HP OpenVMS RTL Library (LIB$) Manual

Order Number: AA–QSBHE–TE

January 2005

This manual documents the library routines contained in the LIB$ and CVT$ facilities of the OpenVMS Run-Time Library.

Revision/Update Information: This manual supersedes HP OpenVMS RTL Library (LIB$) Manual, OpenVMS Alpha Version 7.3.

Software Version: OpenVMS I64 Version 8.2
OpenVMS Alpha Version 8.2

Hewlett-Packard Company
Palo Alto, California
Contents

Preface ............................................................................. xi

Part I LIB$ Overview Section

1 Overview of the LIB$ Facility

1.1 Run-Time Library LIB$ Routines ................................. 1–1
   1.1.1 64-Bit Addressing Support (Alpha and I64 Only) .... 1–2
   1.1.2 The LIB$ Routines ......................................... 1–2

1.2 Translated Version of LIB$ Facility (Alpha and I64 Only) .. 1–9

1.3 Run-Time Library CVT$ Facility ............................... 1–10

Part II LIB$ Reference Section

LIB$ADAWI .................................................. lib–3
LIB$ADDX ............................................... lib–5
LIB$ADD_TIMES ........................................ lib–8
LIB$ANALYZE_SDESC .................................... lib–10
LIB$ANALYZE_SDESC_64 (Alpha and I64 Only) .......... lib–12
LIB$ASN_WTH_MBX ...................................... lib–14
LIB$AST_IN_PROG ....................................... lib–17
LIB$ATTACH ............................................. lib–19
LIB$BBCCI ............................................... lib–21
LIB$BBSSI ................................................ lib–23
LIB$BUILD_NODESPEC .................................. lib–25
LIB$CALLG ................................................ lib–28
LIB$CALLG_64 (Alpha and I64 Only) .......................... lib–29
LIB$CHAR ................................................ lib–30
LIB$COMPARE_NODENAME ................................ lib–32
LIB$COMPRESS_NODENAME ................................ lib–34
LIB$CONVERT_DATE_STRING ................................ lib–37
LIB$CRC ................................................... lib–41
LIB$CRC_TABLE .......................................... lib–43
LIB$CREATE_DIR ......................................... lib–46
LIB$CREATE_USER_VM_ZONE ............................... lib–50
LIB$CREATE_USER_VM_ZONE_64 (Alpha and I64 Only) ...... lib–54
LIB$CREATE_VM_ZONE .................................... lib–57
LIB$CREATE_VM_ZONE_64 (Alpha and I64 Only) ............ lib–63
LIB$CRF_INS_KEY ......................................... lib–69
LIB$CRF_INS_REF ......................................... lib–71
LIB$164_GET_GR (I64 Only) ...................................  lib–308
LIB$164_GET_INVO_CONTEXT (I64 Only) ..........................  lib–310
LIB$164_GET_INVO_HANDLE (I64 Only) ..........................  lib–312
LIB$164_GET_PREV_INVO_CONTEXT (I64 Only) ..................  lib–313
LIB$164_GET_PREV_INVO_HANDLE (I64 Only) ..................  lib–314
LIB$164_GET_UNWIND_HANDLER_FV (I64 Only) ..................  lib–315
LIB$164_GET_UNWIND_LSDA (I64 Only) ..........................  lib–316
LIB$164_GET_UNWIND_OSSD (I64 Only) ..........................  lib–317
LIB$164_INIT_INVO_CONTEXT (I64 Only) .......................  lib–318
LIB$164_IS_AST_DISPATCH_FRAME (I64 Only) ...................  lib–320
LIB$164_IS_EXC_DISPATCH_FRAME (I64 Only) ...................  lib–321
LIB$164_PREV_INVO_END (I64 Only) ...........................  lib–322
LIB$164_PUT_INVO_DISPATCH_FRAME (I64 Only) ...................  lib–323
LIB$164_SET_FR (I64 Only) ...................................  lib–326
LIB$164_SET_GR (I64 Only) ...................................  lib–328
LIB$164_SET_PC (I64 Only) ...................................  lib–330
LIB$ICHAR ................................................................  lib–331
LIB$INDEX ................................................................  lib–333
LIB$INIT_DATE_TIME_CONTEXT ...................................  lib–335
LIB$INIT_TIMER ..................................................  lib–339
LIB$INSERT_TREE ..............................................  lib–341
LIB$INSERT_TREE_64 (Alpha and I64 Only) ....................  lib–350
LIB$INSQHI ................................................................  lib–359
LIB$INSQHIQ (Alpha and I64 Only) ................................  lib–362
LIB$INSQTI ................................................................  lib–365
LIB$INSQTIQ (Alpha and I64 Only) .................................  lib–368
LIB$INSV ..................................................................  lib–371
LIB$INT_OVER ....................................................  lib–373
LIB$LEN ..................................................................  lib–375
LIB$LOCC ..................................................................  lib–376
LIB$LOCK_IMAGE ..................................................  lib–379
LIB$LOOKUP_KEY ..................................................  lib–380
LIB$LOOKUP_TREE ..................................................  lib–384
LIB$LOOKUP_TREE_64 (Alpha and I64 Only) ....................  lib–386
LIB$LP_LINES ......................................................  lib–388
LIB$MATCHC ..........................................................  lib–390
LIB$MATCH_COND ....................................................  lib–392
LIB$MOVC3 .............................................................  lib–395
LIB$MOVC5 .............................................................  lib–397
LIB$MOVTC .............................................................  lib–399
LIB$MOVTUC ............................................................  lib–416
LIB$MULT_DELTA_TIME ............................................  lib–419
LIB$MULTF_DELTA_TIME ..........................................  lib–420
LIB$MULTS_DELTA_TIME (Alpha and I64 Only) ..................  lib–421
LIB$PARSE_ACCESS_CODE ...........................................  lib–422
LIB$PARSE_SOGW_PROT ............................................  lib–425
LIB$PAUSE .............................................................  lib–428
LIB$POLYD ................................................ lib–429
LIB$POLYF ................................................ lib–431
LIB$POLYG ................................................ lib–434
LIB$POLYH ................................................ lib–436
LIB$POLYS (Alpha and I64 Only) ................................ lib–438
LIB$POLYT (Alpha and I64 Only) ................................ lib–440
LIB$PUT_COMMON ........................................ lib–442
LIB$PUT_INVO_REGISTERS (Alpha and I64 Only) ................. lib–444
LIB$PUT_OUTPUT ........................................ lib–446
LIB$RADIX_POINT ........................................ lib–448
LIB$REMQHI ............................................ lib–450
LIB$REMQHIQ (Alpha and I64 Only) ............................. lib–453
LIB$REMQTI ............................................ lib–456
LIB$REMQTIQ (Alpha and I64 Only) ............................. lib–459
LIB$RENAME_FILE ........................................ lib–462
LIB$RESERVE_EF ........................................ lib–471
LIB$RESET_VM_ZONE .................................... lib–473
LIB$RESET_VM_ZONE_64 (Alpha and I64 Only) .................... lib–475
LIB$REVERT ............................................ lib–477
LIB$RUN_PROGRAM ....................................... lib–478
LIB$SCANC ............................................. lib–480
LIB$COPY_DXDX ........................................ lib–482
LIB$COPY_R_DX ........................................ lib–484
LIB$COPY_R_DX_64 (Alpha and I64 Only) ......................... lib–486
LIB$SET_LOGICAL ....................................... lib–488
LIB$SET_SYMBOL ........................................ lib–491
LIB$SFREE1_DD .......................................... lib–494
LIB$SFREEN_DD .......................................... lib–495
LIB$GET1_DD ............................................ lib–497
LIB$GET1_DD_64 (Alpha and I64 Only) .......................... lib–499
LIB$SHOW_TIMER ........................................ lib–501
LIB$SHOW_VM ............................................ lib–505
LIB$SHOW_VM_64 (Alpha and I64 Only) ........................ lib–508
LIB$SHOW_VM_ZONE ..................................... lib–511
LIB$SHOW_VM_ZONE_64 (Alpha and I64 Only) .................... lib–517
LIB$SIGNAL ............................................. lib–523
LIB$SIG_TO_RET ........................................ lib–528
LIB$SIG_TO_STOP ....................................... lib–530
LIB$SIM_TRAP ............................................ lib–532
LIB$SKPC ................................................ lib–534
LIB$SPANC ............................................. lib–536
LIB$SPAWN ............................................. lib–540
LIB$STAT_TIMER ......................................... lib–547
LIB$STAT_VM ............................................. lib–551
LIB$STAT_VM_64 (Alpha and I64 Only) ........................ lib–553
LIB$STOP ................................................ lib–555
LIB$SUBX ............................................. lib–557
Part III  CVT$ Reference Section

CVT$CONVERT_FLOAT ........................................... cvt–3
CVT$FTOF ......................................................... cvt–9

Index

Figures

lib–1  Structure of a Protection Mask .......................................................... lib–47
lib–2  Summary of Symbol Names and Values .................................................. lib–76
lib–3  Summary of Symbol Names, Values, and Names of Referring Modules .......... lib–76
lib–4  Summary Indicating Defining Modules .................................................. lib–77
lib–5  Keyword Table ....................................................................................... lib–381
lib–6  LIB$AB_ASC_EBC ................................................................................. lib–401
lib–7  LIB$AB_ASC_EBC_REV ........................................................................ lib–402
lib–8  LIB$AB_EBC_ASC ................................................................................. lib–403
lib–9  LIB$AB_EBC_ASC_REV ........................................................................ lib–404
lib–10 LIB$AB_CVTPT_O ................................................................................. lib–405
lib–11 LIB$AB_CVTPT_U ................................................................................. lib–406
lib–12 LIB$AB_CVTTP_O ................................................................................. lib–407
lib–13 LIB$AB_CVTTP_U ................................................................................. lib–408
lib–14 LIB$AB_CVT_O_U ................................................................................. lib–409
lib–15 LIB$AB_CVT_U_O ................................................................................. lib–410
lib–16 LIB$AB_CVTPT_Z ................................................................................. lib–411
lib–17 LIB$AB_CVTTP_Z ................................................................................. lib–412
lib–18 LIB$AB_UPCASE ................................................................................. lib–413
lib–19 LIB$AB_LOWERCASE ......................................................................... lib–414
Tables

1–1 LIB$ Routines ........................................ 1–2
1–2 Translated LIB$ Routines (Alpha Only) ................. 1–9
1–3 CVT$ Routines ........................................... 1–10
lib–1 OpenVMS Descriptor Class and Data Type Combinations Accepted by LIB$CVT_DX_DX ........................................... lib–90
lib–2 LIB$CVT_DX_DX Destination NBDS Formats ................ lib–92
lib–3 Symbols for Fields and Values for Operand Access and Data Types Using LIB$DECODE_FAULT ....................... lib–116
lib–4 Formats Used for LIB$GETDVI Strings .................... lib–235
lib–5 Item Code Formats for LIB$GETJPI ........................ lib–239
lib–6 Item Code Formats for LIB$GETQUI ....................... lib–245
lib–7 LIB$SHOW_VM_ZONE Error and Warning Messages ...... lib–513
lib–8 LIB$SHOW_VM_ZONE_64 Error and Warning Messages ... lib–519
lib–9 The Alphabet of LIB$[ABLE]PARSE ........................ lib–576
lib–10 LIB$[ABLE]PARSE Argument Block Fields ................. lib–588
lib–11 Keyword Abbreviation Flags ............................. lib–602
lib–12 Binary Representation of a LIB$[ABLE]PARSE State Transition .................................................... lib–608
lib–13 ASCII Graphics Not Translated to EBCDIC Equivalent by LIB$TRA_ASC_EBC ........................................ lib–637
lib–14 EBCDIC Graphics Not Translated to ASCII Equivalent by LIB$TRA_EBC_ASC ........................................ lib–641
This manual provides users of the HP OpenVMS operating system with detailed usage and reference information on library routines supplied in the LIB$ and CVT$ facilities of the Run-Time Library (RTL).

Intended Audience

This manual is intended for system and application programmers who write programs that call LIB$ and CVT$ Run-Time Library routines.

Document Structure

This manual is organized into three parts as follows:

• The overview chapter provides a brief overview of the LIB$ and CVT$ Run-Time Library facility and lists the LIB$ routines and their functions. It also provides guidelines and information on using the LIB$ facility with VAX and Alpha platforms.
• The LIB$ reference section describes each library routine contained in the LIB$ Run-Time Library facility. This information is presented using the documentation format described in *HP OpenVMS Programming Concepts Manual*. Routine descriptions appear alphabetically by routine name.
• The CVT$ reference section describes the routines contained in the CVT$ Run-Time Library facility. This information is presented using the documentation format described in *HP OpenVMS Programming Concepts Manual*.

Related Documents

The Run-Time Library (RTL) routines are documented in a series of reference manuals.

General descriptions of OpenVMS RTL routines appear in the following manual:

• *HP OpenVMS Programming Concepts Manual*—A description of OpenVMS features and functionality available through calls to the LIB$ Run-Time Library

Specific descriptions of the other RTL facilities and their corresponding routines appear in the following manuals:

• *Compaq Portable Mathematics Library*
• *OpenVMS VAX RTL Mathematics (MTH$) Manual*
• *OpenVMS RTL DECtalk (DTK$) Manual*

1 This manual has been archived but is available on the OpenVMS documentation CD-ROM.
Application programmers using any language can refer to the Guide to Creating OpenVMS Modular Procedures for writing modular and reentrant code.

High-level language programmers will find additional information on calling Run-Time Library routines in their language reference manuals. Additional information may also be found in the language user’s guide provided with your OpenVMS language software.

For a complete list and description of the manuals in the OpenVMS documentation set, see the HP OpenVMS Version 8.2 New Features and Documentation Overview.

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

http://www.hp.com/products/openvms

### Reader’s Comments

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

**Internet**

openvmsdoc@hp.com

**Mail**

Hewlett-Packard Company
OSSG Documentation Group, ZKO3-4/U08
110 Spit Brook Rd.
Nashua, NH 03062-2698

### How To Order Additional Documentation

For information on how to order additional documentation, visit the following World Wide Web address:

http://www.hp.com/go/openvms/doc/order

### Conventions

In this manual, every use of DECwindows and DECwindows Motif refers to HP 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.
In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)

In the HTML version of this document, this convention appears as brackets, rather than a box.

. . . A horizontal ellipsis in examples indicates one of the following possibilities:

- Additional optional arguments in a statement have been omitted.
- The preceding item or items can be repeated one or more times.
- Additional parameters, values, or other information can be entered.

. . . A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.

( ) In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.

[ ] In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.

| In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.

{} In command format descriptions, braces indicate required choices; you must choose at least one of the options listed. Do not type the braces on the command line.

**bold text** This typeface 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*-*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).

**UPPERCASE TEXT** Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
Monospace type 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.
Part I
LIB$ Overview Section

This part contains one chapter that provides a brief overview of the LIB$ and CVT$ Run-Time Library facilities and lists the LIB$ and CVT$ routines and their functions. It also provides guidelines and information on using the LIB$ facility with VAX, Alpha, and HP OpenVMS Industry Standard 64 for Integrity Servers (I64) platforms.
Overview of the LIB$ Facility

This section describes the OpenVMS Run-Time Library (LIB$) facility and lists the function of each routine within the LIB$ facility.

1.1 Run-Time Library LIB$ Routines

This manual discusses the Run-Time Library (RTL) LIB$ routines that perform general purpose (library) functions. One of the functions of the LIB$ facility is to provide a callable interface to components of OpenVMS operating systems that are difficult to use in a high-level language. LIB$ routines allow access to the following:

- System services
- The command language interpreter (CLI)
- Some VAX machine instructions or the equivalent Alpha or I64 instructions

In addition, LIB$ routines allow you to perform the following operations:

- Allocate resources that your process needs, such as virtual memory and event flags
- Convert data types for I/O
- Enable detection of hardware exceptions (VAX only)
- Establish condition handlers (VAX only)
- Generate and display timing statistics while your program is running
- Get and put strings in the process common storage area
- Obtain records from devices
- Obtain the system date and time in various formats
- Process cross-reference data
- Process HP DECnet-Plus for OpenVMS full names
- Search for specified files
- Set up and use binary trees
- Signal exceptions
Overview of the LIB$ Facility
1.1 Run-Time Library LIB$ Routines

### 1.1.1 64-Bit Addressing Support (Alpha and I64 Only)

On Alpha and I64 systems, the Run-Time Library (LIB$) routines provide 64-bit virtual addressing capabilities as follows:

- Most routines now accept 64-bit addresses for arguments passed by reference. Footnotes in the Reference Section of this manual indicate those routines that do not.
- Most routines also accept either 32-bit or 64-bit descriptors for arguments passed by descriptor. Footnotes in the Reference Section of this manual indicate those routines that do not.
- In some cases, a new routine was added to support a 64-bit addressing or data capability. These routines carry the same name as the original routine but with a _64 suffix. In general, both versions of the routine support 64-bit addressing, but the routine with the _64 suffix also supports additional 64-bit capability. The 32-bit capabilities of the original routine are unchanged.
- Specialized routines create and manipulate storage zones in the 64-bit virtual address space. The names of these routines are the same as their 32-bit counterparts but with a _64 suffix. One example is LIB$CREATE_VM_ZONE and LIB$CREATE_VM_ZONE_64. LIB$CREATE_VM_ZONE creates a storage zone in the 32-bit virtual address space, and LIB$CREATE_VM_ZONE_64 creates a storage zone in the 64-bit virtual address space. The function of the original routine is unchanged.

See the *HP OpenVMS Programming Concepts Manual* for more information about 64-bit virtual addressing capabilities.

### 1.1.2 The LIB$ Routines

Table 1–1 lists all of the LIB$ routines and their functions.

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$ADAWI</td>
<td>Add adjacent word with interlock.</td>
</tr>
<tr>
<td>LIB$ADDX</td>
<td>Add two multiple-precision binary numbers.</td>
</tr>
<tr>
<td>LIB$ADD_TIMES</td>
<td>Add two quadwords times.</td>
</tr>
<tr>
<td>LIB$ANALYZE_SDESC</td>
<td>Analyze a string descriptor.</td>
</tr>
<tr>
<td>LIB$ANALYZE_SDESC_64</td>
<td>Analyze a string descriptor.¹</td>
</tr>
<tr>
<td>LIB$ASN_WTH_MBX</td>
<td>Assign a channel to a mailbox.</td>
</tr>
<tr>
<td>LIB$AST_IN_PROG</td>
<td>Check for active AST.</td>
</tr>
<tr>
<td>LIB$ATTACH</td>
<td>Attach a terminal to a process.</td>
</tr>
<tr>
<td>LIB$BBCCI</td>
<td>Test and clear a bit with interlock.</td>
</tr>
<tr>
<td>LIB$BBSSI</td>
<td>Test and set a bit with interlock.</td>
</tr>
<tr>
<td>LIB$BUILD.NODESPEC</td>
<td>Build a node-name specification.</td>
</tr>
<tr>
<td>LIB$CALLG</td>
<td>Call a procedure with a general argument list.</td>
</tr>
<tr>
<td>LIB$CALLG_64</td>
<td>Call a procedure with a general argument list.¹</td>
</tr>
</tbody>
</table>

¹Alpha and I64 specific.
### Table 1–1 (Cont.)  LIB$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$CHAR</td>
<td>Transform a byte to the first character of a string.</td>
</tr>
<tr>
<td>LIB$COMPARE_NODE_NAME</td>
<td>Compare two node names.</td>
</tr>
<tr>
<td>LIB$COMPRESSION_NODE_NAME</td>
<td>Compress a node name to its short form equivalent.</td>
</tr>
<tr>
<td>LIB$CONVERSION_DATE_STRING</td>
<td>Convert a date string to a quadword.</td>
</tr>
<tr>
<td>LIB$CRC</td>
<td>Calculate a cyclic redundancy check (CRC).</td>
</tr>
<tr>
<td>LIB$CRC_TABLE</td>
<td>Construct a cyclic redundancy check (CRC) table.</td>
</tr>
<tr>
<td>LIB$CREATE_DIR</td>
<td>Create a directory.</td>
</tr>
<tr>
<td>LIB$CREATE_USER_VM_ZONE</td>
<td>Create a user-defined storage zone.</td>
</tr>
<tr>
<td>LIB$CREATE_USER_VM_ZONE_64</td>
<td>Create a user-defined storage zone. ¹</td>
</tr>
<tr>
<td>LIB$CREATE_VM_ZONE</td>
<td>Create a new storage zone.</td>
</tr>
<tr>
<td>LIB$CREATE_VM_ZONE_64</td>
<td>Create a new storage zone.</td>
</tr>
<tr>
<td>LIB$CRF_INS_KEY</td>
<td>Insert a key in the cross-reference table.</td>
</tr>
<tr>
<td>LIB$CRF_INS_REF</td>
<td>Insert a reference to a key in the cross-reference table.</td>
</tr>
<tr>
<td>LIB$CRF_OUTPUT</td>
<td>Output some cross-reference table information.</td>
</tr>
<tr>
<td>LIB$CURRENCY</td>
<td>Get the system currency symbol.</td>
</tr>
<tr>
<td>LIB$CVTF_FROM_INTERNAL_TIME</td>
<td>Convert internal time to external time (F-floating value).</td>
</tr>
<tr>
<td>LIB$CVTS_FROM_INTERNAL_TIME</td>
<td>Convert internal time to external time (IEEE S-floating value).</td>
</tr>
<tr>
<td>LIB$CVTF_TO_INTERNAL_TIME</td>
<td>Convert external time to internal time (F-floating value).</td>
</tr>
<tr>
<td>LIB$CVTS_TO_INTERNAL_TIME</td>
<td>Convert external time to internal time (IEEE S-floating value).</td>
</tr>
<tr>
<td>LIB$CVT_DX_DX</td>
<td>Convert the specified data type.</td>
</tr>
<tr>
<td>LIB$CVT_FROM_INTERNAL_TIME</td>
<td>Convert internal time to external time.</td>
</tr>
<tr>
<td>LIB$CVT_TO_INTERNAL_TIME</td>
<td>Convert external time to internal time.</td>
</tr>
<tr>
<td>LIB$CVT_VECTIM</td>
<td>Convert 7-word vector to internal time.</td>
</tr>
<tr>
<td>LIB$CVT_xTB</td>
<td>Convert numeric text to binary.</td>
</tr>
<tr>
<td>LIB$CVT_xTB_64</td>
<td>Convert numeric text to binary. ¹</td>
</tr>
<tr>
<td>LIB$DATE_TIME</td>
<td>Return the date and time as a string.</td>
</tr>
<tr>
<td>LIB$DAY</td>
<td>Return the day number as a longword integer.</td>
</tr>
<tr>
<td>LIB$DAY_OF_WEEK</td>
<td>Return the numeric day of the week.</td>
</tr>
<tr>
<td>LIB$DECLCODE_FAULT</td>
<td>Decode instruction stream during a fault.</td>
</tr>
<tr>
<td>LIB$DEC_OVER</td>
<td>Enable or disable decimal overflow detection.</td>
</tr>
<tr>
<td>LIB$DELETE_FILE</td>
<td>Delete one or more files.</td>
</tr>
<tr>
<td>LIB$DELETE_LOGICAL</td>
<td>Delete a logical name.</td>
</tr>
<tr>
<td>LIB$DELETE_SYMBOL</td>
<td>Delete a CLI symbol.</td>
</tr>
<tr>
<td>LIB$DELETE_VM_ZONE</td>
<td>Delete a virtual memory zone.</td>
</tr>
<tr>
<td>LIB$DELETE_VM_ZONE_64</td>
<td>Delete a virtual memory zone. ¹</td>
</tr>
</tbody>
</table>

¹Alpha and I64 specific.
²Available only on OpenVMS VAX systems and for translated VAX applications running on OpenVMS Alpha or I64 systems.

(continued on next page)
### Overview of the LIB$ Facility

#### 1.1 Run-Time Library LIB$ Routines

**Table 1–1 (Cont.) LIB$ Routines**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$DIGIT_SEP</td>
<td>Get the digit separator symbol.</td>
</tr>
<tr>
<td>LIB$DISABLE_CTRL</td>
<td>Disable CLI interception of control characters.</td>
</tr>
<tr>
<td>LIB$DO_COMMAND</td>
<td>Execute the specified command.</td>
</tr>
<tr>
<td>LIB$EDIV</td>
<td>Perform an extended-precision divide.</td>
</tr>
<tr>
<td>LIB$EMODD</td>
<td>Perform extended multiply and integerize for D-floating values.</td>
</tr>
<tr>
<td>LIB$EMODF</td>
<td>Perform extended multiply and integerize for F-floating values.</td>
</tr>
<tr>
<td>LIB$EMODG</td>
<td>Perform extended multiply and integerize for G-floating values.</td>
</tr>
<tr>
<td>LIB$EMODH</td>
<td>Perform extended multiply and integerize for H-floating values.</td>
</tr>
<tr>
<td>LIB$EMODS</td>
<td>Perform extended multiply and integerize for IEEE S-floating values.</td>
</tr>
<tr>
<td>LIB$EMODT</td>
<td>Perform extended multiply and integerize for IEEE T-floating values.</td>
</tr>
<tr>
<td>LIB$EMUL</td>
<td>Perform an extended-precision multiply.</td>
</tr>
<tr>
<td>LIB$ENABLE_CTRL</td>
<td>Enable CLI interception of control characters.</td>
</tr>
<tr>
<td>LIB$ESTABLISH</td>
<td>Establish a condition handler.</td>
</tr>
<tr>
<td>LIB$EXPAND_NODENAME</td>
<td>Expand a node name to its full name equivalent.</td>
</tr>
<tr>
<td>LIB$EXITV</td>
<td>Extract a field and sign-extend.</td>
</tr>
<tr>
<td>LIB$EXTZV</td>
<td>Extract a zero-extended field.</td>
</tr>
<tr>
<td>LIB$FFx</td>
<td>Find the first clear or set bit.</td>
</tr>
<tr>
<td>LIB$FID_TO_NAME</td>
<td>Convert a device and file ID to a file specification.</td>
</tr>
<tr>
<td>LIB$FILE_SCAN</td>
<td>Perform a file scan.</td>
</tr>
<tr>
<td>LIB$FILE_SCAN_END</td>
<td>End a file scan.</td>
</tr>
<tr>
<td>LIB$FIND_FILE</td>
<td>Find a file.</td>
</tr>
<tr>
<td>LIB$FIND_FILE_END</td>
<td>End of find file.</td>
</tr>
<tr>
<td>LIB$FIND_IMAGE_SYMBOL</td>
<td>Merge activate an image symbol.</td>
</tr>
<tr>
<td>LIB$FIND_VM_ZONE</td>
<td>Find the next valid zone.</td>
</tr>
<tr>
<td>LIB$FIND_VM_ZONE_64</td>
<td>Find the next valid zone.</td>
</tr>
<tr>
<td>LIB$FIT_NODENAME</td>
<td>Fit a node name into an output field.</td>
</tr>
<tr>
<td>LIB$FIXUP_FLT</td>
<td>Fix floating reserved operand.</td>
</tr>
<tr>
<td>LIB$FLT_UNDER</td>
<td>Detect a floating-point underflow.</td>
</tr>
<tr>
<td>LIB$FORMAT_DATE_TIME</td>
<td>Format a date and/or time.</td>
</tr>
<tr>
<td>LIB$FORMAT_SOGW PROT</td>
<td>Format protection mask.</td>
</tr>
<tr>
<td>LIB$FREE_DATE_TIME_CONTEXT</td>
<td>Free the context used to format a date.</td>
</tr>
</tbody>
</table>

1Alpha and I64 specific.
2Available only on OpenVMS VAX systems and for translated VAX applications running on OpenVMS Alpha or I64 systems.
3This routine or an equivalent mechanism is supplied by compilers on OpenVMS Alpha and I64 systems.
4VAX specific.

