>
</tr>
<tr>
<td></td>
<td>Values are TRUE and FALSE (default; the size of the output file is determined by the allocation of the input file).</td>
</tr>
<tr>
<td></td>
<td>(See the BACKUP qualifier /TRUNCATE.)</td>
</tr>
</tbody>
</table>

¹Reserved for use by Media Management Extension (MME) layered products.
Table 3–2 (Cont.) BACKUP Option Structure Types

<table>
<thead>
<tr>
<th>Option Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_OPT_K_UNSHELVE</td>
<td>Flag that is reserved for use with file-shelving layered products. Values are TRUE and FALSE. <strong>Note:</strong> Use of BCK_OPT_K_UNSHELVE and BCK_OPT_K_PHYSICAL in the same call results in a fatal error.</td>
</tr>
<tr>
<td>BCK_OPT_K_VALIDATE_PARAMETERS</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>BCK_OPT_K_VERIFY</td>
<td>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). <strong>(See the BACKUP qualifier /VERIFY.)</strong></td>
</tr>
<tr>
<td>BCK_OPT_K_VOLUME</td>
<td>Integer that specifies the specific disk volume in a disk volume set to be processed (valid only when BCK_OPT_K_IMAGE is TRUE). <strong>(See the BACKUP qualifier /VOLUME.)</strong></td>
</tr>
</tbody>
</table>

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, Compaq 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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Element Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct dsc$descriptor</td>
<td>bckevt_r_event_buffer</td>
<td>Pointer to event data</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckevt_l_event_type</td>
<td>Event type</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckevt_l_event_subtype</td>
<td>Event subtype (if any)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckevt_q_event_ctx [2]</td>
<td>Reserved for use by Compaq</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckevt_l_event_handle</td>
<td>Reserved for use by Compaq</td>
</tr>
</tbody>
</table>

Table 3–4 describes the values returned in the bckEvent data structure.

Table 3–4 Event Callback Buffer Formats

<table>
<thead>
<tr>
<th>Type/Subtype</th>
<th>Format</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_EVENT_K_CONTROL</td>
<td>bckControl</td>
<td>See Table 3–5.</td>
</tr>
<tr>
<td>BCK_EVENT_K_ERROR_MSG</td>
<td>bckMsgVect</td>
<td>Message vector (use $PUTMSG to output message to user).</td>
</tr>
<tr>
<td>(no subtype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_JOURNAL_OPEN</td>
<td>dsc$descriptor</td>
<td>String descriptor (name of file to create).</td>
</tr>
<tr>
<td>(no subtype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_JOURNAL_CLOSE</td>
<td>dsc$descriptor</td>
<td>String descriptor (name of file to close).</td>
</tr>
<tr>
<td>(no subtype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_JOURNAL_WRITE</td>
<td>512-byte block</td>
<td>File descriptor of journal buffer (condensed journal records, refer to the BJLDEF structure definition in the BAPIDEF files).</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 3–4 (Cont.) Event Callback Buffer Formats

<table>
<thead>
<tr>
<th>Type/Subtype</th>
<th>Format</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCK_EVENT_K_LIST_CLOSE</td>
<td>Array of 2</td>
<td>LIST_TOTFILE: Total files listed.</td>
</tr>
<tr>
<td>(no subtype)</td>
<td>longwords</td>
<td>LIST_TOTSIZE: Total blocks listed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The application should close the list file.</td>
</tr>
<tr>
<td>BCK_EVENT_K_LIST_OPEN</td>
<td>dsc$descriptor</td>
<td>File specification of list file to open</td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td>(TRUE = 1, indicates /FULL listing).</td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
<td>(FALSE = 0).</td>
</tr>
<tr>
<td>BCK_EVENT_K_LIST_WRITE</td>
<td>dsc$descriptor</td>
<td>Journal file listing information (TRUE = 1, indicates a change of volume or save set).</td>
</tr>
<tr>
<td>BRH$K_SUMMARY</td>
<td>BSRBLK</td>
<td>List BACKUP save set - save set summary record.</td>
</tr>
<tr>
<td>BRH$K_VOLUME</td>
<td>BSRBLK</td>
<td>List BACKUP save set - volume summary record.</td>
</tr>
<tr>
<td>BRH$K_PHYSVOL</td>
<td>PVABLK</td>
<td>List BACKUP save set - physical volume record.</td>
</tr>
<tr>
<td>BRH$K_FILE</td>
<td>FARBLK</td>
<td>List BACKUP save set - file record.</td>
</tr>
<tr>
<td>BCK_EVENT_K_LISTJ OUR_WRITE</td>
<td></td>
<td>Subtype is a condition value that indicates the type of action that occurred for the specified file/item. Obtain message text with the $GETMSG system service.</td>
</tr>
<tr>
<td>TRUE</td>
<td>bckListourblk</td>
<td>Journal file listing of file/item specification string (descriptor) (TRUE = 1, indicates a change of volume or save set).</td>
</tr>
<tr>
<td>FALSE</td>
<td>dsc$descriptor</td>
<td>Journal file listing information (TRUE = 1, indicates a change of volume or save set).</td>
</tr>
<tr>
<td>BCK_EVENT_K_LOG</td>
<td>dsc$descriptor</td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_AECREATED</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_COMPARED</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_COPIED</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_CREATED</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_KERD</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_HEADCOPIED</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_INCDELETE</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BACKUP$_NEWSAVSET</td>
<td></td>
<td>String descriptor (file logging).</td>
</tr>
<tr>
<td>BCK_EVENT_K_OP_PHASE</td>
<td>Condition Value</td>
<td>Start of verify operation (obtain message text with $GETMSG). (continued on next page)</td>
</tr>
<tr>
<td>Type/Subtype</td>
<td>Format</td>
<td>Value Returned</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BACKUP$_STARTDELETE</td>
<td>Condition Value</td>
<td>Start of delete operation (obtain message text with $GETMSG).</td>
</tr>
<tr>
<td>BACKUP$_STARTRECORD</td>
<td>Condition Value</td>
<td>Start of record operation (obtain message text with $GETMSG).</td>
</tr>
<tr>
<td>BCK_EVENT_K_SAVESET_CLOSE</td>
<td>RMS FOB</td>
<td>A BACKUP save set must be closed.</td>
</tr>
<tr>
<td>(no subtype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_SAVESET_OPEN</td>
<td>RMS FOB</td>
<td>A BACKUP save set must be opened or created.</td>
</tr>
<tr>
<td>(no subtype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_SAVESET_READ</td>
<td>BACKUP Buffer Control Block</td>
<td>A BACKUP save set block/buffer has been read from the input save set.</td>
</tr>
<tr>
<td>(no subtype)</td>
<td>(BCBBLK)</td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_SAVESET_WRITE</td>
<td>BACKUP Buffer Control Block</td>
<td>A BACKUP save set block/buffer is ready to be written to the output save set.</td>
</tr>
<tr>
<td>(no subtype)</td>
<td>(BCBBLK)</td>
<td></td>
</tr>
<tr>
<td>BCK_EVENT_K_STATISTICS</td>
<td>bckMsgVect</td>
<td>Statistics message; one of the following message condition values (use $PUTMSG to output message to user):</td>
</tr>
<tr>
<td>(no subtype)</td>
<td></td>
<td>BACKUP$_STAT_PHYSICAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BACKUP$_STAT_SAVCOP_ACT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BACKUP$_STAT_INACTIVE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BACKUP$_STAT_COMPARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BACKUP$_STAT_RESTORE</td>
</tr>
<tr>
<td>BCK_EVENT_K_USER_MSG</td>
<td>bckMsgVect</td>
<td>Message vector (use $PUTMSG to output message to user).</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Subtype Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bckCntrl_l_event</td>
<td>BCKCNTRL_K_CONFIRM_EVENT</td>
</tr>
<tr>
<td>bckCntrl_l_function</td>
<td>Backup operation type (integer value)</td>
</tr>
<tr>
<td>bckCntrl_a_outmsgvect</td>
<td>Confirmation message (bckMsgVect, BACKUP$_CNTRL_CONFCOMP or BACKUP$_CNTRL_CONFCOPY)</td>
</tr>
<tr>
<td>bckCntrl_v_response_required</td>
<td>TRUE (response is required)</td>
</tr>
<tr>
<td>bckCntrl_r_response_buffer</td>
<td>dsc$descriptor (&quot;Yes/No&quot; string descriptor)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtype Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bckCntrl_l_event</td>
<td>BCKCNTRL_K_USER_ASSIST_EVENT or BCKCNTRL_K_OPER_ASSIST_EVENT</td>
</tr>
<tr>
<td>bckCntrl_l_function</td>
<td>Backup operation type (integer value)</td>
</tr>
<tr>
<td>bckCntrl_a_outmsgvect</td>
<td>bckMsgVect (assist and other messages)</td>
</tr>
<tr>
<td>bckCntrl_v_response_required</td>
<td>TRUE or FALSE (TRUE = 1, if response is required)</td>
</tr>
<tr>
<td>bckCntrl_r_response_buffer</td>
<td>dsc$descriptor (response string descriptor)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtype Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bckCntrl_l_event</td>
<td>BCKCNTRL_K_RESTART_EVENT</td>
</tr>
<tr>
<td>bckCntrl_l_function</td>
<td>Backup operation type (integer value)</td>
</tr>
<tr>
<td>bckCntrl_a_outmsgvect</td>
<td>bckMsgVect (operation restart message vector)</td>
</tr>
<tr>
<td>bckCntrl_v_response_required</td>
<td>FALSE (= 0, no response is required)</td>
</tr>
<tr>
<td>bckCntrl_r_response_buffer</td>
<td>dsc$descriptor (&quot;Yes/No&quot; string descriptor)</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Data Type</th>
<th>Element Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int</td>
<td>bckCntrl_l_event</td>
<td>Control event type.</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckCntrl_l_function</td>
<td>Backup operation type.</td>
</tr>
<tr>
<td>bckMsgVec</td>
<td>*bckCntrl_a_outmsgvect</td>
<td>Output messages and parameters.</td>
</tr>
<tr>
<td>union {</td>
<td>bckCntrl_l_ctlflags</td>
<td>Flags.</td>
</tr>
<tr>
<td>unsigned</td>
<td>bckCntrl_v_response_required</td>
<td>Response required = 1.</td>
</tr>
<tr>
<td>struct {</td>
<td>: 1</td>
<td></td>
</tr>
<tr>
<td>unsigned</td>
<td>bckCntrl_v_fill_5 : 7</td>
<td>Filler.</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>struct dsc$descriptor</td>
<td>bckCntrl_r_response_buffer</td>
<td>Descriptor for buffer to which response text is to be written.</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckCntrl_l_response_status</td>
<td>Reserved for use by Compaq.</td>
</tr>
<tr>
<td>unsigned int</td>
<td>bckCntrl_l_control_options</td>
<td>Reserved for use by Compaq.</td>
</tr>
</tbody>
</table>

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


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:*.*;*
```

```c
#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.
 */
Backup API
BACKUP$START

/ *  
   ** 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);
  }

BCK–26  Backup (BACKUP) Routine
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 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 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

```plaintext
********************************************************************************
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")
```

********************************************************************************
INCOME.FOR
********************************************************************************

PROGRAM INCOME
INTEGER STATUS, 2 CLI$DCL_PARSE, 2 CLI$DISPATCH
INCLUDE ‘($RMSDEF)’
INCLUDE ‘($STSDEF)’
EXTERNAL INCOME_SUBCOMMANDS,
2 LIB$GET_INPUT

(continued on next page)
Example 4–1 (Cont.) Using the CLI Routines to Retrieve Information About Command Lines in a Fortran Program

! 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)
(continued on next page)
Example 4–1 (Cont.) Using the CLI Routines to Retrieve Information About Command Lines in a Fortran Program

! 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$GET_INPUT in the OpenVMS RTL Library (LIB$) 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$GET_INPUT in the OpenVMS RTL Library (LIB$) 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$_INVREQTYP**  Calling process did not have a CLI to perform this function, or the CLI did not support the request.
- **CLI$_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$_NORMAL**  Normal successful completion.
- **RMS$_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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI$_INVREQTYP</td>
<td>Calling process did not have a CLI to perform this function or the CLI did not support the request.</td>
</tr>
<tr>
<td>CLI$_INVROUT</td>
<td>CLI$DISPATCH unable to invoke the routine. An invalid routine is specified in the command table, or no routine is specified.</td>
</tr>
</tbody>
</table>
CLI$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 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$COPY_DX routine will signal the errors. For a list of these errors, see the OpenVMS RTL String Manipulation (STR$) 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 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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI$ABSENT</td>
<td>Specified entity not present, and it is not present by default.</td>
</tr>
<tr>
<td>CLI$DEFAULTED</td>
<td>Specified entity not present, but it is present by default.</td>
</tr>
<tr>
<td>CLI$INVREQTYP</td>
<td>Calling process did not have a CLI to perform this function, or the CLI did not support the request.</td>
</tr>
<tr>
<td>CLI$LOCNEG</td>
<td>Specified qualifier present in negated form (with /NO) and used as a local qualifier.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CLI$_LOCPRES</td>
<td>Specified qualifier present and used as a local qualifier.</td>
</tr>
<tr>
<td>CLI$_NEGATED</td>
<td>Specified qualifier present in negated form (with /NO) and used as a global qualifier.</td>
</tr>
<tr>
<td>CLI$_PRESENT</td>
<td>Specified entity present in the command string. This status is returned for all entities except local qualifiers and positional qualifiers that are used locally.</td>
</tr>
</tbody>
</table>
This chapter describes the common file qualifier (UTIL$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>
<thead>
<tr>
<th>Routine Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTIL$CQUAL_FILE_PARSE</td>
<td>Parses the command line for the file qualifiers listed in Table 5–2, and obtains associated values. Returns a context value that is used when calling the matching and ending routines.</td>
</tr>
<tr>
<td>UTIL$CQUAL_FILE_MATCH</td>
<td>Compares the routine file input to the command line data obtained from the parse routine call.</td>
</tr>
<tr>
<td>UTIL$CQUAL_FILE_END</td>
<td>Deletes all virtual memory allocated during the command line parse routine call.</td>
</tr>
<tr>
<td>UTIL$CQUAL_CONFIRM_ACT</td>
<td>Prompts a user for a response from SYS$COMMAND.</td>
</tr>
</tbody>
</table>

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

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

```c
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:

```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 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, filename, filetype, 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:

```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:

```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:

```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:

```c
define verb foo
    image foo
    parameter p1,prompt="File",value(list,impct,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**

```c
$ 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$C_BID /* determine if HFS support is here */
define HFS_Support 1
#else
#define HFS_Support 0
#endif
#if !HFS_Support /* compensate for lack of HFS support */
define nam$l_rsa nam$l_rsa
define nam$b_rsl nam$b_rsl
define nam$l_long_result nam$l_rsa
#define nam$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

(continued on next page)
Common File Qualifier Routines
5.2 Using the Common File Qualifier Routines

Example 5–1 (Cont.) Using UTIL$CQUAL Routines to Process Files

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) {
                strcpyp(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$w_length = strlen(short_string);
            } else {
                short_prompt.dsc$w_length = 0;
                long_prompt.dsc$w_length = 0;
            }

        if ((status = UTIL$CQUAL_FILE_MATCH(&context, 
                                            &result_desc, 
                                            &short_prompt, &long_prompt, 
                                            0, 
                                            &prompt_form, 
                                            &ignore_flags) & 1) {
            strcpyp(outstring, process);
        }
    
    (continued on next page)
Example 5–1 (Cont.) Using UTIL$CQUAL Routines to Process Files

```
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
```

(continued on next page)
5.2 Using the Common File Qualifier Routines

Example 5–1 (Cont.) Using UTIL$CQUAL Routines to Process Files

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 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,l*) 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 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—Parse the Command Line

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.

Arguments

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

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Mask Value</th>
<th>Field Specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>/CONFIRM</td>
<td>UTIL$M_CQF_CONFIRM</td>
<td>UTIL$V_CQF_CONFIRM</td>
</tr>
<tr>
<td>/EXCLUDE</td>
<td>UTIL$M_CQF_EXCLUDE</td>
<td>UTIL$V_CQF_EXCLUDE</td>
</tr>
<tr>
<td>/BEFORE</td>
<td>UTIL$M_CQF_BEFORE</td>
<td>UTIL$V_CQF_BEFORE</td>
</tr>
<tr>
<td>/SINCE</td>
<td>UTIL$M_CQF_SINCE</td>
<td>UTIL$V_CQF_SINCE</td>
</tr>
<tr>
<td>/CREATED</td>
<td>UTIL$M_CQF_CREATED</td>
<td>UTIL$V_CQF_CREATED</td>
</tr>
<tr>
<td>/MODIFIED</td>
<td>UTIL$M_CQF_MODIFIED</td>
<td>UTIL$V_CQF_MODIFIED</td>
</tr>
<tr>
<td>/EXPIRED</td>
<td>UTIL$M_CQF_EXPIRED</td>
<td>UTIL$V_CQF_EXPIRED</td>
</tr>
<tr>
<td>/BACKUP</td>
<td>UTIL$M_CQF_BACKUP</td>
<td>UTIL$V_CQF_BACKUP</td>
</tr>
<tr>
<td>/BY_OWNER</td>
<td>UTIL$M_CQF_BYOWNER</td>
<td>UTIL$V_CQF_BYOWNER</td>
</tr>
</tbody>
</table>

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
Common File Qualifier Routines

UTIL$CQUAL_FILE_PARSE

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—Match a File with Selection Criteria

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

Format


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.

Arguments

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).
Common File Qualifier Routines

UTIL$CQUAL_FILE_MATCH

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:

- 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.

Abnormal Completion Codes:

- 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).
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.

Arguments

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—Ask User for Confirmation

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.

Arguments

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>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTIL$K_CQF_SHORT</td>
<td>Use prompt_string_1</td>
</tr>
<tr>
<td>UTIL$K_CQF_LONG</td>
<td>Use prompt_string_2</td>
</tr>
<tr>
<td>UTIL$K_CQF_UNSPECIFIED</td>
<td>None specified; use default.</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Positive Response</th>
<th>Negative Response</th>
<th>Stop Processing</th>
<th>Stop Prompting</th>
<th>Switch Prompts</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>NO</td>
<td>QUIT</td>
<td>ALL</td>
<td>CONDENSED</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>Ctrl/Z</td>
<td></td>
<td>EXPANDED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>&lt;Return&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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$_{INVAR}$: 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}$).
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.

These routines cannot be called from the asynchronous system trap (AST) level. In addition, in order to function properly, these routines require ASTs to remain enabled.

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)

(continued on next page)
Example 6–1 (Cont.) Using the CONVERT Routines in a Fortran Program

* Set up parameter list: number of options, CREATE, NOSHARE, FAST_LOAD, MERGE, APPEND, SORT, WORK_FILES, KEY=0, NOPAD, PAD CHARACTER, NOTRUNCATE, NOEXIT, NOPERIODS, FILL_BUCKETS, NOREAD_CHECK, NOWRITE_CHECK, FDL, and NOEXCEPTION.

** INTEGER*4 OPTIONS(19) /18,1,0,1,0,1,2,0,0,0,0,0,0,0,0,1,0/  
** 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>

*/

struct conv$statistics stats;

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

  void callback();

  int stat;

  struct conv$options param_list;

  (continued on next page)
Example 6–2 (Cont.) Using the CONVERT Routines in a C Program

```c
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 */
```

(continued on next page)
Example 6–2 (Cont.) Using the CONVERT Routines in a C Program

```c
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", 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$STOP (status);
    printf(" Load Cpu Time : %s\n", date);
    printf(" Load Page Faults : %d\n", stats.conv$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

```fortran
* * * 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’/
```

(continued on next page)
Example 6–3 (Cont.) Using the CONV$RECLAIM Routine in a Fortran Program

```fortran
* 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

```c
#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

<table>
<thead>
<tr>
<th>Array Element</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CONV$L_STATISTICS_COUNT</td>
<td>Number of statistics specified</td>
</tr>
<tr>
<td>1</td>
<td>CONV$L_FILE_COUNT</td>
<td>Number of files</td>
</tr>
<tr>
<td>2</td>
<td>CONV$L_RECORD_COUNT</td>
<td>Number of records</td>
</tr>
<tr>
<td>3</td>
<td>CONV$L_EXCEPTION_COUNT</td>
<td>Number of exception record</td>
</tr>
<tr>
<td>4</td>
<td>CONV$L_VALID_COUNT</td>
<td>Number of valid records</td>
</tr>
<tr>
<td>5</td>
<td>CONV$L_KEY_NUMBER</td>
<td>Most recent key processed</td>
</tr>
<tr>
<td>6</td>
<td>CONV$L_REC_OUT</td>
<td>Number of records sorted</td>
</tr>
<tr>
<td>7</td>
<td>CONV$L_NODES</td>
<td>Nodes in sort tree</td>
</tr>
<tr>
<td>8</td>
<td>CONV$L_WRK_ALQ</td>
<td>Work file allocation</td>
</tr>
<tr>
<td>9</td>
<td>CONV$L_INI_RUNS</td>
<td>Initial dispersion runs</td>
</tr>
<tr>
<td>10</td>
<td>CONV$L_MRG_ORDER</td>
<td>Maximum merge order</td>
</tr>
<tr>
<td>11</td>
<td>CONV$L_MRG_PASSES</td>
<td>Number of merge passes</td>
</tr>
<tr>
<td>12</td>
<td>CONV$L_SORT_DIO_COUNT</td>
<td>Sort direct IO</td>
</tr>
<tr>
<td>13</td>
<td>CONV$L_SORT_BIO_COUNT</td>
<td>Sort buffered IO</td>
</tr>
<tr>
<td>14</td>
<td>CONV$Q_SORT_ELAPSED_TIME</td>
<td>Sort elapsed time</td>
</tr>
<tr>
<td>15</td>
<td>CONV$Q_SORT_CPU_TIME</td>
<td>Sort CPU time</td>
</tr>
<tr>
<td>16</td>
<td>CONV$L_SORT_PF_COUNT</td>
<td>Number of page faults for sort</td>
</tr>
<tr>
<td>17</td>
<td>CONV$L_LOAD_DIO_COUNT</td>
<td>Load direct IO</td>
</tr>
<tr>
<td>18</td>
<td>CONV$L_LOAD_BIO_COUNT</td>
<td>Load buffered IO</td>
</tr>
<tr>
<td>19</td>
<td>CONV$Q_LOAD_ELAPSED_TIME</td>
<td>Load elapsed time</td>
</tr>
<tr>
<td>20</td>
<td>CONV$Q_LOAD_CPU_TIME</td>
<td>Load CPU time</td>
</tr>
<tr>
<td>21</td>
<td>CONV$L_LOAD_PF_COUNT</td>
<td>Number of page faults for load</td>
</tr>
</tbody>
</table>

**flags**

OpenVMS usage: mask_longword

Type: longword (unsigned)

Access: read only

Mechanism: by reference

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV$V_FDL_STRING</td>
<td>Interprets the <code>fdl_filespec</code> argument supplied in the call to CONV$PASS_FILES as an FDL specification in string form. By default, this argument is interpreted as the file name of an FDL file.</td>
</tr>
<tr>
<td>CONV$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

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

### framebuffer 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.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV$_NOKEY</td>
<td>No such key.</td>
</tr>
<tr>
<td>CONV$_NOTIDX</td>
<td>File is not an indexed file.</td>
</tr>
<tr>
<td>CONV$_NOTSEQ</td>
<td>Output file is not a sequential file.</td>
</tr>
<tr>
<td>CONV$_NOWILD</td>
<td>No wildcard permitted.</td>
</tr>
<tr>
<td>CONV$_OPENEXC</td>
<td>Error opening exception file specification.</td>
</tr>
<tr>
<td>CONV$_OPENIN</td>
<td>Error opening file specification as input.</td>
</tr>
<tr>
<td>CONV$_OPENOUT</td>
<td>Error opening file specification as output.</td>
</tr>
<tr>
<td>CONV$_ORDER</td>
<td>Routine called out of order.</td>
</tr>
<tr>
<td>CONV$_PAD</td>
<td>Packet Assembly/Disassembly (PAD) option ignored; output record format not fixed.</td>
</tr>
<tr>
<td>CONV$_PLV</td>
<td>Unsupported prolog version.</td>
</tr>
<tr>
<td>CONV$_PROERR</td>
<td>Error reading prolog.</td>
</tr>
<tr>
<td>CONV$_PROL_WRT</td>
<td>Prolog write error.</td>
</tr>
<tr>
<td>CONV$_READERR</td>
<td>Error reading file specification.</td>
</tr>
<tr>
<td>CONV$_REX</td>
<td>Record already exists.</td>
</tr>
<tr>
<td>CONV$_RMS</td>
<td>Record caused RMS severe error.</td>
</tr>
<tr>
<td>CONV$_RSK</td>
<td>Record shorter than primary key.</td>
</tr>
<tr>
<td>CONV$_RSZ</td>
<td>Record does not fit in block/bucket.</td>
</tr>
<tr>
<td>CONV$_RTL</td>
<td>Record longer than maximum record length.</td>
</tr>
<tr>
<td>CONV$_RTS</td>
<td>Record too short for fixed record format file.</td>
</tr>
<tr>
<td>CONV$_SEQ</td>
<td>Record not in order.</td>
</tr>
<tr>
<td>CONV$_UDF_BKS</td>
<td>Cannot convert UDF records into spanned file.</td>
</tr>
<tr>
<td>CONV$_UDF_BLK</td>
<td>Cannot fit UDF records into single block bucket.</td>
</tr>
<tr>
<td>CONV$_VALERR</td>
<td>Specified value is out of legal range.</td>
</tr>
<tr>
<td>CONV$_VFC</td>
<td>Record too short to fill fixed part of VFC record.</td>
</tr>
<tr>
<td>CONV$_WRITEERR</td>
<td>Error writing file specification.</td>
</tr>
</tbody>
</table>
CONV$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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV$V_FDL_STRING</td>
<td>Interprets the fdl_filespec argument as an FDL specification in string form. By default, this argument is interpreted as a file name of an FDL file.</td>
</tr>
<tr>
<td>CONV$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>CONV$_INP_FILES</td>
<td>Too many input files.</td>
</tr>
<tr>
<td>CONV$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>CONV$_NARG</td>
<td>Wrong number of arguments.</td>
</tr>
<tr>
<td>CONV$_ORDER</td>
<td>Routine called out of order.</td>
</tr>
</tbody>
</table>
CONV$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 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

<table>
<thead>
<tr>
<th>Element Number</th>
<th>Symbolic Value</th>
<th>Longword Default Value</th>
<th>Qualifier Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CONV$L_OPTIONS_COUNT</td>
<td>None</td>
<td>Not applicable</td>
</tr>
<tr>
<td>1</td>
<td>CONV$L_CREATE</td>
<td>1</td>
<td>/CREATE</td>
</tr>
<tr>
<td>2</td>
<td>CONV$L_SHARE</td>
<td>0</td>
<td>/NOSHARE</td>
</tr>
<tr>
<td>3</td>
<td>CONV$L_FAST</td>
<td>1</td>
<td>/FAST_LOAD</td>
</tr>
<tr>
<td>4</td>
<td>CONV$L_MERGE</td>
<td>0</td>
<td>/NOMERGE</td>
</tr>
<tr>
<td>5</td>
<td>CONV$L_APPEND</td>
<td>0</td>
<td>/NOAPPEND</td>
</tr>
<tr>
<td>6</td>
<td>CONV$L_SORT</td>
<td>1</td>
<td>/SORT</td>
</tr>
<tr>
<td>7</td>
<td>CONV$L_WORK_FILES</td>
<td>2</td>
<td>/WORKFILES=2</td>
</tr>
<tr>
<td>8</td>
<td>CONV$L_KEY</td>
<td>0</td>
<td>/KEY=0</td>
</tr>
<tr>
<td>9</td>
<td>CONV$L_PAD</td>
<td>0</td>
<td>/NOPAD</td>
</tr>
<tr>
<td>10</td>
<td>CONV$L_PAD_CHARACTER</td>
<td>0</td>
<td>Pad character=0</td>
</tr>
<tr>
<td>11</td>
<td>CONV$L_TRUNCATE</td>
<td>0</td>
<td>/NOTRUNCATE</td>
</tr>
<tr>
<td>12</td>
<td>CONV$L_EXIT</td>
<td>0</td>
<td>/NOEXIT</td>
</tr>
<tr>
<td>13</td>
<td>CONV$L_FIXED_CONTROL</td>
<td>0</td>
<td>/NOFIXED_CONTROL</td>
</tr>
<tr>
<td>14</td>
<td>CONV$L_FILL_BUCKETS</td>
<td>0</td>
<td>/NOFILL_BUCKETS</td>
</tr>
<tr>
<td>15</td>
<td>CONV$L_READ_CHECK</td>
<td>0</td>
<td>/NOREAD_CHECK</td>
</tr>
<tr>
<td>16</td>
<td>CONV$L_WRITE_CHECK</td>
<td>0</td>
<td>/NOWRITE_CHECK</td>
</tr>
<tr>
<td>17</td>
<td>CONV$L_FDL</td>
<td>0</td>
<td>/NOFDL</td>
</tr>
<tr>
<td>18</td>
<td>CONV$L_EXCEPTION</td>
<td>0</td>
<td>/NOEXCEPTION</td>
</tr>
<tr>
<td>19</td>
<td>CONV$L_PROLOGUE</td>
<td>None</td>
<td>/PROLOGUE=n²</td>
</tr>
<tr>
<td>20</td>
<td>CONV$L_IGNORE_PROLOGUE</td>
<td>0</td>
<td>Not applicable</td>
</tr>
<tr>
<td>21</td>
<td>CONV$L_SECONDARY</td>
<td>1</td>
<td>SECONDARY=1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV$V_FDL_STRING</td>
<td>Interprets the <code>fdl_filespec</code> argument supplied in the call to <code>CONV$PASS_FILES</code> as an FDL specification in string form. By default, this argument is interpreted as the file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

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

```plaintext
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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>CONV$_BADBLK</td>
<td>Invalid option block.</td>
</tr>
<tr>
<td>CONV$_CONFQUAL</td>
<td>Conflicting qualifiers.</td>
</tr>
<tr>
<td>CONV$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>CONV$_NARG</td>
<td>Wrong number of arguments.</td>
</tr>
<tr>
<td>CONV$_OPENEXC</td>
<td>Error opening exception file specification.</td>
</tr>
<tr>
<td>CONV$_ORDER</td>
<td>Routine called out of order.</td>
</tr>
</tbody>
</table>
CONV$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

<table>
<thead>
<tr>
<th>Array Element</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>RECL$L_STATISTICS_COUNT</code></td>
<td>Number of statistics specified</td>
</tr>
<tr>
<td>1</td>
<td><code>RECL$L_SCAN_COUNT</code></td>
<td>Data buckets scanned</td>
</tr>
<tr>
<td>2</td>
<td><code>RECL$L_DATA_COUNT</code></td>
<td>Data buckets reclaimed</td>
</tr>
<tr>
<td>3</td>
<td><code>RECL$L_INDEX_COUNT</code></td>
<td>Index buckets reclaimed</td>
</tr>
<tr>
<td>4</td>
<td><code>RECL$L_TOTAL_COUNT</code></td>
<td>Total buckets reclaimed</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>CONV$_BADLOGIC</td>
<td>Internal logic error detected.</td>
</tr>
<tr>
<td>CONV$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>CONV$_INVBKT</td>
<td>Invalid bucket at VBN n.</td>
</tr>
<tr>
<td>CONV$_NOTIDX</td>
<td>File is not an indexed file.</td>
</tr>
<tr>
<td>CONV$_NOWILD</td>
<td>No wildcard permitted.</td>
</tr>
<tr>
<td>CONV$_OPENIN</td>
<td>Error opening file specification as input.</td>
</tr>
<tr>
<td>CONV$_PLV</td>
<td>Unsupported prolog version.</td>
</tr>
<tr>
<td>CONV$_PROERR</td>
<td>Error reading prolog.</td>
</tr>
<tr>
<td>CONV$_PROL_WRT</td>
<td>Prolog write error.</td>
</tr>
<tr>
<td>CONV$_READERR</td>
<td>Error reading file specification.</td>
</tr>
<tr>
<td>CONV$_WRITEERR</td>
<td>Error writing output file.</td>
</tr>
</tbody>
</table>
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.
Data Compression/Expansion (DCX) Routines
7.1 Introduction to DCX Routines

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$COMPRESS_DATA routine to compress each record. As you compress each record, use unformatted I/O to write the compressed record to the file containing the compression/expansion function. For each record, write the length of the record and the substring containing the record. See the COMPRESS DATA section in Example 7–1. (The length is stored with
7. Use DCX$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 Compaq 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 Compaq 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 ’(SPORDEF)’
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’)
Example 7–1 (Cont.) Compressing a File in a Compaq Fortran Program

! ************
! 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)

(continued on next page)
Example 7–1 (Cont.) Compressing a File in a Compaq Fortran Program

! 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
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 (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.
(continued on next page)
Example 7–1 (Cont.)  Compressing a File in a Compaq Fortran Program

! dummy arguments
! context variable
INTEGER CONTEXT
! logical unit number
INTEGER LUN
! compression/expansion function
INTEGER MAP, MAP_LEN
! status variable
INTEGER STATUS, IOSTAT, IO_OK, 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$ANALYZE_DATA, DCX$MAKE_MAP

! analyze records
EOF = .FALSE.
READ (UNIT = LUN, FMT = ' (Q,A)', 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$ANALYZE_DATA (CONTEXT, RECORD(1:RECORD_LEN))
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  READ (UNIT = LUN, FMT = ' (Q,A)', 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 IF

END DO

(continued on next page)
Example 7–1 (Cont.) Compressing a File in a Compaq Fortran Program

```fortran
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 Compaq Fortran program.