(continued on next page)
### Table 1–1 (Cont.) LIB$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$FREE_EF</td>
<td>Free an event flag.</td>
</tr>
<tr>
<td>LIB$FREE_LUN</td>
<td>Free a logical unit number.</td>
</tr>
<tr>
<td>LIB$FREE_TIMER</td>
<td>Free timer storage.</td>
</tr>
<tr>
<td>LIB$FREE_VM</td>
<td>Free virtual memory from the program region.</td>
</tr>
<tr>
<td>LIB$FREE_VM_64</td>
<td>Free virtual memory from the program region.</td>
</tr>
<tr>
<td>LIB$FREE_VM_PAGE</td>
<td>Free a virtual memory page.</td>
</tr>
<tr>
<td>LIB$FREE_VM_PAGE_64</td>
<td>Free a virtual memory page.</td>
</tr>
<tr>
<td>LIB$GETDVI</td>
<td>Get device/volume information.</td>
</tr>
<tr>
<td>LIB$GETJPI</td>
<td>Get job/process information.</td>
</tr>
<tr>
<td>LIB$GETQUI</td>
<td>Get queue information.</td>
</tr>
<tr>
<td>LIB$GETSYI</td>
<td>Get systemwide information.</td>
</tr>
<tr>
<td>LIB$GET_ACCNAM</td>
<td>Get access name table for a security object identified by name.</td>
</tr>
<tr>
<td>LIB$GET_ACCNAM_BY_CONTEXT</td>
<td>Get access name table for a security object identified by $GET_SECURITY or $SET_SECURITY context.</td>
</tr>
<tr>
<td>LIB$GET_COMMAND</td>
<td>Get line from SYS$COMMAND.</td>
</tr>
<tr>
<td>LIB$GET_COMMON</td>
<td>Get string from common area.</td>
</tr>
<tr>
<td>LIB$GET_CURR_INVOC_CONTEXT</td>
<td>Get current invocation context.</td>
</tr>
<tr>
<td>LIB$GET_DATE_FORMAT</td>
<td>Return the user’s date input format.</td>
</tr>
<tr>
<td>LIB$GET_EF</td>
<td>Get an event flag.</td>
</tr>
<tr>
<td>LIB$GET_FOREIGN</td>
<td>Get foreign command line.</td>
</tr>
<tr>
<td>LIB$GET_FULLNAME_OFFSET</td>
<td>Get the offset to the starting position of the most significant part of a full name.</td>
</tr>
<tr>
<td>LIB$GET_HOSTNAME</td>
<td>Get host node name.</td>
</tr>
<tr>
<td>LIB$GET_INPUT</td>
<td>Get line from SYS$INPUT.</td>
</tr>
<tr>
<td>LIB$GET_INVOC_CONTEXT</td>
<td>Get invocation context.</td>
</tr>
<tr>
<td>LIB$GET_INVOC_HANDLE</td>
<td>Get invocation handle.</td>
</tr>
<tr>
<td>LIB$GET_LUN</td>
<td>Get logical unit number.</td>
</tr>
<tr>
<td>LIB$GET_MAXIMUM_DATE_LENGTH</td>
<td>Get the maximum possible date/time string length.</td>
</tr>
<tr>
<td>LIB$GET_PREV_INVOC_CONTEXT</td>
<td>Get previous invocation context.</td>
</tr>
<tr>
<td>LIB$GET_PREV_INVOC_HANDLE</td>
<td>Get previous invocation handle.</td>
</tr>
<tr>
<td>LIB$GET_SYMBOL</td>
<td>Get the value of a CLI symbol.</td>
</tr>
<tr>
<td>LIB$GET_USERS_LANGUAGE</td>
<td>Return the user’s language choice.</td>
</tr>
<tr>
<td>LIB$GET_VM</td>
<td>Allocate virtual memory.</td>
</tr>
<tr>
<td>LIB$GET_VM_64</td>
<td>Allocate virtual memory.</td>
</tr>
<tr>
<td>LIB$GET_VM_PAGE</td>
<td>Get a virtual memory page.</td>
</tr>
<tr>
<td>LIB$GET_VM_PAGE_64</td>
<td>Get a virtual memory page.</td>
</tr>
<tr>
<td>LIB$ICHAR</td>
<td>Convert the first character of a string to an integer.</td>
</tr>
</tbody>
</table>

1 Alpha and I64 specific.

4 VAX specific.

(continued on next page)
### Table 1–1 (Cont.) LIB$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$I64_CREATE_INVO_CONTEXT</td>
<td>Allocate and initialize an invocation context block.¹</td>
</tr>
<tr>
<td>LIB$I64_GET_CURR_INVO_CONTEXT</td>
<td>Get current invocation context.</td>
</tr>
<tr>
<td>LIB$I64_FREE_INVO_CONTEXT</td>
<td>Deallocate an invocation context block.</td>
</tr>
<tr>
<td>LIB$I64_GET_CURR_INVO_HANDLE</td>
<td>Get current invocation handle.</td>
</tr>
<tr>
<td>LIB$I64_GET_FR</td>
<td>Get floating-point register value.</td>
</tr>
<tr>
<td>LIB$I64_GET_GR</td>
<td>Get general register value.</td>
</tr>
<tr>
<td>LIB$I64_GET_INVO_HANDLE</td>
<td>Get invocation handle.</td>
</tr>
<tr>
<td>LIB$I64_GET_INVO_CONTEXT</td>
<td>Get invocation context.</td>
</tr>
<tr>
<td>LIB$I64_GET_PREV_INVO_CONTEXT</td>
<td>Get previous invocation context.</td>
</tr>
<tr>
<td>LIB$I64_GET_PREV_INVO_END</td>
<td>Free memory used to process unwind descriptors.</td>
</tr>
<tr>
<td>LIB$I64_GET_UNWIND_CONTEXT</td>
<td>Get previous invocation handle.</td>
</tr>
<tr>
<td>LIB$I64_GET_UNWIND_HANDLER_FV</td>
<td>Given a pc value, find the function value (address of the procedure descriptor) for the condition handler, if present, and write it to handler_fv.</td>
</tr>
<tr>
<td>LIB$I64_GET_UNWIND_LSDA</td>
<td>Find Address of Unwind Information Block Language-Specific Data.</td>
</tr>
<tr>
<td>LIB$I64_GET_UNWIND_OSSD</td>
<td>Find address of the unwind information block operating system-specific data area.</td>
</tr>
<tr>
<td>LIB$I64_INIT_INVO_CONTEXT</td>
<td>Initialize an invocation context block that has already been allocated.</td>
</tr>
<tr>
<td>LIB$I64_IS_AST_DISPATCH_FRAME</td>
<td>Determine whether a given PC value represents an AST dispatch frame.</td>
</tr>
<tr>
<td>LIB$I64_IS_EXC_DISPATCH_FRAME</td>
<td>Determine whether a given PC value represents an exception dispatch frame.</td>
</tr>
<tr>
<td>LIB$I64_PUT_INVO_REGISTERS</td>
<td>Update register contents using a given invocation context.</td>
</tr>
<tr>
<td>LIB$I64_PREV_INVO_END</td>
<td>Free memory used tp process unwind descriptors.</td>
</tr>
<tr>
<td>LIB$I64_SET_FR</td>
<td>Write context of invocation context block.</td>
</tr>
<tr>
<td>LIB$I64_SET_GR</td>
<td>Write invocation block general register value.</td>
</tr>
<tr>
<td>LIB$I64_SET_PC</td>
<td>Write pc_copy value of invocation context block.</td>
</tr>
<tr>
<td>LIB$INDEX</td>
<td>Index to relative position of substring.</td>
</tr>
<tr>
<td>LIB$INIT_DATE_TIME_CONTEXT</td>
<td>Initialize the context used in formatting date/time strings.</td>
</tr>
<tr>
<td>LIB$INIT_TIMER</td>
<td>Initialize times and counts.</td>
</tr>
<tr>
<td>LIB$INSERT_TREE</td>
<td>Insert entry in a balanced binary tree.</td>
</tr>
<tr>
<td>LIB$INSERT_TREE_64</td>
<td>Insert entry in a balanced binary tree.¹</td>
</tr>
<tr>
<td>LIB$INSQHI</td>
<td>Insert entry at the head of a queue.¹</td>
</tr>
<tr>
<td>LIB$INSQHIQ</td>
<td>Insert entry at the head of a queue.</td>
</tr>
<tr>
<td>LIB$INSQTI</td>
<td>Insert entry at the tail of a queue.</td>
</tr>
<tr>
<td>LIB$INSQTIQ</td>
<td>Insert entry at the tail of a queue.¹</td>
</tr>
<tr>
<td>LIB$INSV</td>
<td>Insert a variable bit field.</td>
</tr>
</tbody>
</table>

¹ Alpha and I64 specific.

⁵ I64 specific.
Overview of the LIB$ Facility

1.1 Run-Time Library LIB$ Routines

Table 1–1 (Cont.)  LIB$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$INT_OVER</td>
<td>Detect integer overflow.</td>
</tr>
<tr>
<td>LIB$LEN</td>
<td>Return the length of a string as a longword.</td>
</tr>
<tr>
<td>LIB$LOCC</td>
<td>Locate a character.</td>
</tr>
<tr>
<td>LIB$LOCK</td>
<td>Lock a specified image in the process's working set.</td>
</tr>
<tr>
<td>LIB$LOOKUP_KEY</td>
<td>Look up specified keyword in table.</td>
</tr>
<tr>
<td>LIB$LOOKUP_TREE</td>
<td>Look up an entry in a balanced binary tree.</td>
</tr>
<tr>
<td>LIB$LOOKUP_TREE_64</td>
<td>Look up an entry in a balanced binary tree.</td>
</tr>
<tr>
<td>LIB$LP_LINES</td>
<td>Specify the number of lines on each printer page.</td>
</tr>
<tr>
<td>LIB$MATCHC</td>
<td>Match characters, return relative position.</td>
</tr>
<tr>
<td>LIB$MATCH_COND</td>
<td>Match condition values.</td>
</tr>
<tr>
<td>LIB$MOVC3</td>
<td>Move characters.</td>
</tr>
<tr>
<td>LIB$MOVC5</td>
<td>Move characters with fill.</td>
</tr>
<tr>
<td>LIB$MOVC5</td>
<td>Move translated characters.</td>
</tr>
<tr>
<td>LIB$MOVTUC</td>
<td>Move translated until character.</td>
</tr>
<tr>
<td>LIB$MULTF_DELT_TIME</td>
<td>Multiply delta time by F-floating scalar.</td>
</tr>
<tr>
<td>LIB$MULTS_DELT_TIME</td>
<td>Multiply delta time by IEEE S-floating scalar.</td>
</tr>
<tr>
<td>LIB$MULT_DELT_TIME</td>
<td>Multiply delta time by scalar.</td>
</tr>
<tr>
<td>LIB$PARSE_ACCESS_CODE</td>
<td>Parse access-encoded name string.</td>
</tr>
<tr>
<td>LIB$PARSE_SOGW PROT</td>
<td>Parse protection string.</td>
</tr>
<tr>
<td>LIB$PAUSE</td>
<td>Pause program execution.</td>
</tr>
<tr>
<td>LIB$POLYD</td>
<td>Evaluate polynomials for D-floating values.</td>
</tr>
<tr>
<td>LIB$POLYF</td>
<td>Evaluate polynomials for F-floating values.</td>
</tr>
<tr>
<td>LIB$POLYG</td>
<td>Evaluate polynomials for G-floating values.</td>
</tr>
<tr>
<td>LIB$POLYH</td>
<td>Evaluate polynomials for H-floating values.</td>
</tr>
<tr>
<td>LIB$POLYS</td>
<td>Evaluate polynomials for IEEE S-floating values.</td>
</tr>
<tr>
<td>LIB$POLYT</td>
<td>Evaluate polynomials for IEEE T-floating values.</td>
</tr>
<tr>
<td>LIB$PUT_COMMON</td>
<td>Put string into common area.</td>
</tr>
<tr>
<td>LIB$PUT_INVO_REGISTERS</td>
<td>Put invocation registers.</td>
</tr>
<tr>
<td>LIB$PUT_OUTPUT</td>
<td>Put line to SYS$OUTPUT.</td>
</tr>
<tr>
<td>LIB$RADIX_POINT</td>
<td>Radix point symbol.</td>
</tr>
<tr>
<td>LIB$REMHI</td>
<td>Remove entry from head of queue.</td>
</tr>
<tr>
<td>LIB$REMHIQ</td>
<td>Remove entry from head of queue.</td>
</tr>
<tr>
<td>LIB$REMQT</td>
<td>Remove entry from tail of queue.</td>
</tr>
<tr>
<td>LIB$REMGTI</td>
<td>Remove entry from tail of queue.</td>
</tr>
<tr>
<td>LIB$RENAME_FILE</td>
<td>Rename one or more files.</td>
</tr>
<tr>
<td>LIB$RESERVE_EF</td>
<td>Reserve an event flag.</td>
</tr>
</tbody>
</table>

1 Alpha and I64 specific.
2 Available only on OpenVMS VAX systems and for translated VAX applications running on OpenVMS Alpha or I64 systems.
3 VAX specific.
(continued on next page)
## 1.1 Run-Time Library LIB$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$RESET_VM_ZONE</td>
<td>Reset virtual memory zone.</td>
</tr>
<tr>
<td>LIB$RESET_VM_ZONE_64</td>
<td>Reset virtual memory zone.</td>
</tr>
<tr>
<td>LIB$REVERT</td>
<td>Revert to the handler of the procedure activator.</td>
</tr>
<tr>
<td>LIB$RUN_PROGRAM</td>
<td>Run new program.</td>
</tr>
<tr>
<td>LIB$SCANC</td>
<td>Scan for characters and return relative position.</td>
</tr>
<tr>
<td>LIB$SCOPY_DXDX</td>
<td>Copy source string by descriptor to destination.</td>
</tr>
<tr>
<td>LIB$SCOPY_R_DX</td>
<td>Copy source string by reference to destination.</td>
</tr>
<tr>
<td>LIB$SCOPY_R_DX_64</td>
<td>Copy source string by reference to destination.</td>
</tr>
<tr>
<td>LIB$SET_LOGICAL</td>
<td>Set logical name.</td>
</tr>
<tr>
<td>LIB$SET_SYMBOL</td>
<td>Set the value of a CLI symbol.</td>
</tr>
<tr>
<td>LIB$SFREE1_DD</td>
<td>Free one or more dynamic strings.</td>
</tr>
<tr>
<td>LIB$SFREEN_DD</td>
<td>Free n dynamic strings.</td>
</tr>
<tr>
<td>LIB$SGET1_DD</td>
<td>Get one dynamic string.</td>
</tr>
<tr>
<td>LIB$SGET1_DD_64</td>
<td>Get one dynamic string.</td>
</tr>
<tr>
<td>LIB$SHOW_TIMER</td>
<td>Show accumulated times and counts.</td>
</tr>
<tr>
<td>LIB$SHOW_VM</td>
<td>Show virtual memory statistics.</td>
</tr>
<tr>
<td>LIB$SHOW_VM_64</td>
<td>Show virtual memory statistics.</td>
</tr>
<tr>
<td>LIB$SHOW_VM_ZONE</td>
<td>Display information about a virtual memory zone.</td>
</tr>
<tr>
<td>LIB$SHOW_VM_ZONE_64</td>
<td>Display information about a virtual memory zone.</td>
</tr>
<tr>
<td>LIB$SIGNAL</td>
<td>Signal exception condition.</td>
</tr>
<tr>
<td>LIB$SIG_TO_RET</td>
<td>Convert a signaled message to a return status.</td>
</tr>
<tr>
<td>LIB$SIG_TO_STOP</td>
<td>Convert a signaled condition to a signaled stop.</td>
</tr>
<tr>
<td>LIB$SIM_TRAP</td>
<td>Simulate floating trap.</td>
</tr>
<tr>
<td>LIB$SKPC</td>
<td>Skip equal characters.</td>
</tr>
<tr>
<td>LIB$SPANC</td>
<td>Skip selected characters.</td>
</tr>
<tr>
<td>LIB$SPAWN</td>
<td>Spawn a subprocess.</td>
</tr>
<tr>
<td>LIB$STAT_TIMER</td>
<td>Return accumulated time and count statistics.</td>
</tr>
<tr>
<td>LIB$STAT_VM</td>
<td>Return virtual memory statistics.</td>
</tr>
<tr>
<td>LIB$STAT_VM_64</td>
<td>Return virtual memory statistics.</td>
</tr>
<tr>
<td>LIB$STOP</td>
<td>Stop execution and signal the condition.</td>
</tr>
<tr>
<td>LIB$SUBX</td>
<td>Perform multiple-precision binary subtraction.</td>
</tr>
<tr>
<td>LIB$SUB_TIMES</td>
<td>Subtract two quadword times.</td>
</tr>
<tr>
<td>LIB$SYS_ASCTIM</td>
<td>Invoke $ASCTIM to convert binary time to ASCII.</td>
</tr>
<tr>
<td>LIB$SYS_FAO</td>
<td>Invoke $FAO system service to format output.</td>
</tr>
<tr>
<td>LIB$SYS_FAOL</td>
<td>Invoke $FAOL system service to format output.</td>
</tr>
<tr>
<td>LIB$SYS_FAOL_64</td>
<td>Invoke $FAOL system service to format output.</td>
</tr>
</tbody>
</table>

1 Alpha and I64 specific.
2 Available only on OpenVMS VAX systems and for translated VAX applications running on OpenVMS Alpha or I64 systems.
3 This routine or an equivalent mechanism is supplied by compilers on OpenVMS Alpha and I64 systems.

(continued on next page)
1.1 Run-Time Library LIB$ Routines

Table 1–1 (Cont.) LIB$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$SYS_GETMSG</td>
<td>Invoke $GETMSG system service to get message text.</td>
</tr>
<tr>
<td>LIB$TABLE_PARSE</td>
<td>Implement a table-driven, finite-state parser.</td>
</tr>
<tr>
<td>LIB$TPARSE</td>
<td>Implement a table-driven, finite-state parser.</td>
</tr>
<tr>
<td>LIB$TRAVERSE_TREE</td>
<td>Traverse a balanced binary tree.</td>
</tr>
<tr>
<td>LIB$TRAVERSE_TREE_64</td>
<td>Traverse a balanced binary tree.</td>
</tr>
<tr>
<td>LIB$TRA_ASC_EBC</td>
<td>Translate ASCII to EBCDIC.</td>
</tr>
<tr>
<td>LIB$TRA_EBC_ASC</td>
<td>Translate EBCDIC to ASCII.</td>
</tr>
<tr>
<td>LIB$TRIM_FILESPEC</td>
<td>Fit a long file specification into a fixed field.</td>
</tr>
<tr>
<td>LIB$TRIM_FULLNAME</td>
<td>Trim a full name to fit into a desired output field.</td>
</tr>
<tr>
<td>LIB$UNLOCK</td>
<td>Unlock a specified image in the process's working set.</td>
</tr>
<tr>
<td>LIB$VERIFY_VM_ZONE</td>
<td>Verify a virtual memory zone.</td>
</tr>
<tr>
<td>LIB$VERIFY_VM_ZONE_64</td>
<td>Verify a virtual memory zone.</td>
</tr>
<tr>
<td>LIB$WAIT</td>
<td>Wait a specified period of time.</td>
</tr>
</tbody>
</table>

1 Alpha and I64 specific.
2 Available only on OpenVMS VAX systems and for translated VAX applications running on OpenVMS Alpha or I64 systems.

1.2 Translated Version of LIB$ Facility (Alpha and I64 Only)

The RTL LIB$ facility exists in two forms on OpenVMS Alpha and I64 systems: native and translated. The translated LIB$ library contains routines specific to VAX systems only, and are executed in the Translated Image Environment (TIE). These routines are not available to native OpenVMS Alpha and I64 programs. See Migrating an Application from OpenVMS VAX to OpenVMS Alpha for additional information on using translated images and the TIE.

Table 1–2 lists the translated LIB$ routines.

Table 1–2 Translated LIB$ Routines (Alpha Only)

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$DECODE_FAULT</td>
<td>Decodes VAX instructions.</td>
</tr>
<tr>
<td>LIB$DEC_OVER</td>
<td>Applies to VAX PSL only.</td>
</tr>
<tr>
<td>LIB$ESTABLISH</td>
<td>Supported by compilers on OpenVMS Alpha systems.</td>
</tr>
<tr>
<td>LIB$FIXUP_FLT</td>
<td>Applies to VAX PSL only.</td>
</tr>
<tr>
<td>LIB$FLT_UNDER</td>
<td>Applies to VAX PSL only.</td>
</tr>
<tr>
<td>LIB$INT_OVER</td>
<td>Applies to VAX PSL only.</td>
</tr>
<tr>
<td>LIB$REVERT</td>
<td>Supported by compilers on OpenVMS Alpha systems.</td>
</tr>
<tr>
<td>LIB$SIM_TRAP</td>
<td>Applies to VAX code.</td>
</tr>
</tbody>
</table>

1 This manual has been archived but is available on the OpenVMS Documentation CD-ROM.
Overview of the LIB$ Facility

1.2 Translated Version of LIB$ Facility (Alpha and I64 Only)

Table 1–2 (Cont.) Translated LIB$ Routines (Alpha Only)

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$TPARSE</td>
<td>Requires action routine interface changes. Replaced by LIB$TABLE_PARSE.</td>
</tr>
</tbody>
</table>

LIB$ routines that are called using JSB linkages may function differently on OpenVMS VAX and OpenVMS Alpha systems. See *OpenVMS Programming Interfaces: Calling a System Routine*\(^1\) for more information on using JSB linkages.

1.3 Run-Time Library CVT$ Facility

This manual describes the Run-Time Library CVT$ facility and its routines: CVT$CONVERT_FLOAT and CVT$FTOF. The CVT$ facility lets you convert data stored in one OpenVMS data type into data of another data type. Table 1–3 lists the routines in the CVT$ facility.

Table 1–3 CVT$ Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$CONVERT_FLOAT</td>
<td>Converts data in one of several floating-point data types to another floating-point data type.</td>
</tr>
<tr>
<td>CVT$FTOF</td>
<td>Enhanced version of CVT$CONVERT_FLOAT that provides better performance and more output options than CVT$CONVERT_FLOAT, and also enhances portability between HP-supported platforms.</td>
</tr>
</tbody>
</table>

---

\(^1\) This manual has been archived but is available on the OpenVMS Documentation CD-ROM.
This part contains detailed descriptions of the routines provided by the OpenVMS RTL Library (LIB$) facility.
LIB$ADAWI
Add Aligned Word with Interlock

The Add Aligned Word with Interlock routine allows the user to perform an interlocked add operation using an aligned word.

Format

LIB$ADAWI add ,sum ,sign

Returns

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

Arguments

add
OpenVMS usage: word_signed
type: word (signed)
access: read only
mechanism: by reference

The addend operand to be added to the value of sum. The add argument is the address of a signed word that contains the addend operand.

sum
OpenVMS usage: word_signed
type: word integer (signed)
access: modify
mechanism: by reference

The word to which add is added. The sum argument is the address of a signed word integer containing this value. The add operand is added to the sum operand, and the value of the sum argument is replaced by the result of this addition. The sum argument must be word-aligned; in other words, its address must be a multiple of 2.

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

Sign of the sum argument. The sign argument is the address of a signed word integer that is assigned the value −1, 0, or 1, depending on whether the new value of sum is negative, 0, or positive.
LIB$ Routines
LIB$ADAWI

Description

LIB$ADAWI allows the user to perform an interlocked add operation using an aligned word, and makes the VAX ADAWI\(^1\) instruction available as a callable routine. This routine also enables the user to implement synchronization primitives for multiprocessing.

The add operation is interlocked against similar operations on other processors in a multiprocessor environment. This provides an atomic addition operation. The destination must be aligned on a word boundary; that is, bit 0 of the address of the sum operand must be 0.

If the addend and the sum operand overlap, the result of the addition, the value of the sign argument, and the associated condition codes are unpredictable.

The value of the sign argument is useful when LIB$ADAWI is used to implement locking in a multiprocessing program. For example, a process that is waiting to seize a lock or a resource calls LIB$ADAWI to add 1 to the sum. When the call returns, the waiting process checks the value of sign.

One possible algorithm would interpret the value of sign as follows:

<table>
<thead>
<tr>
<th>Value of sign Argument</th>
<th>Status of Lock or Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Open lock or free resources</td>
</tr>
<tr>
<td>0</td>
<td>Closed lock or no free resources, with no processes waiting</td>
</tr>
<tr>
<td>+1</td>
<td>Closed lock or no free resources, with processes waiting</td>
</tr>
</tbody>
</table>

In this algorithm, if the value of the sign argument is -1, that indicates that the process successfully seized the lock or resource, and other free resources are available. A value of 0 indicates that the process successfully seized the lock or the last available resource. A value of 1 indicates that the process was unable to seize the lock.

It is not sufficient for a waiting process to test the value of sum. The result is unpredictable because other processes can alter the value of sum after the original process executes the ADAWI instruction but before it tests the value of sum. However, a process can safely test the value of sign because its value is determined by the ADAWI instruction and is unaffected by other processes’ activities.

Condition Values Returned

<table>
<thead>
<tr>
<th>LIB$_NORMAL</th>
<th>Routine successfully completed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_INTOVF</td>
<td>Integer overflow error.</td>
</tr>
</tbody>
</table>

\(^1\) On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$ADDX
Add Two Multiple-Precision Binary Numbers

The Add Two Multiple-Precision Binary Numbers routine adds two signed two’s complement integers of arbitrary length.

Format

LIB$ADDX addend-array,augend-array,resultant-array [,array-length]

Returns

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

Arguments

addend-array
OpenVMS usage: vector_longword_signed
type: unspecified
access: read only
mechanism: by reference, array reference

First multiple-precision, signed two’s complement integer that LIB$ADDX adds to the second two’s complement integer. The addend-array argument is the address of the array containing the two’s complement number to be added.

augend-array
OpenVMS usage: vector_longword_signed
type: unspecified
access: read only
mechanism: by reference, array reference

Second multiple-precision, signed two’s complement integer that LIB$ADDX adds to the first two’s complement integer. The augend-array argument is the address of the array containing the two’s complement number.

resultant-array
OpenVMS usage: vector_longword_signed
type: unspecified
access: write only
mechanism: by reference, array reference

Multiple-precision, signed two’s complement integer result of the addition. The resultant-array argument is the address of the array into which LIB$ADDX writes the result of the addition.

array-length
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference
LIB$ Routines
LIB$ADDX

Length in longwords of the arrays to be operated on; each array is of length array-length. The array-length argument is the address of a signed longword integer containing the length. The array-length argument must not be negative. This is an optional argument. If omitted, the default is 2.

Description
LIB$ADDX adds two signed two's complement integers of arbitrary length. The integers are located in arrays of longwords. The higher addresses of these longwords contain the higher precision parts of the values. The highest-addressed longword contains the sign and 31 bits of precision. The remaining longwords contain 32 bits of precision in each. The number of longwords in each array is specified in the optional argument array-length. The default array length is 2, which corresponds to the OpenVMS quadword data type.

Any two or all three of the first three arguments can be the same.

Condition Values Returned
SS$_NORMAL Routine successfully completed.
SS$_INTOVF Integer overflow. The result is correct, except that the sign bit is lost.

Example
C+ C This Fortran example program shows the use C of LIB$ADDX.
C- INTEGER A(2),B(2),C(2),RETURN
DATA A/'00000001'x,'7FFF407F'x/
DATA B/'FFFFFFFF'x,'8000BF80'x/
C+ C The highest addressed longword of "A" is A(2).
C So, "A" represents the integer value ('7FFF407F'x) * 16**7 + 1.
C That is, A(2) is 576447592255193089.
C "B" is the two complement representation of "-A".
C-
RETURN = LIB$ADDX(A,B,C)
TYPE *, 'Let A = 576447592255193089.'
TYPE *, 'Then A + B is 0.'
TYPE 1,C(2),C(1)
1 FORMAT(' "A" - "A" is ',1H',I1,I1,3H'x.)
TYPE *, 'Note that C is C(2) concatenated with C(1).'
C+ C Let "A" have the value 72057594037927937 = '100000000000001'x.
C Let "B" have the value 4294967295 = '00000000FFFFFF'x.
C-
A(1) = '00000000'x
A(2) = '10000000'x
B(1) = 'FFFFFF'x
B(2) = '00000000'x
C+ C Then "A" + "B" is 72057598332895232.
C-
RETURN = LIB$ADDX(A,B,C)
TYPE *,', '
TYPE *,'LET A = 72057594037927937 and B = 4294967295'
TYPE *,'Then A + B is ',C
TYPE 2,C(2),C(1)
2 FORMAT(' 72057598332895232 is represented as ',1H',Z8.8,Z8.8,3H'x.)
TYPE *,'Recall that 72057598332895232 is C(2) concatenated with C(1).'
END

This Fortran example demonstrates how to call LIB$ADDX. The output generated by this program is as follows:

Let A = 576447592255193089. 
Then A + B is 0.
"A" - "A" is '00'x.
Note that C is C(2) concatenated with C(1).
LET A = 72057594037927937 and B = 4294967295 
Then A + B is 0 268435457
72057598332895232 is represented as '10000001 0'x.
Recall that 72057598332895232 is C(2) concatenated with C(1).
LIB$ Routines  
LIB$ADD_TIMES

LIB$ADD_TIMES
Add Two Quadword Times

The Add Two Quadword Times routine adds two internal format times.

Format

LIB$ADD_TIMES time1 ,time2 ,resultant-time

Returns

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

Arguments

**time1**
OpenVMS usage: date_time  
type: quadword (unsigned)  
access: read only  
mechanism: by reference

First time that LIB$ADD_TIMES adds to the second time. The time1 argument is the address of an unsigned quadword containing the first time to be added. The time1 argument may be either a delta time or an absolute time; however, at least one of the arguments, time1 or time2, must be a delta time.