Example 7–2 Expanding a Compressed File in a Compaq Fortran Program

```fortran
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
(continued on next page)
```
Example 7-2 (Cont.) Expanding a Compressed File in a Compaq Fortran Program

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

! open file to hold expanded data
STATUS = LIB$GET_LUN (NEW_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (NEW_FILE,
1  'File to hold expanded data: ',
2  NEW_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = NEW_LUN,
1  STATUS = 'NEW',
2  CARRIAGECONTROL = 'LIST',
3  FILE = NEW_FILE(1:NEW_LEN))

! expand file
! get length of compression/expansion function
READ (UNIT = OLD_LUN) MAP_LEN
STATUS = LIB$GET_VM (MAP_LEN,
1  MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! expand records
CALL EXPAND_DATA (%VAL(MAP),
1  MAP_LEN, ! length of function
2  OLD_LUN, ! compressed data file
3  NEW_LUN) ! expanded data file

! delete virtual memory used for function
STATUS = LIB$FREE_VM (MAP_LEN,
1  MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END

SUBROUTINE EXPAND_DATA (MAP, ! passed
1  MAP_LEN, ! passed
2  OLD_LUN, ! passed
3  NEW_LUN) ! passed

! expand data program
! dummy arguments
INTEGER MAP_LEN, ! length of expansion function
1  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,
1  IOSTAT,
2  IO_OK,
3  STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE ‘$(FORDEF)’

(continued on next page)
Example 7–2 (Cont.) Expanding a Compressed File in a Compaq Fortran Program

! context variable
INTEGER CONTEXT

! logical end_of_file
LOGICAL EOF

! record buffers
CHARACTER*32764 RECORD, RECORD2
INTEGER RECORD_LEN, RECORD2_LEN

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

! read data compression/expansion function
READ (UNIT = OLD_LUN) MAP
! initialize work area
STATUS = DCX$EXPAND_INIT (CONTEXT, %LOC(MAP(1)))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! expand records
EOF = .FALSE.
! read length of compressed record
READ (UNIT = OLD_LUN, 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 IF

DO WHILE (.NOT. EOF)
! read compressed record
READ (UNIT = OLD_LUN) RECORD (1:RECORD_LEN)
! expand record
STATUS = DCX$EXPAND_DATA (CONTEXT, RECORD(1:RECORD_LEN),
RECORD2, RECORD2_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! write expanded record to new file
WRITE (UNIT = NEW_LUN,
FMT = '(A)') RECORD2(1:RECORD2_LEN)
! read length of compressed record
READ (UNIT = OLD_LUN, 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 IF

(continued on next page)
Example 7–2 (Cont.) Expanding a Compressed File in a Compaq Fortran Program

    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$C_ONE_PASS
Data Compression/Expansion (DCX) Routines
DCX$ANALYZE_INIT

**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:

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Corresponding Item Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCX$C_BOUNDED</td>
<td>A Boolean variable. If bit &lt;0&gt; is true (equals 1), you are stating your intention to submit for analysis all data records that will be compressed; doing so often enables DCX to compute a better compression algorithm. If bit &lt;0&gt; is false (equals 0) or if the DCX$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.</td>
</tr>
<tr>
<td>DCX$C_EST_BYTES</td>
<td>A longword value containing your estimate of the total number of data bytes that will be submitted for compression. This estimate is useful in those cases where fewer than the total number of bytes are presented for analysis. If you do not specify the DCX$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.</td>
</tr>
<tr>
<td>DCX$C_EST_RECORDS</td>
<td>A longword value containing your estimate of the total number of data records that will be submitted for compression. This estimate is useful in those cases where fewer than the total number of records are presented for analysis. If you do not specify the DCX$C_EST_RECORDS item code, DCX submits for compression the same number of bytes that was presented for analysis.</td>
</tr>
</tbody>
</table>
### Item Code Corresponding Item Value

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCX$C_LIST</td>
<td>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$C_LIST item code may be specified, in a single call, alone or together with any of the other item-code and item-value pairs.</td>
</tr>
<tr>
<td>DCX$C_ONE_PASS</td>
<td>A Boolean variable. If bit &lt;0&gt; is true (equals 1), you make a binding request that DCX make only one pass over the data to be analyzed. If bit &lt;0&gt; 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.</td>
</tr>
</tbody>
</table>

### 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.
Data Compression/Expansion (DCX) Routines

DCX$COMPRESS_DATA

**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 ValuesReturned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCX$_INVCTX</td>
<td>Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.</td>
</tr>
<tr>
<td>DCX$_INVDATA</td>
<td>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$C_BOUNDED item value means that you must analyze all data records that are to be compressed.</td>
</tr>
<tr>
<td>DCX$_INVMAP</td>
<td>Error; invalid map. The <strong>map</strong> argument was not specified correctly in the DCX$ANALYZE_INIT routine or the context area is invalid.</td>
</tr>
<tr>
<td>DCX$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>DCX$_TRUNC</td>
<td>Error. The compressed data record has been truncated because the <strong>out_rec</strong> descriptor did not specify enough memory to accommodate the record.</td>
</tr>
</tbody>
</table>

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$COMPRESSDONE.

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCX$_INVMAP</td>
<td>Error; invalid map. The map argument was not specified correctly, or the context area is invalid.</td>
</tr>
<tr>
<td>DCX$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
</tbody>
</table>

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$EXPAND_DATA.
DCX$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.
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 a VAX processor or 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 Interfaces to Callable DECTPU

8.1.1 Simplified Callable Interface

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$INITIALIZE)
- Execute DECTPU procedures (TPU$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.

```
Name of your routine

Environment
```

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.

¹ This manual has been archived but is available on the OpenVMS Documentation CD-ROM.
8.1 Introduction to DECTPU Routines

- 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 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 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.

```c
/*
 * 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 $DESCRIPTOR (output_file, "outfile.dat");
main (argc, argv)
int argc;
char *argv[];
{
    /*
     * Call DECTPU to edit text in "infile.dat" and write the result to "outfile.dat". Return the condition code from DECTPU as the status of this program.
     */
```
The next example performs the same task as the previous example. This time, the TPU$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
DEC Text Processing Utility (DECTPU) Routines

8.3 Full Callable Interface

- 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 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; Compaq 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

(continued on next page)
Example 8–1 (Cont.) Sample VAX BLISS Template for Callable DECTPU

! 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_ini_file, ! 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$m_delete_entry,
  tpu$m_last_time,

(continued on next page)
Example 8–1 (Cont.) Sample VAX BLISS Template for Callable DECTPU

```bliss
tpu$_nofileaccess, ! DECTPU status codes
tpu$_openin,
tpu$_inviocde,
tpu$_failure,
tpnu$_closein,
npnu$_closeout,
npnu$_readerr,
npnu$_writeerr,
ntpu$_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.
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
! (continued on next page)
Example 8–1 (Cont.) Sample VAX BLISS Template for Callable DECTPU

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)
!
<table>
<thead>
<tr>
<th>Item Code</th>
<th>Item Length</th>
<th>Buffer Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address (always 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

! 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),

  )

(continued on next page)
Example 8–1 (Cont.) Sample VAX BLISS Template for Callable DECTPU

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;

(continued on next page)
Example 8–1 (Cont.) Sample VAX BLISS Template for Callable DECTPU

```
RETURN .status;
END; ! End of routine TPU_IO
END ! End Module file_io_example
```

Example 8–2 shows normal DECTPU setup in Compaq Fortran.

Example 8–2 Normal DECTPU Setup in Compaq 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$SUCCESS !external error message
EXTERNAL LOC_PARSE !user supplied routine to
C call TPU$CLIPARSE and setup
C
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

(continued on next page)
Example 8–2 (Cont.) Normal DECTPU Setup in Compaq Fortran

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 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 ) )
    GOTO 9999
ENDIF

C Get the value of the option from the external literals. In Fortran you cannot use external literals directly so you must first get the value of the literal from its external location. Here we are getting the 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 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

TPU$CLEANUP wants the address of the flags as the parameter so 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
Example 8–2 (Cont.) Normal DECTPU Setup in Compaq Fortran

```fortran
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

 Declare the DECTPU functions used
 INTEGER*4 TPU$FILEIO
 INTEGER*4 TPU$CLIPARSE

 Declare this routine as external because it is never called directly and
 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'

 parse the command line and build the item list for TPU$INITIALIZE
 LOC_PARSE = TPU$CLIPARSE (EDIT_COMM, BPV , 0)
 RETURN
 END
```

Example 8–3 shows how to build a callback item list with Compaq Fortran.

Example 8–3 Building a Callback Item List with Compaq Fortran

```fortran
PROGRAM TEST_TPU

 IMPLICIT NONE

 Define the expected DECTPU return statuses
 EXTERNAL TPU$_SUCCESS
 EXTERNAL TPU$_QUITTING
 EXTERNAL TPU$_EXITING

 Declare the DECTPU routines and symbols used
 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

 Use LIB$MATCH_COND to compare condition codes
 INTEGER*4 LIB$MATCH_COND

 Declare the external callback routine
 EXTERNAL TPU_STARTUP ! the DECTPU set-up function
 INTEGER*4 TPU_STARTUP
 BPV(2) = 0 ! Set up a bound procedure value
```

(continued on next page)
Example 8–3 (Cont.) Building a Callback Item List with Compaq Fortran

C Declare the functions used for working with the condition handler
C
INTEGER*4 LIB$ESTABLISH
INTEGER*4 LIB$REVERT
C
Local Flags and Indices
C
INTEGER*4 CLEANUP_FLAG ! flag(s) for DECTPU cleanup
INTEGER*4 RET_STATUS
INTEGER*4 MATCH_STATUS
C
Initializations
C
RET_STATUS = 0
CLEANUP_FLAG = %LOC(TPU$M_DELETE_CONTEXT)
C
Establish the default DECTPU condition handler
C
CALL LIB$ESTABLISH(%REF(TPU$HANDLER))
C
Set up the bound procedure value for the initialization callback
C
BPV(1) = %LOC (TPU_STARTUP)
BPV(2) = 0
C
Call the DECTPU procedure for initialization
C
RET_STATUS = TPU$INITIALIZE(BPV)
IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
CALL LIB$SIGNAL (%VAL(RET_STATUS))
ENDIF
C
Execute the DECTPU initialization file
C
RET_STATUS = TPU$EXECUTE_INIFILE()
IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
CALL LIB$SIGNAL (%VAL(RET_STATUS))
ENDIF
C
Pass control to DECTPU
C
RET_STATUS = TPU$CONTROL()
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

(continued on next page)
Example 8–3 (Cont.) Building a Callback Item List with Compaq Fortran

```fortran
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
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)
```

(continued on next page)
Example 8–3 (Cont.) Building a Callback Item List with Compaq Fortran

C Define the name of the initialization section file
C
SECTION_NAME = ’TPU$SECTION’
C
Set up the required I/O routine. Use the DECTPU default.
C
BPV(1) = %LOC(TPU$FILEIO)  
BPV(2) = 0
C
Build the callback item list
C
Set up the edit session options
C
CALLBACK(1).ITEM_CODE = %LOC(TPU$K_OPTIONS)  
CALLBACK(1).BUFFER_ADDRESS = %LOC(OPTION_MASK)  
CALLBACK(1).BUFFER_LENGTH = 4  
CALLBACK(1).RETURN_ADDRESS = 0
C
Identify the section file to be used
C
CALLBACK(2).ITEM_CODE = %LOC(TPU$K_SECTIONFILE)  
CALLBACK(2).BUFFER_ADDRESS = %LOC(SECTION_NAME)  
CALLBACK(2).BUFFER_LENGTH = LEN(SECTION_NAME)  
CALLBACK(2).RETURN_ADDRESS = 0
C
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
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
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>
(continued on next page)
Example 8–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

** 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.

```c
#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_addr; /* 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 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 tpu$k_fileio, tpu$k_options, tpu$k_sectionfile,
        tpu$k_commandfile;

globalvalue tpu$m_display, tpu$m_section, tpu$m_command, tpu$m_create;

globalvalue 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 tpu$k_io, tpu$k_input, tpu$k_output;

globalvalue rms$_fnf; /* OpenVMS RMS File Not Found message code */
```

(continued on next page)
Example 8–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

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$execute_infile (); /* TPU routine to execute initialization code */
int tpu$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 */
        }

    (continued on next page)
Example 8–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

```c
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;
}
```

(continued on next page)
Example 8–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

/* 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;
}

(continued on next page)
Example 8–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

```c
/*
   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.
DEC Text Processing Utility (DECTPU) Routines
8.5 Creating and Calling a USER Routine

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.

```c
/* 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$sget1_dd(),
    vex$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 occurred 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$CALLUSERSYS$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.

```fortran
! 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;
```
8.6 Accessing the USER Routine from DECTPU

```plaintext
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:

```plaintext
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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_DELETE_JOURNAL</td>
<td>Closes and deletes the journal file if it is open.</td>
</tr>
<tr>
<td>TPU$M_DELETE_EXITH</td>
<td>Deletes the DECTPU exit handler.</td>
</tr>
<tr>
<td>TPU$M_DELETE_BUFFERS</td>
<td>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.</td>
</tr>
</tbody>
</table>

1The 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.
## DEC Text Processing Utility (DECTPU) Routines

### TPU$CLEANUP

<table>
<thead>
<tr>
<th>Flag(^1)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_DELETE_WINDOWS</td>
<td>Deletes all windows. If this is not the last time you are calling DECTPU, then all variables referring to these data structures are reset, as if by the built-in procedure DELETE.</td>
</tr>
<tr>
<td>TPU$M_DELETE_CACHE</td>
<td>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.</td>
</tr>
<tr>
<td>TPU$M_PRUNE_CACHE</td>
<td>Frees up any virtual file manager caches that have no pages allocated to buffers. This frees up any caches that may have been created during the session but are no longer needed.</td>
</tr>
<tr>
<td>TPU$M_EXECUTE_FILE</td>
<td>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.</td>
</tr>
<tr>
<td>TPU$M_EXECUTE_PROC</td>
<td>Looks up TPU$INIT_PROCEDURE and executes it the next time TPU$EXECUTE_INIFILE is called.</td>
</tr>
<tr>
<td>TPU$M_DELETE_CONTEXT</td>
<td>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.</td>
</tr>
<tr>
<td>TPU$M_RESET_TERMINAL</td>
<td>Resets the terminal to the state it was in upon entry to DECTPU. The terminal mailbox and all windows are deleted. If the terminal is reset, then it is reinitialized the next time TPU$INITIALIZE is called.</td>
</tr>
<tr>
<td>TPU$M_KILL PROCESSES</td>
<td>Deletes all subprocesses created during the session.</td>
</tr>
</tbody>
</table>

\(^1\)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.
### DEC Text Processing Utility (DECTPU) Routines

#### TPU$CLEANUP

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_CLOSE_SECTION²</td>
<td>Closes the section file and releases the associated memory. All buffers, windows, and processes are deleted. The cache is purged and the flags are set for reexecution of the initialization file and initialization procedure. If the section is closed and if the option bit indicates the presence of the SECTION qualifier, then the next call to TPU$INITIALIZE attempts a new restore operation.</td>
</tr>
<tr>
<td>TPU$M_DELETE_OTHERS</td>
<td>Deletes all miscellaneous preallocated data structures. Memory for these data structures is reallocated the next time TPU$INITIALIZE is called.</td>
</tr>
<tr>
<td>TPU$M_LAST_TIME</td>
<td>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.</td>
</tr>
</tbody>
</table>

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$V_DELETE_EXITH
DEC Text Processing Utility (DECTPU) Routines
TPU$CLEANUP

- 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**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$_SUCCESS</td>
<td>Normal successful completion.</td>
</tr>
</tbody>
</table>
TPU$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.

Arguments

None.

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 journaled" 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 journaling 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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$_\text{EXITING}$</td>
<td>A result of EXIT (when the default condition handler is established).</td>
</tr>
<tr>
<td>TPU$_\text{NONANSICRT}$</td>
<td>A result of operation termination — results when you call DECTPU with TPU$\text{DISPLAYFILE}$ set to nodisplay and you attempt to execute screen-oriented commands.</td>
</tr>
<tr>
<td>TPU$_\text{QUITTING}$</td>
<td>A result of QUIT (when the default condition handler is established).</td>
</tr>
<tr>
<td>TPU$_\text{RECOVERFAIL}$</td>
<td>A recovery operation was terminated abnormally.</td>
</tr>
</tbody>
</table>
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.

Arguments

None.

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$_SUCCESS</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>TPU$_COMPILEFAIL</td>
<td>The compilation of the initialization file was unsuccessful.</td>
</tr>
<tr>
<td>TPU$_EXECUTEFAIL</td>
<td>The execution of the statements in the initialization file was unsuccessful.</td>
</tr>
<tr>
<td>TPU$_EXITING</td>
<td>A result of EXIT. If the default condition handler is being used, the session is terminated.</td>
</tr>
<tr>
<td>TPU$_FAILURE</td>
<td>General code for all other errors.</td>
</tr>
<tr>
<td>TPU$_QUITTING</td>
<td>A result of QUIT. If the default condition handler is being used, the session is terminated.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>File Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFM</td>
</tr>
<tr>
<td>Allocation</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Length</td>
</tr>
<tr>
<td>Address of name</td>
</tr>
</tbody>
</table>

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.
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$KRELATEDFILE 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM,STMLF,STMCR</td>
<td>0,BLK,CR,BLK+CR</td>
</tr>
<tr>
<td>VAR</td>
<td>0,BLK,FTN,CR,BLK+FTN,BLK+CR</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Flag Bit 1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_NODE</td>
<td>Returns the node component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DEV</td>
<td>Returns the device component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DIR</td>
<td>Returns the directory component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_NAME</td>
<td>Returns the name component of the file specification.</td>
</tr>
</tbody>
</table>

1 TPU$M ... indicates a mask. There is a corresponding value for each mask in the form TPU$V ....
### DEC Text Processing Utility (DECTPU) Routines

#### TPU$FILE_PARSE

<table>
<thead>
<tr>
<th>Flag Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_TYPE</td>
<td>Returns the type component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_VER</td>
<td>Returns the version component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_HEAD</td>
<td>Returns NODE, DEVICE and DIRECTORY components of the file specification. If the TPU$M_NODE, TPU$M_DEV or TPU$M_DIR bits are set while TPU$M_HEAD is set, the routine signals the error TPU$_INCKWDCOM and returns control to the caller.</td>
</tr>
<tr>
<td>TPU$M_TAIL</td>
<td>Returns NAME, TYPE and VERSION components of the file specification. If the TPU$M_NAME, TPU$M_TYPE or TPU$M_VER bits are set while TPU$M_TAIL is set, the routine signals the error TPU$_INCKWDCOM and returns control to the caller.</td>
</tr>
</tbody>
</table>

1TPU$M . . . indicates a mask. There is a corresponding value for each mask in the form TPU$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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_NODE</td>
<td>Returns the node component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DEV</td>
<td>Returns the device component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DIR</td>
<td>Returns the directory component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_NAME</td>
<td>Returns the name component of the file specification.</td>
</tr>
</tbody>
</table>

1TPU$M ... indicates a mask. There is a corresponding value for each mask in the form TPU$V ....
DEC Text Processing Utility (DECTPU) Routines
TPU$FILE_SEARCH

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_TYPE</td>
<td>Returns the type component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_VER</td>
<td>Returns the version component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_REPARSE</td>
<td>Reparses the file specification before processing. This is intended to be used to reset the file search.</td>
</tr>
<tr>
<td>TPU$M_HEAD</td>
<td>Returns NODE, DEVICE, and DIRECTORY components of the file specification. If the TPU$M_NODE, TPU$M_DEV or TPU$M_DIR bits are set while TPU$M_HEAD is set, the routine will signal the error TPU$_INCKWDCOM and return.</td>
</tr>
<tr>
<td>TPU$M_TAIL</td>
<td>Returns NAME, TYPE and VERSION components of the file specification. If the TPU$M_NAME, TPU$M_TYPE or TPU$M_VER bits are set while TPU$M_TAIL is set, the routine will signal the error TPU$_INCKWDCOM and return.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$_SUCCESS</td>
<td>Normal successful completion. If the return string contains a null string, the final match operation was detected.</td>
</tr>
<tr>
<td>TPU$_INCKWDCOM</td>
<td>The flags argument had an illegal combination of values.</td>
</tr>
<tr>
<td>TPU$_PARSEFAIL</td>
<td>The requested repeat parse failed.</td>
</tr>
<tr>
<td>TPU$_SEARCHFAIL</td>
<td>An error occurred during the search operation.</td>
</tr>
</tbody>
</table>
TPU$HANDLER—DECTPU Condition 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 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 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 resignedaled.

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 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)
amccess: 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.
DEC Text Processing Utility (DECTPU) Routines

TPU$INITIALIZE

<table>
<thead>
<tr>
<th>Item code</th>
<th>Buffer address</th>
<th>Buffer length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$INITIALIZE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZK</td>
<td>4044</td>
<td>GE</td>
</tr>
</tbody>
</table>

The return address in an item descriptor is usually 0.

The following item codes are available:

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$OPTIONS</td>
<td>Enables the command qualifiers. The bits in the bit mask specified by the buffer address field correspond to the various DECTPU command qualifiers.</td>
</tr>
<tr>
<td>TPU$JOURNALFILE</td>
<td>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.</td>
</tr>
<tr>
<td>TPU$SECTIONFILE</td>
<td>Passes the string that is the name of the binary initialization file (section file) to be mapped in. The buffer length field is the length of the string, and the buffer address field is the address of the string. If the TPU$V SECTION bit is set, this item code must be specified.</td>
</tr>
<tr>
<td>TPU$OUTPUTFILE</td>
<td>Passes the string specified with the /OUTPUT qualifier. The buffer length field is the length of the string, and the buffer address field specifies the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “OUTPUT_FILE”). The string can be a null string.</td>
</tr>
<tr>
<td>TPU$DISPLAYFILE</td>
<td>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.</td>
</tr>
<tr>
<td>TPU$COMMANDFILE</td>
<td>Passes the string specified with the /COMMAND qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, “COMMAND_FILE”). The string can be a null string.</td>
</tr>
<tr>
<td>TPU$FILENAME</td>
<td>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.</td>
</tr>
</tbody>
</table>
### DEC Text Processing Utility (DECTPU) Routines

#### TPU$INITIALIZE

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$_OTHER_FILENAMES</td>
<td>Passes a string that contains the name of an input file that follows the first input file on the command line. The buffer length field specifies the length of this string, and the buffer address field specifies its address. Each additional file specified on the command line requires its own TPU$_OTHER_FILENAMES item entry. These strings are returned by the GET_INFO (COMMAND_LINE,&quot;NEXT_FILE_NAME&quot;) built-in procedure in the order they appear in the item list. This item code must appear after the TPU$_FILENAME item in the item list.</td>
</tr>
<tr>
<td>TPU$_FILEIO</td>
<td>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$_FILEIO, the utility routine provided by DECTPU for handling file operations. The buffer address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that DECTPU loads into R1 before calling the routine.</td>
</tr>
<tr>
<td>TPU$_CALLUSER</td>
<td>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.</td>
</tr>
<tr>
<td>TPU$_INIT_FILE</td>
<td>Passes the string specified with the /INITIALIZATION qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE,&quot;INIT_FILE&quot;).</td>
</tr>
<tr>
<td>TPU$_START_LINE</td>
<td>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,&quot;LINE&quot;). 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.</td>
</tr>
<tr>
<td>TPU$_START_CHAR</td>
<td>Passes the starting column position for the edit. The buffer address field contains the second of the two integer values you specified as part of the /START_POSITION command qualifier. The value is available using the built-in procedure GET_INFO (COMMAND_LINE,&quot;CHARACTER&quot;). Usually an initialization procedure uses this information to set the starting position in the main editing buffer. The first column on a line to character 1.</td>
</tr>
<tr>
<td>TPU$_CHARACTERSET</td>
<td>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 &quot;DEC_MCS&quot; (the default value), &quot;ISO_LATIN1&quot;, and &quot;GENERAL&quot;. If the application tries to pass any other string, the routine signals an error and passes the default string (DEC_MCS).</td>
</tr>
</tbody>
</table>
### DEC Text Processing Utility (DECTPU) Routines
#### TPUS$INITIALIZE

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPUS$WORKFILE</td>
<td>Passes the string specified with the /WORK qualifier. The buffer length field specifies the string length and the buffer address specifies the string address. This string is available with GET_INFO (COMMAND_LINE, &quot;WORK_FILE&quot;).</td>
</tr>
<tr>
<td>TPUS$CHAIN</td>
<td>Passes the address of the next item list to the process specified by the buffer address field.</td>
</tr>
<tr>
<td>TPUS$ENDLIST</td>
<td>Signals the end of the item list.</td>
</tr>
<tr>
<td>TPUS$PARENT_WIDGET</td>
<td>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.</td>
</tr>
<tr>
<td>TPUS$APPLICATION_CONTEXT</td>
<td>Passes the application context to use with the TPUS$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.</td>
</tr>
<tr>
<td>TPUS$DEFAULTSFILE</td>
<td>Specifies which file DECTPU uses to initialize the X defaults database. The buffer length field specifies the string length and the buffer address field specifies the string address. This item code is only valid when using the DECwindows interface.</td>
</tr>
<tr>
<td>TPUS$CTRL_C_ROUTINE</td>
<td>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 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.</td>
</tr>
<tr>
<td>TPUS$DEBUGFILE</td>
<td>Passes the string specified with the /DEBUG command qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string.</td>
</tr>
<tr>
<td>TPUS$FILE_SEARCH</td>
<td>Passes the bound procedure value of a routine to be used to replace the TPUS$FILE_SEARCH routine which is called when the built-in procedure FILE_SEARCH is called from TPU code. See the description of the TPUS$FILE_SEARCH and the user routine FILE_SEARCH for more information.</td>
</tr>
<tr>
<td>TPUS$FILE_PARSE</td>
<td>Passes the bound procedure value of a routine to be used to replace the TPUS$FILE_PARSE routine which is called when the built-in procedure FILE_PARSE is called from TPU code. See the description of the TPUS$FILE_PARSE and the user routine FILE_PARSE for more information.</td>
</tr>
</tbody>
</table>

Table 8-1 lists the bits and corresponding masks enabled by the item code TPUS$OPTIONS and shows how each bit affects TPUS$INITIALIZE operation. Several bits in the TPUS$OPTIONS mask require additional item code entries in the item list. An example of this is TPUS$M_COMMAND which requires a TPUS$COMMANDFILE entry in the item list.
## DEC Text Processing Utility (DECTPU) Routines

### TPUS$INITIALIZE

**Table 8-1 Valid Masks for the TPUS$K_OPTIONS Item Code**

<table>
<thead>
<tr>
<th>Mask1</th>
<th>GET_INFO Request String2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPUS$M_COMMAND</td>
<td>COMMAND</td>
<td>If DECTPU senses the presence of the TPUS$_COMMANDFILE item, it tries to read, compile and execute the unbound TPU code.</td>
</tr>
<tr>
<td>TPUS$M_COMMAND_DFLTED</td>
<td>Not applicable</td>
<td>Specifies that DECTPU should use the default command file name of TPUS$COMMAND.TPU when reading in the command file. No error is reported if the default command file is not found. TPUS$INITIALIZE fails when the TPUS$M_COMMAND_DFLTED bit is set to 0 and no file is specified in the item list.</td>
</tr>
<tr>
<td>TPUS$M_CREATE</td>
<td>CREATE</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
<tr>
<td>TPUS$M_DEBUG</td>
<td>Not applicable</td>
<td>If DECTPU senses the presence of the TPUS$_DEBUGFILE item, it tries to read the file, and then proceeds to compile and execute its contents as TPU statements.</td>
</tr>
<tr>
<td>TPUS$M_DEFAULTS</td>
<td>Not applicable</td>
<td>If DECTPU senses the presence of the TPUS$_DEFAULTFILE item, it uses the specified DECwindows X resource file to initialize the DECwindows X resource database.</td>
</tr>
<tr>
<td>TPUS$M_DISPLAY</td>
<td>DISPLAY</td>
<td>If DECTPU senses the presence of the TPUS$_DISPLAYFILE item, it tries to image activate the specified image as its screen manager. When the bit is 0, DECTPU uses SYS$OUTPUT for display and only the READ_LINE built-in procedure may be used for input.</td>
</tr>
<tr>
<td>TPUS$M_INIT</td>
<td>INITIALIZATION</td>
<td>If DECTPU senses the presence of the TPUS$_INIT_FILE item, it returns the specified string through the built-in procedure GET_INFO (COMMAND_LINE, &quot;INITIALIZATION_FILE&quot;). Processing of the initialization file is left to the application.</td>
</tr>
<tr>
<td>TPUS$M_JOURNAL</td>
<td>JOURNAL</td>
<td>If DECTPU senses the presence of the TPUS$_JOURNALFILE item, it outputs the keystrokes entered during the editing session to the specified file.</td>
</tr>
</tbody>
</table>

**Note:** Compaq recommends the use of buffer change journaling in new applications.

---

1. The prefix can be TPUS$M_ or TPUS$V_. TPUS$M_ denotes a mask corresponding to the specific field in which the bit is set. TPUS$V_ is a bit number.
2. Most bits in the mask have a corresponding GET_INFO (COMMAND_LINE) request string.
DEC Text Processing Utility (DECTPU) Routines

<table>
<thead>
<tr>
<th>Mask</th>
<th>GET_INFO Request String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_MODIFY</td>
<td>MODIFY</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
<tr>
<td>TPU$M_NODEFAULTS</td>
<td>Not applicable</td>
<td>DECTPU initializes the DECwindows X resource database only with resource files that the DECwindows toolkit routine XtAppIInitialize loads into the database.</td>
</tr>
<tr>
<td>TPU$M_NOMODIFY</td>
<td>NOMODIFY</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
<tr>
<td>TPU$M_OUTPUT</td>
<td>OUTPUT</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
<tr>
<td>TPU$M_READ</td>
<td>READ_ONLY</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
<tr>
<td>TPU$M_RECOVER</td>
<td>RECOVER</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
<tr>
<td>TPU$M_SECTION</td>
<td>SECTION</td>
<td>If DECTPU senses the presence of the TPU$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.</td>
</tr>
<tr>
<td>TPU$M_SEC_LNM_MODE</td>
<td>Not applicable</td>
<td>If DECTPU senses the presence of the TPU$SEC_LNM_MODE item, it looks only at executive mode logical names when attempting to read in a section file.</td>
</tr>
<tr>
<td>TPU$M_WORK</td>
<td>WORK</td>
<td>If DECTPU senses the presence of the TPU$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 SYSLOGIN:.TPU$WORK.</td>
</tr>
<tr>
<td>TPU$M_WRITE</td>
<td>WRITE</td>
<td>The behavior of DECTPU is not affected by this bit. Its interpretation is left to the application layered on DECTPU.</td>
</tr>
</tbody>
</table>

1The 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.

2Most 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.
DEC Text Processing Utility (DECTPU) Routines

TPU$INITIALIZE

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.
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$SPECIFYASYNC_ACTION—Register an Asynchronous Action

The TPU$SPECIFYASYNC_ACTION routine allows applications using the DECTPU full callable interface to register asynchronous actions with DECTPU.

Format

TPU$SPECIFYASYNC_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$TRIGGERASYNC_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$TRIGGERASYNC_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$SPECIFYASYNC_ACTION routine, along with TPU$TRIGGERASYNC_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 journaling is enabled.
## Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$_SUCCESS</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>TPU$_COMPILEFAIL</td>
<td>The code specified in <code>tpu_statement</code> did not compile successfully.</td>
</tr>
<tr>
<td>TPU$_INVPARM</td>
<td>An invalid parameter was passed.</td>
</tr>
<tr>
<td>TPU$_JNLACTIVE</td>
<td>Keystroke journaling is active. This routine requires that either journaling be turned off or that buffer change journaling be used.</td>
</tr>
</tbody>
</table>
TTPU$TPU—Invoke DECTPU

The TTPU$TPU routine invokes DECTPU and is equivalent to the DCL command EDIT/TPU.

Format

TTPU$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 TTPU$CLOSE_SECTION. This feature lets you make multiple calls to TTPU$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 TTPU$TPU. This is because TTPU$TPU destroys all parse information obtained and stored before TTPU$TPU was called.

Condition Values Returned

This routine returns any condition value returned by TTPU$INITIALIZE, TTPU$EXECUTE_INIFILE, TTPU$CONTROL, and TTPU$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 (unsigned)
access: write only
mechanism: by reference

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed 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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_NODE</td>
<td>Requests for the node component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DEV</td>
<td>Requests for the device component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DIR</td>
<td>Requests for the directory component of the file specification.</td>
</tr>
</tbody>
</table>

1TPU$M . . . indicates a mask. There is a corresponding value for each mask in the form TPU$V . . . .
DEC Text Processing Utility (DECT PU) Routines

FILE_PARSE

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_NAME</td>
<td>Requests for the name component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_TYPE</td>
<td>Requests for the type component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_VER</td>
<td>Requests for the version component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_HEAD</td>
<td>Requests for the NODE, DEVICE, and DIRECTORY components of the file specification.</td>
</tr>
<tr>
<td>TPU$M_TAIL</td>
<td>Requests for NAME, TYPE, and VERSION components of the file specification.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_NODE</td>
<td>Requests for the node component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_DEV</td>
<td>Requests for the device component of the file specification.</td>
</tr>
</tbody>
</table>

1TPU$M . . . indicates a mask. There is a corresponding value for each mask in the form TPU$V . . . .
DEC Text Processing Utility (DECTPU) Routines
FILE_SEARCH

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPU$M_DIR</td>
<td>Requests for the directory component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_NAME</td>
<td>Requests for the name component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_TYPE</td>
<td>Requests for the type component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_VER</td>
<td>Requests for the version component of the file specification.</td>
</tr>
<tr>
<td>TPU$M_REPARSE</td>
<td>Reparses the file specification before processing.</td>
</tr>
<tr>
<td></td>
<td>This is intended as a way to restart the file search.</td>
</tr>
<tr>
<td></td>
<td>This flag will automatically be set by DECTPU</td>
</tr>
<tr>
<td></td>
<td>if on a previous call to the FILE_SEARCH user routine the result-string has zero length or the routine returns a odd (noneven) status.</td>
</tr>
<tr>
<td>TPU$M_HEAD</td>
<td>Requests for the NODE, DEVICE, and DIRECTORY components of the file specification.</td>
</tr>
<tr>
<td>TPU$M_TAIL</td>
<td>Requests for the NAME, TYPE, and VERSION component of the file specification.</td>
</tr>
</tbody>
</table>

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 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 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 OpenVMS Programming Interfaces: Calling a System Routine.

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 on 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
Access: longword (unsigned)
access: write only
mechanism: by value

Longword condition value.

Arguments

integer
OpenVMS usage: longword_unsigned
Access: 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.
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.

9.1 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.

9.2 Using the EDT Routines: An Example

Example 9–1 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 9–1  Using the EDT Routines in a VAX BASIC Program

100 EXTERNAL INTEGER EDT$FILEIO
200 EXTERNAL INTEGER EDT$WORKIO
250 EXTERNAL INTEGER AXLATE
300 EXTERNAL INTEGER FUNCTION EDT$EDIT
400 DECLARE INTEGER RESULT
450 DIM INTEGER PASSFILE(1%)
460 DIM INTEGER PASSWORK(1%)
465 DIM INTEGER PASSXLATE(1%)
470 PASSFILE(0%) = LOC(EDT$FILEIO)
480 PASSWORK(0%) = LOC(EDT$WORKIO)
485 PASSXLATE(0%) = LOC(AXLATE)

(continued on next page)
Example 9–1 (Cont.) Using the EDT Routines in a VAX BASIC Program

500  RESULT = EDT$EDIT('FILE.BAS','','EDTINI','',0%,
        PASSFILE(0%) BY REF, PASSWORK(0%) BY REF,
        PASSXLATE(0%) BY REF) ③
600  IF (RESULT AND 1%) = 0%
    THEN
        PRINT "SOMETHING WRONG"
        CALL LIB$STOP(RESULT BY VALUE)
900  PRINT "EVERYTHING O.K."
1000 END

① The external entry points EDT$FILEIO, EDT$WORKIO, and AXLATE are defined so they can be passed to callable EDT.

② Arrays are used to construct the two-longword structure needed for data type BPV.

③ 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.

④ 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.

⑤ 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.

9.3 EDT Routines

This section describes the individual EDT routines.
**EDT$EDIT—Edit a File**

The EDT$EDIT routine invokes the EDT editor.

**Format**

```
```

**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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDT$M_RECOVER</td>
<td>If set, bit &lt;0&gt; causes EDT to read the journal file and execute the commands in it, except for the EXIT or QUIT commands, which are ignored. After the journal file commands are processed, editing continues normally. If bit &lt;0&gt; is set, the FILEIO routine is asked to open the journal file for both input and output; otherwise FILEIO is asked only to open the journal file for output. Bit &lt;0&gt; corresponds to the /RECOVER qualifier on the EDT command line.</td>
</tr>
<tr>
<td>EDT$M_COMMAND</td>
<td>If set, bit &lt;1&gt; causes EDT to signal if the startup command file cannot be opened. When bit &lt;1&gt; 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 &lt;1&gt; corresponds to the /COMMAND qualifier on the EDT command line. If EDT$M_NOCOMMAND (bit &lt;4&gt;) is set, bit &lt;1&gt; is overridden because bit &lt;4&gt; prevents EDT from trying to open a command file.</td>
</tr>
<tr>
<td>EDT$M_NOJOURNAL</td>
<td>If set, bit &lt;2&gt; prevents EDT from opening the journal file. Bit &lt;2&gt; corresponds to the /NOJOURNAL or /READ_ONLY qualifier on the EDT command line.</td>
</tr>
</tbody>
</table>
Symbol | Description
---|---
EDT$M_NOOUTPUT | If set, bit &lt;3&gt; prevents EDT from using the input file name as the default output file name. Bit &lt;3&gt; corresponds to the /NOOUTPUT or /READ_ONLY qualifier on the EDT command line.
EDT$M_NOCOMMAND | If set, bit &lt;4&gt; prevents EDT from opening a startup command file. Bit &lt;4&gt; corresponds to the /NOCOMMAND qualifier on the EDT command line.
EDT$M_NOCREATE | If set, bit &lt;5&gt; causes EDT to return to the caller if the input file is not found. The status returned is the error code EDT$_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$FILEIO for this argument, set the second longword to 0. You can pass a 0 for this argument, and EDT will set up EDT$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.
**EDT Routines**

**EDT$EDIT**

```plaintext
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

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:

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDT$K_OPEN_INPUT</td>
<td>The record argument names a file to be opened for input. The rhb argument is the default file name.</td>
</tr>
<tr>
<td>EDT$K_OPEN_OUTPUT_SEQ</td>
<td>The record argument names a file to be opened for output as a sequenced file. The rhb argument is the default file name.</td>
</tr>
<tr>
<td>EDT$K_OPEN_OUTPUT_NOSEQ</td>
<td>The record argument names a file to be opened for output. The rhb argument is the default file name.</td>
</tr>
</tbody>
</table>
Function Code        Description

EDT$K_OPEN_IN_OUT    The *record* argument names a file to be opened for both input and output. The *rhh* argument is the default file name.