**time2**
OpenVMS usage: date_time  
type: quadword (unsigned)  
access: read only  
mechanism: by reference

Second time that LIB$ADD_TIMES adds to the first time. The time2 argument is the address of an unsigned quadword containing the second time to be added. The time2 argument may be either a delta time or an absolute time; however, at least one of the arguments, time1 or time2, must be a delta time.

**resultant-time**
OpenVMS usage: date_time  
type: quadword (unsigned)  
access: write only  
mechanism: by reference

The result of adding time1 and time2. The resultant-time argument is the address of an unsigned quadword containing the result. If both time1 and time2 are delta times, then resultant-time is a delta time. Otherwise, resultant-time is an absolute time.
Description

LIB$ADD_TIMES adds two OpenVMS internal times. It can add two delta times or a delta time and an absolute time. LIB$ADD_TIMES cannot add two absolute times.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$._NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$._IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$._ONEDELTIM</td>
<td>At least one delta time is required.</td>
</tr>
<tr>
<td>LIB$._WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
LIB$ ANALYZE_SDESC
Analyze String Descriptor

The Analyze String Descriptors routine extracts the length and the address at which the data starts for a variety of 32-bit string descriptor classes.

Format

LIB$ANALYZE_SDESC input-descriptor, data-length, data-address

Corresponding JSB Entry Point

LIB$ANALYZE_SDESC_R2

Returns

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

Arguments

input-descriptor
OpenVMS usage: descriptor
type: quadword (unsigned)
access: read only
mechanism: by reference

Input descriptor from which LIB$ANALYZE_SDESC extracts the length of the data and the address at which the data starts. The input-descriptor argument is the address of a descriptor pointing to the input data.

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

Length of the data; LIB$ANALYZE_SDESC extracts this length value from the input descriptor. The data-length argument is the address of an unsigned word integer into which LIB$ANALYZE_SDESC writes the length.

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

Starting address of the data; LIB$ANALYZE_SDESC extracts this address from the input descriptor. The data-address argument is the address of an unsigned longword into which LIB$ANALYZE_SDESC writes the starting address of the data.
Description

LIB$ANALYZE_SDESC extracts the length and the address at which the data starts for a variety of 32-bit string descriptor classes. Following is a description of the classes of string descriptors.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Restrictions/Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Array</td>
<td>DSC$L_ARSIZE must be less than 65,536 bytes.</td>
</tr>
<tr>
<td>D</td>
<td>Decimal string</td>
<td>Treated as class S.</td>
</tr>
<tr>
<td>NCA</td>
<td>Noncontiguous array</td>
<td>Same as class A.</td>
</tr>
<tr>
<td>S</td>
<td>Scalar, string</td>
<td>None.</td>
</tr>
<tr>
<td>SD</td>
<td>Decimal scalar</td>
<td>Treated as class S.</td>
</tr>
<tr>
<td>VS</td>
<td>Varying string</td>
<td>Length returned is CURLEN.</td>
</tr>
<tr>
<td>Z</td>
<td>Unspecified</td>
<td>Treated as class S.</td>
</tr>
</tbody>
</table>

See STR$ANALYZE_SDESC for a similar routine that signals an error rather than returning a status.

Condition Values Returned

- **SS$_NORMAL**  Routine successfully completed.
- **LIB$_INVSTRDES**  Invalid string descriptor. An array descriptor has an ARSIZE greater than 65,535 bytes, or the class is unsupported.
LIB$ Routines
LIB$ANALYZE_SDESC_64 (Alpha and I64 Only)

LIB$ANALYZE_SDESC_64 (Alpha and I64 Only)
Analyze String Descriptor

The Analyze String Descriptor routine extracts the length and the address at which the data starts for a variety of 32-bit and 64-bit string descriptor classes.

Format

LIB$ANALYZE_SDESC_64 input-descriptor ,data-length ,data-address [,descriptor-type]

Corresponding JSB Entry Point

LIB$ANALYZE_SDESC_R2 Refer to the LIB$ANALYZE_SDESC routine for information about the JSB entry point, LIB$ANALYZE_SDESC_R2. This JSB entry point returns 64-bit results on Alpha and I64 systems.

Returns

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

Arguments

input-descriptor
OpenVMS usage: descriptor
type: longword (unsigned) or quadword (unsigned)
access: read only
mechanism: by reference
Input descriptor from which LIB$ANALYZE_SDESC_64 extracts the length of the data and the address at which the data starts. The input-descriptor argument is the address of a descriptor pointing to the input data. The input descriptor can be a longword (unsigned) or a quadword (unsigned).

data-length
OpenVMS usage: quadword_unsigned
type: quadword (unsigned)
access: write only
mechanism: by reference
Length of the data; LIB$ANALYZE_SDESC_64 extracts this length value from the input descriptor. The data-length argument is the address of an unsigned quadword integer into which LIB$ANALYZE_SDESC_64 writes the length.

data-address
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference
Starting address of the data; LIB$ANALYZE_SDESC_64 extracts this address from the input descriptor. The data-address argument is the address of an unsigned quadword into which LIB$ANALYZE_SDESC_64 writes the starting address of the data.
LIB$ Routines
LIB$ANALYZE_SDESC_64 (Alpha and I64 Only)

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

Flag value indicating the type of input descriptor. The descriptor-type argument contains the address of an unsigned longword integer to which LIB$ANALYZE_SDESC_64 writes a 0 for a 32-bit input descriptor or a 1 for a 64-bit descriptor.

This argument is optional.

Description
LIB$ANALYZE_SDESC_64 extracts the length and the address at which the data starts for a variety of 32-bit and 64-bit string descriptor classes. Following is a description of the classes of string descriptors:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Restrictions/Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Array</td>
<td>For 32-bit descriptors, DSC$L_ARSIZE must be less than $2^{16}$, or 65,536, bytes. For 64-bit descriptors, DSC64$Q_ARSIZE must be less than $2^{64}$ bytes.</td>
</tr>
<tr>
<td>D</td>
<td>Decimal string</td>
<td>Treated as class S.</td>
</tr>
<tr>
<td>NCA</td>
<td>Noncontiguous array</td>
<td>Same as class A.</td>
</tr>
<tr>
<td>S</td>
<td>Scalar, string</td>
<td>None.</td>
</tr>
<tr>
<td>SD</td>
<td>Decimal scalar</td>
<td>Treated as class S.</td>
</tr>
<tr>
<td>VS</td>
<td>Varying string</td>
<td>Length returned is CURLEN.</td>
</tr>
<tr>
<td>Z</td>
<td>Unspecified</td>
<td>Treated as class S.</td>
</tr>
</tbody>
</table>

See STR$ANALYZE_SDESC_64 for a similar routine that signals an error rather than returning a status.

Condition Values Returned

| SS$_NORMAL | Routine successfully completed. |
| LIB$_INVSTRDES | Invalid string descriptor. An array descriptor has an ARSIZE greater than 65,535 bytes, or the class is unsupported. |
LIB$ Routines
LIB$ASN_WTH_MBX

LIB$ASN_WTH_MBX
Assign Channel with Mailbox

The Assign Channel with Mailbox routine assigns a channel to a specified device and associates a mailbox with the device. It returns both the device channel and the mailbox channel.

Format

LIB$ASN_WTH_MBX device-name [,maximum-message-size] [,buffer-quota] ,device-channel ,mailbox-channel

Returns

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

Arguments

device-name
OpenVMS usage: device_name
type: character string
access: read only
mechanism: by descriptor

Device name that LIB$ASN_WTH_MBX passes to the $ASSIGN service. The device-name argument is the address of a descriptor pointing to the device name.

maximum-message-size
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Maximum message size that can be sent to the mailbox; LIB$ASN_WTH_MBX passes this argument to the $CREMBX service. The maximum-message-size argument is the address of a signed longword integer containing this maximum message size.

buffer-quota
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Number of system dynamic memory bytes that can be used to buffer messages sent to the mailbox; LIB$ASN_WTH_MBX passes this argument to the $CREMBX service. The buffer-quota argument is the address of a signed longword integer containing this buffer quota.
device-channel
OpenVMS usage: word_unsigned
| type:     | word integer (unsigned) |
| access:   | write only              |
| mechanism:| by reference            |

Device channel that LIB$ASN_WTH_MBX receives from the $ASSIGN service. The device-channel argument is the address of an unsigned word integer into which $ASSIGN writes the device channel.

mailbox-channel
OpenVMS usage: channel
| type:     | word integer (unsigned) |
| access:   | write only              |
| mechanism:| by reference            |

Mailbox channel that LIB$ASN_WTH_MBX receives from the $CREMBX service. The mailbox-channel argument is the address of an unsigned word integer into which $CREMBX writes the mailbox channel.

Description

A mailbox is a virtual device used for communication between processes. A channel is the communication path that a process uses to perform I/O operations to a particular device. LIB$ASN_WTH_MBX assigns a channel to a device and associates a mailbox with the device. It returns both the device channel and the mailbox channel to the mailbox.

Normally, a process calls the $CREMBX system service to create a mailbox and assign a channel and logical name to it. Any process running in the same job and using the same logical name uses the same mailbox.

LIB$ASN_WTH_MBX associates the physical mailbox name with the channel assigned to the device. To create a temporary mailbox for itself and other processes cooperating with it, your program calls LIB$ASN_WTH_MBX. The Run-Time Library routine assigns the channel and creates the temporary mailbox by using the system services $GETDVIW, $ASSIGN, and $CREMBX. Instead of a logical name, the mailbox is identified by a physical device name of the form MBcu. The physical device name MBcu is made up of the following elements:

MB  Indicates that the device is a mailbox
    c  Is the controller
    u  Is the unit number

The routine returns the channel for this device name to the calling program, which then must pass the mailbox channel to the other programs with which it cooperates. In this way, the cooperating processes access the mailbox by its physical name, instead of by a logical name.

The calling program passes the routine a device name, which specifies the device to which the channel is to be assigned. For this argument (called device-name), you may use a logical name. If you do so, the routine attempts one level of logical name translation.
LIB$ Routines
LIB$ASN_WTH_MBX

The privilege restrictions and process quotas required for using this routine are those required by the $GETDVIW, $CREMBX, and $ASSIGN system services.

Note

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.

Condition Values Returned

SS$_NORMAL Routine successfully completed.

Any condition value returned by the called system services $ASSIGN, $CREMBX, $GETDVI, or the RTL routines LIB$GET_EF and LIB$FREE_EF.
LIB$AST_IN_PROG
AST in Progress

The AST in Progress routine indicates whether an AST is currently in progress.

Format
LIB$AST_IN_PROG

Returns
OpenVMS usage: boolean
type: boolean
access: write only
mechanism: by value

Truth value that indicates whether an AST is currently in progress (value = 1) or not (value = 0).

Arguments
None.

Description
An asynchronous system trap (AST) is an OpenVMS mechanism for providing a software interrupt when an external event occurs, such as the user pressing Ctrl/C. When an external event occurs, the OpenVMS operating system interrupts the execution of the current process and calls a routine that you supply. While that routine is active, the AST is said to be in progress, and the process is said to be executing at AST level. When your AST routine returns control to the original process, the AST is no longer active, and execution continues where it left off.

LIB$AST_IN_PROG indicates to the calling program whether an AST is currently in progress. Your program can call LIB$AST_IN_PROG to determine whether it is executing at AST level and then take appropriate action. This routine is useful if you are writing AST-reentrant code, which takes different actions depending on whether an AST is in progress. For example, the routine might have two separate statically allocated storage areas, one for AST level and one for non-AST level.

LIB$AST_IN_PROG calls the RTL routines LIB$FREE_EF and LIB$GET_EF, and the $GETJPI system service. If LIB$AST_IN_PROG or any of these routines encounters an error, LIB$AST_IN_PROG calls LIB$STOP.

Condition Values Returned
None.
Example

```pascal
PROGRAM AST_IN_PROGRESS(INPUT, OUTPUT);
FUNCTION LIB$AST_IN_PROG : INTEGER; EXTERN;
VAR
  ASTVALUE : INTEGER;
BEGIN
  ASTVALUE := LIB$AST_IN_PROG;
  CASE ASTVALUE OF
    0 : WRITELN('AN AST IS NOT IN PROGRESS');
    1 : WRITELN('AN AST IS IN PROGRESS');
  END { of the case statement }
END.
```

This Pascal program determines whether or not an AST is in progress.
LIB$ATTACH

Attach Terminal to Process

The Attach Terminal to Process routine requests the calling process’s command language interpreter (CLI) to detach the terminal of the calling process and to reattach it to a different process.

Format

LIB$ATTACH process-id

Returns

OpenVMS usage: cond_value

type: longword (unsigned)

access: write only

mechanism: by value

Argument

process-id

OpenVMS usage: process_id

type: longword integer (unsigned)

access: read only

mechanism: by reference

Identification of the process to which LIB$ATTACH requests the calling process to attach its terminal. The process-id argument is the address of an unsigned longword integer containing the process identification. The specified process must be currently detached (by means of a SPAWN or ATTACH command or by a call to LIB$SPAWN or LIB$ATTACH) and must be part of the caller’s job.

Description

LIB$ATTACH requests the calling process’s command language interpreter (CLI) to detach the terminal of the calling process and reattach it to a different process. The calling process then hibernates. LIB$ATTACH provides the same function as the DCL command ATTACH. For more information on ATTACH, see the HP OpenVMS DCL Dictionary.

LIB$ATTACH is supported for use with the DCL CLI. If used with the Monitor Control Routine (MCR) CLI, the error status LIB$_NOCLI is returned. If an image is run directly as a subprocess or detached process, no CLI is present to perform this function. In such cases, the error status LIB$_NOCLI is returned.

Condition Values Returned

SS$NORMAL Routine successfully completed.

SS$NONEXPR Nonexistent process. The process specified by process-id does not exist.

LIB$ATTREQREF Attach request refused. The specified process could not be attached to. Either it was not detached or it did not belong to the caller’s job.
<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NOCLI</td>
<td>No CLI present to perform function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.</td>
</tr>
<tr>
<td>LIB$_UNECLIERR</td>
<td>Unexpected CLI error. The CLI returned an error status, which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL CLI, please report the problem to your HP support representative.</td>
</tr>
</tbody>
</table>
LIB$BBCCI
Test and Clear Bit with Interlock

The Test and Clear Bit with Interlock routine tests and clears a selected bit under memory interlock. LIB$BBCCI makes the VAX BBCCI instruction available as a callable routine.1

Format

LIB$BBCCI  position , bit-zero-address

Returns

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

State of the bit before it was cleared by LIB$BBCCI: 1 if the bit was previously set, and 0 if the bit was previously clear.

Arguments

position
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Bit position, relative to bit-zero-address, of the bit that LIB$BBCCI tests and clears. The position argument is the address of a signed longword integer containing the bit position. A position of zero denotes the low-order bit of the byte base. The bit position is equal to the offset of the bit chosen from the base position. This offset may span the entire range of a signed longword integer; negative offsets access bits in lower addressed bytes.

bit-zero-address
OpenVMS usage: unspecified
type: address
access: read only
mechanism: by value

Address of the byte containing bit 0 of the field that LIB$BBCCI references. The bit-zero-address argument is the location of the base position. The bit that LIB$BBCCI tests and clears is position bits offset from the low bit of bit-zero-address.

Description

The single bit specified by position and bit-zero-address is tested, the previous state of the bit remembered, and the bit cleared. The reading of the state of the bit and its clearing are interlocked against similar operations by other processors or devices in the system. The remembered previous state of the bit is then returned as the function value of LIB$BBCCI.

On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$ Routines
LIB$BBCCI

Condition Values Returned

None.

Example

This Fortran program demonstrates the use of LIB$BBCCI.

```
INTEGER*4 STATES(4) ! 128 shared state bits
COMMON /STATES/ STATES ! Could be shared memory
LOGICAL*4 LIB$BBCCI

IF (LIB$BBCCI (42, STATES)) THEN
  TYPE *,'State bit 42 was set'
ELSE
  TYPE *,'State bit 42 was clear'
END IF
```

This Fortran example tests and clears bit 42 of array STATES, which is in a COMMON area (possibly shared between two processors).

The output generated by this program is as follows:

```
$ RUN STATE
State bit 42 was clear.
```
**LIB$BBSSI**

**Test and Set Bit with Interlock**

The Test and Set Bit with Interlock routine tests and sets a selected bit under memory interlock. LIB$BBSSI makes the VAX BBSSI instruction available as a callable routine. ¹

**Format**

LIB$BBSSI position ,bit-zero-address

**Returns**

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

The state of the bit before it was set by LIB$BBSSI: 1 if it was previously set, and 0 if it was previously clear.

**Arguments**

**position**

OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Bit position, relative to **bit-zero-address**, of the bit that LIB$BBSSI tests and sets. The **position** argument is the address of a signed longword integer containing the bit position. A position of zero denotes the low-order bit of the byte base. The bit position is equal to the offset of the bit chosen from the base position. This offset may span the entire range of a signed longword integer; negative offsets access bits in lower addressed bytes.

**bit-zero-address**

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

Address of the byte containing bit 0 of the field that LIB$BBSSI references. The **bit-zero-address** argument is the location of the base position. The bit that LIB$BBSSI tests and sets is **position** bits offset from the low bit of **bit-zero-address**.

**Description**

The single bit specified by **position** and **bit-zero-address** arguments is tested, the previous state of the bit remembered, and the bit set. The reading of the state of the bit and its setting are interlocked against similar operations by other processors or devices in the system. The remembered previous state of the bit is then returned as the function value of LIB$BBSSI.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$ Routines
LIB$BBSSI

Condition Values Returned
None.

Example

C+
C This Fortran example program demonstrates
C the use of LIB$BBSSI.
C-

 INTEGER*4 STATES(4)   ! 128 shared state bits
 COMMON /STATES/ STATES ! Could be shared memory
 LOGICAL*4 LIB$BBSSI

 IF (LIB$BBSSI (104, STATES)) THEN
   TYPE *, 'State bit 104 was set'
 ELSE
   TYPE *, 'State bit 104 was clear'
 END IF
END

This Fortran example tests and sets bit 104 of array STATES, which is in a
COMMON storage area (possibly shared between two processors).

The output generated by this program is as follows:

 $ RUN STATEB
 State bit 104 was clear.
LIB$BUILD_NODESPEC
Build a Node-Name Specification

The Build a Node-Name Specification routine builds a node-name specification from the primary node name. The output node-name specification can be used for other node-name parsing operations. †

Format
LIB$BUILD_NODESPEC primary-nodename, nodespec [,acs] [,secondary-nodename] [,nodespec-length]

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

Arguments
primary-nodename
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Primary node name. The primary-nodename argument contains the address of a descriptor pointing to this node-name string. The primary node name should not contain unnecessary quotation marks (that is, quotation marks (" ") that are not part of a simple name within the node name).

The error LIB$_INVARG is returned if primary-nodename points to a null string. The error LIB$_INVSTRDES is returned if primary-nodename is an invalid descriptor.

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

Node-name specification. The nodespec argument contains the address of a descriptor pointing to this output node-name specification string. LIB$BUILD_NODESPEC writes the output node-name specification into the buffer pointed to by the nodespec descriptor.

The error LIB$_INVSTRDES is returned if nodespec is an invalid descriptor.

The length field of the nodespec descriptor is not updated unless nodespec is a dynamic descriptor with a length less than the resultant node-name specification. Refer to the OpenVMS RTL String Manipulation (STR$) Manual for dynamic string descriptor usage.

The nodespec argument contains an unusable result when LIB$BUILD_NODESPEC returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
**LIB$ Routines**

**LIB$BUILD_NODESPEC**

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

Access control string. The `acs` argument contains the address of a descriptor pointing to this access control string. The access control string must be a quoted string.

The error LIB$_INVSTRDES is returned if `acs` is an invalid descriptor.

**secondary-nodename**
- OpenVMS usage: char_string
- type: character string
- access: read only
- mechanism: by descriptor

Secondary node name. The `secondary-nodename` argument contains the address of a descriptor pointing to this secondary node-name string.

The error LIB$_INVSTRDES is returned if `secondary-nodename` is an invalid descriptor.

**nodespec-length**
- OpenVMS usage: unsigned_word
- type: word (unsigned)
- access: write only
- mechanism: by reference

Length of the output node-name specification. The `nodespec-length` argument is the address of an unsigned word that contains this length in bytes.

The `nodespec-length` argument contains an unusable result when LIB$BUILD_NODESPEC returns in error.

**Description**

This routine builds the parsable form of a node name as the output node-name specification from the network usable form. Refer to LIB$GET_HOSTNAME for the definitions of both the parsable form and the network usable form.

The network usable form is specified by the argument `primary-nodename`. If `primary-nodename` contains special characters, it is enclosed in quotation marks (" " ) to build the node-name specification. The quotation marks prevent the special characters from being recognized as terminator characters and enables correct parsing of the node-name syntax.

If you enclose `primary-nodename` in quotation marks, any quotation marks that are part of any simple names within `primary-nodename` are doubled (that is, each quotation mark ("" ) is turned into two quotation marks ("""" )). LIB$BUILD_NODESPEC checks if the fully quoted primary node name exceeds 1024 characters. The error condition LIB$_NODTOOLNG is returned if this is the case.

To form the output node-name specification, the fully quoted primary node name is concatenated with the access control string (if supplied) and the double colons and is followed by the secondary node name (if supplied).
This routine does not validate any of the input arguments to ensure they can form a syntactically valid node name when they are concatenated.

If the routine overflows the output buffer pointed to by `nodespec`, the output node-name specification is truncated, and the alternate successful status `LIB$_STRTRU` is returned.

The `nodespec-length` argument, if supplied, is always set to the length of the node-name specification that is written into the output buffer pointed to by `nodespec`.

**Condition Values Returned**

- **SS$_NORMAL**  
  Routine successfully completed.

- **LIB$_INVARG**  
  Invalid argument. The `primary-nodename` argument points to a null string.

- **LIB$_INVSTRDES**  
  Invalid string descriptor.

- **LIB$_NODTOOLNG**  
  The primary node name after quoting exceeds 1024 characters.

- **LIB$_STRTRU**  
  Routine successfully completed. Characters are truncated in the output buffer pointed to by the `nodespec` argument.

- **LIB$_WRONUMARG**  
  Wrong number of arguments.

Any condition value returned by `LIB$SCOPY_DXDX`.
LIB$CALLG
Call Routine with General Argument List

The Call Routine with General Argument List routine calls a routine with an argument list specified as an array of longwords, the first of which is a count of the remaining longwords. LIB$CALLG is a callable version of the VAX CALLG instruction.1

Format

LIB$CALLG argument-list ,user-procedure

Returns

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

Return value, if any, of the called routine, unchanged by LIB$CALLG.

Arguments

argument-list
OpenVMS usage: arg_list
type: unspecified
access: read only
mechanism: by reference, array reference

Argument list to be passed to user-procedure. The argument-list argument is the address of an array of longwords that is the argument list. The first longword contains the count of the remaining longwords, to a maximum of 255.

user-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

Routine that LIB$CALLG calls with the specified argument list.

Description

LIB$CALLG is used to call routines that accept variable-length argument lists when the number of arguments to be passed is not known until execution time. LIB$CALLG is also used to call such routines from strongly typed languages, which require routines to be declared as having a fixed number of arguments.

Condition Values Returned

None.

1 On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$CALLG_64 (Alpha and I64 Only)
Call Routine with General Argument List

The Call Routine with General Argument List routine calls a routine with an argument list specified as an array of quadwords, the first of which is a count of the remaining quadwords.

Format

LIB$CALLG_64 argument-list ,user-procedure

Returns

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

Return value, if any, of the called routine, unchanged by LIB$CALLG_64.

Arguments

argument-list

OpenVMS usage: arg_list
type: unspecified
access: read only
mechanism: by reference, array reference

Argument list to be passed to user-procedure. The argument-list argument is the address of an array of quadwords that is the argument list. The first quadword contains the count of the remaining quadwords, to a maximum of 255.

user-procedure

OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

Routine that LIB$CALLG_64 calls with the specified argument list.

Description

LIB$CALLG_64 is useful for calling routines that accept variable-length argument lists when the number of arguments to be passed is not known until execution time. LIB$CALLG_64 can also be used to call such routines from strongly typed languages, which require routines to be declared as having a fixed number of arguments.

Condition Values Returned

None.
LIB$CHAR
Transform Byte to First Character of String

The Transform Byte to First Character of String routine transforms a single 8-bit ASCII character to an ASCII string consisting of a single character followed by trailing spaces, if needed, to fill out the string. The range of the input byte is 0 through 255.

Format

```
LIB$CHAR one-character-string ,ascii-code
```

Returns

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

Arguments

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

ASCII character string consisting of a single character followed by trailing spaces, if needed, that LIB$CHAR creates when it transforms the ASCII character code. The **one-character-string** argument is the address of a descriptor pointing to the character string that LIB$CHAR writes.

**ascii-code**
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

Single 8-bit ASCII character code that LIB$CHAR transforms to an ASCII string. The **ascii-code** argument is the address of an unsigned byte containing the ASCII character code.

Description

LIB$CHAR is the inverse of LIB$ICHAR. (See the description of LIB$ICHAR.) LIB$CHAR is not a binary-to-ASCII conversion routine. LIB$CHAR merely interprets **ascii-code** as an ASCII character code and converts it to a string.
### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed, but the string was truncated. The destination string could not contain all of the characters.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$COMPARE_NODENAME

LIB$COMPARE_NODENAME
Compare Two Node Names

The Compare Two Node Names routine compares two node names to see if they resolve to the same full name. †

Format
LIB$COMPARE_NODENAME node name1 ,nodename2 ,comparison-result

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

Arguments

nodename1
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

First node name to be compared. The nodename1 argument contains the address of a descriptor pointing to this node-name string.

The error LIB$_INVARG is returned if nodename1 contains an invalid node name, points to a null string, or contains more than 1024 characters. The error LIB$_INVSTRDES is returned if nodename1 is an invalid descriptor.

nodename2
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Second node name to be compared. The nodename2 argument contains the address of a descriptor pointing to this node-name string.

The error LIB$_INVARG is returned if nodename2 contains an invalid node name, points to a null string, or contains more than 1024 characters. The error LIB$_INVSTRDES is returned if nodename2 is an invalid descriptor.

comparison-result
OpenVMS usage: longword_unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

Result of the comparison. The comparison-result argument is the address of an unsigned longword that contains the comparison result. If the two node names are equal, 0 is returned. If they are not equal, 1 is returned.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
**Comparison-result** contains an unusable result when LIB$COMPARE_NODENAME returns in error.

**Description**

This routine compares two node names and checks to see if they resolve to the same full name. The two node names are first expanded using LIB$EXPAND_NODENAME. Any errors that result from expanding the input node names are propagated and returned as condition values. A string comparison is performed on the expanded node names to check if they resolve to the same full name. The result of the comparison is returned in **comparison-result** as follows:

<table>
<thead>
<tr>
<th>comparison-result Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Node names are equal.</td>
</tr>
<tr>
<td>1</td>
<td>Node names are not equal.</td>
</tr>
</tbody>
</table>

**Condition Values Returned**

- **SS$_NORMAL**  
  Routine successfully completed.
- **LIB$_INVARG**  
  Invalid argument:
  - `nodename1` or `nodename2` is an invalid node name.
  - `nodename1` or `nodename2` points to a null string.
  - The length of the node name is more than 1024 characters.
  - The expanded DECnet-Plus for OpenVMS node name is invalid in a DECnet for OpenVMS environment.
- **LIB$_INVSTRDES**  
  Invalid string descriptor.
- **LIB$_WRONUMARG**  
  Wrong number of arguments.

Any condition value returned by RTL routine LIB$SCOPY_R_DX or by the $IPC DECnet service.
LIB$COMPRESS_NODENAME
Compress a Node Name to Its Short Form Equivalence

The Compress a Node Name to Its Short Form Equivalence routine compresses a node name to an unambiguous short form usable within the naming environment where the compression is performed. †

Format

LIB$COMPRESS_NODENAME nodename , compressed-nodename [,resultant-length]

Returns

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

Arguments

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

Node name to be compressed. The nodename argument contains the address of a descriptor pointing to this node-name string.

The error LIB$$_INVARG is returned if nodename contains an invalid node name, points to a null string, or contains more than 1024 characters. The error LIB$$_INVSTRDES is returned if the nodename descriptor is invalid.

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

Compressed node name. The compressed-nodename argument contains the address of a descriptor pointing to the compressed node-name string.

LIB$COMPRESS_NODENAME writes the compressed node name into the buffer pointed to by compressed-nodename.

The error LIB$$_INVSTRDES is returned if compressed-nodename is an invalid descriptor.

The length field of the compressed-nodename descriptor is not updated unless compressed-nodename is a dynamic descriptor with a length less than the resulting compressed node name. Refer to the OpenVMS RTL String Manipulation (STR$) Manual for dynamic string descriptor usage.

The compressed-nodename argument contains an unusable result when LIB$COMPRESS_NODENAME returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
resultant-length
OpenVMS usage: word unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Length of the compressed node name. The resultant-length argument is the address of an unsigned word that contains this length in bytes.