EDT$K_GET            The *record* argument is to be filled with data from the next record of the file. If the file has record prefixes, *rhh* is filled with the record prefix. If the file has no record prefixes, *rhh* is not written. When you attempt to read past the end of file, *status* is set to RMS$_EOF.

EDT$K_PUT            The data in the *record* argument is to be written to the file as its next record. If the file has record prefixes, the record prefix is taken from the *rhh* argument. For a file opened for both input and output, EDT$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 *rhh* arguments are not used in the call.

EDT$K_CLOSE          The file is to be closed. The *record* and *rhh* 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.

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 Routines
FILEIO

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDT$K_JOURNAL_FILE</td>
<td>The journal file. If bit 0 of the options is set, it is opened for both input and output and is read completely. Otherwise, it is opened for output only. After it is read or opened for output only, it is used for writing. On a successful termination of the editing session, the journal file is closed and deleted. EXIT/SAVE and QUIT/SAVE close the journal file without deleting it.</td>
</tr>
<tr>
<td>EDT$K_OUTPUT_FILE</td>
<td>The primary output file. It is not opened until you enter the EXIT command.</td>
</tr>
<tr>
<td>EDT$K_WRITE_FILE</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

rhh
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 rhh argument is used. When the code argument starts with EDT$K_OPEN, the rhh argument is the default file name. When the code is EDT$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:

```plaintext
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

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:

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDT$K_OPEN_IN_OUT</td>
<td>Open the work file for both input and output. Neither the record nor recordno argument is used.</td>
</tr>
<tr>
<td>EDT$K_GET</td>
<td>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.</td>
</tr>
<tr>
<td>EDT$K_PUT</td>
<td>Write a record. The recordno argument is the number of the record to be written. The record argument tells the location of the record to be written.</td>
</tr>
<tr>
<td>EDT$K_CLOSE_DEL</td>
<td>Close the work file. After a successful close, the file is deleted. Neither the record nor recordno argument is used.</td>
</tr>
</tbody>
</table>
**EDT Routines**

**WORKIO**

- **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**

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.

---

\(^1\) This manual has been archived but is available on the OpenVMS Documentation CD-ROM.
Condition Value Returned

SS$_NORMAL Normal successful completion.
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.

10.1 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$CREATE routine is the one most likely to be called from a high-level language. It creates a file from an FDL specification and then closes the file. The following Compaq Fortran program segment creates an empty data file named INCOME93.DAT using the file characteristics specified by the FDL file INCOME.FDL. The STATEMENT variable contains the number of the last FDL statement processed by FDL$CREATE; this argument is useful for debugging an FDL file.

```fortran
INTEGER STATEMENT
INTEGER STATUS,
FDL$CREATE
STATUS = FDL$CREATE ('INCOME.FDL',
  'INCOME93.DAT',
  STATEMENT,
  )
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

...
10.1 Introduction to FDL Routines

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$GENERATE routine produces an FDL specification by interpreting a set of RMS control blocks in an existing data file. It then writes the FDL specification either to an FDL file or to a character string. If your programming language does not provide language statements that access RMS control blocks (for example, Compaq Fortran), you must use FDL$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.

10.2 Using the FDL Routines: Examples

This section provides examples that demonstrate the use of the FDL routines in various programming scenarios.

- Example 10–1 shows how to use the FDL$CREATE routine in a Fortran program.
- Example 10–2 shows how to use the FDL$PARSE and FDL$RELEASE routines in a C program.
- Example 10–3 shows a Compaq Pascal program that uses the FDL$PARSE routine to fill in the RMS control blocks in a data file. The program then uses the FDL$GENERATE routine to create an FDL file using the information in the control blocks.
Example 10–1 Using FDL$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/
1 = 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 2 shows how to use the FDL$PARSE and FDL$RELEASE routines in a C program.

Example 10–2 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;

  (continued on next page)
Example 10–2 (Cont.) Using FDL$PARSE and FDL$RELEASE in a C Program

```c
/*
** 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 record 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$RELEASE ( &fab_ptr, &rab_ptr );
```
Example 10–3 shows a Compaq Pascal program that uses the FDL$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 10–3  Using FDL$PARSE and FDL$GENERATE in a Compaq 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;
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;

(continued on next page)
10.2 Using the FDL Routines: Examples

Example 10–3 (Cont.) Using FDL$PARSE and FDL$GENERATE in a Compaq Pascal Program

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.

10.3 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

```
```

Returns

OpenVMS usage: cond_value

<table>
<thead>
<tr>
<th>type</th>
<th>longword (unsigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>write only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by value</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>type</th>
<th>character-coded text string</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by descriptor—fixed-length string descriptor</td>
</tr>
</tbody>
</table>

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
File Definition Language (FDL) Routines
FDL$CREATE

- 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.
File Definition Language (FDL) Routines
FDLS$CREATE

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

Flags (or masks) that control how the \texttt{fdl\_desc} argument is interpreted and how errors are signaled. The \texttt{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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDL$V_FDL_STRING</td>
<td>Interprets the \texttt{fdl_desc} argument as an FDL specification in string form. By default, the \texttt{fdl_desc} argument is interpreted as the file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_LONG_NAMES</td>
<td>Returns the \texttt{RESULT_NAME} using the long result name from a long name access block (NAML). By default, the \texttt{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.</td>
</tr>
<tr>
<td>FDL$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

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

\texttt{stmnt\_num}

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

FDL statement number. The \texttt{stmnt\_num} argument is the address of a longword that receives the FDL statement number. If the routine finishes successfully, the \texttt{stmnt\_num} argument is the number of statements in the FDL specification. If the routine does not finish successfully, the \texttt{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.”

\texttt{retlen}

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

Number of characters returned in the \texttt{result\_name} argument. The \texttt{retlen} argument is the address of a longword that receives this number.

\texttt{sts}

OpenVMS usage: longword unsigned
type: longword (unsigned)
access: write only
mechanism: by reference
RMS status value FAB$L_STS. The \texttt{sts} argument is the address of a longword that receives the status value FAB$L_STS from the $CREATE system service.

\texttt{stv}
OpenVMS usage: longword unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

RMS status value FAB$L_STV. The \texttt{stv} argument is the address of a longword that receives the status value FAB$L_STV from the $CREATE system service.

\texttt{default_fdl_desc}
OpenVMS usage: char_string
type: character-coded text string
access: read only
mechanism: by descriptor—fixed-length string descriptor

The \texttt{default_fdl_desc} argument is one of the following:

\begin{itemize}
  \item A character string descriptor pointing to a file containing the default FDL specification to be parsed
  \item A character string containing the actual default FDL specification
\end{itemize}

See the description of the \texttt{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 \texttt{fdl_desc} argument.

You can code the FDL defaults directly into your program, typically with an FDL specification in string form.

\section*{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.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Source of FDL Specification} & \textbf{Advantages} & \textbf{Disadvantages} \\
\hline
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. Faster access. & Program must be recoded to change FDL specification. \\
\hline
\end{tabular}
\end{table}

If the FDL specification is relatively simple and is not going to change, put the FDL specification in a character string as the \texttt{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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>FDL$_ABKW</td>
<td>Ambiguous keyword in statement number &lt;CRLF&gt; reference-text.</td>
</tr>
<tr>
<td>FDL$_ABPRIKW</td>
<td>Ambiguous primary keyword in statement number &lt;CRLF&gt; reference-text.</td>
</tr>
<tr>
<td>FDL$_BADLOGIC</td>
<td>Internal logic error detected.</td>
</tr>
<tr>
<td>FDL$_CLOSEIN</td>
<td>Error closing filename as input.</td>
</tr>
<tr>
<td>FDL$_CLOSEOUT</td>
<td>Error closing filename as output.</td>
</tr>
<tr>
<td>FDL$_CREATE</td>
<td>Error creating filename.</td>
</tr>
<tr>
<td>FDL$_CREATED</td>
<td>Filename created.</td>
</tr>
<tr>
<td>FDL$_CREATED_STM</td>
<td>Filename created in stream format.</td>
</tr>
<tr>
<td>FDL$_FDLERROR</td>
<td>Error parsing FDL file.</td>
</tr>
<tr>
<td>FDL$_ILL_ARG</td>
<td>Wrong number of arguments.</td>
</tr>
<tr>
<td>FDL$_INSVIREM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>FDL$_INVBLK</td>
<td>Invalid RMS control block at virtual address 'hex-offset'.</td>
</tr>
<tr>
<td>FDL$_MULPRI</td>
<td>Multiple primary definition in statement number.</td>
</tr>
<tr>
<td>FDL$_OPENFDL</td>
<td>Error opening filename.</td>
</tr>
<tr>
<td>FDL$_OPENIN</td>
<td>Error opening filename as input.</td>
</tr>
<tr>
<td>FDL$_OPENOUT</td>
<td>Error opening filename as output.</td>
</tr>
<tr>
<td>FDL$_OUTORDER</td>
<td>Key or area primary defined out of order in statement number.</td>
</tr>
<tr>
<td>FDL$_READERR</td>
<td>Error reading filename.</td>
</tr>
<tr>
<td>FDL$_RFLOC</td>
<td>Unable to locate related file.</td>
</tr>
<tr>
<td>FDL$_SYNTAX</td>
<td>Syntax error in statement number reference-text.</td>
</tr>
<tr>
<td>FDL$_UNPRIKW</td>
<td>Unrecognized primary keyword in statement number &lt;CRLF&gt; reference-text.</td>
</tr>
<tr>
<td>FDL$_UNQUAKW</td>
<td>Unrecognized qualifier keyword in statement number &lt;CRLF&gt; reference-text.</td>
</tr>
<tr>
<td>FDL$_UNSECKW</td>
<td>Unrecognized secondary keyword in statement number &lt;CRLF&gt; reference-text.</td>
</tr>
<tr>
<td>FDL$_VALERR</td>
<td>Specified value is out of legal range.</td>
</tr>
<tr>
<td>FDL$_VALPRI</td>
<td>Value required on primary in statement number.</td>
</tr>
<tr>
<td>FDL$_WARNING</td>
<td>Parsed with warnings.</td>
</tr>
<tr>
<td>FDL$_WRITEERR</td>
<td>Error writing filename.</td>
</tr>
<tr>
<td>RMS$_ACT</td>
<td>File activity precludes operation.</td>
</tr>
<tr>
<td>RMS$_CRE</td>
<td>Ancillary control process (ACP) file create failed.</td>
</tr>
<tr>
<td>RMS$_CREATED</td>
<td>File was created, not opened.</td>
</tr>
<tr>
<td>RMS$_DNF</td>
<td>Directory not found.</td>
</tr>
</tbody>
</table>
### File Definition Language (FDL) Routines

**FDL$CREATE**

- **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$_WLK**: Device currently write locked.
FDLS$GENERATE—Generate an FDL Specification

The FDLS$GENERATE routine produces an FDL specification and writes it to either an FDL file or a character string.

Format

FDLS$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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDL$V_FDL_STRING</td>
<td>Interprets the fdl_str_dst argument as an FDL specification in string form. By default, the fdl_str_dst argument is interpreted as the file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_FULL_OUTPUT</td>
<td>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).</td>
</tr>
<tr>
<td>FDL$V_LONG_NAMES</td>
<td>Returns the FDL_FILE_RESNAME using the long result name from a long name access block (NAML). By default, the FDL_FILE_RESNAME is returned from the short fields of a name access block (NAM) and thus may have a generated specification. This flag is valid for OpenVMS Alpha only.</td>
</tr>
</tbody>
</table>
File Definition Language (FDL) Routines
FDL$GENERATE

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDL$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

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.
**File Definition Language (FDL) Routines**

**FDL$GENERATE**

**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**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>FDL$INVBLK</td>
<td>Invalid block.</td>
</tr>
<tr>
<td>RMS$ACT</td>
<td>File activity precludes operation.</td>
</tr>
<tr>
<td>RMS$CONTROLLC</td>
<td>Operation completed under Ctrl/C.</td>
</tr>
<tr>
<td>RMS$CONTROLO</td>
<td>Output completed under Ctrl/O.</td>
</tr>
<tr>
<td>RMS$CONTROLY</td>
<td>Operation completed under Ctrl/Y.</td>
</tr>
<tr>
<td>RMS$DNR</td>
<td>Device not ready or mounted.</td>
</tr>
<tr>
<td>RMS$EXT</td>
<td>ACP file extend failed.</td>
</tr>
<tr>
<td>RMS$OK_ALK</td>
<td>Record already locked.</td>
</tr>
<tr>
<td>RMS$OK_DUP</td>
<td>Record inserted had duplicate key.</td>
</tr>
<tr>
<td>RMS$OK_IDX</td>
<td>Index update error occurred.</td>
</tr>
<tr>
<td>RMS$PENDING</td>
<td>Asynchronous operation pending completion.</td>
</tr>
<tr>
<td>RMS$PRV</td>
<td>Insufficient privilege or file protection violation.</td>
</tr>
<tr>
<td>RMS$REX</td>
<td>Record already exists.</td>
</tr>
<tr>
<td>RMS$RLK</td>
<td>Target record currently locked by another stream.</td>
</tr>
<tr>
<td>RMS$RSA</td>
<td>Record stream currently active.</td>
</tr>
<tr>
<td>RMS$WLK</td>
<td>Device currently write locked.</td>
</tr>
<tr>
<td>SS$ACCVIO</td>
<td>Access violation.</td>
</tr>
<tr>
<td>STR$FATINERR</td>
<td>Fatal internal error in run-time library.</td>
</tr>
<tr>
<td>STR$ILLSTRCLA</td>
<td>Illegal string class.</td>
</tr>
<tr>
<td>STR$INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
</tbody>
</table>
FDL$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 in the section on FDL$CREATE 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 VAX, this is the RAB; 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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDL$V_DEFAULT_STRING</td>
<td>Interprets the <code>default_fdl_desc</code> argument as an FDL specification in string form. By default, the <code>default_fdl_desc</code> argument is interpreted as the file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_FDL_STRING</td>
<td>Interprets the <code>fdl_desc</code> argument as an FDL specification in string form. By default, the <code>fdl_desc</code> argument is interpreted as the file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_LONG_NAMES</td>
<td>Allocates and returns a long name access block (NAML) linked to the returned RMS file access block (FAB). The appropriate values are set in the NAML and FAB blocks so that the long file name fields of the NAML block will be used. By default, a name block is not allocated and the file name fields of FAB are used. If the FDL$V_LONG_NAMES flag is set, then the FDL$V_LONG_NAMES bit must also be set in the <code>flags</code> argument to the FDL$RELEASE routine to ensure that memory allocated for the NAML block is deallocated properly. This flag is valid for OpenVMS Alpha only.</td>
</tr>
<tr>
<td>FDL$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

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 in the section on FDL$CREATE 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.
**File Definition Language (FDL) Routines**

**FDL$PARSE**

<table>
<thead>
<tr>
<th><strong>stmt_num</strong></th>
<th>OpenVMS usage:</th>
<th>longword_unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>type:</td>
<td>longword (unsigned)</td>
</tr>
<tr>
<td></td>
<td>access:</td>
<td>write only</td>
</tr>
<tr>
<td></td>
<td>mechanism:</td>
<td>by reference</td>
</tr>
</tbody>
</table>

FDL statement number. The **stmt_num** argument is the address of a longword that receives the FDL statement number. If the routine finishes successfully, the **stmt_num** argument is the number of statements in the FDL specification. If the routine does not finish successfully, the **stmt_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**

<table>
<thead>
<tr>
<th>Condition Values Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>Bad block address.</td>
</tr>
<tr>
<td>LIB$_BADBLOSIZ</td>
<td>Bad block size.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>RMS$_DNF</td>
<td>Directory not found.</td>
</tr>
<tr>
<td>RMS$_DNR</td>
<td>Device not ready or not mounted.</td>
</tr>
<tr>
<td>RMS$_WCC</td>
<td>Invalid wildcard context (WCC) value.</td>
</tr>
</tbody>
</table>
FDL$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.
11

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, using the DCL command LIBRARY. For details, see the OpenVMS DCL Dictionary.

11.1 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 11.2 provides sample programs showing how to use various LBR routines. Section 11.3 is a reference section that provides details about each of the LBR routines.

11.1.1 Types of Libraries

You can use the LBR routines to maintain the following types of libraries:

- Object libraries, including Alpha object libraries, which 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 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, which 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. See the VAX MACRO and Instruction Set Reference Manual for information about defining macros.

  A macro library has a default file type of .MLB and defaults the file type of input files to .MAR.

- Help libraries, which contain modules of help messages that provide user information about a program. You can retrieve help messages at the DCL level by executing 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 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, which 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.
11.1 Introduction to 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 and Alpha shareable symbol table libraries which 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 OpenVMS Command Definition, Librarian, and Message Utilities Manual and the OpenVMS Linker Utility Manual.

A shareable image library has a default type of .OLB and defaults the file type of input files to .EEX.

- National character set (NCS) libraries, which 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, which 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.

11.1.2 Structure of Libraries

You create libraries by executing the DCL command LIBRARY or by calling the LBR$OPEN routine. When object, macro, text, help, or shareable image libraries are created, the Librarian utility structures them as described in Figure 11–1 and Figure 11–2. You can create user-developed libraries only by calling LBR$OPEN; they are structured as described in Figure 11–3.

11.1.2.1 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.

11.1.2.2 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.

11.1.2.3 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 11–1) 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.

¹ This manual has been archived but is available on the OpenVMS Documentation CD-ROM.
Object and shareable image libraries (see Figure 11–2) 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.

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 11–3). 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 11–1 Structure of a Macro, Text, or Help Library

<table>
<thead>
<tr>
<th>Library Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index (Module Name Table)</td>
</tr>
<tr>
<td>Key-1</td>
</tr>
<tr>
<td>Each key in the index points to a module.</td>
</tr>
<tr>
<td>Modules</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>
### Figure 11–2 Structure of an Object or Shareable Image Library

<table>
<thead>
<tr>
<th>Library Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index (Module Name Table)</td>
</tr>
<tr>
<td>Key–1</td>
</tr>
<tr>
<td>Each key in the index points to a module.</td>
</tr>
</tbody>
</table>

| Index (Global Symbol Table) |
| Global Symbol | Global Symbol | Global Symbol | Global Symbol | Global Symbol |
| Each global symbol is a key in the index, and points to the module in which it was defined. |

| Modules |
| Header | Header | Header | Header |
| Data | Data | Data | Data |
11.1.3 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 11–1 lists the routines and summarizes their functions.

Table 11–1 LBR Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$CLOSE</td>
<td>Closes an open library.</td>
</tr>
<tr>
<td>LBR$DELETE_DATA</td>
<td>Deletes a specified module's header and data.</td>
</tr>
</tbody>
</table>

(continued on next page)
11.1 Introduction to LBR Routines

Table 11–1 (Cont.) LBR Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$DELETE_KEY</td>
<td>Deletes a key from a library index.</td>
</tr>
<tr>
<td>LBR$FIND</td>
<td>Finds a module by using an address returned by a preceding call to LBR$LOOKUP_KEY.</td>
</tr>
<tr>
<td>LBR$FLUSH</td>
<td>Writes the contents of modified blocks to the library file and returns the virtual memory that contained those blocks.</td>
</tr>
<tr>
<td>LBR$GET_HEADER</td>
<td>Retrieves information from the library header.</td>
</tr>
<tr>
<td>LBR$GET_HELP</td>
<td>Retrieves help text from a specified library.</td>
</tr>
<tr>
<td>LBR$GET_HISTORY</td>
<td>Retrieves library update history records and calls a user-supplied routine with each record returned.</td>
</tr>
<tr>
<td>LBR$GET_INDEX</td>
<td>Calls a routine to process modules associated with some or all of the keys in an index.</td>
</tr>
<tr>
<td>LBR$GET_RECORD</td>
<td>Reads a data record from the module associated with a specified key.</td>
</tr>
<tr>
<td>LBR$INI_CONTROL</td>
<td>Initializes a control index that the Librarian uses to identify a library.</td>
</tr>
<tr>
<td>LBR$INSERT_KEY</td>
<td>Inserts a new key in the current library index.</td>
</tr>
<tr>
<td>LBR$LOOKUP_KEY</td>
<td>Looks up a key in the current index.</td>
</tr>
<tr>
<td>LBR$OPEN</td>
<td>Opens an existing library or creates a new one.</td>
</tr>
<tr>
<td>LBR$OUTPUT_HELP</td>
<td>Retrieves help text from an explicitly named library or from user-supplied default libraries, and optionally prompts you for additional help queries.</td>
</tr>
<tr>
<td>LBR$PUT_END</td>
<td>Terminates the writing of a sequence of records to a module using the LBR$PUT_RECORD routine.</td>
</tr>
<tr>
<td>LBR$PUT_HISTORY</td>
<td>Inserts a library update history record.</td>
</tr>
<tr>
<td>LBR$PUT_RECORD</td>
<td>Writes a data record to the module associated with the specified key.</td>
</tr>
<tr>
<td>LBR$REPLACE_KEY</td>
<td>Replaces an existing key in the current library index.</td>
</tr>
<tr>
<td>LBR$RET_RMSSTV</td>
<td>Returns the last RMS status value.</td>
</tr>
<tr>
<td>LBR$SEARCH</td>
<td>Finds index keys that point to specified data.</td>
</tr>
<tr>
<td>LBR$SET_INDEX</td>
<td>Sets the index number to be used during processing of the library.</td>
</tr>
<tr>
<td>LBR$SET_LOCATE</td>
<td>Sets Librarian subroutine record access to locate mode.</td>
</tr>
<tr>
<td>LBR$SET_MODULE</td>
<td>Reads and optionally updates a module header.</td>
</tr>
<tr>
<td>LBR$SET_MOVE</td>
<td>Sets Librarian subroutine record access to move mode.</td>
</tr>
</tbody>
</table>

11.2 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 Compaq Pascal and the subroutine examples are written in Compaq 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.
11.2 Using the LBR Routines: Examples

**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.

11.2.1 Creating, Opening, and Closing a Text Library

Example 11–1 is a sample Compaq 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 11–1 Creating a New Library Using Compaq Pascal**

```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$_ILLCTL,
  LBR$_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$L_LHMAX,CRE$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 LBR$INI_CONTROL (VAR library_index: UNSIGNED; func: UNSIGNED; typ: UNSIGNED;
                           VAR namblk: ARRAY[l..u:INTEGER] OF INTEGER := %IMMED 0):
  UNSIGNED; EXTERN;

FUNCTION LBR$OPEN (library_index: UNSIGNED;
                    fns: [class_s]PACKED ARRAY[l3..u3:INTEGER] OF CHAR;
                    create_options: Create_Array;
                    dns: [CLASS_S] PACKED ARRAY [l3..u3:INTEGER] OF CHAR;
                    rfnam: 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):
  UNSIGNED; EXTERN;
```

(continued on next page)
Example 11–1 (Cont.) Creating a New Library Using Compaq Pascal

(*Function that closes library*)

```pascal
FUNCTION LBR$CLOSE (library_index: UNSIGNED): INTEGER; EXTERN;
```

(*Error handler to check error codes if open/create not successful*)

```pascal
PROCEDURE Open_Error;
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*)
```

```pascal
BEGIN (* *************** DECLARATIONS COMPLETE ******************* *)

(*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$L_KEYLEN) DIV 4 + 1] := 31;
Options[IADDRESS(CRE$L_ALLOC) DIV 4 + 1] := 8;
Options[IADDRESS(CRE$L_IDXMAX) DIV 4 + 1] := 1;
Options[IADDRESS(CRE$L_ENTALL) DIV 4 + 1] := IADDRESS(CRE$C_VMSV3);
Options[IADDRESS(CRE$L_IDXOPT) DIV 4 + 1] := IADDRESS(CRE$C_MACTXTCAS);

(*Initialize library control index*)
status := LBR$INI_CONTROL (lib_index_ptr, IADDRESS(LBR$C_CREATE), IADDRESS(LBR$C_TYP_TXT));
IF NOT ODD(status) THEN (*Check return status*)
WRITELN('Initialization Failed')
ELSE (*Initialization was successful*)
BEGIN (*Create and open the library*)
status := LBR$OPEN (lib_index_ptr, Lib_Name, Options, File_Type);
IF NOT ODD(status) THEN (*Check return status*)
Open_Error (*Call error handler*)
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*)
11.2 Using the LBR Routines: Examples

Each item in the following list corresponds to a number highlighted in Example 11–1:

1. Use the INHERIT attribute to access the LBR and CRE symbols from SYS$LIBRARY:STARLET.PEN.

2. 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.

3. 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.

4. Initialize the array called Options with the values the Librarian needs to create the library.

5. Call LBR$INI_CONTROL, specifying that the function to be performed is create and that the library type is text.

6. Call LBR$OPEN to create and open the library; pass the Options array initialized in item 5 to the Librarian.

7. If the call to LBR$OPEN was unsuccessful, call the procedure Open_Error (see item 4) to determine the cause of the error.

11.2.2 Inserting a Module

Example 11–2 illustrates the insertion of a module into a library from a Compaq 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 11–2 Inserting a Module into a Library Using Compaq 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 := FALSE; (*True if new mod not already in lib*)

(*------Function Definitions------*)

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 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 LBR$LOOKUP_KEY (library_index: UNSIGNED;
key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF CHAR;
VAR txtrfa: Rfa_Ptr):
INTEGER; EXTERN;

FUNCTION LBR$INSERT_KEY (library_index: UNSIGNED;
key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF CHAR;
taxtrfa: Rfa_Ptr):
INTEGER; EXTERN;

(*Function that writes data records*)

(continued on next page)
Example 11–2 (Cont.) Inserting a Module into a Library Using Compaq Pascal

FUNCTION LBR$PUT_RECORD (library_index: UNSIGNED; (*to modules*)
textline:[CLASS_S] PACKED ARRAY [1..u:INTEGER] OF CHAR;
    txtrfa: Rfa_Ptr):
    INTEGER; EXTERN;
    (*Function that marks end of a module*)
FUNCTION LBR$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,
        ! 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 (*Is module already in the library?*)
            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,
                        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 3
                            Key_Not_Found := TRUE
                    END
        END
    END;

(continued on next page)
Example 11–2 (Cont.) Inserting a Module into a Library Using Compaq Pascal

(* wenn 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
        READ(Textin,Text_Data_Record); (*Read record from
        external file*)
        status := LBR$PUT_RECORD (lib_index_ptr, (*Write*)
        Text_Data_Record, (*record to*)
        txtrfa_ptr); (*library*)
        IF NOT ODD(status) THEN
          WRITELN(‘Put Record Routine Not Successful’)   END; (*of WHILE statement*)
      END; (*of WHILE statement*)
      IF ODD(status) THEN (*True if all the records have been
        successfully written into the library*)
        BEGIN
          status := LBR$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
                status := LBR$INSERT_KEY (lib_index_ptr,
                Module_Name,
                txtrfa_ptr);
                IF NOT ODD(status) THEN
                  WRITELN(‘Insert Key Not Successful’)   END
              END;
              END;
      END; (*of program insertmod*)

Each item in the following list corresponds to a number highlighted in
Example 11–2:

1. Call LBR$INI_CONTROL, specifying that the function to be performed is
   update and that the library type is text.

2. Call LBR$LOOKUP_KEY to see whether the module to be inserted is already
   in the library.

3. 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.)

4. 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.

5. Use LBR$INSERT_KEY to insert a key for the module into the current index.
11.2.3 Extracting a Module

Example 11–3 illustrates the extraction of a library module from a Compaq 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 11–3 Extracting a Module from a Library Using Compaq 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*)
  txttrfa_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;
  rlna: ARRAY [l4..u4:INTEGER] OF CHAR := %IMMED 0;
  rns: [CLASS_S] PACKED ARRAY [l5..u5:INTEGER] OF CHAR :=
    %IMMED 0;
  VAR rnslen: INTEGER := %IMMED 0):
  INTEGER; EXTERN;
Example 11–3 (Cont.) Extracting a Module from a Library Using Compaq Pascal

(*Function that finds a key in an index*)
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
key_name:[CLASS_S] PACKED ARRAY [1..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 [1..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: ')缙REN(Lib_Name);
WRITE('Module Name: ')缙REN(Module_Name);
WRITE('Extract Into File: ')缙REN(Extracted_File);
status := LBR$INI_CONTROL (lib_index_ptr,
IADDRESS(LBR$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
status := LBR$LOOKUP_KEY (lib_index_ptr,
Module_Name, txtrfa_ptr);
IF NOT ODD(status) THEN
WRITELN('Lookup Key Not Successful')
ELSE
BEGIN
OPEN(Textout,Extracted_File,new); REWRITE(Textout)
END
END
END;
WHILE ODD(status) DO
BEGIN
nullstring := ’0’;
FOR i := 1 TO 255 DO
Outtext[i] := nullstring;
status := LBR$GET_RECORD (lib_index_ptr,
Outtext);
IF NOT ODD(status) THEN
BEGIN
IF status = IADDRESS(RMS$_EOF) THEN
WRITELN('RMS end of file')
END
(continued on next page)
Librarian (LBR) Routines

11.2 Using the LBR Routines: Examples

Example 11–3 (Cont.)  Extracting a Module from a Library Using Compaq Pascal

ELSE

BEGIN

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; (*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 11–3:

1. Call LBR$INI_CONTROL, specifying that the function to be performed is update and that the library type is text.

2. Call LBR$LOOKUP_KEY to find the key that points to the module you want to extract.

3. Open an output file to receive the extracted module.

4. Initialize the variable that is to receive the extracted records to null characters.

5. 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.

6. Write the extracted record data to the output file. This record should consist only of the data up to the first null character.

11.2.4 Deleting a Module

Example 11–4 illustrates the deletion of library module from a Compaq 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 11–4 Deleting a Module from a Library Using Compaq 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 LBR$INI_CONTROL (VAR library_index: UNSIGNED; func: UNSIGNED; typ: UNSIGNED;
                       VAR namblk: ARRAY[l..u:INTEGER] OF INTEGER := %IMMED 0):
  INTEGER; EXTERN;

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 LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF CHAR;
                         VAR txtrfa: Rfa_Ptr):
  INTEGER; EXTERN;

FUNCTION LBR$DELETE_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF CHAR):
  INTEGER;
EXTERN;

FUNCTION LBR$DELETE_DATA (library_index: UNSIGNED;
                           txtrfa: Rfa_Ptr):
  INTEGER;
EXTERN;

FUNCTION LBR$CLOSE (library_index: UNSIGNED):
  INTEGER; EXTERN;

(continued on next page)
Example 11–4 (Cont.) Deleting a Module from a Library Using Compaq Pascal

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$INI_CONTROL (lib_index_ptr, IADDRESS(LBR$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 in the library?*)
status := LBR$LOOKUP_KEY (lib_index_ptr, Module_Name,
txtrfa_ptr);
IF NOT ODD(status) THEN (*Check status*)
WRITELN('Lookup Key Not Successful')
END
IF ODD(status) THEN (*Key was found; delete it*)
BEGIN
status := LBR$DELETE_KEY (lib_index_ptr, 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$DELETE_DATA (lib_index_ptr, txtrfa_ptr);
IF NOT ODD(status) THEN
WRITELN('Delete Data Routine Not Successful')
END
status := LBR$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 11–4:

1. Call LBR$INI_CONTROL, specifying that the function to be performed is update and the library type is text.
2. Call LBR$LOOKUP_KEY to find the key associated with the module you want to delete.
3. 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$DELETE_KEY for each key.
4. Call LBR$DELETE_DATA to delete the module (the module header and data) from the library.
11.2.5 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 11–5 illustrates the way that keys can be associated with modules.

Example 11–5  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

(continued on next page)
Example 11–5 (Cont.)  Associating Keys with Modules

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:

```plaintext
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 11–6 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 11–6  Listing Keys Associated with a Module

```fortran
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
```
11.2.6 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 11–7 displays the user data portion of module headers on SYS$OUTPUT and applies updates from SYS$INPUT.

Example 11–7 Displaying the Module Header

```fortran
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
    MODNAME = CLI$GET_VALUE ('MODHEAD', MODNAME)
    MODNAME_LEN = $LEN (MODNAME)

    ! Compare library index to MODNAME
    IF (MODNAME_LEN == INDEX) THEN
        MODNAME = MODNAME(1:INDEX)

        ! Get RFA of module
        TXTRFA = LBR$INI_CONTROL (INDEX)

        ! Get module header
        HEADER = LBR$GET_HEADER (INDEX, MODNAME, MODNAME_LEN)

        ! Move data to module header
        USERDATA_LEN = MHD$B_USRDAT + MODNAME_LEN
        USERDATA = HEADER (USER_START:USER_DATA_LEN)

        ! Display module header
        WRITE (SYS$OUTPUT, *) MODNAME, USERDATA

        ! Apply updates from SYS$INPUT
        WRITE (SYS$INPUT, *) MODNAME, USERDATA
    ELSE
        WRITE (SYS$OUTPUT, *)'Module not found'
    END IF

    RETURN
END MODHEAD
```

(continued on next page)
Example 11–7 (Cont.) Displaying the Module Header

! 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))

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 (%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

11.2.7 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 11–8 reads the library header and displays some information from it.

Example 11–8  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 $LBRED
EXTERNAL LBR$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))
(continued on next page)
11.2 Using the LBR Routines: Examples

Example 11–8 (Cont.) Reading Library Headers

! Exit
END

11.2.8 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 11–9 solicits keywords from SYS$INPUT and displays the text associated with those keywords on SYS$OUTPUT, thus inhibiting the prompting facility.

Example 11–9 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

(continued on next page)
Example 11–9 (Cont.) Displaying Text from a Help Library

! 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

11.2.9 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 11–10 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 11–10  Displaying Index Entries

SUBROUTINE LIST (INDEX)
! Lists modules in the library

INTEGER STATUS, INDEX, ! Return status
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, RFA (2) ! Good return status
PARAMETER (STATUS_OK = 1) ! RFA of module name in index
CHARACTER*(*) MODNAME ! Odd value
! Display the name
TYPE *, MODNAME ! Display module name
! Exit
MATCH = STATUS_OK
END

11.3 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 a Module’s Data

The LBR$DELETE_DATA routine deletes the module header and data associated with the specified module.

Format

LBR$DELETE_DATA 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 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$LOOKUP_exit KEY or LBR$PUT_RECORD.

Description

If you want to delete a library module, you must first call LBR$DELETE_KEY to delete any keys that point to it. If no library index keys are pointing to the module header, LBR$DELETE_DATA deletes the module header and associated data records; otherwise, this routine returns the error LBR$STILLKEYS.

Note that other LBR routines may reuse data blocks that contain no data.
## Librarian (LBR) Routines
### LBR$DELETE_DATA

<table>
<thead>
<tr>
<th>Condition Value Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$ _ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$ _INVRFA</td>
<td>Specified RFA not valid.</td>
</tr>
<tr>
<td>LBR$ _LIBNOTOPEN</td>
<td>Specified library not open.</td>
</tr>
<tr>
<td>LBR$ _STILLKEYS</td>
<td>Keys in other indexes still point at the module header. Therefore, the specified module was not deleted.</td>
</tr>
</tbody>
</table>
LBR$DELETE_KEY—Delete a Key

The LBR$DELETE_KEY routine deletes a key from a library index.

Format

LBR$DELETE_KEY library_index , key_name

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.

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

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:

<table>
<thead>
<tr>
<th>Argument Characteristics</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenVMS usage</td>
<td>char_string</td>
</tr>
<tr>
<td>type</td>
<td>character string</td>
</tr>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>
Librarian (LBR) Routines
LBR$DELETE_KEY

Description
If LBR$DELETE_KEY finds the key specified by key_name in the current index, it deletes the key. Note that, if you want to delete a library module, you should first use LBR$DELETE_KEY to delete any 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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_KEYNOTFND</td>
<td>Specified key not found.</td>
</tr>
<tr>
<td>LBR$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
<tr>
<td>LBR$_UPDURTRAV</td>
<td>Specified index update not valid in a user-supplied routine specified in LBR$SEARCH or LBR$GET_INDEX.</td>
</tr>
</tbody>
</table>
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.

textrfa

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$FIND initializes internal tables so you can read the associated data.
Librarian (LBR) Routines
LBR$FIND

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_INVRFA</td>
<td>Specified RFA not valid.</td>
</tr>
<tr>
<td>LBR$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
</tbody>
</table>
LBR$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.
## Librarian (LBR) Routines

### LBR$FLUSH

#### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$NORMAL</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>LBR$BADPARAM</td>
<td>Error. A value passed to the LBR$FLUSH routine was either out of range or an illegal value.</td>
</tr>
<tr>
<td>LBR$WITERRR</td>
<td>Error. An error occurred during the writing of the cached update blocks to the library file.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Offset in Longwords</th>
<th>Symbolic Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LHI$L_TYPE</td>
<td>Library type (see LBR$OPEN for possible values)</td>
</tr>
<tr>
<td>1</td>
<td>LHI$L_NINDEX</td>
<td>Number of indexes</td>
</tr>
<tr>
<td>2</td>
<td>LHI$L_MAJORID</td>
<td>Library format major identification</td>
</tr>
<tr>
<td>3</td>
<td>LHI$L_MINORID</td>
<td>Library format minor identification</td>
</tr>
<tr>
<td>4</td>
<td>LHI$T_LBRVER</td>
<td>ASCII version of Librarian</td>
</tr>
<tr>
<td>12</td>
<td>LHI$L_CREDAT</td>
<td>Creation date/time</td>
</tr>
<tr>
<td>14</td>
<td>LHI$L_UPDTIM</td>
<td>Date/time of last update</td>
</tr>
</tbody>
</table>
Librarian (LBR) Routines
LBR$GET_HEADER

<table>
<thead>
<tr>
<th>Offset in Longwords</th>
<th>Symbolic Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>LHI$L_UPDHIS</td>
<td>Virtual block number (VBN) of start of update history</td>
</tr>
<tr>
<td>17</td>
<td>LHI$L_FREEVBN</td>
<td>First logically deleted block</td>
</tr>
<tr>
<td>18</td>
<td>LHI$L_FREEBLK</td>
<td>Number of deleted blocks</td>
</tr>
<tr>
<td>19</td>
<td>LHI$B_NEXTRFA</td>
<td>Record's file address (RFA) of end of library</td>
</tr>
<tr>
<td>21</td>
<td>LHI$L_NEXTVBN</td>
<td>Next VBN to allocate at end of file</td>
</tr>
<tr>
<td>22</td>
<td>LHI$L_FREIDXBLK</td>
<td>Number of free preallocated index blocks</td>
</tr>
<tr>
<td>23</td>
<td>LHI$L_FREEIDX</td>
<td>List head for preallocated index blocks</td>
</tr>
<tr>
<td>24</td>
<td>LHI$L_HIPREAL</td>
<td>VBN of highest preallocated block</td>
</tr>
<tr>
<td>25</td>
<td>LHI$L_IDXBLKS</td>
<td>Number of index blocks in use</td>
</tr>
<tr>
<td>26</td>
<td>LHI$L_IDXCNT</td>
<td>Number of index entries (total)</td>
</tr>
<tr>
<td>27</td>
<td>LHI$L_MODCNT</td>
<td>Number of entries in index 1 (module names)</td>
</tr>
<tr>
<td>28</td>
<td>LHI$L_MHDUSZ</td>
<td>Number of bytes of additional information reserved in module header</td>
</tr>
<tr>
<td>29</td>
<td>LHI$L_MAXLUHREC</td>
<td>Maximum number of library update history records maintained</td>
</tr>
<tr>
<td>30</td>
<td>LHI$L_NUMLUHREC</td>
<td>Number of library update history records in history</td>
</tr>
<tr>
<td>31</td>
<td>LHI$L_LIBSTATUS</td>
<td>Library status (false if there was an error closing the library)</td>
</tr>
<tr>
<td>32-128</td>
<td>Reserved by Compaq</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
</tbody>
</table>
**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.
Librarian (LBR) Routines
LBR$GET_HELP

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:

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Return all level 1 help text in the library.</td>
</tr>
<tr>
<td>KEY ...</td>
<td>Return all help text associated with the specified key and its subkeys (valid for level 1 keys only).</td>
</tr>
<tr>
<td>* ...</td>
<td>Return all help text in the library.</td>
</tr>
</tbody>
</table>

**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$_ILLCTL Specified library control index not valid.
- LBR$_LIBNOTOPEN 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

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_NORMAL</td>
<td>Normal exit from the routine.</td>
</tr>
<tr>
<td>LBR$_EMPTYHIST</td>
<td>History empty. This is an informational code, not an error code.</td>
</tr>
<tr>
<td>LBR$_INTRNLERR</td>
<td>Internal Librarian routine error occurred.</td>
</tr>
<tr>
<td>LBR$_NOHISTORY</td>
<td>No update history. This is an informational code, not an error code.</td>
</tr>
</tbody>
</table>
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]

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

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 (see Section 11.1.2.3).

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.
Librarian (LBR) Routines
LBR$GET_INDEX

LBR$GET_INDEX passes two arguments to the routine:

- 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’s 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 routine must return a value to indicate success or failure. If the 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.

**Description**

LBR$GET_INDEX searches through the specified index for a key that matches the argument **match_desc**. Each time it finds a match, it calls the routine specified by the **routine_name** argument. If you do not specify the **match_desc** argument, it calls the routine for every key in the index.

For example, if you call LBR$GET_INDEX with **match_desc** equal to TR* and **index_number** set to 1 (module name table), then LBR$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.
Librarian (LBR) Routines
LBR$GET_RECORD

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
<tr>
<td>LBR$_LKPNOTDON</td>
<td>Requested key lookup not done.</td>
</tr>
<tr>
<td>RMS$_EOF</td>
<td>Error. An attempt has been made to read past the logical end of the data in the module.</td>
</tr>
</tbody>
</table>
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 LBR$C_CREATE, LBR$C_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$C_TYP_OBJ (VAX object)
- LBR$C_TYP_SHSTB (VAX shareable image)
- LBR$C_TYP_EOBJ (Alpha object)
- LBR$C_TYP_ESHSTB (Alpha shareable image)
Librarian (LBR) Routines
LBR$INI_CONTROL

- 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 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$NORMAL</td>
<td>Library control index initialized successfully.</td>
</tr>
<tr>
<td>LBR$ILLFUNC</td>
<td>Requested function not valid.</td>
</tr>
<tr>
<td>LBR$ILLTYP</td>
<td>Specified library type not valid.</td>
</tr>
<tr>
<td>LBR$TOOMNYLIB</td>
<td>Error. An attempt was made to allocate more than 16 control indexes.</td>
</tr>
</tbody>
</table>
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

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.

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:

<table>
<thead>
<tr>
<th>Argument Characteristics</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenVMS usage</td>
<td>char_string</td>
</tr>
<tr>
<td>type</td>
<td>character string</td>
</tr>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>
txtrfa
OpenVMS usage: vector_longword_unsigned
type: longword (unsigned)
access: modify
mechanism: by reference

The record's 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.

Description

You cannot call LBR$INSERT_KEY within the user-supplied routine specified in LBR$SEARCH or LBR$GET_INDEX.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_DUPKEY</td>
<td>Index already contains the specified key.</td>
</tr>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_INVRFA</td>
<td>Specified RFA does not point to valid data.</td>
</tr>
<tr>
<td>LBR$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
<tr>
<td>LBR$_UPDURTRAV</td>
<td>LBR$INSERT_KEY was called by the user-defined routine specified in LBR$SEARCH or LBR$GET_INDEX.</td>
</tr>
</tbody>
</table>
Librarian (LBR) Routines
LBR$LOOKUP_KEY

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

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.

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:

<table>
<thead>
<tr>
<th>Argument Characteristics</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenVMS usage</td>
<td>char_string</td>
</tr>
<tr>
<td>type</td>
<td>character string</td>
</tr>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by descriptor</td>
</tr>
</tbody>
</table>
txtrfa
OpenVMS usage: vector_longword_unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

The record's 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.

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 (consisting of the virtual block number (VBN) and the byte offset) to the 2-longword array referenced by txtrfa. Note that the RFA is only 6 bytes long.

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$OPEN—Open or Create a Library

The LBR$OPEN routine opens an existing library or creates a new one.

Format


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$C_CREATE, you must include the create_options argument. The following table shows the entries that the array must contain. Each programming language provides an appropriate mechanism for accessing the listed symbols.
<table>
<thead>
<tr>
<th>Offset in Longwords</th>
<th>Symbolic Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CRE$L_TYPE</td>
<td>Library type:</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_UNK (0)</td>
<td>Unknown/unspecified</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_OBJ (1)</td>
<td>VAX object</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_MLB (2)</td>
<td>Macro</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_HLP (3)</td>
<td>Help</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_TXT (4)</td>
<td>Text</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_SHSTB (5)</td>
<td>VAX shareable image</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_NCS (6)</td>
<td>NCS</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_EOBJ (7)</td>
<td>Alpha object</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_ESHSTB (8)</td>
<td>Alpha shareable image</td>
</tr>
<tr>
<td>(9–127)</td>
<td></td>
<td>Reserved by Compaq</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_USRLW (128)</td>
<td>User library types — low end of range</td>
</tr>
<tr>
<td></td>
<td>LBR$C_TYP_USRHI (255)</td>
<td>User library types — high end of range</td>
</tr>
<tr>
<td>1</td>
<td>CRE$L_KEYLEN</td>
<td>Maximum length of ASCII keys or, if 0, indicates 32-bit unsigned keys (binary keys)</td>
</tr>
<tr>
<td>2</td>
<td>CRE$L_ALLOC</td>
<td>Initial library file allocation</td>
</tr>
<tr>
<td>3</td>
<td>CRE$L_IDXMAX</td>
<td>Number of library indexes (maximum of eight)</td>
</tr>
<tr>
<td>4</td>
<td>CRE$L_UHDMAX</td>
<td>Number of additional bytes to reserve in module header</td>
</tr>
<tr>
<td>5</td>
<td>CRE$L_ENTALL</td>
<td>Number of index entries to preallocate</td>
</tr>
<tr>
<td>6</td>
<td>CRE$L_LUHMAX</td>
<td>Maximum number of library update history records to maintain</td>
</tr>
<tr>
<td>7</td>
<td>CRE$L_VERTYP</td>
<td>Format of library to create:</td>
</tr>
<tr>
<td></td>
<td>CRE$C_VMSV2</td>
<td>VMS Version 2.0</td>
</tr>
<tr>
<td></td>
<td>CRE$C_VMSV3</td>
<td>VMS Version 3.0</td>
</tr>
<tr>
<td>8</td>
<td>CRE$L_IDXOPT</td>
<td>Index key casing option:</td>
</tr>
<tr>
<td></td>
<td>CRE$C_HLPCASING</td>
<td>Treat character case as it is for help libraries</td>
</tr>
<tr>
<td></td>
<td>CRE$C_OBJCASING</td>
<td>Treat character case as it is for object libraries</td>
</tr>
<tr>
<td></td>
<td>CRE$C_MACTXTCAS</td>
<td>Treat character case as it is for macro and text libraries</td>
</tr>
<tr>
<td>9–19</td>
<td></td>
<td>Reserved by Compaq</td>
</tr>
</tbody>
</table>

The input of uppercase and lowercase characters is treated differently for help, object, macro, and text libraries. For details, see the OpenVMS Command Definition, Librarian, and Message Utilities Manual.
Librarian (LBR) Routines
LBR$OPEN

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 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 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ERRCLOSE</td>
<td>Error. When the library was last modified while opened for write access, the write operation was interrupted. This left the library in an inconsistent state.</td>
</tr>
<tr>
<td>LBR$_ILLCREOPT</td>
<td>Requested create options not valid or not supplied.</td>
</tr>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_ILLFMT</td>
<td>Specified library format not valid.</td>
</tr>
<tr>
<td>LBR$_ILLFUNC</td>
<td>Specified library function not valid.</td>
</tr>
<tr>
<td>LBR$_LIBOPN</td>
<td>Specified library already open.</td>
</tr>
<tr>
<td>LBR$_NOFILNAM</td>
<td>Error. The <strong>fns</strong> argument was not supplied or the RMS NAM block was not filled in.</td>
</tr>
<tr>
<td>LBR$_OLDLIBRARY</td>
<td>Success. The specified library has been opened; the library was created with an old library format.</td>
</tr>
<tr>
<td>LBR$_OLDMISMCH</td>
<td>Requested library function conflicts with old library type specified.</td>
</tr>
<tr>
<td>LBR$_TYPMISMCH</td>
<td>Library type does not match the requested type.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLP$M_PROMPT</td>
<td>Interactive help prompting is in effect.</td>
</tr>
<tr>
<td>HLP$M_PROCESS</td>
<td>The process logical name table is searched for default help libraries.</td>
</tr>
<tr>
<td>HLP$M_GROUP</td>
<td>The group logical name table is searched for group default help libraries.</td>
</tr>
<tr>
<td>HLP$M_SYSTEM</td>
<td>The system logical name table is searched for system default help libraries.</td>
</tr>
<tr>
<td>HLP$M_LIBLIST</td>
<td>The list of default libraries available is output with the list of topics available.</td>
</tr>
<tr>
<td>HLP$M_HELP</td>
<td>The list of topics available in a help library is preceded by the major portion of the text on help.</td>
</tr>
</tbody>
</table>

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$M_PROMPT is set or defaulted).
Librarian (LBR) Routines
LBR$OUTPUT_HELP

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:

<table>
<thead>
<tr>
<th>Response</th>
<th>Action in the Current Prompt Environment¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword [ . . . ]</td>
<td>(1,2) Search all enabled libraries for these keys. (3,4) Search additional help for the current topic (and subtopic) for these keys.</td>
</tr>
<tr>
<td>@filespec [keyword[ . . . ]]</td>
<td>(1,2) Same as above, except that the root library is the library specified by filespec. If the specified library does not exist, treat @filespec as a normal key. (3,4) Same as above; treat @filespec as a normal key.</td>
</tr>
<tr>
<td>?</td>
<td>(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.</td>
</tr>
<tr>
<td>Carriage Return</td>
<td>(1) Exit from LBR$OUTPUT_HELP. (2) Change root library to main library. (3,4) Strip the last keyword from a list of keys defining the current topic (and subtopic) environment.</td>
</tr>
<tr>
<td>Ctrl/Z</td>
<td>(1,2,3,4) Exit from LBR$OUTPUT_HELP.</td>
</tr>
</tbody>
</table>

¹Keyed to the prompt in the preceding list.

### Condition Values Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ILLINROU</td>
<td>Input routine improperly specified or omitted.</td>
</tr>
<tr>
<td>LBR$_ILLOUTROU</td>
<td>Output routine improperly specified or omitted.</td>
</tr>
<tr>
<td>LBR$_NOHLPLIS</td>
<td>Error. No default help libraries can be opened.</td>
</tr>
<tr>
<td>LBR$_TOOMNYARG</td>
<td>Error. Too many arguments were specified.</td>
</tr>
<tr>
<td>LBR$_USRINPERR</td>
<td>Error. An error status was returned by the user-supplied input routine.</td>
</tr>
</tbody>
</table>
LBR$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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_NORMAL</td>
<td>Normal exit from the routine.</td>
</tr>
<tr>
<td>LBR$_INTRNLERR</td>
<td>Internal Librarian error.</td>
</tr>
<tr>
<td>LBR$_NOHISTORY</td>
<td>No update history. This is an informational code, not an error code.</td>
</tr>
<tr>
<td>LBR$_RECLNG</td>
<td>Record length greater than that specified by LBR$C_MAXRECSIZ. The record was not inserted or truncated.</td>
</tr>
</tbody>
</table>
LBR$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

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.

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. The maximum record size for VAX libraries is symbolically defined as LBR$C_MAXRECSIZ; for Alpha libraries, the symbolic maximum record size is ELBR$_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$PUT_RECORD.
Librarian (LBR) Routines
LBR$PUT_RECORD

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
</tbody>
</table>
LBR$REPLACE_KEY—Replace a Library Key

The LBR$REPLACE_KEY routine inserts a key in an index by changing the pointer associated with an existing key or by inserting a new key.

Format

LBR$REPLACE_KEY library_index ,key_name ,oldrfa ,newrfa

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.

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

Library key (for libraries with ASCII keys). The key_name argument is the address of a string descriptor for the key.

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

Library key (for libraries with binary keys). The key_name argument is the address of an unsigned longword value for the key.
Librarian (LBR) Routines
LBR$REPLACE_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.

**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.

Arguments

None.

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

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

Library index number. The index_number argument is the address of a longword containing the number of the index you want to search. Refer to Section 11.1.2.3.

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

Record’s 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.

Description
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 global symbols 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:
  - 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 routine must return a value to indicate success or failure. If the specified 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$_KEYNOTFND Librarian did not find any keys with the specified RFA.
LBR$_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 11.1.2.3.

**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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR$_ILLCTL</td>
<td>Specified library control index not valid.</td>
</tr>
<tr>
<td>LBR$_ILLIDXNUM</td>
<td>Library index number specified not valid.</td>
</tr>
<tr>
<td>LBR$_LIBNOTOPN</td>
<td>Specified library not open.</td>
</tr>
</tbody>
</table>
LBR$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.
Librarian (LBR) Routines
LBR$SET_MODULE

**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.
12 Lightweight Directory Access Protocol (LDAP) Routines

12.1 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.

12.1.1 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.
12.1.2 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 12.20.

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.

12.1.3 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:

```c
#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.
12.1 Introduction

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:

```c
$ 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
```

12.1.4 64-bit Addressing Support

This section describes the LDAP 64-bit addressing support.

12.1.4.1 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.
Lightweight Directory Access Protocol (LDAP) Routines

12.1 Introduction

12.1.4.2 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.

12.1.4.2.1 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.

12.1.4.2.2 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.

```c
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.

```c
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:

```c
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:

```
<table>
<thead>
<tr>
<th>bv_len</th>
<th>bv_val</th>
</tr>
</thead>
<tbody>
<tr>
<td>:A</td>
<td>:A+4</td>
</tr>
</tbody>
</table>
```

In the case of a 64-bit compilation, the layout would be:
12.1 Introduction

The following code would therefore work correctly regardless of pointer size:

```c
#include <ldap.h>

char *buff;
BerValue val;

buff = (char *)malloc(255);
val.bv_len = 255;
val.bv_val = buff;
```

12.1.4.3 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.

12.1.5 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.
12.2 Common Data Structures and Memory Handling

The following are definitions of some data structures that are common to several LDAP API functions.

```c
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 12.17 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:
12.2 Common Data Structures and Memory Handling

ber_bvecfree()
ber_bvfree()
ber_free()
ldap_control_free()
ldap_controls_free()
ldap_memfree()
ldap_msgfree()
ldap_value_free()
ldap_value_free_len()

12.3 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 (0x09) -- new in LDAPv3
LDAP_ADMINLIMIT_EXCEEDED (0x0a) -- new in LDAPv3
LDAP_UNAVAILABLE_CRITICAL_EXTENSION (0x0b) -- 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)
12.3 LDAP Error Codes

- 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)

12.4 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.

```c
LDAP *ldap_init( 
    const char *hostname, 
    int portno); 
```
Use of the following function is deprecated.

```c
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.

### 12.5 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.

```c
int ldap_get_option(
    LDAP *ld,
    int option,
    void *outvalue
);
```
Lightweight Directory Access Protocol (LDAP) Routines

12.5 LDAP Session Handle Options

```c
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)`. 

### 12.5 LDAP Session Handle Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type for Invalue Parameter</th>
<th>Type for Outvalue Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP_OPT_RESTART (0x09)</td>
<td>int (LDAP_OPT_ON or LDAP_OPT_OFF)</td>
<td>int *</td>
<td>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.</td>
</tr>
<tr>
<td>LDAP_OPT_PROTOCOL_VERSION (0x11)</td>
<td>Type for Invalue Parameter: int *</td>
<td>Type for Outvalue Parameter: int *</td>
<td>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).</td>
</tr>
<tr>
<td>LDAP_OPT_SERVER_CONTROLS (0x12)</td>
<td>Type for Invalue Parameter: LDAPControl **</td>
<td>Type for Outvalue Parameter: LDAPControl ***</td>
<td>A default list of LDAP server controls to be sent with each request. See Section 12.6 for more information.</td>
</tr>
<tr>
<td>LDAP_OPT_CLIENT_CONTROLS (0x13)</td>
<td>Type for Invalue Parameter: LDAPControl **</td>
<td>Type for Outvalue Parameter: LDAPControl ***</td>
<td>A default list of client controls that affect the LDAP session. See Section 12.6 for more information.</td>
</tr>
<tr>
<td>LDAP_OPT_HOST_NAME (0x30)</td>
<td>char *</td>
<td>char **</td>
<td>The host name (or list of host) for the primary LDAP server.</td>
</tr>
<tr>
<td>LDAP_OPT_ERROR_NUMBER (0x31)</td>
<td>int *</td>
<td>int *</td>
<td>The code of the most recent LDAP error that occurred for this session.</td>
</tr>
<tr>
<td>LDAP_OPT_ERROR_STRING (0x32)</td>
<td>char *</td>
<td>char **</td>
<td>The message returned with the most recent LDAP error that occurred for this session.</td>
</tr>
</tbody>
</table>

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().
12.5 LDAP Session Handle Options

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.

12.6 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.
Lightweight Directory Access Protocol (LDAP) Routines

12.7 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.

```c
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:

```c
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.
12.7 Authenticating to the Directory

For `ldap_simple_bind()`, the password to compare to the entry’s `userPassword` attribute.

List of LDAP server controls.

List of client controls.

This result parameter will be set to the message id of the request if the `ldap_sasl_bind()` call succeeds.

This result parameter will be filled in with the credentials passed back by the server for mutual authentication, if given. An allocated `ber_bval` 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 12.18 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 12.18 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()`.

12.8 Closing the Session

The following functions are used to unbind from the directory, close the connection, and dispose of the session handle.

```c
int ldap_unbind( LDAP *ld );
int ldap_unbind_s( LDAP *ld );
```

Parameter is as follows:

Id 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.
12.9 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.

```c
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.
12.9 Searching

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 12.18 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 12.18.

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 bind. 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 12.18 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.

12.9.1 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.
12.10 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.

```c
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.
12.10 Comparing a Value Against an Entry

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 12.18 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 bind. 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 LDAPCOMPARE_TRUE or LDAPCOMPARE_FALSE if the operation was successful, or another LDAP error code if it was not. See Section 12.18 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.

12.11 Modifying an Entry

The following functions are used to modify an existing LDAP entry. There are four variations.

```c
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
);
```

12.11 Modifying an Entry

```c
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 12.18 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.
12.11 Modifying an Entry

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 12.18 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.

12.12 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:

- **Id** The session handle.
- **dn** The name of the entry whose DN is to be changed.
- **newrdn** The new RDN to give the entry.
12.12 Modifying the Name of an 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 12.18 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 12.18 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.

12.13 Adding an Entry

The following functions are used to add entries to the LDAP directory. There are four variations.

```c
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
);
```
Lightweight Directory Access Protocol (LDAP) Routines

12.13 Adding an Entry

```c
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 12.18 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 12.18 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.

12.14 Deleting an Entry

The following functions are used to delete a leaf entry from the LDAP directory. There are four variations.

```c
int ldap_delete_ext(
    LDAP *ld,
    const char *dn,
    LDAPControl **serverctrls,
    LDAPControl **clientctrls,
    int *msgidp
);
```

Note that the parent of the entry being deleted must already exist for a delete to succeed.

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 12.18 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 12.18 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.
12.14 Deleting an Entry

```c
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 12.18 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 12.18 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.

12.15 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.
Lightweight Directory Access Protocol (LDAP) Routines

12.15 Extended Operations

```c
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 12.18 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 12.18 for more information about possible errors and how to interpret them. The retoid and retdatap 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.
12.16 Abandoning an Operation

The following calls are used to abandon an operation in progress:

```c
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 12.18 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.

12.17 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.

```c
int ldap_result(
    LDAP *ld, int msgid, int all, struct timeval *timeout, LDAPMessage **res);
```
Lightweight Directory Access Protocol (LDAP) Routines
12.17 Obtaining Results and Looking Inside LDAP Messages

```c
int ldap_msgfree( LDAPMessage *res );
int ldap_msgtype( LDAPMessage *res );
int ldap_msgid( LDAPMessage *res );
```

Parameters are as follows:

- **Id**: 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 (0x68)
- 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.
12.18 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.

```c
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.

```c
int ldap_result2error( 
    LDAP *ld, 
    LDAPMessage *res, 
    int freeit 
); 

void ldap_perror( LDAP *ld, const char *msg );
```

Parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>The session handle.</td>
</tr>
<tr>
<td>res</td>
<td>The result of an LDAP operation as returned by <code>ldap_result()</code> or one of the synchronous API operation calls.</td>
</tr>
<tr>
<td>errcodep</td>
<td>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.</td>
</tr>
<tr>
<td>matcheddnp</td>
<td>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 <code>ldap_memfree()</code>.</td>
</tr>
</tbody>
</table>
Lightweight Directory Access Protocol (LDAP) Routines

12.18 Handling Errors and Parsing Results

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 the res parameter is disposed of or not. Pass any non-zero value to have these functions free res after extracting the requested information. This 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.
12.18 Handling Errors and Parsing Results

12.18.1 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.

```c
LDAPMessage *ldap_first_message( LDAP *ld, LDAPMessage *res );
LDAPMessage *ldap_next_message ( LDAP *ld, LDAPMessage *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()`.

12.19 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.

12.19.1 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.

```c
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()`.

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.

### 12.19.2 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.

```c
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.

### 12.19.3 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.

```c
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>The session handle.</td>
</tr>
<tr>
<td>entry</td>
<td>The entry from which to retrieve values, as returned by ldap_first_entry() or ldap_next_entry().</td>
</tr>
<tr>
<td>attr</td>
<td>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, &quot;mail&quot;).</td>
</tr>
<tr>
<td>vals</td>
<td>The values returned by a previous call to ldap_get_values() or ldap_get_values_len().</td>
</tr>
</tbody>
</table>

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.
12.19 Parsing Search Results

12.19.4 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.

```c
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.

12.19.5 Retrieving Controls from an Entry

The ldap_get_entry_controls() function is used to extract LDAP controls from an entry.

```c
int ldap_get_entry_controls(  
    LDAP *ld,  
    LDAPMessage *entry,  
    LDAPControl ***serverctrlsp  
);  
```
12.19 Parsing Search Results

Parameters are as follows:

- **Id**: 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).

12.19.6 Parsing References

The `ldap_parse_reference()` function is used to extract referrals and controls from a `SearchResultReference` message.

```c
int ldap_parse_reference(
    LDAP *ld,
    LDAPMessage *ref,
    char ***referralsp,
    LDAPControl ***serverctrlsp,
    int freeit
);
```

Parameters are as follows:

- **Id**: 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 the ref parameter is disposed of or not. Pass any non-zero value to have these functions free res after extracting the requested information. This 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).

12.20 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.
12.20 Encoded ASN.1 Value Manipulation

12.20.1 Encoding

The following is an example of encoding:

```c
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:

```c
#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`.

```c
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.
12.20 Encoded ASN.1 Value Manipulation

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 be 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.

```c
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).

12.20.1.1 Encoding Example

The following is an example of encoding the following ASN.1 data type:

```asn1
Example1Request ::= SEQUENCE {
    s OCTET STRING, -- must be printable
    val1 INTEGER,
    val2 [0] INTEGER DEFAULT 0
}
```

```c
int encode_example1(char *s, ber_int_t val1, ber_int_t val2,
                    struct berval **bvPtr)
```
Lightweight Directory Access Protocol (LDAP) Routines

12.20 Encoded ASN.1 Value Manipulation

```c
{  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;
}
```

12.20.2 Decoding

The following two symbols are available to applications.

```
#define LBER_ERROR 0xffffffffL
#define LBER_DEFAULT 0xffffffffL
```

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.

```
BerElement *ber_init (struct berval *bv);
```

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:
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.

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.

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.

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.

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.

Bitstring. A char ** argument should be supplied which point to the allocated bits, followed by a ber_len_t * argument, which 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.

Null. No argument is required. The element is simply skipped if it is recognized as a zero-length element. The tag is ignored.

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.

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.

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 \texttt{ber\_skip\_tag()} function is similar to \texttt{ber\_peek\_tag()}, except that the state pointer in the \texttt{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 *\texttt{lenPtr}.

\begin{verbatim}
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);
\end{verbatim}

The \texttt{ber\_first\_element()} and \texttt{ber\_next\_element()} functions are used to traverse a SET, SET OF, SEQUENCE or SEQUENCE OF data value. The \texttt{ber\_first\_element()} function calls \texttt{ber\_skip\_tag()}, stores internal information in *\texttt{lenPtr} and *\texttt{opaquePtr}, and calls \texttt{ber\_peek\_tag()} for the first element inside the constructed value. \texttt{LBER\_DEFAULT} is returned if the constructed value is empty. The \texttt{ber\_next\_element()} function positions the state at the start of the next element in the constructed type. \texttt{LBER\_DEFAULT} is returned if there are no further values.

The \texttt{len} and \texttt{opaque} values should not be used by applications other than as arguments to \texttt{ber\_next\_element()}, as shown in the following example:

\subsection*{12.20.2.1 Decoding Example}

The following is an example of decoding an ASN.1 data type:

\begin{verbatim}
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;
    }
\end{verbatim}
/** *** 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;
        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,"0",&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;
12.21 Sample 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);
}
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.

13.1 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.

13.1.1 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.
13.1 Introduction to LOGINOUT

13.1.2 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 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.

13.1.3 The LOGINOUT Data Flow

Figure 13–1 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.
13.2 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.

13.2.1 LOGINOUT Callout Routines

LOGINOUT calls a different site-provided callout routine at each important step in its execution. Table 13-1 briefly describes the LOGINOUT callouts. See Section 13.4 for detailed descriptions of these routines.