The resultant-length argument contains an unusable result when LIB$COMPRESS_NODENAME returns in error.

Description

This routine compresses a given node name to a short form that is usable within the local naming environment in which the compression is performed. The local naming environment is defined by the underlying network directory services. Be careful when using the compressed node name for making network connections. Using the compressed node name outside the intended local naming environment may result in an ambiguous reference. Use the full name whenever you need to eliminate ambiguity.

The nodename argument is validated against the supported form of node names. The error LIB$_INVAR$ is returned if the input node name is invalid.

When calling LIB$COMPRESS_NODENAME in a DECnet-Plus for OpenVMS environment, the underlying network layer verifies the existence of the input node name. If the input node name does not resolve to an existing node name in the naming environment, an error condition is returned by the underlying network layer and propagated back to the caller of LIB$COMPRESS_NODENAME.

If the returned compressed node name overflows the buffer pointed to by compressed-nodename, the compressed node name is truncated, and the alternate successful status LIB$_STRTRU$ is returned.

The actual length of the compressed node name written to the output buffer compressed-nodename is returned in resultant-length if this argument is supplied.

In a DECnet environment, compressing a DECnet-Plus node name results in the error condition LIB$_INVAR$.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. Characters are truncated in the output buffer pointed to by compressed-nodename.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$COMPRESS_NODENAME

LIB$_INVARG Invalid argument:
• **nodename** is invalid.
• **nodename** points to a null string.
• The length of the node name is more than 1024 characters.
• The compressed DECnet-Plus for OpenVMS node name is invalid in a DECnet for OpenVMS environment.

LIB$_INVSTRDES Invalid string descriptor.
LIB$_WRONUMARG Wrong number of arguments.

Any condition value returned by RTL routine LIB$SCOPY_R_DX or by the $IPC DECnet service.
LIB$ Routines

LIB$CONVERT_DATE_STRING

Convert Date String to Quadword

The Convert Date String to Quadword routine converts an absolute date string into an OpenVMS internal format date-time quadword. That is, given an input date/time string of a specified format, LIB$CONVERT_DATE_STRING converts this string to an OpenVMS internal format time.

Format

LIB$CONVERT_DATE_STRING date-string ,date-time [,user-context] [,flags] [,defaults] [,defaulted-fields]

Returns

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

Arguments

**date-string**
OpenVMS usage: time_name
type: character-coded text string
access: read only
mechanism: by descriptor

Date string that specifies the absolute time to be converted to an internal system time. The **date-string** argument is the address of a descriptor pointing to this date string. This string must have a format corresponding to the currently defined input format, or it must be one of the relative day strings YESTERDAY, TODAY, or TOMORROW, or their equivalents in the currently selected language.

**date-time**
OpenVMS usage: date_time
type: quadword (unsigned)
access: write only
mechanism: by reference

Receives the converted time. The **date-time** argument is the address of an unsigned quadword that contains this OpenVMS internal format converted time.

**user-context**
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference

Context variable that receives the translation context from a call to LIB$INIT_DATE_TIME_CONTEXT and then retains the translation context over multiple calls to LIB$CONVERT_DATE_STRING. The **user-context** argument is the address of an unsigned longword that contains this context. The user program should not write directly to this variable once it is initialized.
The user-context parameter is optional. However, if a context cell is not passed, the routine LIB$CONVERT_DATE_STRING may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, HP recommends that users specify a different context cell for each calling thread.

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

Specifies which date or time fields of the date-string argument might be omitted so that default values are applied. The flags argument is the address of a longword bit mask that contains these flags. A set bit indicates that the field may be omitted. The bit definitions for the mask correspond to the fields in a $NUMTIM “timbuf” structure as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Number</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Month</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Day of month</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Hours</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Minutes</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Seconds</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>Fractional seconds</td>
<td>6</td>
<td>64</td>
</tr>
</tbody>
</table>

Bits 7 through 31 must be zero and are reserved for use by HP. If this parameter is omitted, a default value of 120 (78H) is used, indicating that the time fields may be defaulted but the date fields may not.

defaults
OpenVMS usage: vector_word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference, array reference

Supplies the defaults to be used for omitted fields. The defaults argument is the address of an array of unsigned words containing these default values. This array corresponds to a 7-word $NUMTIM “timbuf” structure. If the defaults argument is omitted, the following defaults are applied:

- For the date group, the default is the current date.
- For the time group, the default is 00:00:00.00.
**defaulted-fields**

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

Indicates which date or time fields have been defaulted. The **defaulted-fields** argument is the address of a longword bit mask that specifies these fields. The bit definitions are identical to those of the **flags** bit mask. A set bit indicates that the field was defaulted. Bits 7 through 31, which are reserved for use by HP, are zeroed.

**Description**

**LIB$CONVERT_DATE_STRING** converts an absolute date string into an OpenVMS internal format date-time quadword. The input date string can either correspond to the format specified, or it can be the language equivalent of one of the relative date strings YESTERDAY, TODAY, or TOMORROW. The language to be used and the format in which to interpret the information are programmable using either of the following methods:

- The language and format are programmable at compile time through the use of the routine **LIB$INIT_DATE_TIME_CONTEXT**.
- The language and format can be determined at run time through the translation of the logical names SYS$LANGUAGE and LIB$DT_INPUT_FORMAT.

In general, if an application is reading text from internal storage, the language and input format should be specified at compile time. If this is the case, use the routine **LIB$INIT_DATE_TIME_CONTEXT** to specify the language and input format of your choice.

If an application is accepting text from a user, the logical name method of specifying language and format should be used. In this method, the user assigns equivalence names to the logical names SYS$LANGUAGE and LIB$DT_INPUT_FORMAT, thereby selecting the language and input format of the date and time at run time.

The calling program can choose to apply defaults for omitted fields in the date string. To do this, the **flags** argument is used to indicate which fields are to be defaulted, and the **defaults** argument is used to supply the default values. If the **defaults** argument is not supplied, the following default values are applied:

- For the date group, the default is the current date.
- For the time group, the default is 00:00:00.00.

Optionally, you can use the **defaulted-fields** argument to receive information on which input fields were omitted and thus accepted default values.

**Note**

Because the default is the current date for the date group, if you specify a value of 00 with the '!Y2 format, the year is interpreted as 1900. After January 1, 2000, the value 00 will be interpreted as 2000.
See the *HP OpenVMS Programming Concepts Manual* for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.

**Condition Values Returned**

SS$_NORMAL
Routine successfully completed.

LIB$_AMBDATTIM
Ambiguous date or time.

LIB$_DEFFORUSE
Default format used; unable to determine desired format.

LIB$_ENGLUSED
English used by default; unable to translate SYS$LANGUAGE.

LIB$_ILLFORMAT
Illegal format string; too many or not enough fields.

LIB$_INCDATTIM
Incomplete date or time; missing fields with no defaults.

LIB$_INVARG
Invalid argument; a required argument was not specified.

LIB$_INVSTRDES
Invalid input string descriptor.

LIB$_IVTIME
Invalid date or time.

LIB$_REENTRANCY
Reentrancy detected.

LIB$_UNRFORCOD
Unrecognized format code.

LIB$_WRONUMARG
Wrong number of arguments.

Any condition value returned by RTL routines LIB$GET_VM, LIB$FREE_VM, LIB$FREE1_DD, and LIB$SCOPY_R_DX, and system services $NUMTIM and $GETTIM.
LIB$CRC
Calculate a Cyclic Redundancy Check (CRC)

The Calculate a Cyclic Redundancy Check routine calculates the cyclic redundancy check (CRC) for a data stream.

Format

LIB$CRC crc-table, initial-crc, stream

Returns

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

The computed cyclic redundancy check.

Arguments

crc-table
OpenVMS usage: vector_longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference, array reference

The 16-longword cyclic redundancy check table created by a call to LIB$CRC_TABLE. The crc-table argument is the address of a signed longword integer containing this table. Because this table is created by LIB$CRC_TABLE and then used as input in LIB$CRC, your program must call LIB$CRC_TABLE before it calls LIB$CRC.

initial-crc
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Initial cyclic redundancy check. The initial-crc argument is the address of a signed longword integer containing the initial cyclic redundancy check.

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

Data stream for which LIB$CRC is calculating the CRC. The stream argument is the address of a descriptor pointing to the data stream.
Description
Before your program can call LIB$CRC, it must call LIB$CRC_TABLE.
LIB$CRC_TABLE takes a polynomial as its input and builds the table that
LIB$CRC uses to calculate the CRC.

LIB$CRC allows your high-level language program to use the CRC instruction,
which calculates the cyclic redundancy check. This instruction checks the
integrity of a data stream by comparing its state at the sending point and the
receiving point. Each character in the data stream is used to generate a value
based on a polynomial. The values for each character are then added together.
This operation is performed at both ends of the data transmission, and the two
result values compared. If the results disagree, then an error occurred during the
transmission.

Condition Values Returned
None.

Example
For an example on how to use LIB$CRC, refer to the BASIC example at the end
of the description of LIB$CRC_TABLE.

1 On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$CRC_TABLE

Construct a Cyclic Redundancy Check (CRC) Table

The Construct a Cyclic Redundancy Check Table routine constructs a 16-longword table that uses a cyclic redundancy check polynomial specification as a bit mask.

Format

LIB$CRC_TABLE polynomial-coefficient,crc-table

Returns

None.

Arguments

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

A bit mask indicating which polynomial coefficients are to be generated by LIB$CRC_TABLE. The polynomial-coefficient argument is the address of an unsigned longword integer containing this bit mask.

crc-table
OpenVMS usage: vector_longword_signed
type: longword integer (signed)
access: write only
mechanism: by reference, array reference

The 16-longword table that LIB$CRC_TABLE produces. The crc-table argument is the address of a signed longword integer containing the table.

Description

The table created by LIB$CRC_TABLE can be passed to the LIB$CRC routine for generating the cyclic redundancy check value for a stream of characters.

Condition Values Returned

None.

Example

1  %TITLE "Demonstrate LIB$CRC and LIB$CRC_TABLE"
2  %SBTTL "Declarations"
3  %IDENT "1-001"
4  ----
5  OPTION TYPE = EXPLICIT
LIB$ Routines
LIB$CRC_TABLE

DECLARE LONG CRC_TABLE(15), ! CRC table array & 
    LONG CRC_VAL_1, ! CRC for first stream & 
    LONG CRC_VAL_2, ! CRC for second stream & 
    STRING DATA_1, ! First data stream & 
    STRING DATA_2 ! Second data stream &

EXTERNAL LONG FUNCTION LIB$CRC ! Rtn to calculate CRC 
EXTERNAL SUB LIB$CRC_TABLE ! Rtn to set up table for CRC 

OPEN "SYS$INPUT:" FOR INPUT AS FILE 1%
!
! Initialize the CRC table. Use the CRC-16 polynomial (refer to the ! 
"VAX Architecture Reference Manual"). This is the polynomial used by ! 
DDCMP and Bisync. !-
CALL LIB$CRC_TABLE( O'120001'L, CRC_TABLE() BY REF ) !+

! Get data from user. !-
LINPUT #1%, 'Enter string: ';DATA_1 !+
! Calc the CRC for the user's input. This CRC polynomial needs ! 
an initial CRC of 0 (refer to the "VAX Architecture Reference Manual"). ! 
LIB$CRC returns a longword, but only the low-order word is valid ! 
for this polynomial. !-
CRC VAL_1 = LIB$CRC( CRC_TABLE() BY REF, 0%, DATA_1 ) 
CRC VAL_1 = CRC VAL_1 AND 32767%
!
! Get more data from user. !-
LINPUT #1%, 'Enter a second string: ';DATA_2 
CRC VAL_2 = LIB$CRC( CRC_TABLE() BY REF, 0%, DATA_2 ) 
CRC VAL_2 = CRC VAL_2 AND 32767%
!

! Tell the user the results of the CRC comparison. !-
IF CRC VAL_1 = CRC VAL_2 THEN 
    PRINT "The two CRCs";CRC VAL_1;" and ";CRC VAL_2;" were the same" 
ELSE 
    PRINT "The two CRCs";CRC VAL_1;" and ";CRC VAL_2;" were different" 
END IF

IF DATA_1 = DATA_2 THEN 
    PRINT "The two strings were the same" 
ELSE 
    PRINT "The two strings were different" 
END IF

END

This BASIC example program shows the use of LIB$CRC and LIB$CRC_TABLE. 
One example of the output generated by this program is as follows:
$ RUN CRC
Enter string: DOVE
Enter a second string: HOSE
The two CRCs 29915 and 29915 were the same
The two strings were different
LIB$CREATE_DIR
Create a Directory

The Create a Directory routine creates a directory or subdirectory.

Format


Returns

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

Arguments

device-directory-spec
OpenVMS usage: device_name
type: character string
access: read only
mechanism: by descriptor

Directory specification of the directory or subdirectory that LIB$CREATE_DIR will create. The device-directory-spec argument is the address of a descriptor pointing to this directory specification.

The format of the device-directory-spec string conforms to standard OpenVMS Record Management Services (RMS) format. This specification must contain a directory or subdirectory specification. It may contain a disk specification. SMD$:[THIS.IS.IT] is an example of a standard RMS file specification, where SMD$ is the disk specification and [THIS.IS.IT] is the subdirectory specification.

This specification cannot contain a node name, file name, file type, file version, or wildcard characters. The maximum size of this string is 255 characters on VAX, and 4095 characters on Alpha.

owner-UIC
OpenVMS usage: uic
type: longword (unsigned)
access: read only
mechanism: by reference

User identification code (UIC) identifying the owner of the created directory or subdirectory. The owner-UIC argument is the address of an unsigned longword that contains the UIC. If owner-UIC is zero, the owner UIC is that of the parent directory. The specified value for owner-UIC is interpreted as a 32-bit octal number, with two 16-bit fields:

bits 00–15 — Member number
bits 16–31 — Group number

This is an optional argument. The default is the UIC of the current process except when the directory is in UIC format. For a directory in UIC format, for example [123,321], the UIC of the created directory is used.
**protection-enable**

OpenVMS usage: mask_word

type: word (unsigned)

access: read only

mechanism: by reference

Mask specifying the bits of **protection-value** to be set. The **protection-enable** argument is the address of an unsigned word containing this protection mask.

Figure lib–1 shows the structure of a protection mask. Access is allowed for bits set to 0.

**Figure lib–1  Structure of a Protection Mask**

<table>
<thead>
<tr>
<th>World</th>
<th>Group</th>
<th>Owner</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>E</td>
<td>X</td>
<td>R</td>
<td>E</td>
</tr>
<tr>
<td>L</td>
<td>E</td>
<td>I</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>T</td>
<td>U</td>
<td>E</td>
<td>T</td>
</tr>
<tr>
<td>E</td>
<td>T</td>
<td>E</td>
<td>T</td>
</tr>
</tbody>
</table>

15 0

ZK–1979–GE

Bits set in the **protection-enable** mask cause corresponding bits of **protection-value** to be set. Bits not set in the **protection-enable** mask cause corresponding bits of **protection-value** to take the value of the corresponding bit in the parent directory's file protection. Bits in the parent directory's file protection that indicate delete access do not cause corresponding bits of **protection-value** to be set, however.

Following is an example of how the **protection-value** protection mask is defined:

<table>
<thead>
<tr>
<th>Mask Name</th>
<th>Hexadecimal Number</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection enable</td>
<td>%XDBFF</td>
<td>S:None, O:None, G:E, W:W</td>
</tr>
<tr>
<td>Parent directory</td>
<td>%X13FF</td>
<td>S:RWED, O:RWED, G:RW, W:R</td>
</tr>
<tr>
<td>Protection value</td>
<td>%X37FF</td>
<td>S:RWE, O:RWE, G:RWE, W:RW</td>
</tr>
</tbody>
</table>

The **protection-enable** argument is optional. It should be used only when you want to change protection values from the parent directory's default file protection. The default for **protection-enable** is a mask of all zero bits, which results in the propagation of the parent directory's file protection. If the **protection-enable** mask contains zeros, **protection-value** is ignored.
**LIB$ Routines**

**LIB$CREATE_DIR**

**protection-value**
OpenVMS usage: file_protection
type: word (unsigned)
access: read only
mechanism: by reference

System/Owner/Group/World protection value of the directory you are creating. The protection-value argument is the address of an unsigned word that contains this protection mask.

The bits of protection-value are set or cleared in the method described in the definition of protection-enable above.

The protection-value argument is optional. The default is a word of all zero bits, which specifies full access for all access categories. Typically, protection-value is not omitted unless protection-enable is also omitted. If protection-enable is omitted, protection-value is ignored.

**maximum-versions**
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

Maximum number of versions allowed for files created in the newly created directories. The maximum-versions argument is the address of an unsigned word containing the value of the maximum number of versions.

The maximum-versions argument is optional. The default is the parent directory's default version limit. If maximum-versions is zero, the maximum number of versions is not limited.

**relative-volume-number**
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

Relative volume number within a volume set on which the directory or subdirectory is created. The relative-volume-number argument is the address of an unsigned word containing the relative volume number. The relative-volume-number argument is optional. The default is arbitrary placement within the volume set.

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

Initial number of blocks to be allocated to the directory. This argument is useful for creating large directories, for example MAIL.DIR;1. It can improve performance by avoiding the need for later dynamic expansion of the directory.
The initial-allocation argument applies only to Files–11 Level 2 volumes; it is ignored for other volumes. This argument is the address of an unsigned longword that contains the initial number of blocks to be allocated to the directory. The initial-allocation argument is optional. The default allocation is 1 block.

**Description**

LIB$CREATE_DIR creates a directory. You can specify:

- The owner and protection of the directory.
- The maximum number of different versions of a file that can exist in the directory.
- The relative volume number of the volume set member in which the directory is to be created.
- The number of blocks to be allocated initially to the directory.

**Note**

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _CREATED</td>
<td>Routine successfully completed; one or more directories created.</td>
</tr>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed; all specified directories already exist.</td>
</tr>
<tr>
<td>LIB$ _INVARG</td>
<td>Invalid argument to Run-Time Library. Either the required argument was omitted, or device-directory-spec is longer than 4095 characters.</td>
</tr>
<tr>
<td>LIB$ _INVFILSPE</td>
<td>Invalid file specification. Either the file specification did not contain an explicit directory and device name, or it contained a node name, file name, file type, file version, or wildcard. This error is also produced if the device specified was not a disk.</td>
</tr>
</tbody>
</table>

Any condition values returned by system services $ASSIGN, $DASSGN, $PARSE, and $QIO, and RTL routines LIB$ANALYZE_SDESC, LIB$ANALYZE_SDESC_64, and LIB$GET_EF.
LIB$ Routines
LIB$CREATE_USER_VM_ZONE

LIB$CREATE_USER_VM_ZONE
Create User-Defined Storage Zone

The Create User-Defined Storage Zone routine creates a new user-defined storage zone in the 32-bit virtual address space. †

Format


Returns

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

Arguments

zone-id
OpenVMS usage: identifier
type: longword (unsigned)
access: write only
mechanism: by reference

Zone identifier. The zone-id argument is the address of a longword that receives the identifier of the newly created zone.

user-argument
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: by reference

User argument. The user-argument argument is the address of an unsigned longword containing the user argument. LIB$CREATE_USER_VM_ZONE copies the value of user-argument and supplies the value to all user procedures invoked.

user-allocation-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User allocation routine.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
**user-deallocation-procedure**
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User deallocation routine.

**user-reset-procedure**
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User routine invoked each time LIB$RESET_VM_ZONE is called for the zone.

**user-delete-procedure**
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User routine invoked when LIB$DELETE_VM_ZONE is called for the zone.

**zone-name**
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Name to be associated with the zone being created. The optional zone-name argument is the address of a descriptor pointing to the zone name. If zone-name is not specified, the zone will not have an associated name.

**Description**

LIB$CREATE_USER_VM_ZONE creates a user-defined zone in the 32-bit virtual address space. If an error status is returned, the zone is not created.

Each time that one of the heap management routines (LIB$GET_VM, LIB$FREE_VM, LIB$RESET_VM_ZONE, or LIB$DELETE_VM_ZONE) is called to perform an operation on a user-defined zone, the corresponding user routine that you supplied is used.

You may omit any of the optional user routines. However, if you omit a routine and later call the corresponding heap management routine, the error status LIB$_INVOPEZON will be returned.
Call Format for User Routines

The user routines are called with arguments similar to those passed to LIB$GET_VM, LIB$FREE_VM, LIB$RESET_VM_ZONE, or LIB$DELETE_VM_ZONE. In each case, the user-argument argument from LIB$CREATE_USER_VM_ZONE is passed to the user routine rather than a zone-id argument.

The call format for a user get or free routine is as follows:

```
user-rtn num-bytes, base-adr, user-argument
```

**num-bytes**

OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Number of contiguous bytes to allocate or free. The num-bytes argument is the address of a longword integer containing the number of bytes. The value of num-bytes must be greater than zero.

**base-adr**

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

Virtual address of the first contiguous block of bytes allocated or freed. The base-adr argument is the address of an unsigned longword containing this base address. (This argument is write-only for a get routine and read-only for a free routine.)

**user-argument**

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

User argument. LIB$CREATE_USER_VM_ZONE copies user-argument as it is supplied to all user routines invoked.

The status value returned by your routine is returned as the status value for the corresponding call to LIB$GET_VM or LIB$FREE_VM.

The zone-id value that is returned can be used in calls to LIB$SHOW_VM_ZONE and LIB$VERIFY_VM_ZONE.

The call format for a user reset or delete routine is as follows:

```
user-rtn user-argument
```

**user-argument**

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

User argument. LIB$CREATE_USER_VM_ZONE copies user-argument as it is supplied to all user routines invoked.
The status value returned by your routine is returned as the status value for the corresponding call to LIB$RESET_VM_ZONE or LIB$DELETE_VM_ZONE.

**Condition Values Returned**

- **SS$ _NORMAL**  
  Routine successfully completed.

- **LIB$ _INSVIRMEM**  
  Insufficient virtual memory.

- **LIB$ _INVSTRDES**  
  Invalid string descriptor for `zone-name`. 
LIB$ Routines
LIB$CREATE_USER_VM_ZONE_64 (Alpha and I64 Only)

LIB$CREATE_USER_VM_ZONE_64 (Alpha and I64 Only)
Create User-Defined Storage Zone

The Create User-Defined Storage Zone routine creates a new user-defined storage zone in the 64-bit virtual address space.

Format

LIB$CREATE_USER_VM_ZONE_64  zone-id [,user-argument] [,user-allocation-procedure]
[,user-deallocation-procedure] [,user-reset-procedure]
[,user-delete-procedure] [,zone-name]

Returns

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

Arguments

zone-id
OpenVMS usage:  identifier
type:  quadword (unsigned)
access:  write only
mechanism:  by reference

Zone identifier. The zone-id argument is the address of a quadword that receives the identifier of the newly created zone.

user-argument
OpenVMS usage:  user_arg
type:  quadword (unsigned)
access:  read only
mechanism:  by reference

User argument. The user-argument argument is the address of an unsigned quadword containing the user argument. LIB$CREATE_USER_VM_ZONE_64 copies the value of user-argument and supplies the value to all user procedures invoked.

user-allocation-procedure
OpenVMS usage:  procedure
type:  procedure value
access:  function call (before return)
mechanism:  by value

User allocation routine.

user-deallocation-procedure
OpenVMS usage:  procedure
type:  procedure value
access:  function call (before return)
mechanism:  by value

User deallocation routine.
LIB$ Routines

LIB$CREATE_USER_VM_ZONE_64 (Alpha and I64 Only)

user-reset-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User routine invoked each time LIB$RESET_VM_ZONE_64 is called for the zone.

user-delete-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User routine invoked when LIB$DELETE_VM_ZONE_64 is called for the zone.

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

Name to be associated with the zone being created. The optional zone-name argument is the address of a descriptor pointing to the zone name. If zone-name is not specified, the zone will not have an associated name.

Description

LIB$CREATE_USER_VM_ZONE_64 creates a user-defined zone in the 64-bit virtual address space. If an error status is returned, the zone is not created.

Each time that one of the heap management routines (LIB$GET_VM_64, LIB$FREE_VM_64, LIB$RESET_VM_ZONE_64, or LIB$DELETE_VM_ZONE_64) is called to perform an operation on a user-defined zone, the corresponding user routine that you supplied is used.

You may omit any of the optional user routines. However, if you omit a routine and later call the corresponding heap management routine, the error status LIB$_INVOKEZON will be returned.

Call Format for User Routines

The user routines are called with arguments similar to those passed to LIB$GET_VM_64, LIB$FREE_VM_64, LIB$RESET_VM_ZONE_64, or LIB$DELETE_VM_ZONE_64. In each case, the user-argument argument from LIB$CREATE_USER_VM_ZONE_64 is passed to the user routine rather than a zone-id argument.

The call format for a user get or free routine is as follows:

user-rtn num-bytes ,base-adr ,user-argument

num-bytes
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference
Number of contiguous bytes to allocate or free. The num-bytes argument is the address of a quadword integer containing the number of bytes. The value of num-bytes must be greater than zero.

**base-adr**
- OpenVMS usage: address
- type: quadword (unsigned)
- access: modify
- mechanism: by reference

Virtual address of the first contiguous block of bytes allocated or freed. The base-adr argument is the address of an unsigned quadword containing this base address. (This argument is write-only for a get routine and read-only for a free routine.)

**user-argument**
- OpenVMS usage: user_arg
- type: quadword (unsigned)
- access: read only
- mechanism: by reference

User argument. LIB$CREATE_USER_VM_ZONE_64 copies user-argument as it is supplied to all user routines invoked.

The status value returned by your routine is returned as the status value for the corresponding call to LIB$GET_VM_64 or LIB$FREE_VM_64.

The zone-id value that is returned can be used in calls to LIB$SHOW_VM_ZONE_64 and LIB$VERIFY_VM_ZONE_64.

The call format for a user reset or delete routine is as follows:

**user-rtn** user-argument

**user-argument**
- OpenVMS usage: user_arg
- type: quadword (unsigned)
- access: read only
- mechanism: by reference

User argument. LIB$CREATE_USER_VM_ZONE_64 copies user-argument as it is supplied to all user routines invoked.

The status value returned by your routine is returned as the status value for the corresponding call to LIB$RESET_VM_ZONE_64 or LIB$DELETE_VM_ZONE_64.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor for zone-name.</td>
</tr>
</tbody>
</table>
LIB$CREATE_VM_ZONE
Create a New Zone

The Create a New Zone routine creates a new storage zone in the 32-bit virtual address space, according to specified arguments.†

Format


Returns

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

Arguments

zone-id
OpenVMS usage: identifier
type: longword (unsigned)
access: write only
mechanism: by reference

Zone identifier. The zone-id argument is the address of a longword that is set to the zone identifier of the newly created zone.

algorithm
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Algorithm. The algorithm argument is the address of a longword integer that contains a value representing one of the LIB$VM algorithms. Use one of the predefined symbols to specify this value.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_VM_FIRST_FIT</td>
<td>1</td>
<td>First fit</td>
</tr>
<tr>
<td>LIB$K_VM_QUICK_FIT</td>
<td>2</td>
<td>Quick fit, lookaside list</td>
</tr>
<tr>
<td>LIB$K_VM_FREQ_SIZES</td>
<td>3</td>
<td>Frequent sizes, lookaside list</td>
</tr>
<tr>
<td>LIB$K_VM_FIXED</td>
<td>4</td>
<td>Fixed-size blocks</td>
</tr>
</tbody>
</table>

If algorithm is not specified, a default of 1 (first fit) is used.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
algorithm-argument
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Algorithm argument. The **algorithm-argument** argument is the address of a longword integer that contains a value specific to the particular allocation algorithm as shown in the following table.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First fit</td>
<td>Not used, may be omitted.</td>
</tr>
<tr>
<td>Quick fit</td>
<td>The number of lookaside lists used. The number of lists must be between 1 and 128.</td>
</tr>
<tr>
<td>Frequent sizes</td>
<td>The number of lookaside lists used. The number of lists must be between 1 and 16.</td>
</tr>
<tr>
<td>Fixed size blocks</td>
<td>The fixed request size (in bytes) for each get or free request. The request size must be greater than 0.</td>
</tr>
</tbody>
</table>

The **algorithm-argument** argument must be specified if you are using the quick-fit, frequent-sizes or fixed-size-blocks algorithms. However, this argument is optional, but ignored, if you are using the first-fit algorithm.