<table>
<thead>
<tr>
<th>Callout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$ICR_AUTHENTICATE</td>
<td>Authenticates the user account at login</td>
</tr>
<tr>
<td>LGI$ICR_CHKRESTRICT</td>
<td>Checks additional security restrictions</td>
</tr>
<tr>
<td>LGI$ICR_DECWINIT</td>
<td>Prepares for interactive contact with DECwindows users</td>
</tr>
<tr>
<td>LGI$ICR_FINISH</td>
<td>Gives site-specific code final control of the login process</td>
</tr>
<tr>
<td>LGI$ICR_IACT_START</td>
<td>Prepares for interactive contact with users who are not using the DECwindows interface</td>
</tr>
<tr>
<td>LGI$ICR_IDENTIFY</td>
<td>Identifies the user at login</td>
</tr>
<tr>
<td>LGI$ICR_INIT</td>
<td>Initializes context variable</td>
</tr>
<tr>
<td>LGI$ICRJOBSTEP</td>
<td>Indicates the start of each step in a batch job</td>
</tr>
<tr>
<td>LGI$ICR_LOGOUT</td>
<td>Prepares for logout</td>
</tr>
</tbody>
</table>

13.2.2 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 13-2 briefly describes the LOGINOUT callback routines. See Section 13.5 for detailed descriptions of these routines.
13.2 LOGINOUT Callouts

Table 13–2  LOGINOUT Callback Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$ICB_ACCTEXPIRED</td>
<td>Checks for account expiration</td>
</tr>
<tr>
<td>LGI$ICB_AUTOLOGIN</td>
<td>Verifies that standard rules for autologin apply</td>
</tr>
<tr>
<td>LGI$ICB_CHECK_PASS</td>
<td>Checks the entered password against the user authorization file (UAF) record</td>
</tr>
<tr>
<td>LGI$ICB_DISUSER</td>
<td>Checks for DISUSER flag</td>
</tr>
<tr>
<td>LGI$ICB_GET_INPUT</td>
<td>Enables interaction with the user</td>
</tr>
<tr>
<td>LGI$ICB_GET_SYSPWD</td>
<td>Checks system password for character-cell interactive logins</td>
</tr>
<tr>
<td>LGI$ICB_MODALHOURS</td>
<td>Checks for restrictions on access modes and access hours</td>
</tr>
<tr>
<td>LGI$ICB_PASSWORD</td>
<td>Generates prompts, reads input, and optionally validates input against system user authorization file (SYSUAF.DAT)</td>
</tr>
<tr>
<td>LGI$ICB_PWD_EXPIRED</td>
<td>Checks for password expiration</td>
</tr>
<tr>
<td>LGI$ICB_USERPROMPT</td>
<td>Prompts for and reads input for character-cell interactive logins</td>
</tr>
<tr>
<td>LGI$ICB_USERPARSE</td>
<td>Parses input buffer data for character-cell interactive logins</td>
</tr>
<tr>
<td>LGI$ICB_VALIDATE</td>
<td>Validates the user name and password against the system user authorization file (SYSUAF.DAT)</td>
</tr>
</tbody>
</table>

13.3 Using Callout Routines

This section describes:

- The calling environment
- The callout routines and how they are organized and activated
- The callout routines interface

Section 13.3.5 contains a sample LOGINOUT program.

13.3.1 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:
### 13.3 Using Callout Routines

#### Return Status Interpretation

<table>
<thead>
<tr>
<th>Return Status</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Access permitted; continue policy checks. Execute next policy image or OpenVMS policy function associated with this callout, if applicable.</td>
</tr>
<tr>
<td>LGI$_SKIPRELATED</td>
<td>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.</td>
</tr>
<tr>
<td>Other</td>
<td>Disallow the login:</td>
</tr>
<tr>
<td></td>
<td>• Perform break-in detection and intrusion evasion, if appropriate.</td>
</tr>
<tr>
<td></td>
<td>• Perform security audit.</td>
</tr>
<tr>
<td></td>
<td>• Allow additional login attempts up to system-specified repeat limit, if appropriate.</td>
</tr>
</tbody>
</table>

**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$SIGNAL. (See the OpenVMS RTL Library (LIB$) 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.

### 13.3.2 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 13–2 shows how a callout routine vector is organized.
13.3 Using Callout Routines

Figure 13–2 Callout Organization

<table>
<thead>
<tr>
<th>Number of Entry Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callout Routine 1</td>
</tr>
<tr>
<td>Callout Routine 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Callout Routine n</td>
</tr>
</tbody>
</table>

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.

13.3.3 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 Compaq 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.

13.3.4 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:

<table>
<thead>
<tr>
<th>LGI$ICR_ENTRY_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$ICR_INIT</td>
</tr>
<tr>
<td>LGI$ICR_JACT_START</td>
</tr>
<tr>
<td>LGI$ICR_DECWINIT</td>
</tr>
<tr>
<td>LGI$ICR_IDENTIFY</td>
</tr>
<tr>
<td>LGI$ICR_AUTHENTICATE</td>
</tr>
<tr>
<td>LGI$ICR_CHKRESTRICT</td>
</tr>
<tr>
<td>LGI$ICR_FINISH</td>
</tr>
<tr>
<td>LGI$ICR_LOGOUT</td>
</tr>
<tr>
<td>LGI$ICR_JOBSTEP</td>
</tr>
</tbody>
</table>

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 13-2). Other offsets are addresses of useful variable information internal to LOGINOUT. These are described in Table 13-3.

### Table 13-3 Useful LOGINOUT Internal Variables

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$A_ICR_CREPRC_FLAGS</td>
<td>PPD_CREPRC_FLAGS controls program flow based on the major job types of PRC$V_BATCH, PRC$V_NETWRK, PRC$V_INTER, and other values such as PRC$V_NOPASSWORD (used for interactive jobs created on logged-in terminals).</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 13–3 (Cont.) Useful LOGINOUT Internal Variables

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$A_ICR_JOB_TYPE</td>
<td>The job type from the JIB (byte). LOGINOUT does the following:</td>
</tr>
<tr>
<td></td>
<td>• Retrieves the job type with a GETJPI during initialization.</td>
</tr>
<tr>
<td></td>
<td>• Modifies it during execution. (Its value may change between the LGI$ICR_INIT and later callouts.)</td>
</tr>
<tr>
<td></td>
<td>• Writes it back into the JIB before exiting.</td>
</tr>
<tr>
<td>For interactive jobs, this flag indicates JIB$C_LOCAL, JIB$C_REMOTE, or JIB$C_DIALUP.</td>
<td></td>
</tr>
<tr>
<td>LGI$A_ICR_SUBPROCESS</td>
<td>The subprocess flag (byte) indicates whether a subprocess is being logged in.</td>
</tr>
<tr>
<td>LGI$A_ICR_TERMINAL_DEV</td>
<td>The terminal device flag (byte).</td>
</tr>
<tr>
<td>LGI$A_ICR_TT_PHYDEVNAM</td>
<td>A descriptor containing the terminal’s physical device name (null if input is not from a terminal).</td>
</tr>
<tr>
<td>LGI$A_ICR_TT_ACCPORNAM</td>
<td>A descriptor containing the terminal’s access port name (null if input is not from a terminal or is from a terminal without an associated access port).</td>
</tr>
<tr>
<td>LGI$A_ICR_CLINAME</td>
<td>A descriptor containing the command language interpreter (CLI) name, parsed from the user name qualifiers. Valid only for interactive jobs.</td>
</tr>
<tr>
<td>LGI$A_ICR_CLITABLES</td>
<td>A descriptor containing the CLI tables, parsed from the user name qualifiers. Valid only for interactive jobs.</td>
</tr>
<tr>
<td>LGI$A_ICR_NCB</td>
<td>A descriptor containing the network control block. Valid only for network jobs.</td>
</tr>
<tr>
<td>LGI$A_ICR_LOGLINK</td>
<td>A longword containing the local link number. Valid only for network jobs and when doing a SET HOST command from a DECnet-Plus remote terminal.</td>
</tr>
<tr>
<td>LGI$A_ICR_REM_NODE_NAM</td>
<td>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.</td>
</tr>
<tr>
<td>LGI$A_ICR_REM_ID</td>
<td>A descriptor containing the remote ID. This may be the user ID on the remote system if the source operating system sends the user name. Otherwise, it is as defined for the source system. Valid only for network jobs and when doing a SET HOST command from a DECnet-Plus remote terminal.</td>
</tr>
<tr>
<td>LGI$A_ICR_UAF_RECORD</td>
<td>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.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 13–3 (Cont.) Useful LOGINOUT Internal Variables

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$A_ICR_INPUT_RAB</td>
<td>A RAB (record access block) that may be used to communicate with an interactive user.</td>
</tr>
<tr>
<td>LGI$A_ICR_AUTOLOGIN</td>
<td>A flag (byte) indicating whether an autologin is being used for this interactive job.</td>
</tr>
<tr>
<td>LGI$A_ICR_USERNAME</td>
<td>A descriptor for handling the user name.</td>
</tr>
<tr>
<td>LGI$A_ICR_PWD1</td>
<td>A descriptor for handling the primary password.</td>
</tr>
<tr>
<td>LGI$A_ICR_PWD2</td>
<td>A descriptor for handling the secondary password.</td>
</tr>
<tr>
<td>LGI$A_ICR_PWDCOUNT</td>
<td>A longword containing the count of passwords expected for this user. Valid only for interactive jobs.</td>
</tr>
</tbody>
</table>
| 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.  
  - 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. |

### 13.3.5 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.

```c
#define LGISCALLOUT_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 LGISCALLOUT_EXAMPLE.C
 **
 ** This program can be linked with the following example command procedure
 **
 ** $ LINK/SHARE=LGISCALLOUT_EXAMPLE SYS$INPUT/OPT
 **
 ** LGISCALLOUT_EXAMPLE.OBJ
```
** SYMBOL VECTOR=(LGI$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 SYSSSHARE, define it as follows:
** $ DEFINE/SYSTEM/EXEC LGISCALLOUT_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 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_PASSWORD) ();
int (*LGI$ICB_USERPARSE) ();
int (*LGI$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_USERPROMPT;
char *LGI$A_ICR_USERPROMPT;
char *LGI$A_ICR_TERMINAL_DEV;
struct dsc$descriptor_s *LGI$A_ICR_TT_PHYDEVNAM;
struct dsc$descriptor_s *LGI$A_ICR_TT_ACCPRNAM;
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$A_ICR_PWDCOUNT;
short *LGI$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_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);
  }

13.4 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.
L GI$ICR_AUTHENTICATE—Authenticate the Password

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.

ccontext
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_RETRRY_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

Perform standard password prompting and validation.
LGI$ICR_CHKRESTRICT—Check Access Restrictions

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.

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 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:

<table>
<thead>
<tr>
<th>Access Restriction</th>
<th>Callback Routine Used to Check Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account expiration</td>
<td>LGI$ICB_ACCTEXPIRED</td>
</tr>
<tr>
<td>Password expiration</td>
<td>LGI$ICB_PWDEXPIRED</td>
</tr>
<tr>
<td>Account disabled</td>
<td>LGI$ICB_DISUSER</td>
</tr>
<tr>
<td>Access modes and times</td>
<td>LGI$ICB_MODALHOURS</td>
</tr>
</tbody>
</table>
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

Check password expiration, check DISUSER flag, check account expiration, and check restrictions on access time.
LGI$ICR_DECWINIT—DECwindows Initialization

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.

Arguments

arg_vector

OpenVMS usage: vector
type: vector_longword_unsigned
access: modify
mechanism: by reference

Vector containing site-specified callbacks and login information.

calendar

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Access permitted; continue policy checks.</td>
</tr>
<tr>
<td>LGI$_SKIPRELATED</td>
<td>Not supported. Returning this status will cause unpredictable behavior.</td>
</tr>
<tr>
<td>Other</td>
<td>Disallow the login.</td>
</tr>
</tbody>
</table>

Associated OpenVMS Policy Function

Create dialog box, read user name and password, and call the identification and authentication routines.
LGI$ICR_FINISH—Final Site Action

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.

Arguments

arg_vector

OpenVMS usage:  vector  
type:  vector_longword_unsigned  
access:  modify  
mechanism:  by reference

Vector containing callbacks and login information.

countext

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**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$_SKIPRELATED</td>
<td>Access permitted; omit calls to the LGI$ICR_FINISH callout routine in subsequent images.</td>
</tr>
</tbody>
</table>

**Associated OpenVMS Policy Function**

None.
LGI$ICR_IACT_START—Character-Cell Initialization

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.

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

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—Identify the User

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.

Arguments

arg_vector
OpenVMS usage: vector
type: vector_longword_unsigned
access: modify
mechanism: by reference

Vector containing callbacks and useful login information.

collection
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.
LOGINOUT Routines
LGI$ICR_IDENTIFY

– 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

Perform standard OpenVMS user name prompting and parsing.
LGI$ICR_INIT—Initialization Callout Routine

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.

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

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 Function

None.
LGI$ICR_JOBSTEP—Batch Job Step

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.

Arguments

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 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_J OBSTEP.

Typical Condition Values

LGI$_SKIPRELATED or any error value  Access permitted; omit calls to the LGI$ICR_J OBSTEP callout routine in subsequent images.

Associated OpenVMS Policy Function

None.
LGI$ICR_LOGOUT—Installation Logout

The LGI$ICR_LOGOUT callout routine permits the site callout images to respond to the DCL command LOGOUT.

Note

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.

Arguments

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 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$_SKIRELATED or any error value Access permitted; omit calls to the LGI$ICR_
LOGOUT callout routine in subsequent images.

Associated OpenVMS Policy Function

None.
13.5 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—Account Expiration

The LGI$ICB_ACCTEXPIRED callback routine checks for account expiration.

Format

LGI$ICB_ACCTEXPIRED

Returns

No value. Does not return on failure.

Arguments

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.
LGISICB_AUTOLOGIN—Check for Autologin

The site may use the LGISICB_AUTOLOGIN callback routine to determine whether the standard OpenVMS autologin functionality applies for this terminal.

Format

LGISICB_AUTOLOGIN

Returns

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

True (logical 1) if autologin enabled; 0 otherwise.

Arguments

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 LGISICB_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—Check Password

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.

Arguments

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—Check for Disabled User Account

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:

<table>
<thead>
<tr>
<th>If Value of Action Is...</th>
<th>Then...</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGI$DISUSER_STOP</td>
<td>Do not return on error.</td>
</tr>
<tr>
<td>LGI$DISUSER_RETURN</td>
<td>Return LGI$DISUSER or SS$NORMAL.</td>
</tr>
</tbody>
</table>

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—Get User 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.

Arguments

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:

<table>
<thead>
<tr>
<th>Flags</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Normal error message.</td>
</tr>
<tr>
<td>1</td>
<td>LOGINOUT exits quietly.</td>
</tr>
<tr>
<td>2</td>
<td>Normal error message; however, the callback routine returns control to</td>
</tr>
<tr>
<td></td>
<td>the caller rather than exiting on timeout (timeout status is in RAB).</td>
</tr>
</tbody>
</table>

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

No return value. Examine status in RAB to determine the results of the read operation.
LGI$ICB_GET_SYSPWD—Get System Password

The LGI$ICB_GET_SYSPWD callback routine validates the system password.

Format

LGI$ICB_GET_SYSPWD

Returns

No value. Does not return on failure.

Arguments

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—Perform Access Checks

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.

Arguments

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—Produce Password Prompt

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.

Arguments

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Prompt for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Primary password and validate it</td>
</tr>
<tr>
<td>1</td>
<td>Secondary password and validate it</td>
</tr>
<tr>
<td>–1</td>
<td>Primary password but do not validate it</td>
</tr>
<tr>
<td>–2</td>
<td>Secondary password but do not validate it</td>
</tr>
<tr>
<td>–3</td>
<td>Arbitrary 32-character value returned to buffer</td>
</tr>
</tbody>
</table>

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 \texttt{password\_number} argument value is \textbf{-3}.

\textbf{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 \texttt{password\_number} argument:

<table>
<thead>
<tr>
<th>Value of Password_Number Argument</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or \textbf{-1}</td>
<td>\texttt{LGI$A_ICR_PWD1}</td>
</tr>
<tr>
<td>1 or \textbf{-2}</td>
<td>\texttt{LGI$A_ICR_PWD2}</td>
</tr>
<tr>
<td>\textbf{-3}</td>
<td>\texttt{buffer} argument</td>
</tr>
</tbody>
</table>

\textbf{Note}

This routine will do overstriking, if necessary, to support echo local terminals. See the \textit{OpenVMS Programming Concepts Manual} for more information about echo terminals.

\textbf{Condition Values Returned}

- \texttt{SS\$\_NORMAL} Success.
- \texttt{LGI\$\_INVPWD} Password check failed.
- \texttt{LGI\$\_NOSUCHUSER} No UAF record found.
LGI$ICB_PWDEXPIRED—Password Expiration

The LGI$ICB_PWDEXPIRED callback routine checks for password expiration.

Format

LGI$ICB_PWDEXPIRED

Returns

No value. Does not return on failure.

Arguments

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.
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

True (1) if successful; otherwise, any condition code returned by CLI$PARSE.
**LGI$ICB_USERPROMPT—Prompt for Username**

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 "\r\n". 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—Validate User Name and Passwords

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.

Arguments

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
LOGINOUT Routines
LGI$ICB_VALIDATE

Condition Values Returned
Success, or an error indicating the reason for the failure.
Mail Utility (MAIL) 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 OpenVMS User’s Manual.

14.1 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.

Figure 14–1 illustrates the format of a mail message.

Figure 14–1 Standard Message Format

From: MYNODE::USER "The Celestial Navigator" 1
To: NODE::J_DOE 2
CC: USER 3
Subj: Perseids ... 4

(continued on next page)
Figure 14–1 (Cont.) Standard Message Format

Get ready. Tuesday of this week (August 12th), one 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.

The parts of a message are as follows:

- **Header information**
  1. From: field specifies the sender and an optional personal name string
  2. To: field specifies the direct addressee
  3. CC: field specifies the carbon copy addressee
  4. Subj: field specifies the topic of the message

- **Bodypart**
  5. First line of the bodypart
  6. Last line of the bodypart

**External Message Identification Number**

In addition, the file name of an external message uses the following format:

MAIL$nnnnnnnnnnnnnnnn.MAI

where n ... n is the external message identification number.

14.2 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 14–1 describes the three default mail folders.

<table>
<thead>
<tr>
<th>Folder</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEWMAIL</td>
<td>Newly received, unread messages</td>
</tr>
<tr>
<td>MAIL</td>
<td>Messages that have been read and not deleted</td>
</tr>
<tr>
<td>WASTEBASKET</td>
<td>Messages designated for deletion</td>
</tr>
</tbody>
</table>

You can also place messages in any user-defined mail folder and file.
14.3 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.

14.4 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 14–2 summarizes information contained in a user profile entry.

<table>
<thead>
<tr>
<th>Field</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory</td>
<td>Default MAIL subdirectory</td>
</tr>
<tr>
<td>Form</td>
<td>Default print form</td>
</tr>
<tr>
<td>Forwarding address</td>
<td>Forwarding address</td>
</tr>
<tr>
<td>Personal name string</td>
<td>User-specified character string included in the message header</td>
</tr>
<tr>
<td>Queue name</td>
<td>Default print queue name</td>
</tr>
<tr>
<td>Flags</td>
<td></td>
</tr>
<tr>
<td>Automatic purge</td>
<td>Purging of the wastebasket folder on exiting</td>
</tr>
<tr>
<td>CC: prompt</td>
<td>Carbon copy prompt</td>
</tr>
<tr>
<td>Copy self forward</td>
<td>Copy to self when forwarding a message</td>
</tr>
<tr>
<td>Copy self reply</td>
<td>Copy to self when replying to a message</td>
</tr>
<tr>
<td>Copy self send</td>
<td>Copy to self when sending a message</td>
</tr>
<tr>
<td>Signature file</td>
<td>Text file that is automatically appended to the end of the body of a mail message</td>
</tr>
</tbody>
</table>

Both the callable interface and the user interface access the user profile database to determine default processing characteristics.

14.5 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 14–3.
14.5 Mail Utility Processing Contexts

Table 14–3  Levels of Mail Utility Processing

<table>
<thead>
<tr>
<th>Context</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail file</td>
<td>Mail files and folders</td>
</tr>
<tr>
<td>Message</td>
<td>Mail files, folders, and messages</td>
</tr>
<tr>
<td>Send</td>
<td>Messages</td>
</tr>
<tr>
<td>User</td>
<td>User profile database</td>
</tr>
</tbody>
</table>

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.

### 14.5.1 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 14–4 lists Mail utility routines according to context.

Table 14–4  Callable Mail Utility Routines

<table>
<thead>
<tr>
<th>Context</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail file</td>
<td>MAIL$MAILFILE_BEGIN</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_CLOSE</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_COMPRESS</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_END</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_INFO_FILE</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_MODIFY</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_OPEN</td>
</tr>
<tr>
<td></td>
<td>MAIL$MAILFILE_PURGE_WASTE</td>
</tr>
</tbody>
</table>

(continued on next page)
### 14.5 Mail Utility Processing Contexts

#### Table 14–4 (Cont.) Callable Mail Utility Routines

<table>
<thead>
<tr>
<th>Context</th>
<th>Routine</th>
</tr>
</thead>
</table>
| Message | MAIL$MESSAGE_BEGIN  
          | MAIL$MESSAGE_COPY  
          | MAIL$MESSAGE_DELETE  
          | MAIL$MESSAGE_END  
          | MAIL$MESSAGE_GET  
          | MAIL$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 |

#### 14.5.2 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.

#### 14.6 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:

```plaintext
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.
14.6.1 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.

14.6.2 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.

14.6.2.1 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:

<table>
<thead>
<tr>
<th>Item Descriptor</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Each descriptor points to a buffer or file from which Mail reads data.</td>
</tr>
<tr>
<td>Output</td>
<td>Each descriptor points to a buffer or file to which Mail writes data.</td>
</tr>
</tbody>
</table>

An item descriptor is a data structure consisting of three longwords as described in Figure 14–2.

Figure 14–2 Item Descriptor

```
  31  15   0
```

<table>
<thead>
<tr>
<th>Item code</th>
<th>Buffer length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer address</td>
<td></td>
</tr>
<tr>
<td>Return length address</td>
<td></td>
</tr>
</tbody>
</table>
Item descriptor fields are described as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item code</td>
<td>Specifies an action the routine is to perform.</td>
</tr>
<tr>
<td>Buffer length</td>
<td>Specifies the length in bytes of an input or output buffer.</td>
</tr>
<tr>
<td>Buffer address</td>
<td>Specifies the address of the input or output buffer.</td>
</tr>
<tr>
<td>Return length address</td>
<td>Depends on the type of item code specified:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Not used; specify 0.</td>
</tr>
<tr>
<td>Output</td>
<td>Address of a longword that receives the length of the result.</td>
</tr>
</tbody>
</table>

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 14–10 and 14–11.

14.6.2.2 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.

14.6.2.3 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.

14.6.2.4 Terminating an Item List

Terminate an item list with a null item descriptor. Assign the value 0 to each field in the item descriptor.

14.6.3 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 14–5 summarizes the types of action routines and the contexts in which they are used.

### Table 14–5 Types of Action Routines

<table>
<thead>
<tr>
<th>Context</th>
<th>Routine</th>
<th>Action Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail file</td>
<td>MAIL$MAILFILE_INFO_FILE</td>
<td>Provides information about folder and mail files.</td>
</tr>
<tr>
<td>Message</td>
<td>MAIL$MESSAGE_COPY</td>
<td>Copies messages between files and folders.</td>
</tr>
<tr>
<td>Send</td>
<td>MAIL$SEND_MESSAGE</td>
<td>Success and error results; sends a text file to an existing address list.</td>
</tr>
</tbody>
</table>

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.

### 14.7 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 14–6 summarizes each mail file routine and its function.

### Table 14–6 Mail File Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$MAILFILE_BEGIN</td>
<td>Initiates mail file processing</td>
</tr>
<tr>
<td>MAIL$MAILFILE_CLOSE</td>
<td>Closes a mail file</td>
</tr>
<tr>
<td>MAIL$MAILFILE_COMPRESS</td>
<td>Compresses a mail file</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 14–6 (Cont.) Mail File Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$MAILFILE_END</td>
<td>Terminates mail file processing</td>
</tr>
<tr>
<td>MAIL$MAILFILE_INFO_FILE</td>
<td>Obtains information about the mail file</td>
</tr>
<tr>
<td>MAIL$MAILFILE_MODIFY</td>
<td>Changes the wastebasket folder name and the default mail file name</td>
</tr>
<tr>
<td>MAIL$MAILFILE_OPEN</td>
<td>Opens a mail file</td>
</tr>
<tr>
<td>MAIL$MAILFILE_PURGE_WASTE</td>
<td>Purges a mail file</td>
</tr>
</tbody>
</table>

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.

14.7.1 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.

14.7.1.1 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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>User’s disk and directory</td>
<td>Retrieved from the user authorization file (UAF)</td>
</tr>
<tr>
<td>MAIL subdirectory</td>
<td>Retrieved from the user profile entry</td>
</tr>
<tr>
<td>Mail file name and type</td>
<td>MAIL.MAI</td>
</tr>
</tbody>
</table>

### 14.7.1.2 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.
14.7.2 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.

14.7.3 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.

14.7.3.1 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.

14.7.3.2 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.

14.8 Message Context

Message context processing involves manipulating existing messages as well as creating and deleting folders and mail files. Table 14–7 summarizes routines used in the message context.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$MESSAGE_BEGIN</td>
<td>Initiates message processing</td>
</tr>
<tr>
<td>MAIL$MESSAGE_COPY</td>
<td>Copies messages</td>
</tr>
<tr>
<td>MAIL$MESSAGE_DELETE</td>
<td>Deletes messages</td>
</tr>
<tr>
<td>MAIL$MESSAGE_END</td>
<td>Terminates message processing</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 14–7 (Cont.) Message Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$MESSAGE_GET</td>
<td>Retrieves a message</td>
</tr>
<tr>
<td>MAIL$MESSAGE_INFO</td>
<td>Obtains information about a specified message</td>
</tr>
<tr>
<td>MAIL$MESSAGE_MODIFY</td>
<td>Identifies a message as replied, new, or marked</td>
</tr>
<tr>
<td>MAIL$MESSAGE_SELECT</td>
<td>Selects a message or messages from the currently open mail file</td>
</tr>
</tbody>
</table>

### 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.

#### 14.8.1 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.

14.8.2 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 $SNDJ BC system service to manage print jobs and define queue characteristics.

14.8.3 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$V_replied</td>
<td>Flagged as answered</td>
</tr>
<tr>
<td>MAIL$V_marked</td>
<td>Flagged for display purposes</td>
</tr>
</tbody>
</table>

14.8.4 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.
Mail Utility (MAIL) Routines

14.8 Message Context

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—it's folder name.

14.8.4.1 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.

14.8.4.2 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.

14.8.4.3 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.

14.8.5 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.

14.9 Send Context

Send context processing involves creating and sending new and existing messages. Table 14–8 summarizes send routines.
Table 14–8 Send Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$SEND_ABORT</td>
<td>Aborts a send operation</td>
</tr>
<tr>
<td>MAIL$SEND_ADD_ADDRESS</td>
<td>Adds an addressee to the address list</td>
</tr>
<tr>
<td>MAIL$SEND_ADD_ATTRIBUTE</td>
<td>Constructs the message header</td>
</tr>
<tr>
<td>MAIL$SEND_ADD_BODYPART</td>
<td>Constructs the body of the message</td>
</tr>
<tr>
<td>MAIL$SEND_BEGIN</td>
<td>Initiates send processing</td>
</tr>
<tr>
<td>MAIL$SEND_END</td>
<td>Terminates send processing</td>
</tr>
<tr>
<td>MAIL$SEND_MESSAGE</td>
<td>Sends a message</td>
</tr>
</tbody>
</table>

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.

14.9.1 Sending New Messages
You can send new or existing messages to yourself and other users.

14.9.1.1 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

14.9.1.1.1 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.

14.9.1.2 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 body part with one call to MAIL$SEND_ADD_BODYPART by specifying its file name.
14.9.1.2 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.

14.9.2 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:

<table>
<thead>
<tr>
<th>Task</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open a mail file.</td>
<td>MAIL$MAILFILE_OPEN</td>
</tr>
<tr>
<td>Select the message.</td>
<td>MAIL$MESSAGE_SELECT</td>
</tr>
<tr>
<td>Retrieve the message.</td>
<td>MAIL$MESSAGE_GET</td>
</tr>
<tr>
<td>Construct the message.</td>
<td>MAIL$SEND_ADD_ATTRIBUTE</td>
</tr>
<tr>
<td></td>
<td>Construct the message header.</td>
</tr>
<tr>
<td></td>
<td>MAIL$SEND_ADD_BODYPART</td>
</tr>
<tr>
<td>Create an address list.</td>
<td>MAIL$SEND_ADD_ADDRESS</td>
</tr>
<tr>
<td>Send the message.</td>
<td>MAIL$SEND_MESSAGE</td>
</tr>
</tbody>
</table>

14.9.3 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.

14.9.3.1 Success Action Routines

A success action routine performs a task upon successful completion of a send operation.

14.9.3.2 Error Handling Routines

An error action routine is a user-written error handler that signals error conditions during a send operation.
14.9.3.3 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.

14.10 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 14–9 summarizes the user context routines.

Table 14–9 User Profile Context Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$USER_BEGIN</td>
<td>Initiates user profile context</td>
</tr>
<tr>
<td>MAIL$USER_DELETE_INFO</td>
<td>Deletes a user profile entry</td>
</tr>
<tr>
<td>MAIL$USER_END</td>
<td>Terminates user profile context</td>
</tr>
<tr>
<td>MAIL$USER_GET_INFO</td>
<td>Retrieves information about a user from the user profile</td>
</tr>
<tr>
<td>MAIL$USER_SET_INFO</td>
<td>Adds or modifies a user profile entry</td>
</tr>
</tbody>
</table>

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.

14.10.1 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).
14.10 User Profile Context

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.

14.10.1.1 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.

14.10.1.2 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:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Privilege</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$_USER_SET_INFO</td>
<td>SYSPRV</td>
<td>Modifies another user’s profile entry</td>
</tr>
<tr>
<td>MAIL$_USER_GET_INFO</td>
<td>SYSNAM or SYSPRV</td>
<td>Retrieves information about another user</td>
</tr>
</tbody>
</table>

14.11 Input Item Codes
Input item codes direct the called routine to read data from a buffer or file and perform a task. Table 14–10 summarizes input item codes.
<table>
<thead>
<tr>
<th>Item Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mail File Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_MAILFILE_DEFAULT_NAME</td>
<td>Specifies the location (disk and directory) of the default mail file MAIL.MAI.</td>
</tr>
<tr>
<td>MAIL$_MAILFILE_FOLDER_ROUTINE</td>
<td>Displays folder names within a specified mail file.</td>
</tr>
<tr>
<td>MAIL$_MAILFILE_FULL_CLOSE</td>
<td>Requests that the wastebasket folder be purged and that a convert/reclaim operation be performed, if necessary.</td>
</tr>
<tr>
<td>MAIL$_MAILFILE_NAME</td>
<td>Specifies the name of a mail file to be opened.</td>
</tr>
<tr>
<td>MAIL$_MAILFILE_RECLAIM</td>
<td>Overrides the deleted bytes threshold and requests a reclaim operation.</td>
</tr>
<tr>
<td>MAIL$_MAILFILE_USER_DATA</td>
<td>Passes a longword of user context data to an action routine.</td>
</tr>
<tr>
<td>MAIL$_MAILFILE_WASTEBASKET_NAME</td>
<td>Specifies a new name for the wastebasket in a specified mail file.</td>
</tr>
<tr>
<td><strong>Message Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_MESSAGE_AUTO_NEWMAIL</td>
<td>Places newly read messages in the Mail folder automatically.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_BACK</td>
<td>Returns the first record of the preceding message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_BEFORE</td>
<td>Selects a message before a specified date.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_CC_SUBSTRING</td>
<td>Specifies a character string that must match a node or user name substring in the CC: field of the specified message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_CONTINUE</td>
<td>Returns the next text record of the current message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_DEFAULT_NAME</td>
<td>Specifies the default mail file specification.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_DELETE</td>
<td>Deletes a message in the current folder after the message has been copied to a new folder.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FILE_ACTION</td>
<td>Specifies a user-written routine that is called if a mail file is to be created.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FILE_CTX</td>
<td>Specifies mail file context received from MAIL$_MAILFILE_BEGIN.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FILENAME</td>
<td>Specifies the name of a mail file to which the message is to be moved.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FOLDER_ACTION</td>
<td>Specifies a user-written routine that is called if a folder is to be created.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FLAGS</td>
<td>Specifies MAIL system flags to use when selecting messages.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FLAGS_MBZ</td>
<td>Specifies MAIL system flags that must be zero.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FOLDER</td>
<td>Specifies the name of the target folder for moving messages.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_FROM_SUBSTRING</td>
<td>Specifies a character string that must match a node or user name substring in the From: field of the specified message.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 14–10 (Cont.) Input Item Codes

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_MESSAGE_ID</td>
<td>Specifies the message identification number of the message on which an operation is to be performed.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_NEXT</td>
<td>Returns the first record of the message following the current message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_SINCE</td>
<td>Selects a message received on or after a specified date.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_SUBJ_SUBSTRING</td>
<td>Specifies a character string that must match a node or user name substring in the Subject: field of the specified message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_TO_SUBSTRING</td>
<td>Specifies a character string that must match a substring in the To: field of the specified message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_USER_DATA</td>
<td>Specifies a longword to be passed to the folder and mail file action routines.</td>
</tr>
<tr>
<td><strong>Send Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_SEND_CC_LINE</td>
<td>Specifies the CC: field text.</td>
</tr>
<tr>
<td>MAIL$_SEND_DEFAULT_NAME</td>
<td>Specifies the default file specification of a text file to be opened.</td>
</tr>
<tr>
<td>MAIL$_SEND_ERROR_ENTRY</td>
<td>Specifies a user-written routine to process errors that occur during a send operation.</td>
</tr>
<tr>
<td>MAIL$_SEND_FID</td>
<td>Specifies the file identifier.</td>
</tr>
<tr>
<td>MAIL$_SEND_FILENAME</td>
<td>Specifies the input file specification of a text file to be opened.</td>
</tr>
<tr>
<td>MAIL$_SEND_FROM_LINE</td>
<td>Specifies the From: field text.</td>
</tr>
<tr>
<td>MAIL$_SEND_PERS_NAME</td>
<td>Specifies the personal name string.</td>
</tr>
<tr>
<td>MAIL$_SEND_NO_PERS_NAME</td>
<td>Specifies that no personal string be used.</td>
</tr>
<tr>
<td>MAIL$_SEND_RECORD</td>
<td>Specifies the descriptor of a text record to be added to the body of a message.</td>
</tr>
<tr>
<td>MAIL$_SEND_SIGFILE</td>
<td>Specifies a full OpenVMS file specification of the signature file to be used in the message.</td>
</tr>
<tr>
<td>MAIL$_SEND_NO_SIGFILE</td>
<td>Specifies that no signature file be used.</td>
</tr>
<tr>
<td>MAIL$_SEND_SUBJECT</td>
<td>Specifies the Subject: field text.</td>
</tr>
<tr>
<td>MAIL$_SEND_SUCCESS_ENTRY</td>
<td>Specifies a user-written routine to process successfully completed events during a send operation.</td>
</tr>
<tr>
<td>MAIL$_SEND_TO_LINE</td>
<td>Specifies the To: field text.</td>
</tr>
<tr>
<td>MAIL$_SEND_USER_DATA</td>
<td>Specifies a longword passed to the send action routines.</td>
</tr>
<tr>
<td>MAIL$_SEND_USERNAME</td>
<td>Adds a specified user name to the address list.</td>
</tr>
<tr>
<td>MAIL$_SEND_USERNAME_TYPE</td>
<td>Specifies the type of user name added to the address list.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 14–10 (Cont.) Input Item Codes

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$_USER_CREATE_IF</td>
<td>Creates a user profile entry.</td>
</tr>
<tr>
<td>MAIL$_USER_FIRST</td>
<td>Returns information about the first user in the user profile database.</td>
</tr>
<tr>
<td>MAIL$_USER_NEXT</td>
<td>Returns information about the next user in the user profile database.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_AUTO_PURGE</td>
<td>Sets the automatic purge flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_AUTO_PURGE</td>
<td>Clears the automatic purge flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_CC_PROMPT</td>
<td>Sets the CC prompt flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_CC_PROMPT</td>
<td>Clears the CC prompt flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_COPY_FORWARD</td>
<td>Sets the copy self forward flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_COPY_FORWARD</td>
<td>Clears the copy self forward flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_COPY_REPLY</td>
<td>Sets the copy self reply flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_COPY_REPLY</td>
<td>Clears the copy self reply flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_COPY_SEND</td>
<td>Sets the copy self send flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_COPY_SEND</td>
<td>Clears the copy self send flag.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_EDITOR</td>
<td>Specifies the default editor.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_EDITOR</td>
<td>Clears the default editor field.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_FORM</td>
<td>Specifies the default print form string.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_FORM</td>
<td>Clears the default print form field.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_FORWARDING</td>
<td>Specifies the forwarding address string.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_FORWARDING</td>
<td>Clears the forwarding address field.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NEW_MESSAGES</td>
<td>Specifies the new messages count.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_PERSONAL_NAME</td>
<td>Specifies the personal name string.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_PERSONAL_NAME</td>
<td>Clears the personal name field.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_QUEUE</td>
<td>Specifies the default print queue name string.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_QUEUE</td>
<td>Clears the default print queue name field.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_SIGFILE</td>
<td>Specifies a signature file specification for the specified user.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_SIGFILE</td>
<td>Clears a signature file field for the specified user.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_SUB_DIRECTORY</td>
<td>Specifies a MAIL subdirectory.</td>
</tr>
<tr>
<td>MAIL$_USER_SET_NO_SUB_DIRECTORY</td>
<td>Clears the MAIL subdirectory field.</td>
</tr>
<tr>
<td>MAIL$_USER_USERNAME</td>
<td>Points to the user name string to specify the user profile entry to be modified.</td>
</tr>
</tbody>
</table>

### 14.12 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 14–11 summarizes output item codes.
### Table 14–11 Output Item Codes

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mail File Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL_MAILFILE_INDEXED</td>
<td>Determines whether the mail file format is indexed.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_DIRECTORY</td>
<td>Returns the mail file subdirectory specification to the caller.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_RESULTSPEC</td>
<td>Returns the result mail file specification.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_WASTEBASKET</td>
<td>Returns the wastebasket folder name for the specified file.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_DELETED_BYTES</td>
<td>Returns the number of deleted bytes in a specified mail file.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_MESSAGES_DELETED</td>
<td>Returns the number of deleted messages.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_DATA_RECLAIM</td>
<td>Returns the number of data buckets reclaimed.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_DATA_SCAN</td>
<td>Returns the number of data buckets scanned.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_INDEX_RECLAIM</td>
<td>Returns the number of index buckets reclaimed.</td>
</tr>
<tr>
<td>MAIL_MAILFILE_TOTAL_RECLAIM</td>
<td>Returns the total number of bytes reclaimed.</td>
</tr>
<tr>
<td><strong>Message Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL_MESSAGE_BINARY_DATE</td>
<td>Returns the date and time received as a binary value.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_CC</td>
<td>Returns the text in the CC: field of the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_CURRENT_ID</td>
<td>Returns the message identification number of the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_DATE</td>
<td>Returns the message creation date string.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_EXTID</td>
<td>Returns the external message identification number of the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_FILE_CREATED</td>
<td>Returns the value of the mail file created flag.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_FOLDER_CREATED</td>
<td>Returns the value of the folder created flag.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_FROM</td>
<td>Returns the text in the From: field of the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_RECORD</td>
<td>Returns a record from the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_RECORD_TYPE</td>
<td>Returns the record type.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_REPLY_PATH</td>
<td>Returns the reply path.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_RESULTSPEC</td>
<td>Returns the resultant mail file specification.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_RETURN_FLAGS</td>
<td>Returns the MAIL system flag value of the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_SELECTED</td>
<td>Returns the number of selected messages.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_SENDER</td>
<td>Returns the name of the sender of the current message.</td>
</tr>
<tr>
<td>MAIL_MESSAGE_SIZE</td>
<td>Returns the size in records of the current message. (continued on next page)</td>
</tr>
</tbody>
</table>
Table 14–11 (Cont.) Output Item Codes

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail Utility (MAIL) Routines</td>
<td></td>
</tr>
<tr>
<td><strong>Message Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_MESSAGE_SUBJECT</td>
<td>Returns the text in the Subject: field of the specified message.</td>
</tr>
<tr>
<td>MAIL$_MESSAGE_TO</td>
<td>Returns the text in the To: field of the specified message.</td>
</tr>
<tr>
<td><strong>Send Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_SEND_COPY_FORWARD</td>
<td>Returns the value of the caller’s copy forward flag.</td>
</tr>
<tr>
<td>MAIL$_SEND_COPY_REPLY</td>
<td>Returns the value of the caller’s copy reply flag.</td>
</tr>
<tr>
<td>MAIL$_SEND_COPY_SEND</td>
<td>Returns the value of the caller’s copy send flag.</td>
</tr>
<tr>
<td>MAIL$_SEND_RESULTSPEC</td>
<td>Returns the resultant file specification of the file to be sent.</td>
</tr>
<tr>
<td>MAIL$_SEND_USER</td>
<td>Returns the process owner’s user name.</td>
</tr>
<tr>
<td><strong>User Context</strong></td>
<td></td>
</tr>
<tr>
<td>MAIL$_USER_AUTO_PURGE</td>
<td>Returns the value of the automatic purge mail flag.</td>
</tr>
<tr>
<td>MAIL$_USER_CAPTIVE</td>
<td>Returns the value of the UAF captive flag.</td>
</tr>
<tr>
<td>MAIL$_USER_CC_PROMPT</td>
<td>Returns the value of the CC prompt flag.</td>
</tr>
<tr>
<td>MAIL$_USER_COPY_FORWARD</td>
<td>Returns the value of the copy self forward flag.</td>
</tr>
<tr>
<td>MAIL$_USER_COPY_REPLY</td>
<td>Returns the value of the copy self reply flag.</td>
</tr>
<tr>
<td>MAIL$_USER_COPY_SEND</td>
<td>Returns the value of the copy self send flag.</td>
</tr>
<tr>
<td>MAIL$_USER_EDITOR</td>
<td>Returns the name of the default editor.</td>
</tr>
<tr>
<td>MAIL$_USER_FORM</td>
<td>Returns the default print form string.</td>
</tr>
<tr>
<td>MAIL$_USER_FORWARDING</td>
<td>Returns the forwarding address string.</td>
</tr>
<tr>
<td>MAIL$_USER_FULL_DIRECTORY</td>
<td>Returns the complete directory path of the mail file subdirectory.</td>
</tr>
<tr>
<td>MAIL$_USER_NEW_MESSAGES</td>
<td>Returns the new message count.</td>
</tr>
<tr>
<td>MAIL$_USER_PERSONAL_NAME</td>
<td>Returns the personal name string.</td>
</tr>
<tr>
<td>MAIL$_USER_QUEUE</td>
<td>Returns the default queue name string.</td>
</tr>
<tr>
<td>MAIL$_USER_RETURN_USERNAME</td>
<td>Returns the user name string.</td>
</tr>
<tr>
<td>MAIL$_USER_SIGFILE</td>
<td>Returns the default signature file specification.</td>
</tr>
<tr>
<td>MAIL$_USER_SUB_DIRECTORY</td>
<td>Returns the subdirectory specification.</td>
</tr>
</tbody>
</table>

14.13 Using the MAIL Routines: Examples

This section provides examples of using the MAIL routines in various programming scenarios including the following:

- Example 14–1 is a C program that sends a Mail message to another user.
- Example 14–2 is a C program that displays a user’s folders and returns how many messages are in each folder.
- Example 14–3 is a C program that displays fields in the user’s Mail profile.
Example 14–1 Sending a File

```c
/* send_message.c */
#include <stdio>
#include <descrip>
#include <ssdef>
#include <maildef>
#include <nam>

typedef struct itmlst
{
  short buffer_length;
  short item_code;
  long buffer_address;
  long return_length_address;
} ITMLST;

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);
}

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]
    ;
  long resultspeclen;
  int
    status = SS$NORMAL,
    file_len = 0,
    subject_line_len = 0,
    to_user_len = 0
    ;
```
Example 14–1 (Cont.) Sending a File

```c
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$C_MAXRSS) - 1] = '\0';
    address_itmlst[0].buffer_length = strlen(to_user);
    address_itmlst[0].buffer_address = to_user;
    status = mail$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].buffer_length = strlen(to_user);
    attribute_itmlst[0].buffer_address = to_user;
    /* Subject: line */
    attribute_itmlst[1].buffer_length = strlen(subject_line);
    attribute_itmlst[1].buffer_address = 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$C_MAXRSS) - 1] = '\0';
    bodypart_itmlst[0].buffer_length = strlen(file);
    bodypart_itmlst[0].buffer_address = file;
    status = mail$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);
```

(continued on next page)
Example 14–1 (Cont.) Sending a File

/* Send the message */
status = mail$send_message(&send_context, nulllist, nulllist);
if (status != SS$NORMAL)
    return(status);

/* Done processing with the SEND context */
status = mail$send_end(&send_context, nulllist, nulllist);
if (status != SS$NORMAL)
    return(status);
return (status);

Example 14–2 shows a C program that displays folders.

Example 14–2 Displaying Folders

/* show_folders.c */

#include <stdio>
#include <descrip>
#include <ctype>
#include <ssdef>
#include <maildef>

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);
}

(continued on next page)
Example 14–2 (Cont.) Displaying Folders

```
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$MESSAGE_FILE_CTX,&file_context,0},  
    {0,0,0,0}},  
mailfile_info_itmlst[] = {  
    {4,MAIL$MAILFILE_FOLDER_ROUTINE,folder_routine,0},  
    {4,MAIL$MAILFILE_USER_DATA,&list,0},  
    {0,0,0,0}},  
message_select_in_itmlst[] = {  
    {0,MAIL$MESSAGE_FOLDER,0,0},  
    {0,0,0,0}},  
message_select_out_itmlst[] = {  
    {sizeof(messages_selected),MAIL$MESSAGE_SELECTED,&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 = 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++;  
            }  
        }
    }
}
```
Mail Utility (MAIL) Routines
14.13 Using the MAIL Routines: Examples

```
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 14–3 shows a C program that displays user profile information.

**Example 14–3 Displaying User Profile Information**

/* show_profile.c */
#include <stdio>
#include <ssdef>
#include <jpidef>
#include <maildef>
#include <stsdef>
#include <ctype>
#include <nam>

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
    ;
```

(continued on next page)
Example 14–3 (Cont.) Displaying User Profile Information

```
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$_USERNAME, user, &userlen},
    {0, 0, 0, 0}},
user_itmlst[] = {
    {0, MAIL$_USER_USERNAME, 0, 0},
    {0, 0, 0, 0}},
out_itmlst[] = {
    /* Full directory spec */
    {sizeof(full_directory), MAIL$_USER_FULL_DIRECTORY, full_directory, &full_directory_len},
    /* New message count */
    {sizeof(new_messages), MAIL$_USER_NEW_MESSAGES, &new_messages, 0},
    /* Forwarding field */
    {sizeof(forwarding), MAIL$_USER_FORWARDING, forwarding, &forwarding_len},
    /* Personal name field */
    {sizeof(personal_name), MAIL$_USER_PERSONAL_NAME, personal_name, &personal_name_len},
    /* Editor field */
    {sizeof(editor), MAIL$_USER_EDITOR, editor, &editor_len},
    /* CC prompting flag */
    {sizeof(cc_prompt), MAIL$_USER_CC_PROMPT, &cc_prompt, 0},
    /* Copy send flag */
    {sizeof(copy_send), MAIL$_USER_COPY_SEND, &copy_send, 0},
    /* Copy reply flag */
    {sizeof(copy_reply), MAIL$_USER_COPY_REPLY, &copy_reply, 0},
    /* Copy forward flag */
    {sizeof(copy_forward), MAIL$_USER_COPY_FORWARD, &copy_forward, 0},
    /* Auto purge flag */
    {sizeof(auto_purge), MAIL$_USER_AUTO_PURGE, &auto_purge, 0},
    /* Queue field */
    {sizeof(queue), MAIL$_USER_QUEUE, queue, &queue_len},
    /* Form field */
    {sizeof(form), MAIL$_USER_FORM, form, &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,jpi_list,0,0,0);
    user[userlen] = '0';
}
```

(continued on next page)
Example 14–3  (Cont.) Displaying User Profile Information

```c
while(isspace(user[--userlen]))
    user[userlen] = '\0';
user_itmlst[0].buffer_length = strlen(user);
user_itmlst[0].buffer_address = 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\n");
if (copy_send == TRUE)
    printf(" SEND");
if (copy_reply == TRUE) {
    if (copy_send == TRUE)
        printf(" ,\n");
    printf(" REPLY\n");
}
if (copy_forward == TRUE) {
    if ((copy_reply == TRUE) || (copy_send == TRUE))
        printf(" ,\n");
    printf(" FORWARD\n");
}
if ((copy_reply == FALSE) && (copy_send == FALSE) && (copy_forward == FALSE))
    printf(" Nothing\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);
```
14.14 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:

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item code</td>
<td>Buffer length</td>
<td></td>
</tr>
<tr>
<td>Buffer address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return length address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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: itm_lst_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: itm_lst_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 recclaims 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 Compaq.
Mail Utility Routines
MAIL$MAILFILE_CLOSE

out_item_list
OpenVMS usage: itm1st_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.
**Mail Utility Routines**  
**MAIL$MAILFILE_COMPRESS**

### Condition Values Returned

- **SS$$_{NORMAL}**: Normal successful completion.
- **MAIL$_{INVITMCO}D** : The specified item code is invalid.
- **MAIL$_{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.
Mail Utility Routines
MAIL$MAILFILE_END

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$MAILFILE_INFO_FILE uses the current default directory as the default mail file specification.
Mail Utility Routines
MAIL$MAILFILE_INFO_FILE

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: itmlist_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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$_ILLFOLNAM</td>
<td>The specified folder name is illegal.</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_NOTISAM</td>
<td>The message file is not an indexed file.</td>
</tr>
<tr>
<td>MAIL$_OPENIN</td>
<td>Mail cannot open the file as input.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>

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
- 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.
Mail Utility Routines
MAIL$MAILFILE_OPEN

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.
Mail Utility Routines
MAIL$MAILFILE_OPEN

Description
The default mail file specification is MAIL.MAI in the MAIL subdirectory.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$_FILEOPEN</td>
<td>The mail file is already open.</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_NOMSGS</td>
<td>No messages are available.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>

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.
Mail Utility Routines
MAIL$MAILFILE_PURGE_WASTE

**out_item_list**

- **OpenVMS usage**:  itmImt_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.
Mail Utility Routines
MAIL$MAILFILE_PURGE_WASTE

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_NOFILEOPEN</td>
<td>No mail file is currently open.</td>
</tr>
<tr>
<td>MAIL$_NOTISAM</td>
<td>The message file is not an indexed file.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>
MAIL$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$_MESSAGE_FILE_CTX is required.
Mail Utility Routines
MAIL$MESSAGE_BEGIN

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_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.
Mail Utility Routines
MAIL$MESSAGE_COPY

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>MAIL$_BADVALUE</td>
<td>The specified keyword value is invalid.</td>
</tr>
<tr>
<td>MAIL$_CONITMCOD</td>
<td>The specified item codes define conflicting operations.</td>
</tr>
<tr>
<td>MAIL$_DATIMUSED</td>
<td>The date and time is currently used in the specified file.</td>
</tr>
<tr>
<td>MAIL$_DELMMSG</td>
<td>The message is deleted.</td>
</tr>
<tr>
<td>MAIL$_ILLCTXADR</td>
<td>The context block address is illegal.</td>
</tr>
<tr>
<td>MAIL$_INVTMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVTMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_MSGINFO</td>
<td>Informational records are successfully returned.</td>
</tr>
<tr>
<td>MAIL$_MSGTEXT</td>
<td>Text record is successfully returned.</td>
</tr>
<tr>
<td>MAIL$_NOFILEOPEN</td>
<td>The mail file is not open.</td>
</tr>
<tr>
<td>MAIL$_NOMOREREC</td>
<td>No more records can be found.</td>
</tr>
<tr>
<td>MAIL$_NOTREADIN</td>
<td>The operation is invalid; you are not reading a message.</td>
</tr>
<tr>
<td>MAIL$_RECTOBIG</td>
<td>The record is too large for the MAIL buffer.</td>
</tr>
<tr>
<td>MAIL$_WRONGCTX</td>
<td>The context block is incorrect.</td>
</tr>
<tr>
<td>MAIL$_WRONGFILE</td>
<td>The specified file is incorrect in this context.</td>
</tr>
<tr>
<td>SS$_IVDEVNAM</td>
<td>The device name is invalid.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>

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$_MESSAGE_ID is required.
Mail Utility Routines
MAIL$MESSAGE_DELETE

**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.
Mail Utility Routines
MAIL$MESSAGE_END

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 255 characters long.
Specify a value from 0 to 255 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 255 characters long.
Specify a value from 0 to 255 in the buffer length field of the item descriptor.

MAIL$_MESSAGE_RECORD
When you specify MAIL$_MESSAGE_RECORD, MAIL$_MESSAGE_GET returns a record of the message. The buffer address field of the item descriptor points to a buffer that receives a character string 0 to 255 characters long.
Specify a value from 0 to 255 in the buffer length field of the item descriptor.
MAIL$_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$_MESSAGE_NEXT
Mail Utility Routines
MAIL$MESSAGE_GET

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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$MSGINFO</td>
<td>Informational records are successfully returned.</td>
</tr>
<tr>
<td>MAIL$MSGTEXT</td>
<td>Text record is successfully returned.</td>
</tr>
<tr>
<td>MAIL$ILLCTXADR</td>
<td>The context block address is illegal.</td>
</tr>
<tr>
<td>MAIL$INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
</tbody>
</table>
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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIL$_CONITMCOD</td>
<td>The specified item codes define conflicting operations.</td>
</tr>
<tr>
<td>MAIL$_DELMES</td>
<td>The message is deleted.</td>
</tr>
<tr>
<td>MAIL$_ILLCTXADR</td>
<td>The context block address is illegal.</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_NOFILEOPEN</td>
<td>The mail file is not open.</td>
</tr>
<tr>
<td>MAIL$_NOMOREMSG</td>
<td>No more messages.</td>
</tr>
<tr>
<td>MAIL$_WRONGCTX</td>
<td>The context block is incorrect.</td>
</tr>
<tr>
<td>MAIL$_WRONGFILE</td>
<td>The specified file is incorrect in this context.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>

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.

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$_CONITMCOD The specified item codes define conflicting operations.
MAIL$_DELMMSG The message is deleted.
MAIL$_ILLCTXADR The context block address is illegal.
MAIL$_INVITMCOD The specified item code is invalid.
Mail Utility Routines
MAIL$MESSAGE_MODIFY

MAIL$_INVITMLEN  The specified item length is invalid.
MAIL$_MISREQITEM  The required item is missing.
MAIL$_NOMFILEOPEN  The mail file is not open.
MAIL$_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_SELECT

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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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$_NOTEEXIST The specified folder does not exist.
MAIL$_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_MESSAGE.

Condition Value Returned

SS$_NORMAL  Normal successful completion.
MAIL$SEND_ADD_ADDRESS—Add Address to List

Add 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 255 characters long.

Specify a value from 0 to 255 in the buffer length field of the item descriptor.
MAIL$ SEND_ADD_ATTRIBUTE
MAIL$SEND_ADD_ATTRIBUTE 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 255 characters long.

Specify a value from 0 to 255 in the buffer length field of the item descriptor.

Calls to MAIL$SEND_ADD_ATTRIBUTE using this input item code should be made before any calls to MAIL$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 255 characters long.

Specify a value from 0 to 255 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 255 characters long.

Specify a value from 0 to 255 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 ValuesReturned.

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$_SEND_DEFAULT_NAME item code. More information about using a file ID for specifying files can be found in OpenVMS Record Management Services Reference Manual. Note that the MAIL$_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 255 characters long.

Specify a value from 0 to 255 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 SYSS$CRATCH 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.
Mail Utility Routines
MAIL$SEND_ADD_BODYPART

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

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>MAIL$_CONITMCOD</td>
<td>The specified item codes define conflicting operations.</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_OPENIN</td>
<td>The required file is missing.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>
MAIL$SEND_BEGIN—Start Sending Message

Initiates processing to send a message to the users on the address list. You must call MAIL$SENDBEGIN before you call any other send routine.

Format

MAIL$SENDBEGIN 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_Foward**
---

When you specify the Boolean item code Mail$_Send_Copy_Foward, Mail$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.
Mail Utility Routines
MAIL$SEND_BEGIN

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$GET_VLM.
MAIL$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

custom
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.
Mail Utility Routines
MAIL$SEND_END

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>

Any condition value returned by LIB$FREE_VM.
MAIL$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 OpenVMS Programming Concepts Manual for more information about the signal array and its use by condition-handling routines.

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
Mail Utility Routines
MAIL$SEND_MESSAGE

are passed as input to the routine. Refer to the 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.

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.
Mail Utility Routines
MAIL$USER_BEGIN

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$USERBEGIN 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$_USER_FULLDIRECTORY
When you specify MAIL$_USER_FULLDIRECTORY, 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$_USER_NEWMESSAGES
When you specify MAIL$_USER_NEWMESSAGES, 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. 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: itm_list_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
MAIL$ USER SET NO 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
MAIL$ USER SET NO 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
MAIL$ USER SET NO 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
MAIL$ USER SET NO 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
MAIL$ USER SET NO 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 NO 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_INFO

MAIL$USER_SET_FORM
MAIL$USER_SET_NO_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_NO_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_NO_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_NO_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_NO_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$_USER_SET_INFO
MAIL$_USER_SET_SIGFILE
MAIL$_USER_SET_NO_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_NO_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.

out_item_list

OpenVMS usage: itmlst_3

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.
Mail Utility Routines
MAIL$USER_SET_INFO

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>MAIL$_CONITMCOD</td>
<td>The specified item codes perform conflicting operations.</td>
</tr>
<tr>
<td>MAIL$_ILLCHAR</td>
<td>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.</td>
</tr>
<tr>
<td>MAIL$_ILLPERNAM</td>
<td>Personal name formatted improperly. Returns an FAO argument containing the improperly formatted personal name.</td>
</tr>
<tr>
<td>MAIL$_ILLSUBDIR</td>
<td>Illegal subdirectory specification. Returns an FAO argument containing the subdirectory string.</td>
</tr>
<tr>
<td>MAIL$_INVITMCOD</td>
<td>The specified item code is invalid.</td>
</tr>
<tr>
<td>MAIL$_INVITMLEN</td>
<td>The specified item length is invalid.</td>
</tr>
<tr>
<td>MAIL$_MISREQITEM</td>
<td>The required item is missing.</td>
</tr>
<tr>
<td>MAIL$_NAMTOOBIG</td>
<td>Specified name exceeds 255-character limit.</td>
</tr>
<tr>
<td>MAIL$_NOTSUBDIR</td>
<td>No such subdirectory. Returns an FAO argument containing the subdirectory string.</td>
</tr>
<tr>
<td>MAIL$_NOSUCHUSR</td>
<td>No such user. Returns the name of the unfound user.</td>
</tr>
<tr>
<td>MAIL$_NOSYSNAM</td>
<td>Caller needs SYSNAM privileges.</td>
</tr>
<tr>
<td>MAIL$_NOSYSPRV</td>
<td>Caller needs system privileges.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation.</td>
</tr>
</tbody>
</table>
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.

15.1 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.

15.1.1 List of NCS Routines

Table 15–1 lists the individual NCS routines.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCS$COMPARE</td>
<td>Compares two strings using a specified collating sequence as comparison basis.</td>
</tr>
<tr>
<td>NCS$CONVERT</td>
<td>Converts a string using the specified conversion function.</td>
</tr>
<tr>
<td>NCS$END_CF</td>
<td>Terminates the use of a conversion function by the calling program.</td>
</tr>
<tr>
<td>NCS$END_CS</td>
<td>Terminates the use of a collating sequence by the calling program.</td>
</tr>
<tr>
<td>NCS$GET_CF</td>
<td>Retrieves the definition of the named conversion function from the NCS library.</td>
</tr>
</tbody>
</table>

(continued on next page)
15.1 Introduction to NCS Routines

Table 15–1 (Cont.) NCS Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCS$GET_CS</td>
<td>Retrieves the definition of the named collating sequence from the NCS library.</td>
</tr>
<tr>
<td>NCS$RESTORE_CF</td>
<td>Permits the calling program to restore the definition of a “saved” conversion function from a database or an OpenVMS RMS file.</td>
</tr>
<tr>
<td>NCS$RESTORE_CS</td>
<td>Permits the calling program to restore the definition of a “saved” collating sequence from a database or an RMS file.</td>
</tr>
<tr>
<td>NCS$SAVE_CF</td>
<td>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.</td>
</tr>
<tr>
<td>NCS$SAVE_CS</td>
<td>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.</td>
</tr>
</tbody>
</table>

15.1.2 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.

15.2 Using the NCS Utility Routines: Examples

This section includes two examples of how to use NCS utility routines in program applications:

Example 15–1 illustrates the use of NCS utility routines in a Compaq Fortran for OpenVMS program.
Example 15–1 Using NCS Routines in a Compaq 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 Read the name of the collating sequence.
    WRITE (6,30) READ (5,15,END=999) CSLENGTH, CSSTRING
    30 FORMAT(' Collating Sequence: ')

    C Get the collating sequence from the NCS library
    CSID = 0
    STATUS = NCS$GET_CS (CSID, CSSTRING(1:CSLENGTH))
    IF ((STATUS .AND. 1) .NE. 1) THEN
        CALL LIB$SIGNAL (%VAL(STATUS))
    ENDIF

    C Read two strings to be compared according to the collating sequence
    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 Compare the strings
    result = ncs$compare (csid, string1(1:length1), string2(1:length2))

    C Display the results of the comparison
    WRITE (6,40) STRING1(1:LENGTH1), CMP(RESULT+2), STRING2(1:LENGTH2)
    40 FORMAT('',a,'',a,'',a)
    GOTO 100

    C Come here if both inputs are blank -- we are done.
    C Call NCS$END_CS to free any storage used to hold the CS.
    200 STATUS = NCS$END_CS (CSID)
    IF ((STATUS .AND. 1) .NE. 1) THEN
        CALL LIB$SIGNAL (%VAL(STATUS))
    ENDIF
    CALL EXIT

(continued on next page)
Example 15–1 (Cont.) Using NCS Routines in a Compaq Fortran for OpenVMS Program