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

Flags. The **flags** argument is the address of a longword integer that contains flag bits that control various options, as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIB$M_VM_BOUNDARY_TAGS</td>
<td>Boundary tags for faster freeing. Adds a minimum of 8 bytes to each block.</td>
</tr>
<tr>
<td>1</td>
<td>LIB$M_VM_GET_FILL0</td>
<td>LIB$GET_VM; fill with bytes of 0.</td>
</tr>
<tr>
<td>2</td>
<td>LIB$M_VM_GET_FILL1</td>
<td>LIB$GET_VM; fill with bytes of FF (hexadecimal).</td>
</tr>
<tr>
<td>3</td>
<td>LIB$M_VM_FREE_FILL0</td>
<td>LIB$FREE_VM; fill with bytes of 0.</td>
</tr>
<tr>
<td>4</td>
<td>LIB$M_VM_FREE_FILL1</td>
<td>LIB$FREE_VM; fill with bytes of FF (hexadecimal).</td>
</tr>
<tr>
<td>5</td>
<td>LIB$M_VM_EXTEND_AREA</td>
<td>Adds extents to existing areas if possible.</td>
</tr>
<tr>
<td>Bit</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6</td>
<td>LIB$M_VM_NO_EXTEND</td>
<td>Prevents zone from being extended beyond its initial size. If you specify</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this flag, you must also specify an initial-size. The extend-size argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is not used.</td>
</tr>
<tr>
<td>7</td>
<td>LIB$M_VM_TAIL_LARGE</td>
<td>Adds areas larger than extend-size areas to the end of the area list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allocations that are larger than extend-size can result in new areas. These</td>
</tr>
<tr>
<td></td>
<td></td>
<td>areas are added to the end of the area list. (This provides better memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reuse when allocating small and very large blocks from the same zone.)</td>
</tr>
</tbody>
</table>

Bits 8 through 31 are reserved and must be 0.

This is an optional argument. If flags is omitted, the default of 0 (no fill and no boundary tags) is used.

**extend-size**

OpenVMS usage: longword_signed  
Type: longword integer (signed)  
Access: read only  
Mechanism: by reference  

Zone extend size. The extend-size argument is the address of a longword integer that contains the number of (512-byte) pages on VAX systems or pagelets on Alpha and I64 systems to be added to the zone each time it is extended.

The value of extend-size must be greater than or equal to 1.

This is an optional argument. If extend-size is not specified, a default of 16 pages on VAX systems or pagelets on Alpha and I64 systems is used.

**Note**

The extend-size argument does not limit the number of blocks that can be allocated from the zone. The actual extension size is the greater of extend-size and the number of pages on VAX systems or pagelets on Alpha and I64 systems needed to satisfy the LIB$GET_VM call that caused the extension.

**initial-size**

OpenVMS usage: longword_signed  
Type: longword integer (signed)  
Access: read only  
Mechanism: by reference  

Initial size for the zone. The initial-size argument is the address of a longword integer that contains the number of (512-byte) pages on VAX systems or pagelets on Alpha and I64 systems to be allocated for the zone as the zone is created.
This is an optional argument. If you specify a value for initial-size, the value must be greater than or equal to 0; otherwise, LIB$_INVARG is returned. If initial-size is not specified or is specified as 0, no pages on VAX systems or pagelets on Alpha and I64 systems are allocated when the zone is created. The first call to LIB$GET_VM for the zone allocates extend-size pages on VAX systems or pagelets on Alpha and I64 systems.

**block-size**

OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Block size of the zone. The block-size argument is the address of a longword integer specifying the allocation quantum (in bytes) for the zone. All blocks allocated are rounded up to a multiple of block-size.

The value of block-size must be a power of 2 between 8 and 512. This is an optional argument. If block-size is not specified, a default of 8 is used.

**alignment**

OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Block alignment. The alignment argument is the address of a longword integer that specifies the required address alignment (in bytes) for each block allocated.

The value of alignment must be a power of 2 between 4 and 512. This is an optional argument. If alignment is not specified, a default of 8 (quadword alignment) is used.

**page-limit**

OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Maximum page limit. The page-limit argument is the address of a longword integer that specifies the maximum number of (512-byte) pages on VAX systems or pagelets on Alpha and I64 systems that can be allocated for the zone. The value of page-limit must be greater than or equal to 0. Note that part of the zone is used for header information.

This is an optional argument. If page-limit is not specified or is specified as 0, the only limit is the total process virtual address space limit imposed by OpenVMS. If page-limit is specified, then initial-size must also be specified.

**smallest-block-size**

OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Smallest block size. The smallest-block-size argument is the address of a longword integer that specifies the smallest block size (in bytes) that has a lookaside list for the quick fit algorithm.
If **smallest-block-size** is not specified, the default of **block-size** is used. That is, lookaside lists are provided for the first \( n \) multiples of **block-size**.

**zone-name**

OpenVMS usage: char_string

type: character string

access: read only

mechanism: by descriptor

Name to be associated with the zone being created. The optional **zone-name** argument is the address of a descriptor pointing to the zone name. If **zone-name** is not specified, the zone will not have an associated name.

**get-page**

OpenVMS usage: procedure

type: procedure value

access: read only

mechanism: by value

Routine that allocates memory. The number and type of the arguments to this routine must match those of the LIB$GET_VM_PAGE routine. If **get-page** is not specified or is specified as 0, the LIB$GET_VM_PAGE routine is used to allocate memory.

**free-page**

OpenVMS usage: procedure

type: procedure value

access: read only

mechanism: by value

Routine that deallocates memory. The number and type of the arguments to this routine must match those of the LIB$FREE_VM_PAGE routine. If **free-page** is not specified or if **free-page** is specified as 0, the LIB$FREE_VM_PAGE routine is used to deallocate memory.

**Description**

LIB$CREATE_VM_ZONE creates a new storage zone. The zone identifier value that is returned can be used in calls to LIB$GET_VM, LIB$FREE_VM, LIB$RESET_VM_ZONE, LIB$DELETE_VM_ZONE, LIB$SHOW_VM_ZONE, LIB$VERIFY_VM_ZONE, and LIB$CREATE_USER_VM_ZONE.

The following restrictions apply when you are creating a zone:

- If you want the zone to be accessible from another process or processes, you must map the global section into the same virtual addresses in all processes. You can use PPL$CREATE_SHARED_MEM to map to a global section after you have first called PPL$INITIALIZE.

- The zone cannot expand; in other words, additional areas cannot be added to the zone.

- The restrictions for LIB$RESET_VM_ZONE also apply to shared zones. That is, it is the caller’s responsibility to ensure that the caller has exclusive access to the zone while the reset operation is being performed.

If an error status is returned, the zone is not created.
LIB$ Routines
LIB$CREATE_VM_ZONE

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor for zone-name.</td>
</tr>
</tbody>
</table>
LIB$CREATE_VM_ZONE_64 (Alpha and I64 Only)

Create a New Zone

The Create a New Zone routine creates a new storage zone in the 64-bit virtual address space, according to specified arguments.

Format


Returns

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

Arguments

zone-id
OpenVMS usage: identifier
type: quadword (unsigned)
access: write only
mechanism: by reference

Zone identifier. The zone-id argument is the address of a quadword that is set to the zone identifier of the newly created zone.

algorithm
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Algorithm. The algorithm argument is the address of a quadword integer that represents the code for one of the LIB$VM algorithms. Use one of the following predefined symbols to specify this value:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_VM_FIRST_FIT</td>
<td>1</td>
<td>First fit</td>
</tr>
<tr>
<td>LIB$K_VM_QUICK_FIT</td>
<td>2</td>
<td>Quick fit, lookaside list</td>
</tr>
<tr>
<td>LIB$K_VM_FREQ_SIZES</td>
<td>3</td>
<td>Frequent sizes, lookaside list</td>
</tr>
<tr>
<td>LIB$K_VM_FIXED</td>
<td>4</td>
<td>Fixed-size blocks</td>
</tr>
</tbody>
</table>

If algorithm is not specified, a default of 1 (first fit) is used.
**LIB$ Routines**
**LIB$CREATE_VM_ZONE_64 (Alpha and I64 Only)**

**algorithm-argument**
OpenVMS usage: quadword_signed  
type: quadword integer (signed)  
access: read only  
mechanism: by reference

Algorithm argument. The **algorithm-argument** argument is the address of a quadword integer that contains a value specific to the particular allocation algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First fit</td>
<td>Not used, may be omitted.</td>
</tr>
<tr>
<td>Quick fit</td>
<td>The number of lookaside lists used. The number of lists must be between 1 and 128.</td>
</tr>
<tr>
<td>Frequent sizes</td>
<td>The number of lookaside lists used. The number of lists must be between 1 and 16.</td>
</tr>
<tr>
<td>Fixed size blocks</td>
<td>The fixed request size (in bytes) for each get or free request. The request size must be greater than 0.</td>
</tr>
</tbody>
</table>

The **algorithm-argument** argument must be specified if you are using the quick-fit, frequent-sizes or fixed-size-blocks algorithms. However, this argument is optional, but ignored, if you are using the first-fit algorithm.

**flags**
OpenVMS usage: mask_quadword  
type: quadword (unsigned)  
access: read only  
mechanism: by reference

Flags. The **flags** argument is the address of a quadword integer that contains flag bits that control various options, as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIB$M_VM_BOUNDARY_TAGS</td>
<td>Boundary tags for faster freeing. Adds a minimum of 16 bytes to each block.</td>
</tr>
<tr>
<td>1</td>
<td>LIB$M_VM_GET_FILL0</td>
<td>LIB$GET_VM_64; fill with bytes of 0.</td>
</tr>
<tr>
<td>2</td>
<td>LIB$M_VM_GET_FILL1</td>
<td>LIB$GET_VM_64; fill with bytes of FF (hexadecimal).</td>
</tr>
<tr>
<td>3</td>
<td>LIB$M_VM_FREE_FILL0</td>
<td>LIB$FREE_VM_64; fill with bytes of 0.</td>
</tr>
<tr>
<td>4</td>
<td>LIB$M_VM_FREE_FILL1</td>
<td>LIB$FREE_VM_64; fill with bytes of FF (hexadecimal).</td>
</tr>
<tr>
<td>5</td>
<td>LIB$M_VM_EXTEND_AREA</td>
<td>Adds extents to existing areas if possible.</td>
</tr>
</tbody>
</table>
LIB$ Routines

LIB$CREATE_VM_ZONE_64 (Alpha and I64 Only)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>LIB$M_VM_NO_EXTEND</td>
<td>Prevents zone from being extended beyond its initial size. If you specify</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this flag, you must also specify an initial-size. Extend-size is not used.</td>
</tr>
<tr>
<td>7</td>
<td>LIB$M_VM_TAIL_LARGE</td>
<td>Adds areas larger than extend-size areas to the end of the area list. Allocations that are larger than extend-size can result in new areas. These areas are added to the end of the area list. (This provides better memory re-use when allocating small and very large blocks from the same zone.)</td>
</tr>
</tbody>
</table>

Bits 8 through 63 are reserved and must be 0.

This is an optional argument. If flags is omitted, the default of 0 (no fill and no boundary tags) is used.

**extend-size**

OpenVMS usage: quadword_signed

<table>
<thead>
<tr>
<th>type</th>
<th>quadword integer (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by reference</td>
</tr>
</tbody>
</table>

Zone extend size. The **extend-size** argument is the address of a quadword integer that contains the number of Alpha and I64 pagelets to be added to the zone each time it is extended.

The value of **extend-size** must be greater than or equal to 1.

This is an optional argument. If **extend-size** is not specified, a default of 16 Alpha or I64 pagelets is used.

**Note**

The **extend-size** argument does not limit the number of blocks that can be allocated from the zone. The actual extension size is the greater of **extend-size** and the number of Alpha or I64 pagelets needed to satisfy the LIB$GET_VM_64 call that caused the extension.

**initial-size**

OpenVMS usage: quadword_signed

<table>
<thead>
<tr>
<th>type</th>
<th>quadword integer (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>read only</td>
</tr>
<tr>
<td>mechanism</td>
<td>by reference</td>
</tr>
</tbody>
</table>

Initial size for the zone. The **initial-size** argument is the address of a quadword integer that contains the number of Alpha or I64 pagelets to be allocated for the zone as the zone is created.
This is an optional argument. If you specify a value for `initial-size`, the value must be greater than or equal to 0; otherwise, LIB$_INVARG is returned. If `initial-size` is not specified or is specified as 0, no Alpha pagelets or I64 are allocated when the zone is created. The first call to LIB$GET_VM_64 for the zone allocates `extend-size` pagelets on Alpha or I64 systems.

**block-size**

OpenVMS usage: quadword_signed  
type: quadword integer (signed)  
access: read only  
mechanism: by reference

Block size of the zone. The `block-size` argument is the address of a quadword integer specifying the allocation quantum (in bytes) for the zone. All blocks allocated are rounded up to a multiple of `block-size`.

The value of `block-size` must be a power of 2 between 16 and 512. This is an optional argument. If `block-size` is not specified, a default of 16 is used.

**alignment**

OpenVMS usage: quadword_signed  
type: quadword integer (signed)  
access: read only  
mechanism: by reference

Block alignment. The `alignment` argument is the address of a quadword integer that specifies the required address alignment (in bytes) for each block allocated.

The value of `alignment` must be a power of 2 between 8 and 512. This is an optional argument. If `alignment` is not specified, a default of 16 (octaword alignment) is used.

**page-limit**

OpenVMS usage: quadword_signed  
type: quadword integer (signed)  
access: read only  
mechanism: by reference

Maximum page limit. The `page-limit` argument is the address of a quadword integer that specifies the maximum number of Alpha or I64 pagelets that can be allocated for the zone. The value of `page-limit` must be greater than or equal to 0. Note that part of the zone is used for header information.

This is an optional argument. If `page-limit` is not specified or is specified as 0, the only limit is the total process virtual address space limit imposed by OpenVMS. If `page-limit` is specified, then `initial-size` must also be specified.

**smallest-block-size**

OpenVMS usage: quadword_signed  
type: quadword integer (signed)  
access: read only  
mechanism: by reference

Smallest block size. The `smallest-block-size` argument is the address of a quadword integer that specifies the smallest block size (in bytes) that has a lookaside list for the quick fit algorithm.

If `smallest-block-size` is not specified, the default of `block-size` is used. That is, lookaside lists are provided for the first $n$ multiples of `block-size`. 
LIB$ Routines
LIB$CREATE_VM_ZONE_64 (Alpha and I64 Only)

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

Name to be associated with the zone being created. The optional zone-name argument is the address of a descriptor pointing to the zone name. If zone-name is not specified, the zone will not have an associated name.

get-page
OpenVMS usage: procedure
type: procedure value
access: read only
mechanism: by value

Routine that allocates memory. The number and type of the arguments to this routine must match those of the LIB$GET_VM_PAGE_64 routine. If get-page is not specified or is specified as 0, the LIB$GET_VM_PAGE_64 routine is used to allocate memory.

free-page
OpenVMS usage: procedure
type: procedure value
access: read only
mechanism: by value

Routine that deallocates memory. The number and type of the arguments to this routine must match those of the LIB$FREE_VM_PAGE_64 routine. If free-page is not specified or if free-page is specified as 0, the LIB$FREE_VM_PAGE_64 routine is used to deallocate memory.

Description
LIB$CREATE_VM_ZONE_64 creates a new storage zone. The zone identifier value that is returned can be used in calls to LIB$GET_VM_64, LIB$FREE_VM_64, LIB$RESET_VM_ZONE_64, LIB$DELETE_VM_ZONE_64, LIB$SHOW_VM_ZONE_64, LIB$VERIFY_VM_ZONE_64, and LIB$CREATE_USER_VM_ZONE_64.

The following restrictions apply when you are creating a zone:

• If you want the zone to be accessible from another process or processes, you must map the global section into the same virtual addresses in all processes.

• The zone cannot expand; in other words, additional areas cannot be added to the zone.

• The restrictions for LIB$RESET_VM_ZONE_64 also apply to shared zones. That is, it is the caller’s responsibility to ensure that the caller has exclusive access to the zone while the reset operation is being performed.

If an error status is returned, the zone is not created.
LIB$ Routines
LIB$CREATE_VM_ZONE_64 (Alpha and I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>LIB$ _INVARG</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>LIB$ _INVSTRDES</td>
<td>Invalid string descriptor for zone-name.</td>
</tr>
</tbody>
</table>

lib–68
LIB$CRF_INS_KEY
Insert Key in Cross-Reference Table

The Insert Key in Cross-Reference Table routine inserts information about a key into a cross-reference table. †

Format

LIB$CRF_INS_KEY control-table,key-string,symbol-value,flags

Returns

None.

Arguments

control-table
OpenVMS usage: vector_longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference, array reference

Cross-reference table into which LIB$CRF_INS_KEY inserts information about the key. The control-table argument is the address of a signed longword integer pointing to the cross-reference table. You must name this table each time you call a cross-reference routine because you can accumulate information for more than one cross-reference table at a time.

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

A counted ASCII string that contains a symbol name or an unsigned binary longword. The key-string argument is the address of a descriptor pointing to the key.

symbol-value
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Symbol value, the address of which LIB$CRF_INS_KEY inserts in the cross-reference table. The symbol-value argument is the address of a signed longword integer containing this value. Both the key and value addresses must be permanent addresses in the user's symbol table.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$CRF_INS_KEY

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

Value used in selecting the contents of the KEY2 and VAL2 fields; **flags** is stored with the entry. The flags argument is the address of an unsigned longword containing the flags. When preparing the output line, LIB$CRF_OUTPUT uses flags and the 16-bit mask in the field descriptor table to extract the data. The high-order bit of the word is reserved for LIB$CRF_INS_KEY.

**Description**
LIB$CRF_INS_KEY stores information to be printed in the KEY1, KEY2, VAL1, and VAL2 fields. When you call this routine, an entry for the key is made in the cross-reference table if the key is not present in the table. If the key is present, only the value address and value flag fields are updated.

Using LIB$CRF_INS_KEY involves the following steps:

1. Define a table of control information using the $CRFCTLTABLE macro.
2. Define each field of the output line using the $CRFFIELD macro.
3. Using the $CRFFIELDEND macro, specify the end of each set of macros that define a field in the output line.
4. Provide data by calling LIB$CRF_INS_KEY to insert an entry for the specify key in the specified symbol table. This data is used to build tables in virtual memory.
5. Call LIB$CRF_OUTPUT, the cross-reference output routine, to summarize and format the data. Supply a routine that LIB$CRF_OUTPUT calls to print each line in the output file. Because you supply this routine, you can control the number of lines per page and the header lines.

**Condition Values Returned**
None.
LIB$CRF_INS_REF
Insert Reference to a Key in the Cross-Reference Table

The Insert Reference to a Key in the Cross-Reference Table routine inserts a reference to a key in a cross-reference symbol table. †

**Format**

```
LIB$CRF_INS_REF control-table,longword-integer-key,reference-string,longword-integer-reference,ref-definition-indicator
```

**Returns**

None.

**Arguments**

- **control-table**
  - OpenVMS usage: vector_longword_signed
  - type: longword integer (signed)
  - access: read only
  - mechanism: by reference, array reference
  - Control table associated with this cross-reference. The control-table argument is the address of an array containing the control table.

- **longword-integer-key**
  - OpenVMS usage: longword_signed
  - type: longword integer (signed)
  - access: read only
  - mechanism: by reference
  - Key referred to by LIB$CRF_INS_REF. The longword-integer-key argument is the address of a signed longword integer containing the key. The key is a counted ASCII string that contains a symbol name or an unsigned binary longword. It must be a permanent address in the user's symbol table.

- **reference-string**
  - OpenVMS usage: char_string
  - type: character string
  - access: read only
  - mechanism: by descriptor
  - Counted ASCII string with a maximum of 31 characters, not including the byte count. The reference-string argument is the address of a descriptor pointing to the counted ASCII string.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$CRF_INS_REF

longword-integer-reference
OpenVMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by reference

The 16-bit value used in selecting the contents of the REF1 field. The longword-integer-reference argument is the address of a signed longword integer containing this value. When preparing the output line, LIB$CRF_OUTPUT uses longword-integer-reference and the bit mask in the field descriptor table to extract the data. The high-order bit of the word is reserved for LIB$CRF_INS_REF.

ref-definition-indicator
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Reference/definition indicator that LIB$CRF_INS_REF uses to distinguish between a reference to a symbol and the definition of the symbol. The ref-definition-indicator argument is the address of a signed longword integer containing this indicator. The only difference between processing a symbol reference and a symbol definition is where LIB$CRF_INS_REF stores the information.

The reference/definition indicator can have either of the following values:

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRF$K_REF</td>
<td>Reference to a symbol</td>
</tr>
<tr>
<td>CRF$K_DEF</td>
<td>Definition of a symbol</td>
</tr>
</tbody>
</table>

Description
LIB$CRF_INS_REF inserts a reference to a key in the cross-reference symbol table. If you attempt to insert reference information for a key that was not specified in a call to LIB$CRF_INS_KEY, LIB$CRF_INS_REF uses the address of the key to locate the symbol name and set the KEY1 field. Once set, either as a result of LIB$CRF_INS_KEY or LIB$CRF_INS_REF, the KEY1 field is never changed. A KEY1 field set by LIB$CRF_INS_REF has a space-filled VAL1 field associated with it unless it is overridden by a subsequent call to LIB$CRF_INS_KEY.

Using LIB$CRF_INS_REF involves the following steps:
1. Define a table of control information using the $CRFCTLTABLE macro.
2. Define each field of the output line using the $CRFFIELD macro.
3. Using the $CRFFIELDEND macro, specify the end of each set of macros that define a field in the output line.
4. Provide data by calling LIB$CRF_INS_REF to insert a reference to a key in the specified symbol table. This data is used to build tables in virtual memory.

5. Call LIB$CRF_OUTPUT, the cross-reference output routine, to summarize and format the data. Supply a routine that LIB$CRF_OUTPUT calls to print each line in the output file. Because you supply this routine, you can control the number of lines per page and the header lines.

**Condition Values Returned**

None.
LIB$CRF_OUTPUT
Output Cross-Reference Table Information

The Output Cross-Reference Table Information routine extracts the information from the cross-reference tables and formats the output pages. †

Format

LIB$CRF_OUTPUT control-table ,output-line-width ,page1 ,page2 ,mode-indicator ,delete-save-indicator

Returns

None.

Arguments

control-table
OpenVMS usage: vector_longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference, array reference

Control table associated with the cross-reference. The control-table argument is the address of an array containing the control table. The table contains the address of the user-supplied routine that prints the lines formatted by LIB$CRF_OUTPUT.

output-line-width
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Width of the output line. The output-line-width argument is the address of a signed longword integer containing the width.

page1
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Number of lines on the first page of the output. The page1 argument is the address of a signed longword integer containing this number. This allows the user to reserve space to print header information on the first page of the cross-reference.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
**page2**

OpenVMS usage: longword_signed  
**type:** longword integer (signed)  
**access:** read only  
**mechanism:** by reference

Number of lines per page for the other pages. The `page2` argument is the address of a signed longword integer containing this number.

**mode-indicator**

OpenVMS usage: longword_signed  
**type:** longword integer (signed)  
**access:** read only  
**mechanism:** by reference

Output mode indicator. The `mode-indicator` argument is the address of a signed longword integer containing the mode indicator.

This indicator allows the user to select which of three output modes is desired.

<table>
<thead>
<tr>
<th>Output Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRF$K_VALUES</td>
<td>Only the value and key fields are to be printed. LIB$CRF_OUTPUT creates multiple columns across the page. Each column consists of the KEY1, KEY2, VAL1, and VAL2 fields. A minimum of one space between each column is guaranteed.</td>
</tr>
<tr>
<td>CRF$K_VALS_REFS</td>
<td>Requests a cross-reference summary that has no column space saved for a defining reference. If the user inserted a reference with the CRF$K_DEF indicator, the entry is ignored.</td>
</tr>
<tr>
<td>CRF$K_DEFS_REFS</td>
<td>Requests a cross-reference summary with the first REF1 and REF2 fields used only for definition references. If no definition reference is provided, the fields are filled with spaces.</td>
</tr>
</tbody>
</table>

**delete-save-indicator**

OpenVMS usage: longword_signed  
**type:** longword integer (signed)  
**access:** read only  
**mechanism:** by reference

Delete/save indicator, which LIB$CRF_OUTPUT uses to determine whether the table's built-in accumulating symbol information is to be saved or deleted once the cross-reference is produced. The `delete-save-indicator` argument is the address of a signed longword integer containing the delete/save indicator.

The indicator can be either of the following:

- CRF$K_SAVE To preserve the tables for subsequent processing
- CRF$K_DELETE To delete the tables
Description

LIB$CRF_OUTPUT can format output lines for three types of cross-reference listings:

- A summary of symbol names and their values, as shown in Figure lib–2.
- A summary of symbol names, their values, and the names of modules that refer to each symbol, as shown in Figure lib–3.
- A summary of symbol names, their values, the names of the defining modules, and the names of those modules that refer to each symbol, as shown in Figure lib–4.

Figure lib–2 Summary of Symbol Names and Values

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAS$INSTR</td>
<td>00002080-RU</td>
<td>BAS$SCRATCH</td>
<td>00002308-RU</td>
</tr>
<tr>
<td>BAS$IN_D_R</td>
<td>000021F0-RU</td>
<td>BAS$STATUE</td>
<td>00002318-RU</td>
</tr>
<tr>
<td>BAS$IN_F_R</td>
<td>000021E8-RU</td>
<td>BAS$STR_D</td>
<td>000020C0-RU</td>
</tr>
<tr>
<td>BAS$IN_L_R</td>
<td>000021E0-RU</td>
<td>BAS$STR_F</td>
<td>000020B8-RU</td>
</tr>
<tr>
<td>BAS$IN_T_DX</td>
<td>000021F8-RU</td>
<td>BAS$STR_L</td>
<td>000020C8-RU</td>
</tr>
<tr>
<td>BAS$IN_W_R</td>
<td>000021D8-RU</td>
<td>BAS$UNLOCK</td>
<td>00002110-RU</td>
</tr>
<tr>
<td>BAS$IO_END</td>
<td>000021D0-RU</td>
<td>BAS$UPDATE</td>
<td>000022E8-RU</td>
</tr>
<tr>
<td>BAS$LINKAGE</td>
<td>00001674-R</td>
<td>BAS$UPDATE_COUN</td>
<td>000022F0-RU</td>
</tr>
<tr>
<td>BAS$INPUT</td>
<td>000021A8-R</td>
<td>BAS$VAL_D</td>
<td>00002110-RU</td>
</tr>
<tr>
<td>BAS$MAT_INPUT</td>
<td>00002268-R</td>
<td>BAS$VAL_P</td>
<td>00002108-RU</td>
</tr>
</tbody>
</table>

Figure lib–3 Summary of Symbol Names, Values, and Names of Referring Modules

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Referenced By ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAS$DIVBY_ZER</td>
<td>0000003D</td>
<td>ALLGBL, BAS$ERROR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAS$POWDJ, BAS$POWER</td>
</tr>
<tr>
<td>BAS$DUPKEYDET</td>
<td>00000086</td>
<td>ALLGBL, BAS$SIGNAL_IO</td>
</tr>
<tr>
<td>BAS$ENDFILEDEV</td>
<td>0000000B</td>
<td>ALLGBL, BAS$REC_PROC</td>
</tr>
<tr>
<td>BAS$ENDOF_STA</td>
<td>0000006C</td>
<td>ALLGBL</td>
</tr>
</tbody>
</table>
Regardless of the format of the output, LIB$CRF_OUTPUT considers the output line as consisting of six different field types:

- **KEY1**: Is the first field in the line. It contains a symbol name.
- **KEY2**: Is the second field in the line. It contains a set of flags (for example, -R) that provide information about the symbol.
- **VAL1**: Is the third field in the line. It contains the value of the symbol.
- **VAL2**: Is the fourth field in the line. It contains a set of flags describing VAL1.
- **REF1** and **REF2** fields: Within each REF1 and REF2 pair, REF1 provides a set of flags, and REF2 provides the name of a module that references the symbol.

Any of these fields can be omitted from the output.

For example:

```
Symbol    Value    Defined By    Referenced By ...
----------- ----------- -------------------------
LIB$FREE_VM 0001E185-R LIB$VM ALLGBL
            BAS$MARGIN
            BAS$XLATE
            FOR$VM
            STR$APPEND
            STR$DUPL_CHAR
            STR$REPLACE
LIB$GET_COMMAND 0001E2B0-R LIB$GET_INPUT ALLGBL
                ALLGBL
LIB$GET_COMMON 0001E4D6-R LIB$GET_COMMON ALLGBL

ZK–1971–GE
```

**Condition Values Returned**

None.
LIB$CURRENCY
Get System Currency Symbol

The Get System Currency Symbol routine returns the system’s currency symbol.

Format

LIB$CURRENCY currency-string [,resultant-length]

Returns

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

Arguments

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

Currency symbol. The currency-string argument is the address of a descriptor pointing to the currency symbol.

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

Number of characters that LIB$CURRENCY has written into the currency-string argument, not counting padding in the case of a fixed-length string. The resultant-length argument is the address of an unsigned word containing the length of the currency symbol. If the input string is truncated to the size specified in the currency-string argument, resultant-length is set to this size. Therefore, resultant-length can always be used by the calling program to access a valid substring of currency-string.