```fortran
999 CONTINUE
END
```

Example 15–2 illustrates the use of NCS routines in a Compaq 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 15–2 Using NCS Routines in a Compaq C for OpenVMS VAX Program

/*
 *===========================================================================
 * NCS_EXAMPLE.C
 *===========================================================================
 */

/*
 * 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 */
#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 ()
{

(continued on next page)
Example 15–2 (Cont.) Using NCS Routines in a Compaq C for OpenVMS VAX Program

/*
 ** 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 */
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 */
}

(continued on next page)
Example 15–2 (Cont.) Using NCS Routines in a Compaq C for OpenVMS VAX Program

/*
**  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;
}

void status_check(status,expected_status)
{
    if (status != expected_status)
        sys$exit(status);
    else
        return;
}

15.3 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.
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:

<table>
<thead>
<tr>
<th>Returned Value</th>
<th>Comparison Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>string_1 is less than string_2</td>
</tr>
<tr>
<td>0</td>
<td>string_1 is equal to string_2</td>
</tr>
<tr>
<td>1</td>
<td>string_1 is greater than string_2</td>
</tr>
</tbody>
</table>

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

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: by reference

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.
NCS$CONVERT

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCS$_DIAG</td>
<td>Operation completed with signaled diagnostics.</td>
</tr>
<tr>
<td>NCS$_NOT_CF</td>
<td>Name of identifier does not refer to a conversion function.</td>
</tr>
<tr>
<td>NCS$_NOT_FOUND</td>
<td>Name of identifier not found in the NCS library.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCS$_DIAG</td>
<td>Operation completed with signaled diagnostics.</td>
</tr>
<tr>
<td>NCS$_NOT_CS</td>
<td>Name of identifier does not refer to a collating sequence.</td>
</tr>
<tr>
<td>NCS$_NOT_FOUND</td>
<td>Name of identifier not found in the NCS library.</td>
</tr>
</tbody>
</table>

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.
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$RESTORE_C F routine, used in conjunction with the NCS$SAVE_C F routine, permits the application program to keep a local copy of the conversion function. The NCS$SAVE_C F routine obtains the length and location of the conversion function and returns it to the application program. The application program subsequently provides this information to the NCS$RESTORE_C F 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

NCSS_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.
National Character Set (NCS) Utility Routines
NCS$RESTORE_CS

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

<table>
<thead>
<tr>
<th>Condition Value Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCS$ NOT_CS</td>
<td>Name of identifier does not refer to a collating sequence.</td>
</tr>
</tbody>
</table>

Condition Values Signaled

LBR messages (prefaced by an NCS message) might signal errors detected while the process is accessing the NCS library.
National Character Set (NCS) Utility Routines
NCS$SAVE_CFRoutines

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.
16

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.

16.1 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, Compaq 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 16.2 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 16.3 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 16.4 contains an example of a simple modification to the print symbiont.
16.1 Introduction to PSM Routines

- Section 16.5 describes each PSM routine and the interface used by the routines you substitute for the standard PSM routines.

16.2 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.

16.2.1 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 17.

All of these routines are contained in a shareable image with the file specification SYS$SHARE:SMBSRVSHR.EXE.

16.2.2 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 for 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.
16.2.3 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 16–1 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.

16.2.4 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.
16.2 Print Symbiont Overview

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.

16.2.5 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 16–2 illustrates the steps taken by the symbiont in the processing of a print task.

**Figure 16–2 Symbiont Execution Sequence or Flow of Control**

![Diagram of Symbiont Execution Sequence](image)

As Figure 16–2 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.

16.3 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 16–1 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 16.3.6 provides more information about these codes.

16.3.1 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 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 \texttt{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 \texttt{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.

### 16.3.2 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 PSM$K_OPEN, PSM$K_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 16.3.5 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 16-1 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.

#### 16.3.2.1 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.
Print Symbiont Modification (PSM) Routines
16.3 Symbiont Modification Procedure

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Action Taken by the Input Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM$K_OPEN</td>
<td>An OPEN request. When the main input routine receives this request code, it does the following:</td>
</tr>
<tr>
<td></td>
<td>1. Opens the input file.</td>
</tr>
<tr>
<td></td>
<td>2. Stores information about the input file.</td>
</tr>
<tr>
<td></td>
<td>3. Returns the type of carriage control used in the input file. If this routine cannot open the file, it returns an error.</td>
</tr>
<tr>
<td></td>
<td>Note that the print symbiont’s main input routine performs these tasks when it receives the PSM$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.</td>
</tr>
<tr>
<td>PSM$K_READ</td>
<td>A READ request. When the main input routine receives this request, it returns the next record from the file. In addition, when the carriage control used by the data file is PSM$K_CC_PRINT, the main input routine returns the associated record header.</td>
</tr>
<tr>
<td>PSM$K_CLOSE</td>
<td>A CLOSE request. When the main input routine receives this request, it closes the input file.</td>
</tr>
</tbody>
</table>

16.3.2.2 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 OpenVMS Record Management Services Reference Manual.

<table>
<thead>
<tr>
<th>Type of Carriage Control</th>
<th>Symbiont Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded</td>
<td>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.</td>
</tr>
<tr>
<td>Fortran</td>
<td>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.</td>
</tr>
<tr>
<td>PRN</td>
<td>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 <code>funcarg</code> argument of the main input routine; this occurs with the PSM$K$READ request. The <code>funcarg</code> argument specifies a longword; your routine writes the 2-byte PRN carriage control specifier into the first two bytes of this longword.</td>
</tr>
<tr>
<td>Implied</td>
<td>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.</td>
</tr>
</tbody>
</table>

16.3.3 Writing a Format Routine

To write a format routine, follow the modification procedure described in Section 16.3. 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 16.3.5 for additional information about other function codes used in the user-written formatting routine.
16.3.3.1 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.

16.3.4 Writing an Output Routine

To write an output routine, follow the modification procedure described in Section 16.3.

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, PSM$K_WRITE_NOFORMAT, and PSM$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 \texttt{func} argument of the \texttt{USER-OUTPUT-ROUTINE} routine.

See Section 16.3.5 for additional information about other function codes.

\subsection*{16.3.4.1 Internal Logic of the Symbiont's Main Output Routine}

When the symbiont calls the main output routine with the \texttt{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 \texttt{funcarg} argument (for more information, see the description of the \texttt{SMBMSG$K\_DEVICE\_STATUS} message item in Chapter 17)
5. Returns an error if the device is not a terminal or a printer

When this routine receives a \texttt{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 \texttt{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 \texttt{PSM$K\_CANCEL} service request code, it requests the device driver to cancel any outstanding output operations.

When this routine receives a \texttt{PSM$K\_CLOSE} service request code, it deassigns the channel to the device and deallocates the device.

\subsection*{16.3.5 Other Function Codes}

A status \texttt{PSM\_$PENDING} might not be returned whenever the symbiont notifies user-written input, output, and format routines using the following message function codes:

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{PSM$K_START_STREAM}</td>
<td>Job controller sends a message to the symbiont to start a queue</td>
</tr>
<tr>
<td>\texttt{PSM$K_START_TASK}</td>
<td>Symbiont parses a message from job controller directing it to start a queue</td>
</tr>
<tr>
<td>\texttt{PSM$K_PAUSE_TASK}</td>
<td>Job controller sends a message to the symbiont to suspend processing of the current task</td>
</tr>
<tr>
<td>\texttt{PSM$K_STOP_STREAM}</td>
<td>Job controller sends a message to the symbiont to stop the queue</td>
</tr>
<tr>
<td>\texttt{PSM$K_STOP_TASK}</td>
<td>Job controller sends a message to the symbiont to stop the task</td>
</tr>
<tr>
<td>\texttt{PSM$K_RESUME_TASK}</td>
<td>Job controller sends a message to the symbiont to resume processing of the current task</td>
</tr>
<tr>
<td>\texttt{PSM$K_RESET_STREAM}</td>
<td>Same as \texttt{PSM$K_STOP_STREAM}</td>
</tr>
</tbody>
</table>
16.3.6 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 16–1 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 16–1 Routine Codes for Specification to PSM$REPLACE

<table>
<thead>
<tr>
<th>Routine Code</th>
<th>Sequence</th>
<th>Function</th>
<th>Supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM$K_JOB_SETUP</td>
<td>1</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FORM_SETUP</td>
<td>2</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_JOB_FLAG</td>
<td>3</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_JOB_BURST</td>
<td>4</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_SETUP</td>
<td>5</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_FLAG</td>
<td>6</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_BURST</td>
<td>7</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_SETUP_2</td>
<td>8</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_MAIN_INPUT</td>
<td>9</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_INFORMATION</td>
<td>10</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_ERRORS</td>
<td>11</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_FILE_TRAILER</td>
<td>12</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_JOB_RESET</td>
<td>13</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_JOB_TRAILER</td>
<td>14</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_JOB_COMPLETION</td>
<td>15</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_PAGE_SETUP</td>
<td>x</td>
<td>Input</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The job completion (PSM$K_JOB_COMPLETION) and output (PSM$K_OUTPUT) routines are not replaceable when using the LAT protocol option.
16.3 Symbiont Modification Procedure

Table 16-1 (Cont.) Routine Codes for Specification to PSM$REPLACE

<table>
<thead>
<tr>
<th>Routine Code</th>
<th>Sequence</th>
<th>Function</th>
<th>Supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM$K_PAGE_HEADER</td>
<td>x</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_LIBRARY_INPUT</td>
<td>x</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_INPUT_FILTER</td>
<td>x</td>
<td>Formatting</td>
<td>No</td>
</tr>
<tr>
<td>PSM$K_MAIN_FORMAT</td>
<td>x</td>
<td>Formatting</td>
<td>Yes</td>
</tr>
<tr>
<td>PSM$K_OUTPUT_FILTER</td>
<td>x</td>
<td>Formatting</td>
<td>No</td>
</tr>
<tr>
<td>PSM$K_OUTPUT</td>
<td>x</td>
<td>Output</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1The job completion (PSM$K_JOB_COMPLETION) and output (PSM$K_OUTPUT) routines are not replaceable when using the LAT protocol option.

16.3.7 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:
   $$ \text{LINK/DEBUG your-symbiont} $$
   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$SHARE:SMBSRVSRH.EXE and SYS$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:
   $$ \text{SET TERMINAL/NODISCONNECT/PERMANENT _TTcu:} $$
   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:
   $$ \text{DEFINE/GROUP DBG$INPUT _TTcu:} $$
   $$ \text{DEFINE/GROUP DBG$OUTPUT _TTcu:} $$
   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:
   $$ \text{INITIALIZE/QUEUE/PROCESSOR= your-symbiont /ON= printer_name} $$
16.3 Symbiont Modification Procedure

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.

16.4 Using the PSM Routines: An Example

Example 16–1 shows how to use PSM routines to supply a page header routine in a VAX MACRO program.

Example 16–1 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

(continued on next page)
Example 16–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a VAX MACRO Program

; 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_DESC: .LONG 80 ; work buffer descriptor
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

(continued on next page)
Example 16–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a VAX MACRO Program

; 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

(continued on next page)
Print Symbiont Modification (PSM) Routines
16.4 Using the PSM Routines: An Example

Example 16–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a VAX MACRO Program

; 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
$FAO_S CTRSTR = FAO_Ctrl_2, - ; FAO control string desc
OUTBUF = FAO_DESC, - ; output buffer descriptor
P1 = PAGE ; page number
MOVAB FAO_BUFF,FAO_DESC+4 ; point to work buffer
BLBC R0,55$ ; return if error

; 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$:
PUSHL FUNC_DESC(AP) ; symbiont descriptor
PUSHL LINSIZ ; number of bytes to reserve
CALLS #2,G^STR$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$: MOVB #SS$_NORMAL,R0 ; set success
67$: RET ; return

(continued on next page)
Print Symbiont Modification (PSM) Routines
16.4 Using the PSM Routines: An Example

Example 16–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a VAX
MACRO Program

; ; Done with this page header ;
70$: MOVL #PSM$_EOF,R0 ; return end of input MOVBL #2,LINE ; reset line counter RET ; return .END START

16.5 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


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.

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.
Print Symbiont Modification (PSM) Routines
PSM$PRINT

The print symbiont uses the value specified by **buftime** 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 17.
PSM$READ_ITEM_DX—Obtain Value of Message Items

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.

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 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 17.
Print Symbiont Modification (PSM) Routines

PSM$READ_ITEM_DX

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 16-1 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
Print Symbiont Modification (PSM) Routines

PSM$REPLACE

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 16–1 lists each routine code in the order in which the location it identifies is reached within the symbiont execution stream.

**Condition Value Returned**

<table>
<thead>
<tr>
<th>Condition Value Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
</tbody>
</table>
PSM$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 16.3.5, 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:

<table>
<thead>
<tr>
<th>31</th>
<th>23</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Count</td>
<td>Character</td>
<td>Count</td>
<td></td>
</tr>
<tr>
<td>Trailing Carriage-Control Information</td>
<td>Leading Carriage-Control Information</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
Print Symbiont Modification (PSM) Routines
USER-FORMAT-ROUTINE

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 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 16.3.1 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

<table>
<thead>
<tr>
<th>Type</th>
<th>longword (unsigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>read only</td>
</tr>
<tr>
<td>Mechanism</td>
<td>by reference</td>
</tr>
</tbody>
</table>

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$K_GET_KEY and PSM$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. Compaq 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.

PSMK_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 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:

<table>
<thead>
<tr>
<th>Carriage Control Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM$K_CC_IMPLIED</td>
<td>Implied carriage control. For this type, the symbiont inserts a leading line feed (LF) and trailing carriage return (CR) in each input record. This is the default carriage control type; it is used if your routine does not supply a carriage control type in the funcarg argument in response to the PSM$K_OPEN function call.</td>
</tr>
<tr>
<td>PSM$K_CC_FORTRAN</td>
<td>Fortran carriage control. For this type, the symbiont extracts the first byte of each input record and interprets the byte as a Fortran carriage control character, which it then applies to the input record.</td>
</tr>
<tr>
<td>PSM$K_CC_PRINT</td>
<td>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.</td>
</tr>
<tr>
<td>PSM$K_CC_INTERNAL</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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. Compaq 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 16.3.5, 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.

**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 16.3.1 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. Compaq 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. Compaq 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 service.

**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 17.1.6, 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.

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 16.3.1 for more information about error condition values.
17

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.

17.1 Introduction to SMB Routines

Always use the SMB interface routines or the $SNDJ BC 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.

17.1.1 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 symbions.

17.1.2 Symbionts Supplied with the Operating System

The operating system supplies two symbions:

• SYS$SYSTEM:PRTEM.BEXE (PRTEM.BEXE for short), an output symbiont for use with printers and printing terminals

PRTEM.BEXE 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 PRTEM.BEXE using the Print Symbiont Modification (PSM) routines.

• SYS$SYSTEM:INPSMB.BEXE (INPSMB.BEXE 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.

17.1.3 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 17–1 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 17–1 Symbionts in the OpenVMS Environment

1 You request a printing job with the DCL command PRINT. DCL calls the "$SNDJ BC 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.

### 17.1.4 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.

---

### 17.1.5 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
Symbiont/Job Controller Interface (SMB) Routines

17.1 Introduction to SMB Routines

- 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 $SNDJ BC 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 symbions, 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.

17.1.6 The Symbiont/Job Controller Interface Routines

The five SMB routines form a public interface to the job controller. The job controller delivers requests to symbions by means of this interface, and the symbions communicate their responses to those requests through this interface. A user-written symbiont uses the following routines to exchange messages with the job controller:
17.1 Introduction to SMB Routines

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.

SMB$CHECK_FOR_MESSAGE Checks to see if a message from the job controller to the symbiont has arrived (used with synchronous symbions)

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.

17.1.7 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.

17.1.7.1 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.
Symbiont/Job Controller Interface (SMB) Routines
17.1 Introduction to SMB Routines

Figure 17–2 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.

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 17–3 and Figure 17–4 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 17–3 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.
17.1 Introduction to SMB Routines

Figure 17–3 Flowchart for a Single-Threaded, Asynchronous Symbiont (MAIN Routine)
17.1.7.2 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.
17.1.8 Reading Job Controller Requests

Table 17–1 lists the seven general functions the job controller can request of the symbiont.

Table 17–1 Job Controller Functions

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$K_START_STREAM</td>
<td>Start Stream</td>
</tr>
<tr>
<td>SMBMSG$K_STOP_STREAM</td>
<td>Stop Stream</td>
</tr>
<tr>
<td>SMBMSG$K_START_TASK</td>
<td>Start Task</td>
</tr>
<tr>
<td>SMBMSG$K_PAUSE_TASK</td>
<td>Pause Task</td>
</tr>
<tr>
<td>SMBMSG$K_RESUME_TASK</td>
<td>Resume Task</td>
</tr>
<tr>
<td>SMBMSG$K_STOP_TASK</td>
<td>Stop Task</td>
</tr>
<tr>
<td>SMBMSG$K_RESET_STREAM</td>
<td>Reset Stream</td>
</tr>
</tbody>
</table>

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.

17.1.9 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 symbions in particular but can serve as a guideline for other kinds of symbions 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. Compaq 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.)
Symbiont/Job Controller Interface (SMB) Routines
17.1 Introduction to SMB Routines

**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.

17.1.10 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.

17.2 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.

Arguments

None.

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$NORMAL</td>
<td>One or more messages waiting.</td>
</tr>
<tr>
<td>0</td>
<td>No messages waiting.</td>
</tr>
</tbody>
</table>
SMB$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$CHECK_FOR_MESSAGE routine to check for waiting messages.
Symbiont/Job Controller Interface (SMB) Routines
SMB$INITIALIZE

**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.
Symbiont/Job Controller Interface (SMB) Routines

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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Routine completed unsuccessfully because of an invalid argument.</td>
</tr>
</tbody>
</table>

This routine also returns any of the condition codes returned by the Run-Time Library string-handling (STR$) routines.
Symbiont/Job Controller Interface (SMB) Routines
SMB$READ_MESSAGE_ITEM

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:

<table>
<thead>
<tr>
<th>Request</th>
<th>Message Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$K_START_TASK</td>
<td>SMBMSG$K_ACCOUNT_NAME</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_AFTER_TIME</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_BOTTOM_MARGIN</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_CHARACTERISTICS</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_CHECKPOINT_DATA</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_ENTRY_NUMBER</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FILE_COPIES</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FILE_COUNT</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FILE_IDENTIFICATION</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FILE_SETUP_MODULES</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FILE_SPECIFICATION</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FIRST_PAGE</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FORM_LENGTH</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FORM_NAME</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FORM_SETTINGS_MODULES</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_FORM_WIDTH</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_JOB_COPIES</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_JOB_COUNT</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_JOB_NAME</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_JOB_RESET_MODULES</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_LAST_PAGE</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_LEFT_MARGIN</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_MESSAGE_VECTOR</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_NOTE</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PAGE_SETTINGS_MODULES</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_1</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_2</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_3</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_4</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_5</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_6</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_7</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PARAMETER_8</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_PRINT_CONTROL</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_SEPARATION_CONTROL</td>
</tr>
<tr>
<td></td>
<td>SMBMSG$K_REQUEST_CONTROL</td>
</tr>
</tbody>
</table>
Symbiont/Job Controller Interface (SMB) Routines
SMB$READ_MESSAGE_ITEM

<table>
<thead>
<tr>
<th>Request</th>
<th>Message Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$K_STOP_TASK</td>
<td>SMBMSG$K_STOP_CONDITION</td>
</tr>
<tr>
<td>SMBMSG$K_PAUSE_TASK</td>
<td>None</td>
</tr>
<tr>
<td>SMBMSG$K_RESUME_TASK</td>
<td>SMBMSG$K_ALIGNMENT_PAGES</td>
</tr>
<tr>
<td>SMBMSG$K_REQUEST_CONTROL</td>
<td>SMBMSG$K_RESUME_TASK</td>
</tr>
<tr>
<td>SMBMSG$K_SEARCH_STRING</td>
<td>SMBMSG$K_LIBRARY_SPECIFICATION</td>
</tr>
<tr>
<td>SMBMSG$K_STOP_STREAM</td>
<td>None</td>
</tr>
<tr>
<td>SMBMSG$K_RESET_STREAM</td>
<td>None</td>
</tr>
</tbody>
</table>

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$K_AFTER_TIME
This code identifies a 64-bit, absolute-time value specifying the system time after which the job controller can process this job.
Symbiont/Job Controller Interface (SMB) Routines
SMB$READ_MESSAGE_ITEM

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 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$T_DVI field
Symbiont/Job Controller Interface (SMB) Routines
SMB$READ_MESSAGE_ITEM

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$K_JOB_RESET_MODULES
This code identifies a string specifying a list of one or more module names (separated by commas) that the symbiont should copy from the device-control library after processing the task. These modules can be used to reset programmable devices to a known state.
SMB$READ_MESSAGE_ITEM

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.

SMB$K_LEFT_MARGIN

This code identifies a longword specifying the number of spaces to be inserted at the beginning of each line.

SMB$K_LIBRARY_SPECIFICATION

This code identifies a string specifying the name of the device-control library.

SMB$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 SMB$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.

SMB$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.

SMB$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.

SMB$K_PARAMETER_1 through SMB$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.

SMB$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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMB$V_DOUBLE_SPACE</td>
<td>The symbiont uses a double-spaced format; it skips a line after each line it prints.</td>
</tr>
<tr>
<td>SMB$V_NO_INITIAL_FF</td>
<td>The symbiont suppresses the initial form feed if this bit is turned on.</td>
</tr>
<tr>
<td>SMB$V_NORECORD_BLOCKING</td>
<td>The symbiont performs single record output, issuing a single output record for each input record.</td>
</tr>
<tr>
<td>SMB$V_PAGE_HEADER</td>
<td>The symbiont prints a page header at the top of each page.</td>
</tr>
</tbody>
</table>
### Symbiont/Job Controller Interface (SMB) Routines

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$V_PAGINATE</td>
<td>The symbiont inserts a form feed character when it detects an attempt to print in the bottom margin of the current form.</td>
</tr>
<tr>
<td>SMBMSG$V_PASSALL</td>
<td>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.</td>
</tr>
<tr>
<td>SMBMSG$V_RECORD_BLOCKING</td>
<td>The symbiont performs record blocking, buffering output to the device.</td>
</tr>
<tr>
<td>SMBMSG$V_SEQUENCED</td>
<td>This bit is reserved by Compaq.</td>
</tr>
<tr>
<td>SMBMSG$V_SHEET_FEED</td>
<td>The symbiont pauses the queue after each page it prints.</td>
</tr>
<tr>
<td>SMBMSG$V_TRUNCATE</td>
<td>The symbiont truncates input lines that exceed the right margin of the current form.</td>
</tr>
<tr>
<td>SMBMSG$V_WRAP</td>
<td>The symbiont wraps input lines that exceed the right margin, printing the additional characters on a new line.</td>
</tr>
</tbody>
</table>

**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.
Symbiont/Job Controller Interface (SMB) Routines

SMB$READ_MESSAGE_ITEM

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$V_ALIGNMENT_MASK</td>
<td>The symbiont is to replace all alphabetic characters with the letter X, and all numeric characters with the number 9. Other characters (punctuation, carriage control, and so on) are left unchanged. This bit is ordinarily specified in connection with the SMBMSG$K_ALIGNMENT_PAGES item.</td>
</tr>
<tr>
<td>SMBMSG$V_PAUSE_COMPLETE</td>
<td>The symbiont is to pause when it completes the current request.</td>
</tr>
<tr>
<td>SMBMSG$V_RESTARTING</td>
<td>Indicates that this job was previously interrupted and requeued, and is now restarting.</td>
</tr>
<tr>
<td>SMBMSG$V_TOP_OF_FILE</td>
<td>The symbiont is to rewind the input file before it resumes printing.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$V_FILE_BURST</td>
<td>The symbiont is to print a file burst page.</td>
</tr>
<tr>
<td>SMBMSG$V_FILE_FLAG</td>
<td>The symbiont is to print a file flag page.</td>
</tr>
<tr>
<td>SMBMSG$V_FILE_TRAILER</td>
<td>The symbiont is to print a file trailer page.</td>
</tr>
<tr>
<td>SMBMSG$V_FILE_TRAILER_ABORT</td>
<td>The symbiont is to print a file trailer page when a task completes abnormally.</td>
</tr>
<tr>
<td>SMBMSG$V_FIRST_FILE_OF_JOB</td>
<td>The current file is the first file of the job. When specified with SMBMSG$V_LAST_FILE_OF_JOB, the current job contains a single file.</td>
</tr>
</tbody>
</table>
### Symbol Description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$V_JOB_FLAG</td>
<td>The symbiont is to print a job flag page.</td>
</tr>
<tr>
<td>SMBMSG$V_JOB_BURST</td>
<td>The symbiont is to print a job burst page.</td>
</tr>
<tr>
<td>SMBMSG$V_JOB_RESET</td>
<td>The symbiont is to execute a job reset sequence when the task completes.</td>
</tr>
<tr>
<td>SMBMSG$V_JOB_RESET_ABORT</td>
<td>The symbiont is to execute a job reset sequence when a task completes abnormally.</td>
</tr>
<tr>
<td>SMBMSG$V_JOB_TRAILER</td>
<td>The symbiont is to print a job trailer page.</td>
</tr>
<tr>
<td>SMBMSG$V_JOB_TRAILER_ABORT</td>
<td>The symbiont is to print a job trailer page when a task completes abnormally.</td>
</tr>
<tr>
<td>SMBMSG$V_LAST_FILE_OF_JOB</td>
<td>The current file is the last file of the job. When specified with SMBMSG$V_FIRST_FILE_OF_JOB, the current job contains a single job.</td>
</tr>
</tbody>
</table>

#### 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:

```
| 31 | 0 |
|----------------|
| Number of Pages Printed for the Job |
| Number of Reads from Disk or Tape |
| Number of Writes to the Printing Device |
| Unused |
```

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.
Symbiont/Job Controller Interface (SMB) Routines

SMB$SEND_TO_JOBCTL

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:

<table>
<thead>
<tr>
<th>Device Status Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$V_LOWERCASE</td>
<td>The device to which the symbiont is connected supports lowercase characters.</td>
</tr>
<tr>
<td>SMBMSG$V_PAUSE_TASK</td>
<td>The symbiont sends this message to inform the job controller that the symbiont has paused on its own initiative.</td>
</tr>
<tr>
<td>SMBMSG$V_REMOTE</td>
<td>The device is connected to the symbiont by means of a modem.</td>
</tr>
<tr>
<td>SMBMSG$V_SERVER</td>
<td>The symbiont is not connected to a device.</td>
</tr>
<tr>
<td>SMBMSG$V_STALLED</td>
<td>Symbiont processing is temporarily stalled.</td>
</tr>
<tr>
<td>SMBMSG$V_STOP_STREAM</td>
<td>The symbiont requests that the job controller stop the queue.</td>
</tr>
<tr>
<td>SMBMSG$V_TERMINAL</td>
<td>The symbiont is connected to a terminal.</td>
</tr>
<tr>
<td>SMBMSG$V_UNAVAILABLE</td>
<td>The device to which the symbiont is connected is not available.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Request Code</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$K_START_STREAM</td>
<td>request device_status error</td>
</tr>
<tr>
<td>SMBMSG$K_STOP_STREAM</td>
<td>request</td>
</tr>
<tr>
<td>SMBMSG$K_RESET_STREAM</td>
<td>request</td>
</tr>
<tr>
<td>SMBMSG$K_START_TASK</td>
<td>request</td>
</tr>
<tr>
<td>SMBMSG$K_PAUSE_TASK</td>
<td>request</td>
</tr>
<tr>
<td>SMBMSG$K_RESUME_TASK</td>
<td>request error1</td>
</tr>
<tr>
<td>SMBMSG$K_STOP_TASK</td>
<td>request error1</td>
</tr>
</tbody>
</table>

1This 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMBMSG$K_TASK_COMPLETE</td>
<td>request accounting error</td>
</tr>
<tr>
<td>SMBMSG$K_TASK_STATUS</td>
<td>request checkpoint device_status</td>
</tr>
</tbody>
</table>
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 SMBMSG$K_START_TASK as the request argument. This response means that the symbiont has started the task; it does not mean the task has been completed. When the symbiont has completed a task, it sends a SMB$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.
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.

### 18.1 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 18.1.2 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:

```bash
$ 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.
18.1 High-Performance Sort/Merge (Alpha Only)

18.1.1 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 18–1.

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 18–1 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$SPEC_FILE routine is not supported. (Implementation of this feature is deferred to a future OpenVMS Alpha release.) |
| Key data types | DSC$K_DTYPE_O, DSC$K_DTYPE_OU, DSC$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.) |

(continued on next page)
## 18.1 High-Performance Sort/Merge (Alpha Only)

### Table 18–1 (Cont.) High-Performance Sort/Merge: Differences in SOR$ Routine Behavior

<table>
<thead>
<tr>
<th>Feature</th>
<th>High-Performance Sort/Merge Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistical summary information</td>
<td>The following statistics are currently supported:</td>
</tr>
<tr>
<td></td>
<td>Records read/input (SOR$K_REC_INP)</td>
</tr>
<tr>
<td></td>
<td>Records sorted (SOR$K_REC_SOR)</td>
</tr>
<tr>
<td></td>
<td>Records output (SOR$K_REC_OUT)</td>
</tr>
<tr>
<td></td>
<td>Input record length (SOR$K_LRL_INP)</td>
</tr>
<tr>
<td></td>
<td>The following statistics are currently unavailable:</td>
</tr>
<tr>
<td></td>
<td>Internal length</td>
</tr>
<tr>
<td></td>
<td>Output record length</td>
</tr>
<tr>
<td></td>
<td>Sort tree size</td>
</tr>
<tr>
<td></td>
<td>Number of initial runs</td>
</tr>
<tr>
<td></td>
<td>Maximum merge order</td>
</tr>
<tr>
<td></td>
<td>Number of merge passes</td>
</tr>
<tr>
<td></td>
<td>Work file allocation</td>
</tr>
<tr>
<td></td>
<td>Full implementation of this feature is deferred to a future OpenVMS Alpha release.</td>
</tr>
<tr>
<td>User-supplied action routines</td>
<td>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 argument list if other arguments follow the customary position of the <code>user_compare</code> or <code>user_equal</code> argument.</td>
</tr>
<tr>
<td>user_compare</td>
<td>Compares records to determine their sort or merge order.</td>
</tr>
<tr>
<td>user_equal</td>
<td>Resolves the sort or merge order when records have duplicate keys.</td>
</tr>
</tbody>
</table>

### 18.1.2 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.
18.2 Introduction to SOR Routines

The following SOR routines are available for use in a sort or merge operation:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$BEGIN_MERGE</td>
<td>Sets up key arguments and performs the merge. This is the only routine unique to MERGE.</td>
</tr>
<tr>
<td>SOR$BEGIN_SORT</td>
<td>Initializes the sort operation by passing key information and sort options. This is the only routine unique to SORT.</td>
</tr>
<tr>
<td>SOR$DTYPE</td>
<td>Defines a key data-type that is not normally supported by SORT/MERGE. (This feature is not currently supported by the high-performance Sort/Merge utility.)</td>
</tr>
<tr>
<td>SOR$END_SORT</td>
<td>Performs cleanup functions, such as closing files and releasing memory.</td>
</tr>
<tr>
<td>SOR$PASS_FILES</td>
<td>Passes names of input and output files to SORT or MERGE; must be repeated for each input file.</td>
</tr>
<tr>
<td>SOR$RELEASE_REC</td>
<td>Passes one input record to SORT or MERGE; must be called once for each record.</td>
</tr>
<tr>
<td>SOR$RETURN_REC</td>
<td>Returns one sorted or merged record to a program; must be called once for each record.</td>
</tr>
<tr>
<td>SOR$SORT_MERGE</td>
<td>Sorts the records.</td>
</tr>
<tr>
<td>SOR$SPEC_FILE</td>
<td>Passes a specification file or specification text. A call to this routine must precede all other calls to the SOR routines. (This feature is not currently supported by the high-performance Sort/Merge utility.)</td>
</tr>
<tr>
<td>SOR$STAT</td>
<td>Returns a statistic about the sort or merge operation. (This feature is partially supported by the high-performance Sort/Merge utility.)</td>
</tr>
</tbody>
</table>

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.

18.2.1 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).
18.2 Introduction to SOR Routines

- 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.

18.2.2 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.

18.2.2.1 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$END_SORT routine can be called at any time to abort a sort or to merge and release all resources allocated to the sort or merge process.
18.2.2.2 Sort Operation Using Record Interface
For a sort operation using the record interface, first call SOR$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.

18.2.2.3 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.

18.2.2.4 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.

18.2.3 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.
18.3 Using the SOR Routines: Examples

This section provides examples of using the SOR routines for various operations including the following:

- Example 18–1 is a Compaq Fortran program that demonstrates a merge operation using a record interface.
- Example 18–2 is a Compaq Fortran program that demonstrates a sort operation using a file interface on input and a record interface on output.
- Example 18–3 is a Compaq Pascal program that demonstrates a merge operation using a file interface.
- Example 18–4 is a Compaq Pascal program that demonstrates a sort operation using a record interface.
- Example 18–5 is a Compaq C program that demonstrates a sort operation using the STABLE option and two text keys.
Example 18–1 Using SOR Routines to Perform a Merge Using Record Interface in a Compaq Fortran Program

Fortran Program

C... This program demonstrates the Fortran calling sequences for the merge record interface.
C...
C
C THE INPUT FILES ARE LISTED BELOW.
C
C INFILE1.DAT
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 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 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 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
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

(continued on next page)
Example 18–1 (Cont.) Using SOR Routines to Perform a Merge Using Record Interface in a Compaq Fortran Program

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... Open the output file.
C
OPEN (UNIT=8, FILE='TEMP.TMP', TYPE='NEW')

C... Initialize the merge. Pass the merge order, the largest record length, the compare routine address, and the input routine address.
C...

ISTAT = SOR$BEGIN_MERGE (,LRL,,ORDER, * KOMPAR,,READ_REC)
IF (.NOT. ISTAT) GOTO 10 ! Check for error.

C... Now loop getting merged records. SOR$RETURN_REC will 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... 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... Here if an error occurred. Write out the error status and exit.
C...

10 WRITE(8,201) ISTAT
201 FORMAT('ERROR CODE', I20)
CALL EXIT
END

FUNCTION READ_REC (RECX, FILE, SIZE)
C... This routine reads a record from one of the input files for merging. It will be called by SOR$BEGIN_MERGE and by SOR$RETURN_REC.
C... Parameters:
C...
RECX.wcp.ds character buffer to hold the record after it is read in.
C...

(continued on next page)
Example 18–1 (Cont.) Using SOR Routines to Perform a Merge Using Record Interface in a Compaq Fortran Program

C... FILE.r1.r indicates which file the record is
to be read from. 1 specifies the first file, 2 specifies the second etc.
C... LENGTH.wl.r is the actual number of bytes in the record. This is set by READ_REC.
C... 
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... This routine compares two records. It returns -1
if the first record is smaller than the second,
0 if the records are equal, and 1 if the first record
is larger than the second.
C... 
PARAMETER KEYSIZ=10
IMPLICIT INTEGER (A-Z)
LOGICAL*1 REC1(KEYSZ),REC2(KEYSZ)
(continued on next page)
Example 18–1 (Cont.) Using SOR Routines to Perform a Merge Using Record Interface in a Compaq Fortran Program

```
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 18–2 is a Compaq Fortran program that demonstrates a sort operation using a file interface on input and a record interface on output.

Example 18–2 Using SOR Routines to Sort Using Mixed Interface in a Compaq Fortran Program

Program

PROGRAM CALLSORT
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$OUTPUT. The key is an 80-byte
C character ascending key starting in position 1 of
C each record.
C
A short version of the input and output files follows:
C
Input file R010SQ.DAT
C
1 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD.........................END OF RECORD
C 1 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD.........................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD.........................END OF RECORD
C 1 QQQQQQQQQQ REST OF DATA IN RECORD.........................END OF RECORD
C 2 AAAAAAAAAA REST OF DATA IN RECORD.........................END OF RECORD
C 2 QQQQQQQQQQ REST OF DATA IN RECORD.........................END OF RECORD
C
Output file SYS$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 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
(continued on next page)
Example 18–2 (Cont.) Using SOR Routines to Sort Using Mixed Interface in a Compaq Fortran Program

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 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 Call the SORT -- each call is a function.
C
C Pass SORT the file names.
C
ISTATUS = SOR$PASS_FILES(INPUTNAME)
IF (.NOT. ISTATUS) GOTO 10

C Initialize the work areas and keys.
C
ISTATUS = SOR$BEGIN_SORT(KEYBUF,,,,,,SRTTYPE,%REF(3))
IF (.NOT. ISTATUS) GOTO 10

C Sort the records.
C
ISTATUS = SOR$SORT_MERGE()
IF (.NOT. ISTATUS) GOTO 10

(continued on next page)
Example 18–2 (Cont.)  Using SOR Routines to Sort Using Mixed Interface in a Compaq Fortran Program

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 18–3 is a Compaq Pascal program that demonstrates a merge operation using a file interface.

**Example 18–3 Using SOR Routines to Merge Three Input Files in a Compaq 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;

(continued on next page)
Example 18–3 (Cont.) Using SOR Routines to Merge Three Input Files in a Compaq Pascal Program

```pascal
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; 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 );
beginn
  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;
      readin(filename);
    end;
end;

(continued on next page)
```
Example 18–3 (Cont.) Using SOR Routines to Merge Three Input Files in a Compaq Pascal Program

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);
(continued on next page)
Example 18–3 (Cont.) Using SOR Routines to Merge Three Input Files in a Compaq Pascal Program

```pascal
status := sor$end_sort;
if not odd(status) then error(status);
writeln( 'merge successful.' );
end.
```
Example 18–4 is a Compaq Pascal program that demonstrates a sort operation using a record interface.

**Example 18–4 Using SOR Routines to Sort Records from Two Input Files in a Compaq Pascal Program**

Pascal Program

```pascal
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 ;
```

(continued on next page)
Example 18–4 (Cont.) Using SOR Routines to Sort Records from Two Input Files in a Compaq Pascal Program

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 *)
    Recordbuffer : 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 SOR functions *)
(* Note that the following SOR 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 ;
    %STDESCR Recordsize : $UWORD )
    : INTEGER ; EXTERN ;

FUNCTION SOR$END_SORT
    : INTEGER ; EXTERN ;

(* End of the SOR 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( ‘ SOR unsuccessful. Error status = ’ , status:8 hex ) ;
            SYS$EXIT( status ) ;
        end ;
    end ; (* end checkstatus *)

(continued on next page)
Example 18–4 (Cont.) Using SOR Routines to Sort Records from Two Input Files in a Compaq
Pascal Program

(* 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 SOR data types. *)
        Keyorder := Ascending;
        Keyoffset := 0;
        Keylength := 8;
        END;
    END;

Sorttype := SOR$GK_RECORD; (* Use the global SOR constant to
define the sort process. *)
Sortoptions := SOR$M_STABLE; (* Use the global SOR constant to
define the stable sort option. *)

Numworkfiles := 3;

(* call the sort routines as a series of functions *)

(* pass the first filename to SOR *)
Status := SOR$PASS_FILES( Inputname ) ;
CHECKSTATUS( Status ) ;
(* pass the second filename to SOR *)
Inputname := 'PASINPUT2.DAT';
Status := SOR$PASS_FILES( Inputname ) ;
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 SOR. *)
OPEN( SORTOUT, 'TEMP.TMP' ) ;
REWRITE( SORTOUT ) ;

(* Now get the sorted records from SOR. *)
Recordlength := 0 ;
REPEAT
  Status := SOR$RETURN_REC( Rec, Recordlength ) ;

(continued on next page)
Example 18–4 (Cont.) Using SOR Routines to Sort Records from Two Input Files in a Compaq Pascal Program

```pascal
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 18–5 is a Compaq C program that demonstrates a sort operation using the STABLE option and two test keys.

Example 18–5  Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a Compaq 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 DONALD
JONES DAVID
JONES DAVID
JONES DONALD
RUSSO JOSEPH
SMART SHERYL
SMITH RANDY
SMITTS JAMES
WARNER LIZZY

*/

(continued on next page)
## Example 18–5 (Cont.) Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a Compaq C Program

/*
**=================================================================
**  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 <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <descrip.h>
#include <ssdef.h>
#include <sor$routines.h>

/*
** 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.
*/
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.
*/
int SORM_STABLE;

(continued on next page)
Example 18–5 (Cont.) Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a Compaq C Program

/* ----------------------------------------------------------------------------
** 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];
    SDESCRIPTOR (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);
    }

    (continued on next page)
Example 18–5 (Cont.) Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a Compaq C Program

/* 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);

(continued on next page)
Example 18–5 (Cont.) Using SOR Routines to Sort Records Using the STABLE Option and Two Text Keys in a Compaq C Program

/* 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);

18.4 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† Octaword integer (signed)

†Not currently supported by the high-performance Sort/Merge utility.
Sort/Merge (SOR) Routines

SOR$BEGIN_MERGE

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSC$K_DTYPE_OU†</td>
<td>Octaword (unsigned)</td>
</tr>
<tr>
<td>DSC$K_DTYPE_F</td>
<td>Single-precision floating</td>
</tr>
<tr>
<td>DSC$K_DTYPE_D</td>
<td>Double-precision floating</td>
</tr>
<tr>
<td>DSC$K_DTYPE_G</td>
<td>G-format floating</td>
</tr>
<tr>
<td>DSC$K_DTYPE_H†</td>
<td>H-format floating</td>
</tr>
<tr>
<td>DSC$K_DTYPE_FS‡</td>
<td>IEEE single-precision S floating</td>
</tr>
<tr>
<td>DSC$K_DTYPE_FT‡</td>
<td>IEEE double-precision T floating</td>
</tr>
<tr>
<td>DSC$K_DTYPE_T</td>
<td>Text (may be influenced by collating sequence)</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NU</td>
<td>Numeric string, unsigned</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NL</td>
<td>Numeric string, left separate sign</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NLO</td>
<td>Numeric string, left overpunched sign</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NR</td>
<td>Numeric string, right separate sign</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NRO</td>
<td>Numeric string, right overpunched sign</td>
</tr>
<tr>
<td>DSC$K_DTYPE_NZ†</td>
<td>Numeric string, zoned sign</td>
</tr>
<tr>
<td>DSC$K_DTYPE_P</td>
<td>Packed decimal string</td>
</tr>
</tbody>
</table>

†Not currently supported by the high-performance Sort/Merge utility.
‡Alpha specific.