Description

LIB$CURRENCY attempts to translate the logical name SYS$CURRENCY as a process, group, or system logical name, in that order. If the translation fails, the routine returns the United States currency symbol (\$). If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS$CURRENCY as a systemwide logical name to provide a default for all users, and an individual user with a special need can define SYS$CURRENCY as a process logical name to override the system default.

For example, if you want to use the British pound sign (£) as the currency symbol within your process but you want to leave the dollar sign as the system’s default, define SYS$CURRENCY to be the pound sign in your process logical name table. After this, any call to LIB$CURRENCY within your process returns the pound sign (£), while any call outside your process returns the dollar sign (\$).
Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Successfully completed, but the currency string was truncated.</td>
</tr>
</tbody>
</table>

Example

```
10 !+  
! This BASIC program uses LIB$CURRENCY to  
! return the default system currency symbol. 
!-  
  OUTLEN = 1  
CALL LIB$CURRENCY (CURR$, OUTLEN)  
PRINT CURR$  
99 END  
```

This BASIC program uses LIB$CURRENCY to display the system currency symbol default. The output generated by the program is a dollar sign ($).
LIB$ CVTF\_FROM\_INTERNAL\_TIME

Convert Internal Time to External Time (F-Floating-Point Value)

The Convert Internal Time to External Time (F-Floating-Point Value) routine converts a delta internal OpenVMS system time into an external F-floating time.

**Format**

LIB$ CVTF\_FROM\_INTERNAL\_TIME  operation ,resultant-time,input-time

**Returns**

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

**Arguments**

**operation**

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

The conversion to be performed. The **operation** argument is the address of an unsigned longword specifying the operation. Valid values for operation are the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_DELTA_WEEKS_F</td>
<td>Fractional weeks</td>
</tr>
<tr>
<td>LIB$K_DELTA_DAYS_F</td>
<td>Fractional days</td>
</tr>
<tr>
<td>LIB$K_DELTA_HOURS_F</td>
<td>Fractional hours</td>
</tr>
<tr>
<td>LIB$K_DELTA_MINUTES_F</td>
<td>Fractional minutes</td>
</tr>
<tr>
<td>LIB$K_DELTA_SECONDS_F</td>
<td>Fractional seconds</td>
</tr>
</tbody>
</table>

**resultant-time**

OpenVMS usage: floating_point  
type: F\_floating  
access: write only  
mechanism: by reference

The external time that results from the conversion. The **resultant-time** argument is the address of an F-floating-point value containing the result.
input-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

Delta time to be converted. The input-time argument is the address of an unsigned quadword containing the time.

Description
LIB$CVTF_FROM_INTERNAL_TIME converts a delta internal OpenVMS system time into an external F-floating-point time. The operation argument specifies the conversion. LIB$CVTF_FROM_INTERNAL_TIME converts the value of input-time into one of the external formats listed in the operation argument description. LIB$CVTF_FROM_INTERNAL_TIME then places the result into resultant-time.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_DELTIMREQ</td>
<td>Delta time required but absolute time supplied.</td>
</tr>
<tr>
<td>LIB$_INVOPER</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>LIB$_IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$CVTS_FROM_INTERNAL_TIME (Alpha and I64 Only)

LIB$CVTS_FROM_INTERNAL_TIME (Alpha and I64 Only)
Convert Internal Time to External Time (S-Floating-Point Value)

The Convert Internal Time to External Time (IEEE S-Floating-Point Value) routine converts a delta internal OpenVMS system time into an external IEEE S-floating time.

Format

LIB$CVTS_FROM_INTERNAL_TIME operation ,resultant-time ,input-time

Returns

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

Arguments

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

The conversion to be performed. The operation argument is the address of an unsigned longword specifying the operation. Valid values for operation are the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_DELTA_WEEKS_F</td>
<td>Fractional weeks</td>
</tr>
<tr>
<td>LIB$K_DELTA_DAYS_F</td>
<td>Fractional days</td>
</tr>
<tr>
<td>LIB$K_DELTA_HOURS_F</td>
<td>Fractional hours</td>
</tr>
<tr>
<td>LIB$K_DELTA_MINUTES_F</td>
<td>Fractional minutes</td>
</tr>
<tr>
<td>LIB$K_DELTA_SECONDS_F</td>
<td>Fractional seconds</td>
</tr>
</tbody>
</table>

resultant-time
OpenVMS usage: floating_point
type: IEEE S_floating
access: write only
mechanism: by reference

The external time that results from the conversion. The resultant-time argument is the address of an IEEE S-floating-point value containing the result.
LIB$ Routines
LIB$CVTS_FROM_INTERNAL_TIME (Alpha and I64 Only)

**input-time**
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

Delta time to be converted. The *input-time* argument is the address of an unsigned quadword containing the time.

**Description**
LIB$CVTS_FROM_INTERNAL_TIME converts a delta internal OpenVMS system time into an external IEEE S-floating-point time. The *operation* argument specifies the conversion. LIB$CVTS_FROM_INTERNAL_TIME converts the value of *input-time* into one of the external formats listed in the *operation* argument description. LIB$CVTS_FROM_INTERNAL_TIME then places the result into *resultant-time*.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_DELTIMREQ</td>
<td>Delta time required but absolute time supplied.</td>
</tr>
<tr>
<td>LIB$_INVOPER</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>LIB$_IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
LIB$ CVTF_TO_INTERNAL_TIME
Convert External Time to Internal Time (F-Floating-Point Value)

The Convert External Time to Internal Time (F-Floating-Point Value) routine converts an external time interval into an OpenVMS internal format F-floating delta time.

Format

LIB$ CVTF_TO_INTERNAL_TIME  operation , input-time , resultant-time

Returns

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

Arguments

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

The conversion to be performed. The operation argument is the address of an unsigned longword specifying the operation. Valid values for operation are the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K DELTA_WEEKS_F</td>
<td>Fractional weeks</td>
</tr>
<tr>
<td>LIB$K DELTA_DAYS_F</td>
<td>Fractional days</td>
</tr>
<tr>
<td>LIB$K DELTA_HOURS_F</td>
<td>Fractional hours</td>
</tr>
<tr>
<td>LIB$K DELTA_MINUTES_F</td>
<td>Fractional minutes</td>
</tr>
<tr>
<td>LIB$K DELTA_SECONDS_F</td>
<td>Fractional seconds</td>
</tr>
</tbody>
</table>

input-time
OpenVMS usage: varying_arg
type: F_floating
access: read only
mechanism: by reference

Delta time to be converted. The input-time argument is the address of this input time. The value you supply for input-time must be greater than 0.
resultant-time
OpenVMS usage:  date_time
type: quadword (unsigned)
access: write only
mechanism: by reference

The OpenVMS internal format delta time that results from the conversion. The resultant-time argument is the address of an unsigned quadword containing the result.

Description
LIB$CVTF_TO_INTERNAL_TIME converts an external time interval, such as 3.5 weeks, into an OpenVMS internal format F-floating delta time. The operation argument specifies the conversion. LIB$CVTF_TO_INTERNAL_TIME converts the value of input-time into one of the internal format delta times listed in the operation argument description. LIB$CVTF_TO_INTERNAL_TIME then places the result into resultant-time.

Condition Values Returned

LIB$_NORMAL     Routine successfully completed.
LIB$_INVOPER    Invalid operation.
LIB$_IVTIME     Invalid time.
LIB$_WRONUMARG  Incorrect number of arguments.
LIB$ Routines
LIB$CVTS_TO_INTERNAL_TIME (Alpha and I64 Only)

LIB$CVTS_TO_INTERNAL_TIME (Alpha and I64 Only)
Convert External Time to Internal Time (S-Floating-Point Value)

The Convert External Time to Internal Time (IEEE S-Floating-Point Value) routine converts an external time interval into an OpenVMS internal format IEEE S-floating delta time.

Format

LIB$CVTS_TO_INTERNAL_TIME operation,input-time,resultant-time

Returns

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

Arguments

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

The conversion to be performed. The operation argument is the address of an unsigned longword specifying the operation. Valid values for operation are the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_DELTA_WEEKS_F</td>
<td>Fractional weeks</td>
</tr>
<tr>
<td>LIB$K_DELTA_DAYS_F</td>
<td>Fractional days</td>
</tr>
<tr>
<td>LIB$K_DELTA_HOURS_F</td>
<td>Fractional hours</td>
</tr>
<tr>
<td>LIB$K_DELTA_MINUTES_F</td>
<td>Fractional minutes</td>
</tr>
<tr>
<td>LIB$K_DELTA_SECONDS_F</td>
<td>Fractional seconds</td>
</tr>
</tbody>
</table>

input-time
OpenVMS usage: varying_arg
type: IEEE S_floating
access: read only
mechanism: by reference

Delta time to be converted. The input-time argument is the address of this input time. The value you supply for input-time must be greater than 0.
LIB$ Routines

LIB$CVTS_TO_INTERNAL_TIME (Alpha and I64 Only)

resultant-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: write only
mechanism: by reference

The OpenVMS internal format delta time that results from the conversion. The resultant-time argument is the address of an unsigned quadword containing the result.

Description
LIB$CVTS_TO_INTERNAL_TIME converts an external time interval, such as 3.5 weeks, into an OpenVMS internal format IEEE S-floating delta time. The operation argument specifies the conversion. LIB$CVTS_TO_INTERNAL_TIME converts the value of input-time into one of the internal format delta times listed in the operation argument description. LIB$CVTS_TO_INTERNAL_TIME then places the result into resultant-time.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVOPER</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>LIB$_IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
The General Data Type Conversion routine converts OpenVMS standard atomic or string data described by a source descriptor to OpenVMS standard atomic or string data described by a destination descriptor. This conversion is supported over a subset of the OpenVMS standard data types.

Format

LIB$CVT_DX_DX source-item , destination-item [, word-integer-dest-length]

Returns

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

Arguments

source-item
OpenVMS usage: unspecified
type: unspecified
access: read only
mechanism: by descriptor

Source item to be converted by LIB$CVT_DX_DX. The source-item argument is the address of a descriptor pointing to the source item to be converted. The type of the item to be converted is contained in the descriptor.

The combination of source descriptor class and data type is restricted as described in Table lib–1 and Table lib–2.

destination-item
OpenVMS usage: unspecified
type: unspecified
access: write only
mechanism: by descriptor

Destination of the conversion. The destination-item argument is the address of a descriptor pointing to the destination item. The destination descriptor specifies the data type to which the source item is converted.

The combination of destination descriptor class and data type is restricted as described in Table lib–1 and Table lib–2.

word-integer-dest-length
OpenVMS usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Length in bytes of the destination item (when that item is a string) that has been converted by LIB$CVT_DX_DX, not including any space filling. The word-integer-dest-length argument contains the address of an unsigned word containing this length.
If the destination string is truncated, the returned length reflects the truncation. This word can be used by the calling program to determine if truncation has occurred or to extract the exact length of the string when the string contains space filling.

Description

LIB$CVT_DX_DX is a universal conversion utility routine. Table lib–1 shows the complete matrix of data type and descriptor class combinations (as specified in the fields of the descriptor) supported by LIB$CVT_DX_DX.

Conversion is defined over three sets of data types: atomic, string, and numeric byte data strings. Although some of the functions of this routine may be found in other Run-Time Library routines, LIB$CVT_DX_DX packages the conversion functions with a general interface. Because of this general interface, the calling program does not have to specify what conversion should be done for which data type.

Refer to LIB$CVT_xtB if you want to convert the ASCII text string representation of a decimal, hexadecimal, or octal number into a binary representation.

The description of this routine has been divided into the following parts:

- Guidelines for Using LIB$CVT_DX_DX
- Use of Numeric Byte Data Strings (NBDS)

For more information about numeric byte data strings, see the section called Use of Numeric Byte Data Strings (NBDS). Although the set of data types in NBDS is actually a subset of the atomic and string data types, the three sets are mutually exclusive in this routine. For more information on the OpenVMS atomic and string data types and the argument descriptor classes supported by this routine, see the HP OpenVMS Calling Standard manual.
## Table lib–1  OpenVMS Descriptor Class and Data Type Combinations Accepted by LIB$CVT_DX_DX

<table>
<thead>
<tr>
<th>DSC$K_ DTYPETYPE_yyy</th>
<th>A</th>
<th>D</th>
<th>NCA</th>
<th>S</th>
<th>SD</th>
<th>VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BU</td>
<td>NBDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NL</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLO</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NR</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NRO</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NU</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NZ</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td>NBDS</td>
<td>NBDS</td>
<td>NBDS</td>
</tr>
<tr>
<td>VT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NBDS</td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WU</td>
<td></td>
<td></td>
<td></td>
<td>Non-NBDS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Invalid combinations are blank. Any source data can be converted into any other destination data as long as they are both represented by one of the valid combinations.

**Note:** LIB$CVT_DX_DX treats an array, described by a CLASS_A or CLASS_NCA descriptor, as a character string. NBDS must have the format defined in Table lib–2.

### Guidelines for Using LIB$CVT_DX_DX

The data type and descriptor class of the source and destination arguments determine how LIB$CVT_DX_DX performs the conversion, according to the following rules:

- Scale is applied when indicated in the descriptor (descriptor CLASS_SD only), and scaling is defined for the data type.

- No language-specific semantics are applied, such as BASIC scale for DSC$K_ DTYPETYPE_D.

- Some conversions must use intermediate values to arrive at the destination requested. Although some loss of speed is inevitable, intermediate values will not cause a loss of precision.
• Results are always rounded instead of truncated, except for the case described below. Note that loss of precision or range may be inherent in the destination data type or in the NBDS destination size. No errors are reported if there is a loss of precision or range as a result of destination data type.

• When the destination is an NBDS and has fixed-string semantics, then if the source does not fill the destination, the destination is padded with blanks.

• When the source and destination are both NBDS and no scaling is requested, then a straight copy is done without translation or conversion, and truncation is possible. If scaling is requested, then a conversion takes place as defined in Table lib–2.

• When the source is an NBDS and the destination is non-NBDS, if there is an invalid character in the source or the value is outside the range that can be represented by the destination, then LIB$_INVNBDS is returned.

• Attempts to convert a negative value to an unsigned data type cause the LIB$_INVCVT error to be returned.

• If the destination is an NBDS of descriptor CLASS_D, then a new string of appropriate size is allocated for it, if necessary.

• Invalid conversions resulting in an error produce an unpredictable result.

Use of Numeric Byte Data Strings (NBDS)

For simplicity, and to define a generic numeric string that LIB$CVT_DX_DX understands to be a numeric string, the set Numeric Byte Data String (NBDS) is defined to be the set of NBDS descriptors shown in Table lib–1.

The combination of data type and descriptor class determines whether an argument is an NBDS. For example, LIB$CVT_DX_DX treats the combination DSC$K_DTYPE_B/DSC$K_CLASS_S (unsigned byte scalar) as an atomic data type. However, the routine considers DSC$K_DTYPE_BU/DSC$K_CLASS_NCA (noncontiguous array of unsigned bytes) to be an NBDS.

A destination NBDS is always left-justified.

If a destination NBDS requires more than 50 digits for its format (including the sign, if any), then it is expressed in exponential format.

For a conversion of NBDS to NBDS, this format is used if scaling is requested. Otherwise, a straight copy is done. The format of a source NBDS is the same as the format defined for the input argument inp in OTS$_CVT_T_z, with bits 0, 2, and 4 set in the flags argument. That is, blanks are ignored, underflow causes an error, and tabs are ignored.

Table lib–2 defines the format of a destination NBDS.
### Table lib–2 LIB$CVT_DX_DX Destination NBDS Formats

<table>
<thead>
<tr>
<th>Source Data Type</th>
<th>Destination NBDS Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte integer (signed)</td>
<td>sdigits</td>
</tr>
<tr>
<td>Byte (unsigned)</td>
<td>digits</td>
</tr>
<tr>
<td>Word integer (signed)</td>
<td>sdigits</td>
</tr>
<tr>
<td>Word (unsigned)</td>
<td>digits</td>
</tr>
<tr>
<td>Longword integer (signed)</td>
<td>sdigits</td>
</tr>
<tr>
<td>Longword (unsigned)</td>
<td>digits</td>
</tr>
<tr>
<td>Quadword integer (signed)</td>
<td>sdigits</td>
</tr>
<tr>
<td>D_floating</td>
<td>s0.min(16,w-7)E±nn</td>
</tr>
<tr>
<td>F_floating</td>
<td>s0.min( 7,w-7)E±nn</td>
</tr>
<tr>
<td>G_floating</td>
<td>s0.min(15,w-8)E±nnn</td>
</tr>
<tr>
<td>H_floating</td>
<td>s0.min(33,w-9)E±nnnn</td>
</tr>
<tr>
<td>FS_floating (IEEE)</td>
<td>s0.min(7,w-7)E±nn</td>
</tr>
<tr>
<td>FT_floating (IEEE)</td>
<td>s0.min(15,w-8)E±nnn</td>
</tr>
<tr>
<td>NBDS</td>
<td>s0.min(33,w-9)E±nnnn</td>
</tr>
<tr>
<td>Decimal string</td>
<td>sdigits (as defined by VAX architecture)</td>
</tr>
</tbody>
</table>

#### Key to Destination NBDS Formats

- **digits**: Digits 0 through 9, and a decimal point only if source descriptor specifies the value of the SCALE field as less than 0.
- **w**: Width of destination in bytes.
- **s**: Sign. For positive numbers, the sign is implied.
- **min**: Minimum of two values.

The A and NCA array descriptor classes are supported with the following restrictions:

An array is written with the semantics of a fixed string.

- **DIMCT = 1**: Only one-dimensional arrays are recognized.
- **LENGTH = 1**: The length of each array element must be a byte.
- **ARSIZE ≤ 65,535**: The total size of the array must be less than 65,535 bytes. If ARSIZE = 0, the array has a length of zero.
- **S1 = 1**: The stride of an array passed by a noncontiguous array descriptor must be 1. That is, even if the class of the array's descriptor is noncontiguous array (NCA), the array itself must be contiguous.

For more information about the semantics of writing output strings, see the OpenVMS RTL String Manipulation (STR$) Manual.

If the calling program passes a descriptor to LIB$CVT_DX_DX that does not comply with Table lib–1, one of the following error messages is returned:

- LIB$_INVDTYDSC
- LIB$_INVCLADSC
- LIB$_INVCLADTY
- LIB$_INVBNDSD
<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _DECOVF</td>
<td>Packed decimal overflow error. Severe error.</td>
</tr>
<tr>
<td>LIB$ _FLTDOVF</td>
<td>Floating overflow error. Severe error.</td>
</tr>
<tr>
<td>LIB$ _FLTUND</td>
<td>Floating underflow error. Severe error.</td>
</tr>
<tr>
<td>LIB$ _INTOVF</td>
<td>Integer overflow error. Severe error.</td>
</tr>
<tr>
<td>LIB$ _INVCLADSC</td>
<td>Invalid class in descriptor. This class of descriptor is not supported. Severe error.</td>
</tr>
<tr>
<td>LIB$ _INVCLADTY</td>
<td>Invalid class and data type in descriptor. This class and data type combination is not supported. Severe error.</td>
</tr>
<tr>
<td>LIB$ _INVCVT</td>
<td>If the source value is negative and the destination data type is unsigned, this error is returned.</td>
</tr>
<tr>
<td>LIB$ _INVDTYDSC</td>
<td>Invalid data type in descriptor. This data type is not supported. Severe error.</td>
</tr>
<tr>
<td>LIB$ _INVBDS</td>
<td>Invalid NBDS. There is an invalid character in the input, or the value is outside the range that can be represented by the destination, or the NMDS descriptor is invalid. This error is also signaled when the array size of an NBDS is larger than 65,535 bytes or the array is multi-dimensional.</td>
</tr>
<tr>
<td>LIB$ _OUTSTRTRU</td>
<td>Output string truncated. This is returned only when NBDS is both source and destination and no scaling is requested. The result is truncated.</td>
</tr>
<tr>
<td>LIB$ _ROPRAND</td>
<td>Reserved operand error. Severe error.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$CVT_FROM_INTERNAL_TIME

LIB$CVT_FROM_INTERNAL_TIME
Convert Internal Time to External Time

The Convert Internal Time to External Time routine converts an internal
OpenVMS system time (either absolute or delta) into an external time.

Format

LIB$CVT_FROM_INTERNAL_TIME operation,resultant-time [,input-time]

Returns

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

Arguments

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

The conversion to be performed. The operation argument is the address of an
unsigned longword containing the operation. The following table shows valid
values for operation:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Return Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_MONTH_OF_YEAR</td>
<td>Absolute</td>
<td>1 to 12</td>
</tr>
<tr>
<td>LIB$K_DAY_OF_YEAR</td>
<td>Absolute</td>
<td>1 to 366</td>
</tr>
<tr>
<td>LIB$K_HOUR_OF_YEAR</td>
<td>Absolute</td>
<td>1 to 8784</td>
</tr>
<tr>
<td>LIB$K_MINUTE_OF_YEAR</td>
<td>Absolute</td>
<td>1 to 527,040</td>
</tr>
<tr>
<td>LIB$K_SECOND_OF_YEAR</td>
<td>Absolute</td>
<td>1 to 31,622,400</td>
</tr>
<tr>
<td>LIB$K_DAY_OF_MONTH</td>
<td>Absolute</td>
<td>1 to 31</td>
</tr>
<tr>
<td>LIB$K_HOUR_OF_MONTH</td>
<td>Absolute</td>
<td>1 to 744</td>
</tr>
<tr>
<td>LIB$K_MINUTE_OF_MONTH</td>
<td>Absolute</td>
<td>1 to 44,640</td>
</tr>
<tr>
<td>LIB$K_SECOND_OF_MONTH</td>
<td>Absolute</td>
<td>1 to 2,678,400</td>
</tr>
<tr>
<td>LIB$K_DAY_OF_WEEK</td>
<td>Absolute</td>
<td>1 to 7</td>
</tr>
<tr>
<td>LIB$K_HOUR_OF_WEEK</td>
<td>Absolute</td>
<td>1 to 168</td>
</tr>
<tr>
<td>LIB$K_MINUTE_OF_WEEK</td>
<td>Absolute</td>
<td>1 to 10,080</td>
</tr>
<tr>
<td>LIB$K_SECOND_OF_WEEK</td>
<td>Absolute</td>
<td>1 to 604,800</td>
</tr>
<tr>
<td>LIB$K_HOUR_OF_DAY</td>
<td>Absolute</td>
<td>0 to 23</td>
</tr>
<tr>
<td>LIB$K_MINUTE_OF_DAY</td>
<td>Absolute</td>
<td>0 to 1439</td>
</tr>
</tbody>
</table>

1Day 1 is Monday.
2Hours since midnight on previous Monday.
3Minutes since midnight on previous Monday.
4Seconds since midnight on previous Monday.
LIB$ Routines
LIB$CVT_FROM_INTERNAL_TIME

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Return Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_SECOND_OF_DAY</td>
<td>Absolute</td>
<td>0 to 86,399</td>
</tr>
<tr>
<td>LIB$K_MINUTE_OF_HOUR</td>
<td>Absolute</td>
<td>0 to 59</td>
</tr>
<tr>
<td>LIB$K_SECOND_OF_HOUR</td>
<td>Absolute</td>
<td>0 to 3599</td>
</tr>
<tr>
<td>LIB$K_SECOND_OF_MINUTE</td>
<td>Absolute</td>
<td>0 to 59</td>
</tr>
<tr>
<td>LIB$K_JULIAN_DATE</td>
<td>Absolute</td>
<td>Julian date</td>
</tr>
<tr>
<td>LIB$K_DELTA_WEEKS</td>
<td>Delta 6</td>
<td></td>
</tr>
<tr>
<td>LIB$K_DELTA_DAYS</td>
<td>Delta 7</td>
<td></td>
</tr>
<tr>
<td>LIB$K_DELTA_HOURS</td>
<td>Delta 8</td>
<td></td>
</tr>
<tr>
<td>LIB$K_DELTA_MINUTES</td>
<td>Delta 9</td>
<td></td>
</tr>
<tr>
<td>LIB$K_DELTA SECONDS</td>
<td>Delta 10</td>
<td></td>
</tr>
</tbody>
</table>

5Number of days since system zero time (17–Nov–1858).
6Whole weeks.
7Whole days.
8Whole hours.
9Whole minutes.
10Whole seconds.

resultant-time
OpenVMS usage: longword_unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

The external time that results from the conversion. The resultant-time argument is the address of an unsigned longword containing the result.

input-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

Optional absolute or delta time to be converted. The input-time argument is the address of an unsigned quadword containing the time. If you do not supply a value for input-time, the current system time is used.

Description

LIB$CVT_FROM_INTERNAL_TIME converts an internal OpenVMS system time (either absolute or delta) into an external time. The operation argument specifies the conversion. LIB$CVT_FROM_INTERNAL_TIME converts the value of input-time (or the current system time if input-time is not supplied) into one of the external formats listed in the operation argument description. LIB$CVT_FROM_INTERNAL_TIME then places the result into resultant-time.

See the HP OpenVMS Programming Concepts Manual for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.
### Condition Values Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _ABSTIMREQ</td>
<td>Absolute time required but delta time supplied.</td>
</tr>
<tr>
<td>LIB$ _DELTIMREQ</td>
<td>Delta time required but absolute time supplied.</td>
</tr>
<tr>
<td>LIB$ _INVOPER</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>LIB$ _IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$ _WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
The Convert External Time to Internal Time routine converts an external time interval into an OpenVMS internal format delta time.

**Format**

```
LIB$CVT_TO_INTERNAL_TIME operation ,input-time ,resultant-time
```

**Returns**

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

**Arguments**

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

The conversion to be performed. The **operation** argument is the address of an unsigned longword specifying the operation. Valid values for **operation** are the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_DELTA_WEEKS</td>
<td>Whole weeks in delta time</td>
</tr>
<tr>
<td>LIB$K_DELTA_DAYS</td>
<td>Whole days in delta time</td>
</tr>
<tr>
<td>LIB$K_DELTA_HOURS</td>
<td>Whole hours in delta time</td>
</tr>
<tr>
<td>LIB$K_DELTA_MINUTES</td>
<td>Whole minutes in delta time</td>
</tr>
<tr>
<td>LIB$K_DELTA_SECONDS</td>
<td>Whole seconds in delta time</td>
</tr>
</tbody>
</table>

**input-time**  
OpenVMS usage: varying_arg  
type: longword (signed)  
access: read only  
mechanism: by reference

Delta time to be converted. The **input-time** argument is the address of this input time. The value you supply for **input-time** must be greater than 0.
LIB$ CVT_TO_INTERNAL_TIME

resultant-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: write only
mechanism: by reference

The OpenVMS internal format delta time that results from the conversion. The resultant-time argument is the address of an unsigned quadword containing the result.

Description

LIB$CVT_TO_INTERNAL_TIME converts an external time interval, such as three weeks, into an OpenVMS internal format delta time. The operation argument specifies the conversion. LIB$_CVT_TO_INTERNAL_TIME converts the value of input-time into one of the internal format delta times listed in the operation argument description. LIB$_CVT_TO_INTERNAL_TIME then places the result into resultant-time.

See the HP OpenVMS Programming Concepts Manual for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.

Condition Values Returned

LIB$_NORMAL Routine successfully completed.
LIB$_INVOPER Invalid operation.
LIB$_IVTIME Invalid time.
LIB$_WRONUMARG Incorrect number of arguments.
LIB$CVT_VECTIM
Convert 7-Word Vector to Internal Time

The Convert 7-Word Vector to Internal Time routine converts a 7-word vector into an OpenVMS internal format delta or absolute time.

Format

LIB$CVT_VECTIM  input-time ,resultant-time

Returns

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

Arguments

input-time
OpenVMS usage:  vector_word_unsigned
type:            word (unsigned)
access:          read only
mechanism:       by reference, array reference

Time to be converted. The input-time argument is the address of a 7-word structure containing this time. This vector directly corresponds to a $NUMTIM $numbuf structure. The following diagram depicts the fields in this structure:

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month of Year</td>
<td>Year Since 0</td>
<td></td>
</tr>
<tr>
<td>Hour of Day</td>
<td>Day of Month</td>
<td></td>
</tr>
<tr>
<td>Second of Minute</td>
<td>Minute of Hour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hundredths of Second</td>
<td></td>
</tr>
</tbody>
</table>

ZK–7968–GE

The input-time argument can represent an absolute or a delta time. In order for input-time to represent a delta time, the year since 0 and month of year fields must equal zero. If those fields do not equal zero, an absolute time is returned.

resultant-time
OpenVMS usage:  date_time
type:            quadword (unsigned)
access:          write only
mechanism:       by reference

The OpenVMS internal format delta or absolute time that results from the conversion. The resultant-time argument is the address of an unsigned quadword containing the result.
LIB$ Routines
LIB$CVT_VECTIM

Description
LIB$CVT_VECTIM converts a 7-word vector (in the format output by the
$NUMTIM system service) into an OpenVMS internal format delta or absolute
time. LIB$CVT_VECTIM then places the result into **resultant-time**.

See the *HP OpenVMS System Services Reference Manual: GETUTC-Z* for more
information about $NUMTIM.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
LIB$CVT_xTB
Convert Numeric Text to Binary

The Convert Numeric Text to Binary routines return a binary representation of the ASCII text string representation of a decimal, hexadecimal, or octal number.

Format

LIB$CVT_DTB  byte-count ,numeric-string ,result
LIB$CVT_HTB  byte-count ,numeric-string ,result
LIB$CVT_OTB  byte-count ,numeric-string ,result

Returns

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

Arguments

byte-count
OpenVMS usage:  longword_signed
type:  longword integer (signed)
access: read only
mechanism: by value

Byte count of the input ASCII text string. The byte-count argument is a signed longword integer containing the byte count of the input string.

numeric-string
OpenVMS usage:  char_string
type:  character string
access: read only
mechanism: by reference

ASCII text string representation of a decimal, hexadecimal, or octal number that LIB$CVT_xTB converts to binary representation. The numeric-string argument is the address of a character string containing this input string to be converted.

The syntax of a valid ASCII text input string is as follows:

```
  [ +  -  <radix-characters> ]
```

LIB$CVT_xTB allows only an optional plus (+) or minus (–) sign followed by a string of decimal, hexadecimal, or octal characters appropriate to the routine being called.

result
OpenVMS usage:  longword_signed
type:  longword integer (signed)
access: write only
mechanism: by reference
Binary representation of the input string. The **result** argument is the address of a signed longword integer containing the converted string.

**Description**

LIB$CVT_DTB converts the ASCII text string representation of a decimal number into binary representation. LIB$CVT_HTB converts the ASCII text string representation of a hexadecimal number into binary representation. LIB$CVT_OTB converts the ASCII text string representation of an octal number into binary representation.

**Note**

LIB$CVT_DTB, LIB$CVT_HTB, and LIB$CVT_OTB are intended to be called primarily from BLISS and MACRO programs. Therefore, the routines expect input scalar arguments to be passed by value and strings by reference.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>0</td>
<td>Nonradix character in the input string or a sign in any position other than the first character. An overflow from 32 bits (unsigned) causes an error.</td>
</tr>
</tbody>
</table>
LIB$CVT_xTB_64 (Alpha and I64 Only)
Convert Numeric Text to Binary

The Convert Numeric Text to Binary routines return a binary representation of the ASCII text string representation of a decimal, hexadecimal, or octal number.

Format

LIB$CVT_DTB_64 byte-count ,numeric-string ,result
LIB$CVT_HTB_64 byte-count ,numeric-string ,result
LIB$CVT_OTB_64 byte-count ,numeric-string ,result

Returns

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

Arguments

byte-count
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by value

Byte count of the input ASCII text string. The byte-count argument is a signed longword integer containing the byte count of the input string.

numeric-string
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by reference

ASCII text string representation of a decimal, hexadecimal, or octal number that LIB$CVT_xTB_64 converts to binary representation. The numeric-string argument is the address of a character string containing this input string to be converted.

The syntax of a valid ASCII text input string is as follows:

\[
\begin{array}{c}
+ \\
- \\
\end{array} <\text{radix-characters}>
\]

LIB$CVT_xTB_64 allows only an optional plus (+) or minus (−) sign followed by a string of decimal, hexadecimal, or octal characters appropriate to the routine being called.
LIB$ Routines
LIB$CVT_xTB_64 (Alpha and I64 Only)

**result**
- OpenVMS usage: quadword_signed
- type: quadword integer (signed)
- access: write only
- mechanism: by reference

Binary representation of the input string. The **result** argument is the address of a signed quadword integer containing the converted string.

**Description**

LIB$CVT_DTB_64 converts the ASCII text string representation of a decimal number into binary representation. LIB$CVT_HTB_64 converts the ASCII text string representation of a hexadecimal number into binary representation. LIB$CVT_OTB_64 converts the ASCII text string representation of an octal number into binary representation.

**Note**

LIB$CVT_DTB_64, LIB$CVT_HTB_64, and LIB$CVT_OTB_64 are intended to be called primarily from BLISS and MACRO programs. Therefore, the routines expect input scalar arguments to be passed by value and strings by reference.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>0</td>
<td>Nonradix character in the input string or a sign in any position other than the first character. An overflow from 64 bits (unsigned) causes an error.</td>
</tr>
</tbody>
</table>
LIB$DATE_TIME
Date and Time Returned as a String

The Date and Time Returned as a String routine returns the OpenVMS system date and time in the semantics of a user-provided string.

Format

```
LIB$DATE_TIME  date-time-string
```

Returns

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

Argument

**date-time-string**
OpenVMS usage: time_name
type: character string
access: write only
mechanism: by descriptor

Destination string into which LIB$DATE_TIME writes the system date and time. The **date-time-string** argument is the address of a descriptor pointing to the destination string. This string is 23 characters long; its format is as follows:

```
dd-mmm-yyyy hh:mm:ss.hh
```

See the *HP OpenVMS Programming Concepts Manual* for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.

Condition Values Returned

- **SS$_NORMAL** — Routine successfully completed.
- **LIB$_STRTRU** — Success, but destination string was truncated.
- **LIB$_INSVIRMEM** — Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.
- **LIB$_INVSTRDES** — Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.
LIB$ ROUTINES
LIB$DATE_TIME

Example

```
10 !+  ! This BASIC program demonstrates the use of LIB$DATE_TIME.
1-     CALL LIB$DATE_TIME(DSTSTR$)
          PRINT DSTSTR$
99 END
```

This BASIC program uses LIB$DATE_TIME to display the current system date and time. The output generated by one run of this program follows:

**LIB$DAY**  
**Day Number Returned as a Longword Integer**

The Day Number Returned as a Longword Integer routine returns the number of days since the system zero date of November 17, 1858, or the number of days from November 17, 1858, to a user-supplied date.

**Format**

```
LIB$DAY  number-of-days [,user-time] [,day-time]
```

**Returns**

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

**Arguments**

**number-of-days**  
OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: write only  
mechanism: by reference

Number of days since the system zero date. The `number-of-days` argument is the address of a signed longword integer containing the day number.

**user-time**  
OpenVMS usage: date_time  
type: quadword (unsigned)  
access: read only  
mechanism: by reference

User-supplied time, in 100-nanosecond units. The `user-time` argument is the address of a signed quadword integer containing the user time. A positive value indicates an absolute time, while a negative value indicates a delta time. This is an optional argument. If `user-time` is omitted, the default is the current system time. This quadword time value is obtained by calling the $BINTIM system service.

If time is passed as zero by value, the numeric value for the current day is returned. If time is passed as a zero by reference, the number returned represents the day of November 17, 1858, rather than the current day.

**day-time**  
OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: write only  
mechanism: by reference

Number of 10-millisecond units since midnight of the `user-time` argument. The `day-time` argument is the address of a signed longword integer into which LIB$DAY writes this number of units.
LIB$ Routines
LIB$DAY

Description
LIB$DAY returns the number of days since the system zero date of November 17, 1858. Optionally, the caller can supply a time in system time format to be used instead of the current system time. In this case, LIB$DAY returns the number of days from November 17, 1858, to the user-supplied date.

The number of 10-millisecond units since midnight is an optional return argument.

Note
If the caller supplies a quadword time, it is not verified. If it is negative (bit 63 on), the number-of-days value returned is negative.

The Run-Time Library provides the date/time utility routines for languages that do not have built-in time and date functions and for particular applications that require the time or date in a different format from the one that the language supplies. In general, it is simpler to call the Run-Time Library routines for the system date and time than to call a system service.

Condition Values Returned

SS$_NORMAL
Routine successfully completed.

SS$_INTOVF
The optional argument user-time is present and represents a date and time well beyond the year 9999.

Example

PROGRAM DAY(INPUT, OUTPUT);
{+}
{ This is a VAX Pascal example program showing
{ the use of LIB$DAY.
{->
VAR
   DAYNUMBER : INTEGER;
routine LIB$DAY(VAR DAYNUM : INTEGER);
   EXTERN;
BEGIN
   LIB$DAY(DAYNUMBER);
   WRITELN(‘The day number is ’, DAYNUMBER);
END.

This Pascal program retrieves and prints the day number. A sample of the output generated by this program is as follows.

   The day number is 46738
LIB$DAY_OF_WEEK
Show Numeric Day of Week

The Show Numeric Day of Week routine returns the numeric day of the week for an input time value. If 0 is the input time value, the current day of the week is returned. The days are numbered 1 through 7, with Monday as day 1 and Sunday as day 7.

Format

LIB$DAY_OF_WEEK [user-time,] day-number

Returns

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

Arguments

user-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

Time to be translated to a day of the week, or zero. The optional user-time argument is the address of an unsigned quadword containing the value of time. Time must be supplied as an absolute system time. To obtain this time value in proper quadword format, call the $BINTIM system service.

If time is passed as zero by value, the numeric value for the current day is returned. If time is passed as a zero by reference, the number returned represents the day of November 17, 1858. If the user-time argument is omitted, it is equivalent to passing a zero by value.

day-number
OpenVMS usage: longword_unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

Numeric day of week. The day-number argument is the address of a longword into which LIB$DAY_OF_WEEK writes the integer value representing the day of the week.

Condition Values Returned

SS$_NORMAL           Routine successfully completed.
**Example**

```pascal
PROGRAM DAYOFWEEK(INPUT, OUTPUT);

{+}
{ This is an example showing
{ the use of LIB$DAY_OF_WEEK.
{-}

VAR

OUTDAT : INTEGER;

routine LIB$DAY_OF_WEEK(TIM : INTEGER; %REF OUTDA : INTEGER); EXTERN;
BEGIN

LIB$DAY_OF_WEEK(%IMMED 0, OUTDAT);
WRITELN(OUTDAT);
END.

This Pascal program shows the use of LIB$DAY_OF_WEEK. This example was
tested on a Monday, and the output generated was 1.
```
LIB$DECODE_FAULT
Decode Instruction Stream During Fault

The Decode Instruction Stream During Fault routine is a tool for building condition handlers that process instruction fault exceptions. It is called from a condition handler. †

This routine is not available to native OpenVMS Alpha and I64 programs but is available to translated VAX images.

Format

LIB$DECODE_FAULT signal-arguments ,mechanism-arguments ,user-procedure
[,unspecified-user-argument] [,instruction-definitions]

Returns

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

Arguments

signal-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: read only
mechanism: by reference, array reference

Signal arguments array that was passed from the OpenVMS operating system to your condition handler. The signal-arguments argument is the address of the signal arguments array.

mechanism-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: read only
mechanism: by reference, array reference

Mechanism arguments array that was passed from OpenVMS to your condition handler. The mechanism-arguments argument is the address of the mechanism arguments array.

user-procedure
OpenVMS usage: procedure
type: procedure value
access: call after stack unwind
mechanism: by descriptor, procedure descriptor

User-supplied action routine that LIB$DECODE_FAULT calls to handle the exception. The user-procedure argument is the address of a descriptor pointing to your user action routine. The user-procedure argument may be of type...

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
“procedure value” when called by languages with up-level addressing. If user-procedure is not of type “bound routine value,” it is assumed to be the address of an entry mask.

For further information on the user action routine, see the section called Call Format for a User Action Routine in the Description section.

unspecified-user-argument
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: by value

Additional information passed from your handler without interpretation to your user action routine. The unspecified-user-argument argument contains the value of this additional information. The unspecified-user-argument argument is optional; if it is omitted, zero is used as the default.

instruction-definitions
OpenVMS usage: vector_byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference, array reference

Array of bytes specifying instruction opcodes and operand definitions that are to replace or supplement the standard instruction definitions. The instruction-definitions argument is the address of this array.

If instruction-definitions is omitted, only the standard instruction definitions are used. If supplied, instruction-definitions is searched first, followed by the standard definitions.

Each instruction definition consists of a series of bytes, the first one or two of which is the instruction opcode. If the instruction is a 2-byte opcode, the escape byte, which must be hex FD, FE, or FF, is placed in the first of the two bytes. Following the opcode may be from 0 to 16 operand definition bytes. These bytes indicate the operand’s access type and data type.

The end of each instruction definition is denoted by a byte containing the value LIB$K_DCFOPR_END (zero). The list of instruction definitions is terminated by two bytes, each of which contains the value –1 (hexadecimal FF). For further information, see the section called Instruction Operand Definition Codes in the Description section.

Description
The Description section of the LIB$DECODE_FAULT routine is divided into the following parts:

- Guidelines for Using LIB$DECODE_FAULT
- Exceptions Recognized by LIB$DECODE_FAULT
- Instruction Operand Definition Codes
- Call Format for a User Action Routine
- Call Format for a Signal Routine
Guidelines for Using LIB$DECODE_FAULT

LIB$DECODE_FAULT is a tool for building condition handlers that process instruction fault exceptions. Called from a condition handler, LIB$DECODE_FAULT performs the following actions:

1. Unwinds intermediate stack frames back to that of the exception
2. Decodes the instruction stream to determine the operation and its operands
3. Calls a user-supplied action routine and passes it a consistent and easy-to-access description of the instruction's context

Your user action routine performs whatever tasks are necessary to handle the fault and returns to LIB$DECODE_FAULT. LIB$DECODE_FAULT then restores the context as modified by your user action routine and continues execution.

Your condition handler must first decide whether or not it wants to handle the exception. The signal arguments list contains the exception code and the address of the program context (PC) that is usually sufficient for this determination. Once LIB$DECODE_FAULT is called, if the exception is a fault LIB$DECODE_FAULT can analyze, control does not return to the condition handler. Therefore, your handler must not depend on regaining control by a routine return once it has called LIB$DECODE_FAULT. With your user action routine, LIB$DECODE_FAULT makes the original fault disappear.

---

**Note**

Your user action routine is capable of generating a new exception, including one that looks identical to the original exception. Your user action routine may also resignal, but if the decision to resignal is made inside the user action routine, all post-signal stack frames are lost.

---

Once your condition handler has decided that it wants to handle the exception, it calls LIB$DECODE_FAULT, passing as arguments the addresses of the signal and mechanism argument lists and a descriptor for your user action routine entry point. LIB$DECODE_FAULT then performs the following actions:

1. Determines if the exception is a fault it understands. If not, it returns SS$_RESIGNAL.
2. Determines the context in which the exception occurred, including register and processor status longword (PSL) contents, and saves it.
3. Unwinds all stack frames back to that frame in which the exception occurred.
4. Evaluates each operand’s addressing mode, computing the resulting location for the operand. Immediate mode operands are expanded into their full form. If an invalid addressing mode is found, an SS$_RADRMOD exception is generated.
5. Unless the original exception was SS$_ACCVIO, tests each operand for accessibility. If necessary, an access violation is signaled as if the instruction had tried to execute normally. See the paragraph following this list for more information.
6. Unless the original exception was SS$_ROPRAND, tests each floating-point operand that is to be read for a reserved floating operand. If necessary, a reserved operand fault is signaled. See the paragraph following this list for more information.
7. Determines the address of the next sequential instruction.

8. Calls your user action routine with arguments as described below.

9. Upon return from your user action routine, reflects changes to the registers and PSL and continues execution at the instruction address specified by your user action routine. Optionally, your user action routine may resignal the original exception.

Some instructions can generate more than one fault if evaluation of one operand causes a fault that occurs before a later operand (which would also cause a fault). An example of this is the possibility that a floating-point divide instruction might report a divide-by-zero fault upon seeing a zero divisor before noticing that the dividend was a reserved operand or was inaccessible.

In these cases, operand-specific faults are signaled immediately by LIB$DECODE_FAULT in the expectation that another condition handler (or the same one) can repair the situation. This may reorder the sequence of exceptions as seen by a program. If the operand exception is corrected, the original exception reoccurs, and the proper action is taken.

If at all possible, try to determine if a resignal is necessary inside the condition handler that calls LIB$DECODE_FAULT, rather than inside your user action routine. The reason for this is that LIB$DECODE_FAULT removes all post-signal stack frames before calling your user action routine.

Your user action routine may fetch and store the operands, registers, and PSL as necessary for handling the exception. You should follow the VAX architecture rule of reading all input operands in left-to-right order, then writing all output operands in left-to-right order, to avoid inconsistent results with overlapping operands. This is especially necessary with register operands.

PSL may be modified in a manner consistent with the VAX architecture. If the T-bit in the PSL was set at the beginning of the instruction, LIB$DECODE_FAULT sets the TP bit. To initiate tracing, you must set only the T bit. To disable tracing, you must clear both the T and TP bits. See the VAX Architecture Reference Manual for more information.

If the first-part-done (FPD) bit in the PSL was set when the instruction faulted, LIB$DECODE_FAULT only advances the PC over the instruction; it does not reevaluate the operands, and it sets operand-count to zero. It is assumed that if FPD is set, the operands are in known locations (typically the registers).

For the CASEB, CASEW, and CASEL instructions, only the selector, base, and limit operands are represented in operand-count and read-operand-locations. The element of registers that corresponds to the PC, described in the following text as R15, points to the first of the word-length displacements. Your user action routine must modify R15 to reflect the location of the next instruction to execute.

The standard instruction definitions used by LIB$DECODE_FAULT specify the XFC instruction (which causes an SS$_OPCCUS fault) as having zero operands. You may redefine XFC if needed using the instruction-definitions argument to LIB$DECODE_FAULT.

If you do not want instruction execution to resume with the next sequential instruction, you must modify R15 appropriately. Your user action routine then returns to LIB$DECODE_FAULT, which restores the registers and PSL, and resumes instruction execution. See also the LIB$_RESTART condition value in the section called Condition Values Returned from the User Action Routine.
LIB$ Routines
LIB$DECODE_FAULT

Note
Vector context is not saved or restored.

Exceptions Recognized by LIB$DECODE_FAULT
LIB$DECODE_FAULT recognizes the following VAX faults:

- SS$_ACCVIO, access violation.
- SS$_BREAK, breakpoint fault.
- SS$_FLTDIV_F, floating divide by zero.
- SS$_FLTOVF_F, floating overflow.
- SS$_FLTUND_F, floating underflow.
- SS$_OPCCUS, opcode reserved to customers.
- SS$_OPCDEC, opcode reserved to HP.
- SS$_ROPRAND, reserved operand.
- SS$_TBIT, T-bit pending trap. This is actually a fault caused by the TP bit being set at the beginning of instruction execution. It allows you to interpret all instructions by setting the PSL T-bit and allowing each instruction to trace-fault.

All other exceptions, including SS$_COMPAT and SS$_RADRMOD, cause LIB$DECODE_FAULT to return immediately with the return status SS$_RESIGNAL.

SS$_COMPAT is generated by compatibility-mode instructions. LIB$DECODE_FAULT does not handle compatibility-mode instructions.

SS$_RADRMOD is generated by a reserved addressing-mode fault. LIB$DECODE_FAULT assumes that all instructions follow VAX addressing-mode specifications.

Instruction Operand Definition Codes
Each instruction operand has an access type (read, write, . . .) and a data type (byte, word, . . .) associated with it. The operand definition codes used in both the instruction-definitions argument passed to LIB$DECODE_FAULT and in the operand-types argument passed to the user action routine encode the access and data types in a byte. The fields and values for operand access and data types are described using the symbols in Table lib–3. These symbols are defined in definition libraries supplied by HP as macro or module name $LIBDCFDEF.
### Table lib–3 Symbols for Fields and Values for Operand Access and Data Types Using LIB$DECODE_FAULT

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$V_DCFACC</td>
<td>The field of the operand description code that describes the operand access type (bits 0–2).</td>
</tr>
<tr>
<td>LIB$S_DCFACC</td>
<td>The size of the access type field (3 bits).</td>
</tr>
<tr>
<td>LIB$M_DCFACC</td>
<td>The mask for the access type field. This is a 3-bit field that can contain any binary value from 000 through 111. The integer value of these bit settings defines the operand access type code for the LIB$M_DCFACC field. Currently, six codes are defined. These codes have symbolic names and are explained below. It is important to remember that LIB$M_DCFACC is not a bit mask. The values 0 through 6 do not refer to bits 0 through 6. They represent the binary values 001 through 110 as contained in the 3-bit field. The operand access type codes defined for the LIB$M_DCFACC field are:</td>
</tr>
<tr>
<td></td>
<td>LIB$K_DCFACC_R = 1 Operand is read-only.</td>
</tr>
<tr>
<td></td>
<td>LIB$K_DCFACC_M = 2 Operand is to be modified.</td>
</tr>
<tr>
<td></td>
<td>LIB$K_DCFACC_W = 3 Operand is write-only.</td>
</tr>
<tr>
<td></td>
<td>LIB$K_DCFACC_A = 4 Operand is an address (must not be a register).</td>
</tr>
<tr>
<td></td>
<td>LIB$K_DCFACC_V = 5 Operand is the base of a bit field (same as address except that it may be a register).</td>
</tr>
<tr>
<td></td>
<td>LIB$K_DCFACC_B = 6 Operand is a branch address.</td>
</tr>
<tr>
<td>LIB$V_DCFTYP</td>
<td>The field of the operand descriptor code that describes the operand data type (bits 3–7).</td>
</tr>
<tr>
<td>LIB$S_DCFTYP</td>
<td>The size of the operand data type field (5 bits).</td>
</tr>
</tbody>
</table>

(continued on next page)
Table lib–3 (Cont.) Symbols for Fields and Values for Operand Access and Data Types Using LIB$DECODE_FAULT

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$M_DCFTYP</td>
<td>The mask for the operand data type field. This is a 5-bit field (bits 3–7) that can contain any binary value from 00000 through 11111. The integer value of these bit settings defines the operand access type code for the LIB$M_DCFACC field. Currently, nine codes are defined. These codes have symbolic names and are explained below. It is important to remember that LIB$M_DCFTYP is not a bit mask. The values 0 through 9 do not refer to bits 0 through 9. They represent the binary values 00001 through 01001 as contained in the 5-bit field. The operand access type codes defined for the LIB$V_DCFTYP field are:</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_B</td>
<td>Operand is a byte.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_W</td>
<td>Operand is a word.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_L</td>
<td>Operand is a longword.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_Q</td>
<td>Operand is a quadword.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_O</td>
<td>Operand is an octaword.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_F</td>
<td>Operand is F_floating.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_D</td>
<td>Operand is D_floating.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_G</td>
<td>Operand is G_floating.</td>
</tr>
<tr>
<td>LIB$K_DCFTYP_H</td>
<td>Operand is H_floating.</td>
</tr>
</tbody>
</table>

Symbols of the form LIB$K_DCFOPR_xy, where x is the access type and y is the data type, are also defined. These combine the notions of access and data type. For example, LIB$K_DCFOPR_MF has the following value:

\[ 50 \times (2^{+}6^{*}8) \]

It denotes modify access of an F_floating item. For the branch access type, only the types BB, BW, and BL are defined; otherwise, all combinations are available.

**Call Format for a User Action Routine**

LIB$DECODE_FAULT calls the user action routine when it finds an exception to be handled. Your user action routine handles the exception in any manner that you specify and then returns to LIB$DECODE_FAULT.

**action-routine** opcode , instr-PC , PSL , registers , operand-count , operand-types , read-operand-locations , write-operand-locations , signal-arguments , signal-procedure , context , unspecified-user-argument , original-registers

**opcode**

OpenVMS usage: longword_unsighed
type: longword (unsigned)
access: read only
mechanism: by reference
LIB$ Routines
LIB$DECODE_FAULT

Opcode of the instruction that caused the fault. The opcode argument is the address of a longword that contains this opcode. LIB$DECODE_FAULT supplies this opcode when it calls the user action routine.

For 2-byte opcodes, the escape code (for example, hex FD) is in the low-order byte. You must use this argument to examine the opcode instead of reading the bytes pointed to by instr-PC. This is because if a debugger breakpoint has been set on the instruction, only opcode contains the original instruction.

**instr-PC**

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

Value of the PC for the instruction that caused the fault. The instr-PC argument is the address of a longword that contains the PC value.

Note the difference between this value and the contents of the registers array element that corresponds to the PC. R15 of the registers array element contains the address of the byte after the instruction that caused the fault.

**PSL**

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

Processor status longword (PSL) at the time of the fault. The PSL argument is the address of a longword that contains this PSL. Your user action routine may modify this PSL within the restrictions of the VAX architecture.

**registers**

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

Contents of registers R0 through R15 (PC) at the time of the fault but after operand addressing-mode processing. This includes any autoincrements or autodecrements. The registers argument is the address of this 16-longword array. Each longword of the registers array contains the contents of one register.

Your user action routine may modify these values. If it does, the new values will be reflected when instruction execution continues.

To modify vector registers, execute a vector instruction. Executing a vector instruction in the handler modifies the state of the vector processor. The state of the vector processor is not restored when the handler returns. This has the effect of altering the state when the execution continues.

R15 denotes the sixteenth longword in the registers array, which corresponds to the PC. R15 contains the address of the next byte after the current instruction. Unless this value is modified by your user action routine, instruction execution will resume at that address. An exception is for the CASEB, CASEW, and CASEL instructions; R15 contains the address of the first displacement word. For these instructions, your user action routine must modify R15 to point to the next instruction to execute.
Upon instruction completion, registers R0-R15 are restored from this array. However, if `signal-procedure` is used to cause a fault or if instruction restart is specified by returning `LIB$_RESTART`, `original-registers` is used instead.

**operand-count**

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

Number of operands in the instruction currently being decoded. The `operand-count` is the address of a longword that contains this number.

**operand-types**

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

Array of longwords, one element for each operand, that contains the type codes for the associated operand. The `operand-types` argument is the address of this array.

The operand type codes are further defined in the section called Instruction Operand Definition Codes.

**read-operand-locations**

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

Array of longwords, one element for each operand, that contains the addresses of the operands to be read. The `read-operand-locations` argument is the address of this array.

The address given in the array may not be the actual address of the operand if the operand is not a memory location. If the operand is a register, the address indicates a copy of the register values at the time of operand evaluation. If the operand access type is ADDRESS or FIELD and the operand is not a register, the address is the address of the item. If the operand access type is FIELD and the operand is a register, the address refers to the appropriate element in the `registers` array. If the operand access type is BRANCH, the address is the destination PC of the branch. For WRITE access operands, the address value is zero.

**write-operand-locations**

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

Array of longwords, one element for each operand, that contains the addresses of operands that are to be written. The `write-operand-locations` argument is the address of this array. If the operand access type is not MODIFY, WRITE, ADDRESS, or FIELD, the pointer value is zero.
**signal-arguments**
OpenVMS usage: vector_longwordUnsigned
Type: longword (unsigned)
Access: read only
Mechanism: by reference, array reference

Signal arguments list of the original exception, as passed from OpenVMS to your condition handler and then to LIB$DECODE_FAULT. The `signal-arguments` argument is the address of an array of longwords that contains these signal arguments.

**signal-procedure**
OpenVMS usage: procedure
Type: procedure value
Access: call without stack unwinding
Mechanism: by reference

Entry mask of a routine that your user action routine must call if it wants to report an exception for the instruction that faulted. The `signal-procedure` argument is the address of this entry mask.

For further information, see the section called Call Format for a Signal Routine.

**context**
OpenVMS usage: context
Type: unspecified
Access: read only
Mechanism: by value

Context in which the exception occurs, including the register and PSL contents, to be used when calling the signal-procedure. The `context` argument contains the value of this context.

**unspecified-user-argument**
OpenVMS usage: user_arg
Type: longword (unsigned)
Access: read only
Mechanism: by value

Optional argument passed to LIB$DECODE_FAULT. If the argument was not specified, the value zero is substituted. The `unspecified-user-argument` argument contains the value of this optional argument.

**original-registers**
OpenVMS usage: vector_longwordUnsigned
Type: longword (unsigned)
Access: modify
Mechanism: by reference, array reference

Array containing the values of registers R0 through R15 (PC) at the time of the fault, before operand processing. The `original-registers` argument is the address of this 16-longword array.

If the action routine specifies that the instruction should restart or that a fault should be generated, the registers are restored from `original-registers`. See also the description of `registers` above.
LIB$ Routines
LIB$DECODE_FAULT

Condition Values Returned from the User Action Routine: The user action routine can return the following condition values to LIB$DECODE_FAULT:

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_CONTINUE</td>
<td>If the user action routine returns a value of SS$_CONTINUE, instruction execution will continue as specified by the current contents of the registers element for the PC.</td>
</tr>
<tr>
<td>SS$_RESIGNAL</td>
<td>If the user action routine returns SS$_RESIGNAL, the original exception is resigaled, with the only changes reflected being those specified by registers elements for R0 and R1 (which are stored in the mechanism arguments vector), PC, and PSL. All other registers are restored from original registers.</td>
</tr>
<tr>
<td>LIB$_RESTART</td>
<td>If the user action routine returns LIB$_RESTART, the current instruction is restarted with registers restored from original-registers and a PSL from PSL. This feature is useful for writing trace handlers.</td>
</tr>
</tbody>
</table>

Call Format for a Signal Routine
Your action routine calls the signal routine using this format:

```
signal-procedure fault-flag ,context ,signal-arguments
```

**fault-flag**
OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by reference

Longword flag whose low-order bit determines whether the exception is to be signaled as a fault or as a trap. The fault-flag argument contains the address of this longword.