The OpenVMS Programming Interfaces: Calling a System Routine 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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$M_STABLE</td>
<td>Keeps records with equal keys in the same order as they appeared on input.</td>
</tr>
<tr>
<td>SOR$M_EBCDIC</td>
<td>Orders ASCII character keys according to EBCDIC collating sequence. No translation takes place.</td>
</tr>
<tr>
<td>SOR$M_MULTI</td>
<td>Orders character keys according to the multinational collating sequence, which collates the international character set.</td>
</tr>
<tr>
<td>SOR$M_NOSIGNAL</td>
<td>Returns a status code instead of signaling errors.</td>
</tr>
<tr>
<td>SOR$M_NODUPS</td>
<td>Omits records with duplicate keys. You cannot use this option if you specify your own equal-key routine.</td>
</tr>
<tr>
<td>SOR$M_SEQ_CHECK</td>
<td>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.</td>
</tr>
</tbody>
</table>

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
Sort/Merge (SOR) Routines
SOR$BEGIN_MERGE

**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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$DELETE1</td>
<td>Delete the first record from the merge.</td>
</tr>
<tr>
<td>SOR$DELETE2</td>
<td>Delete the second record from the merge.</td>
</tr>
<tr>
<td>SOR$DELBOTH</td>
<td>Delete both records from the merge.</td>
</tr>
<tr>
<td>SS$NORMAL</td>
<td>Keep both records in the merge.</td>
</tr>
</tbody>
</table>

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.

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$_NORMAL Success.
SOR$_BADDDTYPE 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$_BAD_LRL Record length n greater than specified longest record length.
<p>| SOR$_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$_ENDDIAGS | Completed with diagnostics. |
| SOR$_ILLEBAS | 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$_MILENOFF | Length and offset required. |
| SOR$_MISS_PARAM | A required subroutine argument is missing. |
| SOR$_MULTIDIM | Invalid multidimensional OCCURS. |
| SOR$_NODUPEX | 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$_RTNERROR | Unexpected error status from user-written routine. |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$_SIGNCOMPQ</td>
<td>Absolute Date and Time data type represented in 1-second units.</td>
</tr>
<tr>
<td>SOR$_SORT_ON</td>
<td>Sort or merge routines called in incorrect order.</td>
</tr>
<tr>
<td>SOR$_SPCIVC</td>
<td>Invalid collating sequence specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVD</td>
<td>Invalid data type at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVF</td>
<td>Invalid field specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVI</td>
<td>Invalid include or omit specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVK</td>
<td>Invalid key or data specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVP</td>
<td>Invalid sort process at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVS</td>
<td>Invalid specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCIVX</td>
<td>Invalid condition specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCMIS</td>
<td>Invalid merge specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCOVR</td>
<td>Overridden specification at line n.</td>
</tr>
<tr>
<td>SOR$_SPCSI</td>
<td>Invalid sort specification at line n.</td>
</tr>
<tr>
<td>SOR$_SRTIWA</td>
<td>Insufficient space. The specification file is too complex.</td>
</tr>
<tr>
<td>SOR$_STABLEEX</td>
<td>Equal-key routine and stable option cannot both be specified.</td>
</tr>
<tr>
<td>SOR$_SYSERROR</td>
<td>System service error.</td>
</tr>
<tr>
<td>SOR$_UNDOPTION</td>
<td>Undefined option flag was set.</td>
</tr>
<tr>
<td>SOR$_UNSUPLEVL</td>
<td>Unsupported core level for record name.</td>
</tr>
<tr>
<td>SOR$_WRITEERR</td>
<td>Error writing file.</td>
</tr>
</tbody>
</table>
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

                [,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† Octaword integer (signed)
DSC$K_DTYPE_OU† Octaword (unsigned)

†Not currently supported by the high-performance Sort/Merge utility.
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† H-format floating
DSC$K_DTYPE_FS‡ IEEE single-precision S floating
DSC$K_DTYPE_FT‡ 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$K_DTYPE_NZ† Numeric string, zoned sign
DSC$K_DTYPE_P Packed decimal string

†Not currently supported by the high-performance Sort/Merge utility.
‡Alpha specific.

The OpenVMS Programming Interfaces: Calling a System Routine 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.

Irl
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 Irl 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.
### Sort/Merge (SOR) Routines

#### SOR$BEGIN_SORT

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$M_STABLE</td>
<td>Keeps records with equal keys in the same order in which they appeared on input. With multiple input files that have records that collate as equal, records from the first input file are placed before the records from the second input file, and so on.</td>
</tr>
<tr>
<td>SOR$M_EBCDIC</td>
<td>Orders ASCII character keys according to EBCDIC collating sequence. No translation takes place.</td>
</tr>
<tr>
<td>SOR$M_MULTI</td>
<td>Orders character keys according to the multinational collating sequence, which collates the international character set.</td>
</tr>
<tr>
<td>SOR$M_NOSIGNAL</td>
<td>Returns a status code instead of signaling errors.</td>
</tr>
<tr>
<td>SOR$M_NODUPS</td>
<td>Omits records with duplicate keys. You cannot use this option if you specify your own equal-key routine.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$DELETE1</td>
<td>Delete the first record from the sort.</td>
</tr>
<tr>
<td>SOR$DELETE2</td>
<td>Delete the second record from the sort.</td>
</tr>
<tr>
<td>SOR$DELBOTH</td>
<td>Delete both records from the sort.</td>
</tr>
<tr>
<td>SS$NORMAL</td>
<td>Keep both records in the sort.</td>
</tr>
</tbody>
</table>

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)
Sort/Merge (SOR) Routines

**SOR$BEGIN_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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>SOR$_BADLOGIC</td>
<td>Internal logic error detected.</td>
</tr>
<tr>
<td>SOR$_BAD_KEY</td>
<td>Invalid key specification.</td>
</tr>
<tr>
<td>SOR$_BAD_LRL</td>
<td>Record length $n$ greater than specified longest record length.</td>
</tr>
<tr>
<td>SOR$_BAD_MERGE</td>
<td>Number of work files must be between 0 and 10. (For the high-performance Sort/Merge utility, the maximum number is 255.)</td>
</tr>
<tr>
<td>SOR$_BAD_TYPE</td>
<td>Invalid sort process specified.</td>
</tr>
<tr>
<td>SOR$_ENDDIAGS</td>
<td>Completed with diagnostics.</td>
</tr>
<tr>
<td>SOR$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>SOR$_KEYAMBINC</td>
<td>Key specification is ambiguous or inconsistent.</td>
</tr>
<tr>
<td>SOR$_KEY_LEN</td>
<td>Invalid key length, key number $n$, length $n$.</td>
</tr>
<tr>
<td>SOR$_LRL_MISS</td>
<td>Longest record length must be specified.</td>
</tr>
<tr>
<td>SOR$_NODUPEXC</td>
<td>Equal-key routine and no-duplicates option cannot both be specified.</td>
</tr>
<tr>
<td>SOR$_NUM_KEY</td>
<td>Too many keys specified.</td>
</tr>
<tr>
<td>SOR$_NYI</td>
<td>Not yet implemented.</td>
</tr>
<tr>
<td>SOR$_RTNERROR</td>
<td>Unexpected error status from user-written routine.</td>
</tr>
<tr>
<td>SOR$_SORT_ON</td>
<td>Sort or merge routine called in incorrect order.</td>
</tr>
<tr>
<td>SOR$_STABLEEXC</td>
<td>Equal-key routine and stable option cannot both be specified.</td>
</tr>
<tr>
<td>SOR$_SYSERROR</td>
<td>System service error.</td>
</tr>
<tr>
<td>SOR$_UNDOPTION</td>
<td>Undefined option flag was set.</td>
</tr>
</tbody>
</table>
SOR$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

```plaintext
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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$K_ROUTINE</td>
<td>The p1 argument should be interpreted as the address of the procedure value of a routine that SORT/MERGE will call to compare keys described by the dtype_code returned by the call to SOR$DTYPE.</td>
</tr>
<tr>
<td>SOR$K_NCS_TABLE</td>
<td>The p1 argument should be interpreted as the address of a collating sequence identification returned by a call to NCS$GET_CS. SORT/MERGE will use this collating sequence to compare keys described by the dtype_code returned by the call to SOR$DTYPE.</td>
</tr>
</tbody>
</table>

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$DTYPE to allocate a key data type code that can then be used in the the key_buffer argument to SOR$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.
Sort/Merge (SOR) Routines

SOR$END_SORT

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>SOR$_CLOSEIN</td>
<td>Error closing file as input.</td>
</tr>
<tr>
<td>SOR$_CLOSEOUT</td>
<td>Error closing file as output.</td>
</tr>
<tr>
<td>SOR$_ENDDIAGS</td>
<td>Completed with diagnostics.</td>
</tr>
<tr>
<td>SOR$_END_SORT</td>
<td>SORT/MERGE terminated, context = context.</td>
</tr>
<tr>
<td>SOR$_SYSERROR</td>
<td>System service error.</td>
</tr>
</tbody>
</table>
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


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 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 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
Sort/Merge (SOR) Routines

SOR$PASS_FILES

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-Digital 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:

<table>
<thead>
<tr>
<th>File Organization</th>
<th>Acceptable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>0 to 32,767</td>
</tr>
<tr>
<td>Relative</td>
<td>0 to 16,383</td>
</tr>
<tr>
<td>Indexed sequential</td>
<td>0 to 16,362</td>
</tr>
</tbody>
</table>

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 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

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>SOR$_DUP_OUTPUT</td>
<td>Output file has already been specified.</td>
</tr>
<tr>
<td>SOR$_ENDDIAGS</td>
<td>Completed with diagnostics.</td>
</tr>
<tr>
<td>SOR$_INP_FILES</td>
<td>Too many input files specified.</td>
</tr>
<tr>
<td>SOR$_NYI</td>
<td>Not yet implemented.</td>
</tr>
<tr>
<td>SOR$_SORT_ON</td>
<td>Sort or merge routine called in incorrect order.</td>
</tr>
<tr>
<td>SOR$_SYSERROR</td>
<td>System service error.</td>
</tr>
</tbody>
</table>
SOR$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$_SYSErrR  System service error.
SOR$_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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>SOR$BADLOGIC</td>
<td>Internal logic error detected.</td>
</tr>
<tr>
<td>SOR$ENDDIAGS</td>
<td>Completed with diagnostics.</td>
</tr>
<tr>
<td>SOR$EXTEND</td>
<td>Unable to extend work file for needed space.</td>
</tr>
<tr>
<td>SOR$MISS_PARAM</td>
<td>A required subroutine argument is missing.</td>
</tr>
<tr>
<td>SOR$OPERFAIL</td>
<td>Error requesting operator service.</td>
</tr>
<tr>
<td>SOR$READERR</td>
<td>Error reading file.</td>
</tr>
<tr>
<td>SOR$REQ_ALT</td>
<td>Specify alternate name file (or specify nothing to simply try again).</td>
</tr>
<tr>
<td>SOR$RTNERROR</td>
<td>Unexpected error status from user-written routine.</td>
</tr>
<tr>
<td>SOR$SORT_ON</td>
<td>Sort or merge routines called in incorrect order.</td>
</tr>
<tr>
<td>SOR$SYSERROR</td>
<td>System service error.</td>
</tr>
<tr>
<td>SOR$USE_ALT</td>
<td>Using alternate file name.</td>
</tr>
<tr>
<td>SOR$WORK_DEV</td>
<td>Work file name must be on random access local device.</td>
</tr>
</tbody>
</table>
SOR$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$_NORMAL**: Normal successful completion.
- **SOR$_BADTYPE**: Invalid or unsupported CDD data type.
- **SOR$_BADLENOFF**: Length and offset must be multiples of 8 bits.
- **SOR$_BADLOGIC**: Internal logic error detected.
- **SOR$_BADOCURS**: Invalid OCCURS clause.
- **SOR$_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$_ILLBASE**: Nondecimal base is invalid.
- **SOR$_ILL_LITERAL**: 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.
<table>
<thead>
<tr>
<th><strong>SOR$</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_READERR</code></td>
<td>Error reading file.</td>
</tr>
<tr>
<td><code>REQ_ALT</code></td>
<td>Specify alternate name file (or nothing to try again).</td>
</tr>
<tr>
<td><code>RTNERROR</code></td>
<td>Unexpected error status from user-written routine.</td>
</tr>
<tr>
<td><code>SIGNCOMPQ</code></td>
<td>Absolute Date and Time data type represented in 1-second units.</td>
</tr>
<tr>
<td><code>SORT_ON</code></td>
<td>Sort or merge routines called in incorrect order.</td>
</tr>
<tr>
<td><code>SPCIVC</code></td>
<td>Invalid collating sequence specification, at line n.</td>
</tr>
<tr>
<td><code>SPCIVD</code></td>
<td>Invalid data type, at line n.</td>
</tr>
<tr>
<td><code>SPCIVF</code></td>
<td>Invalid field specification, at line n.</td>
</tr>
<tr>
<td><code>SPCIVI</code></td>
<td>Invalid include or omit specification, at line n.</td>
</tr>
<tr>
<td><code>SPCIVK</code></td>
<td>Invalid key or data specification, at line n.</td>
</tr>
<tr>
<td><code>SPCIVP</code></td>
<td>Invalid sort process, at line n.</td>
</tr>
<tr>
<td><code>SPCIVS</code></td>
<td>Invalid specification, at line n.</td>
</tr>
<tr>
<td><code>SPCIVX</code></td>
<td>Invalid condition specification, at line n.</td>
</tr>
<tr>
<td><code>SPCMIS</code></td>
<td>Invalid merge specification, at line n.</td>
</tr>
<tr>
<td><code>SCOVR</code></td>
<td>Overridden specification, at line n.</td>
</tr>
<tr>
<td><code>SPCSIS</code></td>
<td>Invalid sort specification, at line n.</td>
</tr>
<tr>
<td><code>SRTIWA</code></td>
<td>Insufficient space. Specification file is too complex.</td>
</tr>
<tr>
<td><code>SYSERROR</code></td>
<td>System service error.</td>
</tr>
<tr>
<td><code>UNSUPLEVL</code></td>
<td>Unsupported core level for record name.</td>
</tr>
<tr>
<td><code>USE_ALT</code></td>
<td>Using alternate file name.</td>
</tr>
<tr>
<td><code>WORK_DEV</code></td>
<td>Work file name must be on random access local device.</td>
</tr>
<tr>
<td><code>WRITEERR</code></td>
<td>Error writing file.</td>
</tr>
</tbody>
</table>
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

```plaintext
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 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: SOR$K_REC_INP, SOR$K_REC_SOR, SOR$K_REC_OUT, SOR$K_LRL_INP.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$K_IDENT</td>
<td>Address of ASCII string for version number</td>
</tr>
<tr>
<td>SOR$K_REC_INP</td>
<td>Number of records input</td>
</tr>
<tr>
<td>SOR$K_REC_SOR</td>
<td>Records sorted</td>
</tr>
<tr>
<td>SOR$K_REC_OUT</td>
<td>Records output</td>
</tr>
<tr>
<td>SOR$K_LRL_INP</td>
<td>Longest record length (LRL) for input</td>
</tr>
<tr>
<td>SOR$K_LRL_INT</td>
<td>Internal LRL</td>
</tr>
<tr>
<td>SOR$K_LRL_OUT</td>
<td>LRL for output</td>
</tr>
<tr>
<td>SOR$K_NODES</td>
<td>Nodes in sort tree</td>
</tr>
<tr>
<td>SOR$K_INI_RUNS</td>
<td>Initial dispersion runs</td>
</tr>
<tr>
<td>SOR$K_MRG_ORDER</td>
<td>Maximum merge order</td>
</tr>
<tr>
<td>SOR$K_MRG_PASSES</td>
<td>Number of merge passes</td>
</tr>
<tr>
<td>SOR$K_WRK_ALQ</td>
<td>Work file allocation</td>
</tr>
</tbody>
</table>
Sort/Merge (SOR) Routines

SOR$STAT

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$K_MBC_INP</td>
<td>Multiblock count for input</td>
</tr>
<tr>
<td>SOR$K_MBC_OUT</td>
<td>Multiblock count for output</td>
</tr>
<tr>
<td>SOR$K_MBF_INP</td>
<td>Multibuffer count for input</td>
</tr>
<tr>
<td>SOR$K_MBF_OUT</td>
<td>Multibuffer count for output</td>
</tr>
</tbody>
</table>

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.
Access control list editor routines
   See ACL editor routines
ACLEDIT$EDIT routine, ACL–3
ACL editor routines
   editing, ACL–3
   example of BLISS program, ACL–1
   manipulating, ACL–1
   options available, ACL–4
ACLEDT$SECTION logical name, ACL–5
ACLs (access control lists)
   See ACL editor routines
BACKUP$START, BCK–4
Backup API, BCK–1
   example of C program, BCK–2
Binary data, compression of, DCX–1
Buckets
   reclaiming with CONV$RECLAIM routine, CONV–20
   reclamation statistics, CONV–21
Command language definition file
   template for UTIL$CQUAL_FILE_PARSE, CQUAL–6
Command language interface routines
   See CLI routines
Command line qualifiers, CQUAL–2
Command strings
   See also CLI routines
   checking for presence of, CLI–12
   dispatching to action routine, CLI–8
   obtaining values, CLI–9
   parsing a DCL command string, CLI–5
   positional qualifiers, CLI–13
   processing with CLI routines, CLI–1
   prompting for input, CLI–6
Command tables
   using with CLI routines, CLI–1, CLI–6
Common file qualifier routines, CQUAL–1
Context variables, with DCX routines, DCX–17
CONV$CONVERT routine, CONV–8
CONV$PASS_FILES routine, CONV–12
CONV$PASS_OPTIONS routine, CONV–15
CONV$RECLAIM routine, CONV–20
CONVERT command
   list of qualifiers, CONV–16
   passing options, CONV–15, CONV–18
   setting qualifiers, CONV–15
CONVERT routines, CONV–1
   examples, CONV–1 to CONV–6
   list of, CONV–1
   using wildcard characters in, CONV–13
Convert utility (CONVERT)
   conversion statistics, CONV–8
Convert utility routines
   See CONVERT routines
Callable interface of DECTPU routines, DECTPU–1
CLI$DCL_PARSE routine, CLI–5
CLI$DISPATCH routine, CLI–8
CLI$GET_VALUE routine, CLI–9
CLI$PRESENT routine, CLI–12
CLI routines, CLI–1
   See also Command strings
   example in a Fortran program, CLI–2
   list of, CLI–1
   when to use, CLI–1
Data compression
   See also DCX routines
   algorithm for submitting all data records, DCX–16
   analysis preceding compression, DCX–14
   size of data after compression, DCX–1
Database compression facility routines
   See DCX routines
Data expansion, DCX–23
   See also DCX routines
   initializing, DCX–26
Data Expansion facility routines
See DCX routines

Data records
analysis, DCX–13
compression, DCX–1
conversion, CONV–1
expansion, DCX–1

DCL command strings
See Command strings

DCX$ANALYZE_DATA routine, DCX–12
DCX$ANALYZE_DONE routine, DCX–14
DCX$ANALYZE_INIT routine, DCX–15
DCX$COMPRESS_DATA routine, DCX–18
DCX$COMPRESS_DONE routine, DCX–20
DCX$COMPRESS_INIT routine, DCX–21
DCX$EXPAND_DATA routine, DCX–23
DCX$EXPAND_DONE routine, DCX–25
DCX$EXPAND_INIT routine, DCX–26
DCX$MAKE_MAP routine, DCX–28
DCX routines, DCX–1
examples, DCX–11 to DCX–29

DEC Text Processing Utility routines
See DECTPU routines

DECTPU callable interface
See DECTPU routines

DECTPU routines, DECTPU–1
bound procedure parameter value, DECTPU–3
callable DECTPU, DECTPU–1 to DECTPU–5
condition handler, DECTPU–3 to DECTPU–4
examples, DECTPU–4, DECTPU–7 to
DECTPU–27
list of, DECTPU–6
shareable image, DECTPU–1, DECTPU–3
user-written
FILEIO, DECTPU–68
FILE_PARSE, DECTPU–70
FILE_SEARCH, DECTPU–72
HANDLER, DECTPU–75
INITIALIZE, DECTPU–76
requirements, DECTPU–7
USER, DECTPU–77

Device access, controlling through access control lists, ACL–1

Index–2
Job controllers
  function, PSM–3
  request to symbiont, SMB–5

Keys (in records)
  See Sort/Merge utility
Keyword paths
  obtaining values of command string keywords, CLI–9
  referencing command string keywords, CLI–12

LBR$CLOSE routine, LBR–7, LBR–8, LBR–28
LBR$DELETE_DATA routine, LBR–16, LBR–29
LBR$DELETE_KEY routine, LBR–16, LBR–31
LBR$FIND routine, LBR–33
LBR$FLUSH routine, LBR–35
LBR$GET_HEADER routine, LBR–23, LBR–37
LBR$GET_INDEX routine, LBR–14, LBR–46
LBR$GET_MODULE routine, LBR–22, LBR–75
LBR$GET_MOVE routine, LBR–77
LBR$GET_HELP routine, LBR–39
LBR$GET_HISTORY routine, LBR–42
LBR$GET_INDEX routine, LBR–26, LBR–44
LBR$GET_LIBRARY routine, LBR–14, LBR–46
LBR$INIni CONTROL routine, LBR–8, LBR–48
LBR$INSERT_KEY routine, LBR–10, LBR–50
LBR$LOOKUP_KEY routine, LBR–10, LBR–14, LBR–16, LBR–22, LBR–52
LBR$OPEN routine, LBR–7, LBR–8, LBR–54
LBR$OUTPUT_HELP routine, LBR–25, LBR–58
LBR$PUT_END routine, LBR–10, LBR–62
LBR$PUT_INDEX routine, LBR–26, LBR–63
LBR$PUT_LIBRARY routine, LBR–10, LBR–65
LBR$REPLACE_HEADER routine, LBR–10, LBR–67
LBR$REPLACE_KEY routine, LBR–10, LBR–50
LBR$RETURN RMSSTV routine, LBR–69
LBR$SEARCH routine, LBR–70
LBR$SET_INDEX routine, LBR–72
LBR$SET_LOCATE routine, LBR–74
LBR$SET_MODULE routine, LBR–22, LBR–75
LBR$SET.Move routine, LBR–77
LBR routines, LBR–1
  control index, LBR–6
  data records
    reading, LBR–46
    writing, LBR–65
  end-of-module record, writing, LBR–62
  examples, LBR–6 to LBR–18
    creating a new library, LBR–8
    deleting a module from a library, LBR–16 to LBR–18
    inserting a module into a library, LBR–10 to LBR–13
  header, LBR–2
  help text
  outputting, LBR–58
  retrieving, LBR–39
  index, LBR–2
  index searching, LBR–70
  library
    closing, LBR–28
    creating, LBR–54
    opening, LBR–54
    structure, LBR–2 to LBR–5
    types, LBR–1
  library file, flushing, LBR–35
  library header information, LBR–37
  library index
    getting contents, LBR–44
    initializing, LBR–48
    searching for key, LBR–44
  library key, LBR–2
    creating ASCII or binary, LBR–55
    deleting, LBR–31
    finding, LBR–33
    inserting, LBR–50
    looking up, LBR–52
    replacing, LBR–67
  library update history record, retrieving, LBR–42
  locate mode, setting record access mode to, LBR–74
  module, LBR–2
    accessing with RFA, LBR–33
    deleting data records, LBR–29
    deleting header, LBR–29
    headers, LBR–75
  move mode, setting record access to, LBR–77
  RMS status value, returning, LBR–69
  setting current index number, LBR–72
  summary, LBR–6
  update history records, writing, LBR–63
  virtual memory, recovering, LBR–35

LDAP, LDAP–1
LGI$ICB_ACCTEXPIRED callback routine, LGI–35
LGI$ICB_AUTOCHECKIN callback routine, LGI–36
LGI$ICB_AUTOLOGIN callback routine, LGI–37
LGI$ICB_CHECK_PASS callback routine, LGI–38
LGI$ICB_DELETE callback routine, LGI–39
LGI$ICB_GET_INPUT callback routine, LGI–40
LGI$ICB_MODALHOURS callback routine, LGI–41
LGI$ICB_PASSWORD callback routine, LGI–42
LGI$ICB_PWDEXPIRED callback routine, LGI–43
LGI$ICB_USERPARSE callback routine, LGI–45
LGICB_USERPROMPT callback routine, LGI–46
LGICB_VALIDATE callback routine, LGI–47
LGICR_AUTHENTICATE callout routine, LGI–15
LGICR_CHKRESTRIC callout routine, LGI–18
LGICR_DECWINIT callout routine, LGI–20
LGICR_FINISH callout routine, LGI–22
LGICR_JACT_START callout routine, LGI–24
LGICR_IDENTIFY callout routine, LGI–26
LGICR_INIT routine, LGI–28
LGICR_JOBSTEP callout routine, LGI–30
LGICR_LOGOUT callout routine, LGI–32
LGIC callouts, LGI–3
LIB_INSERT_KEY routine, LBR–19
LIB_SET_INDEX routine, LBR–19
Librarian utility routines
   See LBR routines
Libraries
   closing, LBR–7, LBR–8
   deleting module, LBR–16
   initializing, LBR–8
   inserting module, LBR–10
   listing index entries, LBR–26
   module header, LBR–22
   multiple indexes, LBR–19
   multiple keys, LBR–19
   opening, LBR–7, LBR–8
   processing index entry, LBR–26
   replacing module, LBR–10
Lightweight Directory Access Protocol (LDAP), LDAP–1
LOGINOUT callback routines, LGI–34 to LGI–48
LOGINOUT callout routines, LGI–14 to LGI–33
LOGINOUT callouts
   See LGI callouts
LOGINOUT sample program, LGI–10

M
MAIL_MAILFILE_BEGIN routine, MAIL–32
MAIL_MAILFILE_CLOSE routine, MAIL–34
MAIL_MAILFILE_COMPRESS routine, MAIL–36
MAIL_MAILFILE_END routine, MAIL–39
MAIL_MAILFILE_INFO_FILE routine, MAIL–41
MAIL_MAILFILE_MODIFY routine, MAIL–44
MAIL_MAILFILE_OPEN routine, MAIL–47
MAIL_MAILFILE_PURGE_WASTE routine, MAIL–50
MAIL_MESSAGE_BEGIN routine, MAIL–53
MAIL_MESSAGE_COPY routine, MAIL–55
MAIL_MESSAGE_DELETE routine, MAIL–59
MAIL_MESSAGE_END routine, MAIL–61
MAIL_MESSAGE_GET routine, MAIL–63
MAIL_MESSAGE_INFO routine, MAIL–68
MAIL_MESSAGE_MODIFY routine, MAIL–72
MAIL_MESSAGE_SELECT routine, MAIL–75
MAIL_SEND_ABORT routine, MAIL–78
MAIL_SEND_ADD_ADDRESS routine, MAIL–80
MAIL_SEND_ADD_ATTRIBUTE routine, MAIL–82
MAIL_SEND_ADD_BODYPART routine, MAIL–85
MAIL_SEND_BEGIN routine, MAIL–88
MAIL_SEND_END routine, MAIL–91
MAIL_SEND_MESSAGE routine, MAIL–93
MAIL_USER_BEGIN routine, MAIL–95
MAIL_USER_DELETE_INFO routine, MAIL–98
MAIL_USER_END routine, MAIL–100
MAIL_USER_GET_INFO routine, MAIL–102
MAIL_USER_SET_INFO routine, MAIL–106
Mailboxes, controlling access through access control lists, ACL–1
Mail utility (MAIL), MAIL–1
   action routine, MAIL–7
      calling sequence, MAIL–8
      folder, MAIL–11, MAIL–14
      mail file, MAIL–14
      send, MAIL–16
      address list, MAIL–16
      creating, MAIL–16
      user name type, MAIL–16
   bodypart, creating, MAIL–15
   condition handling, MAIL–6
   context, MAIL–3
      initiating, MAIL–4
      mail file, MAIL–9
      message, MAIL–11
      send, MAIL–14
      terminating, MAIL–4
      user profile, MAIL–17
   deleted bytes threshold, MAIL–11
   disk space, reclaiming, MAIL–11
   folder, MAIL–2
      creating, MAIL–14
      deleting, MAIL–14
   folder names, displaying, MAIL–11
   item code, MAIL–7
      input, MAIL–7, MAIL–18
      output, MAIL–7, MAIL–21
   item descriptor
      declaring, MAIL–7
      null, MAIL–7
   item list, MAIL–6
      declaring, MAIL–7
      terminating, MAIL–7
   mail file, MAIL–3
      alternate, MAIL–10
      closing, MAIL–9
      compressing, MAIL–10
      creating, MAIL–14
      default, MAIL–9
National character set (NCS) routines

See NCS routines

NC$COMPARE routine, NCS–8
NC$CONVERT routine, NCS–10
NC$END_CF routine, NCS–12
NC$END_CS routine, NCS–13
NC$GET_CF routine, NCS–14
NC$GET_CS routine, NCS–16
NC$RESTORE_CF routine, NCS–18

NCS$RESTORE_CS routine, NCS–20
NC$SAVE_CF routine, NCS–22
NC$SAVE_CS routine, NCS–24

NCS routines

C program sample, NCS–4
Fortran program sample, NCS–2
list of, NCS–1
typical application of, NCS–2

OpenVMS RMS

See RMS

Options, creating with LBR$OPEN routine, LBR–8

Print Symbiont Modification routines

See PSM routines

Print symbionts

See Symbionts

Prolog 3 indexed files

creating with CONVERT routines, CONV–16
reclaiming, CONV–20
with Convert/Reclaim utility, CONV–1

Prolog files, using with CONVERT routines, CONV–16

Prompt strings, setting with CLI$DCL_PARSE, CLI–7

PSM$PRINT routine, PSM–21
PSM$READ_ITEM_DX routine, PSM–23
PSM$REPLACE routine, PSM–25
PSM$REPORT routine, PSM–30

PSM routines, PSM–1

examples, PSM–16 to PSM–20
USER-FORMAT-ROUTINE, PSM–33
USER-INPUT-ROUTINE, PSM–37
USER-OUTPUT-ROUTINE, PSM–43

Queues

execution of, PSM–3
generic, PSM–3

Records

See Data records

RMS

control blocks with FDL routines, FDL–16, FDL–19
S

Simplified callable interface
  See DECTPU routines
  SMB$CHECK_FOR_MESSAGE routine, SMB–14
  SMB$INITIALIZE routine, SMB–15
  SMB$READ_MESSAGE routine, SMB–17
  SMB$READ_MESSAGE_ITEM routine, SMB–20
  SMB$SEND_TO_JOBCTL routine, SMB–30
  SMB routines, SMB–1
  SOR$STAT routine, SOR–61
  SOR$BEGIN_MERGE routine, SOR–29
  SOR$BEGIN_SORT routine, SOR–36
  SOR$DTYPE routine, SOR–42
  SOR$END_SORT routine, SOR–45
  SOR$FILE routine, SOR–47
  SOR$RELEASE_REC routine, SOR–52
  SOR$RETURN_REC routine, SOR–54
  SOR$SORT_MERGE routine, SOR–56
  SOR$SPEC_FILE routine, SOR–59
  SOR routines, SOR–4
    examples, SOR–8 to SOR–28
    interface
      file, SOR–5
      record, SOR–5
    list of, SOR–4
    reentrancy using context argument, SOR–6
Sort/Merge routines
  See SOR routines
  Sort/Merge utility (high-performance), SOR–1
  Sort/Merge utility (SORT/MERGE)
    See also SOR routines
    See also Sort/Merge utility (high-performance)
    keys, SOR–4
Symbiont/Job Controller Interface routines
  See SMB routines
Symbionts
  See also Queues
  active stream, PSM–3
  allocating memory, SMB–4
  carriage control, processing of, PSM–10
  connecting to a device, SMB–4
  demand input routines, PSM–5
  device, SMB–2
  environments, SMB–5
  function of, PSM–3, SMB–2
  input, SMB–2, SMB–1
    INPSMB.EXE file, SMB–1
  internal logic, PSM–4
    main format routine, PSM–12
    main input routine, PSM–9
    main output routine, PSM–13
  invoking, PSM–21
  job controller, communication with, SMB–1
  job controller requests, SMB–5
  asynchronous, SMB–6

Symbionts
  job controller requests (cont’d)
    processing, SMB–10
    reading, SMB–10
    responding, SMB–13
    synchronous, SMB–5
  modiﬁed, PSM–1, SMB–6, SMB–9
  format routine, PSM–11
  guidelines, PSM–7
  initialization routine, PSM–14
  input routine, PSM–9
  integration of routines, PSM–15
  output routine, PSM–12
  restrictions, PSM–7
  multiple streams, PSM–3, SMB–9
  multithreaded, PSM–3
  output, PSM–2, SMB–1
    PRTSMB.EXE file, SMB–1
  printer, SMB–1
    processing, PSM–1
  process-permanent file, SMB–4
  server, SMB–2, SMB–1
  single stream, PSM–3, SMB–9
  system MAXBUF parameter, PSM–6
  type, SMB–1
  user-written, PSM–1, SMB–1, SMB–3
  user-written guidelines, SMB–3
  user-written interfaces, PSM–6
  Symbiont thread, PSM–3
  SYS$OUTPUT_HELP routine, LBR–8

T

Text compression, DCX–1
Text-processing routines
  See DECTPU routines
  TPU$CLEANUP routine, DECTPU–28
  TPU$CLIPARSE routine, DECTPU–32
  TPU$CLOSE_TERMINAL routine, DECTPU–34
  TPU$CONTROL routine, DECTPU–35
    regaining control from, DECTPU–35,
      DECTPU–53
  TPU$EDIT routine, DECTPU–37
  TPU$EXECUTE_COMMAND routine, DECTPU–39
  TPU$EXECUTE_INIFILE routine, DECTPU–40
  TPU$FILE10 routine, DECTPU–42
  TPU$HANDLER routine, DECTPU–52
  TPU$MESSAGE routine, DECTPU–61
  TPU$PARSEINFO routine, DECTPU–62
  TPU$SPECIFY_ASYNC_ACTION routine, DECTPU–6,
    DECTPU–35, DECTPU–53, DECTPU–64
  TPU$TPU routine, DECTPU–66
TPU$TRIGGER_ASYNC_ACTION routine,
DECTPU–6, DECTPU–35, DECTPU–53,
DECTPU–67

U
User-written DECTPU routines
See DECTPU routines

W
Wildcard characters, using with CONVERT
routines, CONV–13

UTIL$CQUAL
example, CQUAL–7
UTIL$CQUAL routines, CQUAL–1
UTIL$CQUAL_CONFIRM_ACT, CQUAL–5,
CQUAL–18
UTIL$CQUAL_FILE_END, CQUAL–5,
CQUAL–17
UTIL$CQUAL_FILE_MATCH, CQUAL–3,
CQUAL–13
UTIL$CQUAL_FILE_PARSE, CQUAL–11
Utility routines
See also SMB routines, SMB–1