If the low-order bit of fault-flag is set to 1, the exception is signaled as a fault. If the low-order bit of fault-flag is set to 0, the exception is signaled as a trap; the current contents of the registers array are used. In either case, the current contents of PSL are used to set the exception PSL.

**context**
OpenVMS usage: context
type: unspecified
access: read only
mechanism: by reference

Context in which the new exception is to occur, as passed to your user action routine by LIB$DECODE_FAULT. The context argument is the address of this context value.

**signal-arguments**
OpenVMS usage: arg_list
type: longword (unsigned)
access: read only
mechanism: by reference, array reference
LIB$ Routines
LIB$DECODE_FAULT

Signal arguments to be used. The signal-arguments argument is the address of an array of longwords that contains these signal arguments.

The first longword contains the number of following longwords; the remainder of the list contains signal names and arguments. Unlike the signal argument list passed to a condition handler, no PC or PSL is present.

Before the exception is signaled, the stack frames are unwound back to the original exception. You should be careful when causing a new signal that a loop of faults is not inadvertently generated. For example, the condition handler that called LIB$DECODE_FAULT will usually be called for the second signal. If the handler does not analyze the second signal as such, it may cycle through the identical path as for the first signal.

To resignal the current exception, have the user action routine return a value of SS$_RESIGNAL instead of calling the signal routine (unless you want previously called condition handlers to be called again).

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_RESIGNAL</td>
<td>Resignal condition to next handler. The exception described by signal-arguments was not an instruction fault handled by LIB$DECODE_FAULT. If LIB$DECODE_FAULT can process the fault, it does not return to its caller.</td>
</tr>
</tbody>
</table>

Condition Value Signaled

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument to Run-Time Library. The instruction definition contained more than 16 operands or an operand definition contained an invalid data type or access code. This message is signaled after the stack frames have been unwound so that it appears to have been signaled from a routine that was called by the instruction that faulted.</td>
</tr>
</tbody>
</table>

Example

The following Fortran example implements a simple recovery scheme for floating underflow and overflow faults, replacing the result of the instruction with the correctly signed, smallest possible value for underflows or largest possible value for overflows.
LIB$ Routines
LIB$DECODE_FAULT

Example condition handler and user-action routine using LIB$DECODE_FAULT. This example demonstrates the use of most of the features of LIB$DECODE_FAULT. Its purpose is to handle floating underflow and overflow faults, replacing the result of the instruction with the correctly signed smallest possible value for underflows, or greatest possible value for overflows.

For simplicity, faults involving the POLYx instructions are not handled.

***

FIXUP_RESULT is the condition handler enabled by the program desiring the fixup of overflows and underflows.

**

INTEGER*4 FUNCTION FIXUP_RESULT(SIGARGS, MECHARGS)
IMPLICIT NONE
INCLUDE '($SSDEF)' ! SS$ symbols
INCLUDE '($LIBDCFDEF)' ! LIB$DECODE_FAULT symbols
INTEGER*4 SIGARGS(1:*) ! Signal arguments list
INTEGER*4 MECHARGS(1:*) ! Mechanism arguments list

**

This is a sample redefinition of MULH3 instruction.

**

BYTE OPTABLE(8) /'FD'X,'65'X, ! MULH3 opcode
1 LIB$K_DCFOPR_RH, ! Read H_floating
2 LIB$K_DCFOPR_RH, ! Read H_floating
3 LIB$K_DCFOPR_WH, ! Write H_floating
4 LIB$K_DCFOPR_END, ! End of operands
5 'FF'X,'FF'X/ ! End of instructions

INTEGER*4 LIB$DECODE_FAULT ! External function
EXTERNAL FIXUP_ACTION ! Action routine to do the fixup

**

Determine if the exception is one we want to handle.

**

IF ((SIGARGS(2) .EQ. SS$_FLTOVF_F) .OR. (SIGARGS(2) .EQ. SS$_FLTUND_F)) THEN

**

We think we can handle the fault. Call LIB$DECODE_FAULT and pass it the signal arguments and the address of our action routine and opcode table.

**

FIXUP_RESULT = LIB$DECODE_FAULT (SIGARGS, MECHARGS, %DESCR(FIXUP_ACTION), OPTABLE)
RETURN
END IF

We can only get here if we couldn't handle the fault. Resignal the exception.

**

FIXUP_RESULT = SS$_RESIGNAL
RETURN
END
LIB$ Routines
LIB$DECODE_FAULT

C+ User action routine to handle the fault.
C-

INTEGER*4 FUNCTION FIXUP_ACTION (OPCODE, INSTR_PC, PSL, REGISTERS, OP_COUNT, OP_TYPES, READ_OPS, WRITE_OPS, SIGARGS, SIGNAL_ROUT, CONTEXT, USER_ARG, ORIG_REGS)

IMPLICIT NONE
INCLUDE '($SSDEF)' ! SS$ definitions
INCLUDE '($PSLDEF)' ! PSL$ definitions
INCLUDE '($LIBDCFDEF)' ! LIB$DECODE_FAULT definitions

INTEGER*4 OPCODE ! Instruction opcode
INTEGER*4 INSTR_PC ! PC of this instruction
INTEGER*4 PSL ! Processor status
INTEGER*4 REGISTERS(0:15) ! R0-R15 contents
INTEGER*4 OP_COUNT ! Number of operands
INTEGER*4 OP_TYPES(1:*) ! Types of operands
INTEGER*4 READ_OPS(1:*) ! Addresses of read operands
INTEGER*4 WRITE_OPS(1:*) ! Addresses of write operands
INTEGER*4 SIGARGS(1:*) ! Signal argument list
INTEGER*4 SIGNAL_ROUT ! Signal routine address
INTEGER*4 CONTEXT ! Signal routine context
INTEGER*4 USER_ARG ! User argument value
INTEGER*4 ORIG_REGS(0:15) ! Original registers

C+ Declare and initialize table of class codes for each of the "real" opcodes. We’ll index into this by the first byte of one-byte opcodes, the second byte of two-byte opcodes. The class codes will be used in a computed GOTO (CASE). The codes are:
C-

0 - Unsupported
1 - ADD
2 - SUB
3 - MUL,DIV
4 - ACB
5 - CVT
6 - EMOD

The class mainly determines how we compute the sign of the result, except for ACB.

BYTE INST_CLASS_TABLE(0:255)
DATA INST_CLASS_TABLE /
1 48*0, ! 00-2F
2 0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0, ! 30-3F
3 1,1,2,2,3,3,3,3,0,0,0,0,0,0,0,4, ! 40-4F
4 0,0,0,0,6,0,0,0,0,0,0,0,0,0,0,0, ! 50-5F
5 1,1,2,2,3,3,3,0,0,0,0,0,0,0,0,4, ! 60-6F
6 0,0,0,0,6,0,5,0,0,0,0,0,0,0,0,0, ! 70-7F
7 112*0, ! 80-EF
8 0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0/ ! F0-PF

C+ Table of operand sizes in 8-bit bytes, indexed by the datatype code contained in the OP_TYPES array. Only floating types matter.
C-

BYTE OP_SIZES(9) /0,0,0,0,0,4,8,8,16/
LIB$ Routines
LIB$DECODE_FAULT

INTEGER*4 LIB$EXTV ! External function
INTEGER*4 RESULT_NEGATIVE ! -1 if result negative, _0 if positive
INTEGER*4 SIGN1,SIGN2,SIGN3 ! Signs of operands
INTEGER*4 INST_BYTE ! Current opcode byte
INTEGER*4 INST_CLASS ! Class of instruction from table
INTEGER*4 OP_DTYPE ! Datatype of operand
INTEGER*4 OP_SIZE ! Size of operand in 8-bit bytes
INTEGER*4 RESULT_OP ! Position of result in WRITE_OPS array
LOGICAL*4 OVERFLOW ! TRUE if SS$_FLTOVF_F
LOGICAL*4 SMALLER ! Function which compares operands
PARAMETER ESCD = '0FD'X ! First byte of G,H instructions
INTEGER*2 SMALL_F(2) ! Smallest F_floating
DATA SMALL_F /'0080'X,0/
INTEGER*2 SMALL_D(4) ! Smallest D_floating
DATA SMALL_D /'0080'X,0,0,0/
INTEGER*2 SMALL_G(4) ! Smallest G_floating
DATA SMALL_G /'0010'X,0,0,0/
INTEGER*2 SMALL_H(8) ! Smallest H_floating
DATA SMALL_H /'0001'X,0,0,0,0,0,0,0/
INTEGER*2 BIGGEST(8) ! Biggest value (all datatypes)
DATA BIGGEST /'7FFF'X,7*'FFFF'X/
INTEGER*4 SIGNAL_ARRAY(2) ! Array for signalling new exception

NOTE: Because the operands arrays contain the locations of the operands, rather than the operands themselves, we must call a routine using the %VAL function to "fool" the called routine into considering the contents of an operands array element as the address of an item. This would not be necessary in a language that understood the concept of pointer variables, such as PASCAL.

If FPD is set in the PSL, signal SS$_ROPRAND (reserved operand). In reality this shouldn’t happen since none of the instructions we handle can set FPD, but do it as an example.

IF (BTEST(PSL,PSL$V_FPD)) THEN
  SIGNAL_ARRAY(1) = 1 ! Count of signal arguments
  SIGNAL_ARRAY(2) = SS$_ROPRAND ! Error status value
  CALL SIGNAL_ROUT (1, SIGNAL_ARRAY(2) = SS$_ROPRAND
                    2 SIGNAL_ARRAY, ! Signal arguments array
                    3 CONTEXT) ! Context as passed to us
  ! Call will never return
END IF

Set OVERFLOW according to the exception type. We assume that the only alternatives are SS$_FLTOVF_F and SS$_FLTUND_F.

OVERFLOW = (SIGARGS(2) .EQ. SS$_FLTOVF_F)
 Determine the datatype of the instruction by that of its second operand, since that is always the type of the destination.

```c
OP_DTYPE = IBITS(OP_TYPES(2),LIB$V_DCFTYP,LIB$S_DCFTYP)
```

Get the size of the datatype in words.

```c
OP_SIZE = OP_SIZES(OP_DTYPE)
```

Determine the class of instruction and dispatch to the appropriate routine.

```c
INST_BYTE = IBITS(OPCODE,0,8) ! Get first byte
IF (INST_BYTE .EQ. ESCD) INST_BYTE = IBITS(OPCODE,8,8)
INST_CLASS = INST_CLASS_TABLE(INST_BYTE)
GO TO (1000,2000,3000,4000,5000,6000),INST_CLASS
```

If we get here, the instruction’s entry in the INST_CLASS_TABLE is zero. This might happen if the instruction was a POLYx, or was some other unsupported instruction. Resignal the original exception.

```c
FIXUP_ACTION = SS$_RESIGNAL ! Resignal condition to next handler RETURN ! Return to LIB$DECODE_FAULT
```

1000 - ADDF2, ADDF3, ADDD2, ADDD3, ADDG2, ADDG3, ADDH2, ADDH3

Result’s sign is the same as that of the first operand, unless this is an underflow, in which case the magnitudes of the values may change the sign.

```c
RESULT_NEGATIVE = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
IF (.NOT. OVERFLOW) THEN
  IF (SMALLER(OP_SIZE,%VAL(READ_OPS(1)),
    1 %VAL(READ_OPS(2))))
    RESULT_NEGATIVE = .NOT. RESULT_NEGATIVE
END IF
GO TO 9000
```

2000 - SUBF2, SUBF3, SUBD2, SUBD3, SUBG2, SUBG3, SUBH2, SUBH3

Result’s sign is the opposite of that of the first operand, unless this is an underflow, in which case the magnitudes of the values may change the sign.

```c
RESULT_NEGATIVE = .NOT. LIB$EXTV (15,1,%VAL(READ_OPS(1)))
IF (.NOT. OVERFLOW) THEN
  IF (SMALLER(OP_SIZE,%VAL(READ_OPS(1)),
    1 %VAL(READ_OPS(2))))
    RESULT_NEGATIVE = .NOT. RESULT_NEGATIVE
END IF
GO TO 9000
```
LIB$ Routines
LIB$DECODE_FAULT

C 3000 - MULF2, MULF3, MULD2, MULD3, MULG2, MULG3, MULH2, MULH3,
C    DIVF2, DIVF3, DIVD2, DIVD3, DIVG2, DIVG3, DIVH2, DIVH3,
C
C If the signs of the first two operands are the same, then the
C result’s sign is positive, if they are not it is negative.
C-
3000  SIGN1 = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      SIGN2 = LIB$EXTV (15,1,%VAL(READ_OPS(2)))
      RESULT_NEGATIVE = SIGN1 .XOR. SIGN2
      GOTO 9000

C 4000 - ACBF, ACBD, ACBG, ACBH
C
C The result’s sign is the same as that of the second operand
C (addend), unless this is underflow, in which case the
C magnitudes of the addend and index may change the sign.
C We must also determine if the branch is to be taken.
C-
4000  SIGN2 = LIB$EXTV (15,1,%VAL(READ_OPS(2)))
      RESULT_NEGATIVE = SIGN2
      IF (.NOT. OVERFLOW) THEN
          IF (SMALLER(OP_SIZE,%VAL(READ_OPS(2)),
                       1 %VAL(READ_OPS(3))))
              RESULT_NEGATIVE = .NOT. RESULT_NEGATIVE
          END IF
      END IF

C If this is overflow, then the branch is not taken, since the
C result is always going to be greater or equal in magnitude
to the limit, and will be the correct sign. If underflow,
C the branch is ALMOST always taken. The only case where the
C branch might not be taken is when the result is exactly
equal to the limit. For this example, we are going to ignore
C this exceptional case.
C-
      IF (.NOT. OVERFLOW)
          1 REGISTERS(15) = READ_OPS(4) ! Branch destination
          GO TO 9000
      END IF

C 5000 - CVTDF, CVTGF, CVTHF, CVTHD, CVTHG
C
C Result’s sign is the same as that of the first operand.
C-
5000  RESULT_NEGATIVE = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      GO TO 9000

C 6000 - EMODF, EMODD, EMODG, EMODH
C
C If the signs of the first and third operands are the same, then the
C result’s sign is positive, else it is negative.
C-
6000  SIGN1 = LIB$EXTV (15,1,%VAL(READ_OPS(1)))
      SIGN2 = LIB$EXTV (15,1,%VAL(READ_OPS(3)))
      RESULT_NEGATIVE = SIGN1 .XOR. SIGN2
      GOTO 9000
All code paths merge here to store the result value. We also set the PSL appropriately. First, determine which operand is the result.

```
9000 RESULT_OP = OP_COUNT
IF (INST_CLASS EQ 4) RESULT_OP = RESULT_OP - 1 ! ACBx
```

Select result based on datatype and exception type.

```
IF (OVERFLOW) THEN
   CALL LIB$MOVC3 (OP_SIZE,BIGGEST,%VAL(WRITE_OPS(RESULT_OP)))
ELSE
   GO TO (9100,9200,9300,9400), OP_DTYPE-(LIB$DCPTYP_F-1)
```

Should never get here. Resignal original exception.

```
FIXUP_ACTION = SS$_RESIGNAL
RETURN
```

9100 - F_floating result

```
9100 CALL LIB$MOVC3 (OP_SIZE,SMALL_F,%VAL(WRITE_OPS(RESULT_OP)))
GOTO 9500
```

9200 - D_floating result

```
9200 CALL LIB$MOVC3 (OP_SIZE,SMALL_D,%VAL(WRITE_OPS(RESULT_OP)))
GOTO 9500
```

9300 - G_floating result

```
9300 CALL LIB$MOVC3 (OP_SIZE,SMALL_G,%VAL(WRITE_OPS(RESULT_OP)))
GOTO 9500
```

9400 - H_floating result

```
9400 CALL LIB$MOVC3 (OP_SIZE,SMALL_H,%VAL(WRITE_OPS(RESULT_OP)))
GOTO 9500
```

9500 END IF

Modify the PSL to reflect the stored result. If the result was negative, set the N bit. Clear the V (overflow) and Z (zero) bits. If the instruction was an ACBx, leave the C (carry) bit unchanged, otherwise clear it.

```
IF (RESULT_NEGATIVE) THEN
   PSL = IBSET (PSL,PSL$V_N) ! Set N bit
ELSE
```

lib–128
LIB$ Routines
LIB$DECODE_FAULT

PSL = IBCLR (PSL, PSL$V_N) ! Clear N bit
END IF
PSL = IBCLR (PSL, PSL$V_V) ! Clear V bit
PSL = IBCLR (PSL, PSL$V_Z) ! Clear Z bit
IF (INST_CLASS .NE. 4)
  PSL = IBCLR (PSL, PSL$V_C) ! Clear C bit if not ACBx
C+ Set the sign of result.
C-
  IF (RESULT NEGATIVE)
    CALL LIB$INSV (1,15,1,%VAL(WRITE_OPS(RESULT_OP)))
  END IF
C+ Fixup is complete. Return to LIB$DECODE_FAULT.
C-
  FIXUP_ACTION = SS$ _CONTINUE
  RETURN
END

C Function which compares two floating values. It returns .TRUE. if
C the first argument is smaller in magnitude than the second.
C-
LOGICAL*4 FUNCTION SMALLER(NBYTES,VAL1,VAL2)
INTEGER*4 NBYTES ! Number of bytes in values
INTEGER*2 VAL1(*),VAL2(*) ! Floating values to compare
INTEGER*4 WORDA,WORDB
SMALLER = .TRUE. ! Initially return true
C+ Zero extend to a longword for unsigned compares.
C Compare first word without sign bit.
C-
  WORDA = IBCLR(ZEXT(VAL1(1)),15)
  WORDB = IBCLR(ZEXT(VAL2(1)),15)
  IF (WORDA .LT. WORDB) RETURN
DO I=2,NBYTES/2
  WORDA = ZEXT(VAL1(I))
  WORDB = ZEXT(VAL2(I))
  IF (WORDA .LT. WORDB) RETURN
END DO
SMALLER = .FALSE. ! VAL1 not smaller than VAL2
RETURN
END
LIB$ DEC_OVER
Enable or Disable Decimal Overflow Detection

The Enable or Disable Decimal Overflow Detection routine enables or disables decimal overflow detection for the calling routine activation. The previous decimal overflow setting is returned. †

This routine is available on OpenVMS Alpha and I64 systems in translated form and is applicable to translated VAX images only.

**Format**

```
LIB$DEC_OVER new-setting
```

**Returns**

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

The old decimal overflow enable setting (the previous contents of SF$W_PS_W[PSW$V_DV] in the caller’s frame).

**Argument**

```
new-setting
```

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

New decimal overflow enable setting. The `new-setting` argument is the address of an unsigned longword that contains the new decimal overflow enable setting. Bit 0 set to 1 means enable; bit 0 set to 0 means disable.

**Description**

The caller’s stack frame is modified by this routine.

A call to LIB$DEC_OVER affects only the current routine activation and does not affect any of its callers or any routines that it may call. However, the setting does remain in effect for any routines that are subsequently entered through a JSB entry point.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
Example

DECOVF: ROUTINE OPTIONS (MAIN);

DECLARE LIB$DEC_OVER ENTRY (FIXED BINARY (7)) /* Address of byte for
    /* enable/disable
    /* setting */
    RETURNS (FIXED BINARY (31)); /* Old setting */

DECLARE DISABLE FIXED BINARY (7) INITIAL (0) STATIC READONLY;
DECLARE RESULT FIXED BINARY (31);
DECLARE (A,B) FIXED DECIMAL (4,2);

ON FIXEDOVERFLOW PUT SKIP LIST ('Overflow');
RESULT = LIB$DEC_OVER (DISABLE); /* Disable recognition of decimal
    /* overflow in this block */
A = 99.99;
B = A + 2;
PUT SKIP LIST ('In MAIN');
BEGIN;
    B = A + 2;
    PUT LIST ('In BEGIN block');
    CALL Q;
        Q: ROUTINE;
            B = A + 2;
            PUT LIST ('In Q');
        END Q;
    END /* Begin */;
END DECOVF;

This PL/I program shows how to use LIB$DEC_OVER to enable or disable the
detection of decimal overflow. Note that in PL/I, disabling decimal overflow
using this routine causes the condition to be disabled only in the current block;
descendent blocks will enable the condition unless this routine is called in each
block.
LIB$ Routines
LIB$DELETE_FILE

LIB$DELETE_FILE
Delete One or More Files

The Delete One or More Files routine deletes one or more files. The specification of the files to be deleted may include wildcards.

LIB$DELETE_FILE is similar in function to the DCL command DELETE.

Format


Returns

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

Arguments

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

String containing the OpenVMS Record Management Services (RMS) file specification of the files to be deleted. The filespec argument is the address of a descriptor pointing to the file specification. If the specification includes wildcards, each file that matches the specification is deleted. If running on Alpha or I64 and flag LIB$M_FIL_LONG_NAMES is set, the string must not contain more characters than specified by NAML$C_MAXRSS, otherwise the string must not contain more than 255 characters. Any string class is supported.

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

Default file specification of the files to be deleted. The default-filespec argument is the address of a descriptor pointing to the default file specification. This is an optional argument; if the argument is omitted, the default is the null string. Any string class is supported.

See the OpenVMS Record Management Services Reference Manual for information about default file specifications.

related-filespec
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor
Related file specification of the files to be deleted. The related-filspec argument is the address of a descriptor pointing to the related file specification. Any string class is supported. This is an optional argument; if the argument is omitted, the default is the null string.

Input file parsing is used. See the OpenVMS Record Management Services Reference Manual for information on related file specifications and input file parsing.

The related file specification is useful when you are processing lists of file specifications. Unspecified portions of the file specification are inherited from the last file processed.

**user-success-procedure**

OpenVMS usage: procedure

type: procedure value

access: function call (before return)

mechanism: by value

User-supplied success routine that LIB$DELETE_FILE calls after it successfully deletes a file.

The success routine can be used to display a log of the files that were deleted. For more information on the success routine, see Call Format for a Success Routine in the Description section.

**user-error-procedure**

OpenVMS usage: procedure

type: procedure value

access: function call (before return)

mechanism: by value

User-supplied error routine that LIB$DELETE_FILE calls when it detects an error.

The error routine returns a success/fail value that LIB$DELETE_FILE uses to determine if more files should be processed. For more information on the error routine, see Call Format for an Error Routine in the Description section.

**user-confirm-procedure**

OpenVMS usage: procedure

type: procedure value

access: function call (before return)

mechanism: by value

User-supplied confirm routine that LIB$DELETE_FILE calls before each file is deleted. The value returned by the confirm routine determines whether or not the file will be deleted. The confirm routine can be used to select specific files for deletion based on criteria such as expiration date, size, and so on. For more information about the confirm routine, see Call Format for a Confirm Routine in the Description section.

**user-specified-argument**

OpenVMS usage: user_arg

type: longword (unsigned)

access: read only

mechanism: by value
User-supplied argument that LIB$DELETE_FILE passes to the error, success, and confirm routines each time they are called. Whatever mechanism is used to pass user-specified-argument to LIB$DELETE_FILE is also used to pass it to the routines. This is an optional argument; if the argument is omitted, zero is passed by value.

**resultant-name**

OpenVMS usage: char_string

Type: character string

Access: write only

Mechanism: by descriptor

String into which LIB$DELETE_FILE writes the RMS resultant file specification of the last file processed. The resultant-name argument is the address of a descriptor pointing to the resultant name.

If present, resultant-name is used to store the file specification passed to the user-supplied routines, instead of a default class S, type T string. Therefore, this argument should be specified when the user-supplied routines are used and those routines require a descriptor type other than class S, type T. Any string class is supported.

If you specify one or more of the user-supplied action routines, the descriptor used to pass resultant-name must be:

- Of the same class as the descriptor required by the filespec argument of any action routines. For example, VAX Ada requires a class SB descriptor for string arguments to Ada routines but will use a class A descriptor by default when calling external routines. Refer to your language manual to determine the proper descriptor class to use.

- (Alpha and I64 only) Of the same form as the descriptor required by the filespec argument of all action routines. For example, if the filespec argument of an action routine uses a 64-bit descriptor, then the resultant-name argument must also use a 64-bit descriptor.

**file-scan-context**

OpenVMS usage: context

Type: longword (unsigned)

Access: modify

Mechanism: by reference

Context for deleting a list of file specifications. The file-scan-context argument is the address of a longword containing the context value.

You must initialize the file scan context to zero before the first of a series of calls to LIB$DELETE_FILE. LIB$FILE_SCAN uses this context to retain the file context for multiple input files. You must specify this context only when you are dealing with multiple input files, as the DCL command DELETE does. You may deallocate the context allocated by LIB$FILE_SCAN by calling LIB$FILE_SCAN_END after all calls to LIB$DELETE_FILE have been completed.

**flags**

OpenVMS usage: mask_longword

Type: longword (unsigned)

Access: read only

Mechanism: by reference
User flags. The flags argument is the address of an unsigned longword containing the user flags.

The flag bits and their corresponding symbols are described in the following table:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved to HP.</td>
<td>Reserved to HP.</td>
</tr>
<tr>
<td>1</td>
<td>Reserved to HP.</td>
<td>Reserved to HP.</td>
</tr>
<tr>
<td>2</td>
<td>LIB$M_FIL_LONG_NAMES (Alpha or I64 only) If set, LIB$DELETE_FILE can process file names with a maximum length of NAML$C_MAXRSS. If clear, LIB$DELETE_FILE can process file specifications with a maximum length of 255 (default).</td>
<td></td>
</tr>
</tbody>
</table>

Description

This Description section is divided into the following parts:

- Call Format for a Success Routine
- Call Format for an Error Routine
- Call Format for a Confirm Routine

**Call Format for a Success Routine**

The success routine is called only if the user-success-procedure argument was specified in the LIB$DELETE_FILE argument list.

The calling format of a success routine is as follows:

```
user-success-procedure filespec [,user-specified-argument]
```

**filespec**

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

RMS resultant file specification of the file being deleted. The filespec argument is the address of a descriptor pointing to the file specification. If the resultant-name argument was specified, it is used to pass the string to the success routine. Otherwise, a class S, type T string is passed. Any string class is supported.

On Alpha and I64 systems, the descriptor specified by each of the action routines for the filespec argument and the descriptor specified by the LIB$DELETE_FILE resultant-name argument, if any, must be of the same form. They must all be 32-bit descriptors or all 64-bit descriptors. If you do not specify a resultant-name argument, then the filespec argument must use a 32-bit descriptor.

**user-specified-argument**

- **OpenVMS usage:** user_arg
- **type:** longword (unsigned)
- **access:** read only
- **mechanism:** unspecified
Value of **user-specified-argument** passed by LIB$DELETE_FILE to the success routine. The same passing mechanism that was used to pass **user-specified-argument** to LIB$DELETE_FILE is used by LIB$DELETE_FILE to pass **user-specified-argument** to the success routine. This is an optional argument.

**Call Format for an Error Routine**

The error routine is called only if the **user-error-procedure** argument was specified in the LIB$DELETE_FILE argument list.

The calling format of an error routine is as follows:

```
user-error-procedure filespec ,rms-sts ,rms-stv ,error-source [,user-specified-argument]
```

**filespec**

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

String containing the RMS resultant file specification of the file being deleted. If **resultant-name** was specified, it is used to pass the string to the error routine. Otherwise, a class S, type T string is passed. Any string class is supported.

On Alpha and I64 systems, the descriptor specified by each of the action routines for the **filespec** argument and the descriptor specified by the LIB$DELETE_FILE **resultant-name** argument, if any, must be of the same form. They must all be 32-bit descriptors or all 64-bit descriptors. If you specify no **resultant-name** argument, then the **filespec** argument must use a 32-bit descriptor.

**rms-sts**

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

Primary condition code (FAB$L_STS) that describes the error that occurred. The **rms-sts** argument is the address of an unsigned longword that contains the primary condition code.

**rms-stv**

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

Secondary condition code (FAB$L_STV) that describes the error that occurred. The **rms-stv** argument is the address of an unsigned longword that contains the secondary condition code.

**error-source**

OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Integer code that indicates the point at which the error was found. The **error-source** argument is the address of a longword integer containing the code of the error source.
Possible values for the error code are as follows:

0  Error searching for file specification
1  Error deleting file

**user-specified-argument**
- OpenVMS usage: user_arg
- type: unspecified
- access: read only
- mechanism: unspecified

Value passed to LIB$DELETE_FILE that is then passed to **user-error-procedure** using the same passing mechanism that was used to pass it to LIB$DELETE_FILE. This is an optional argument.

If the error routine returns a success status (bit 0 set), then LIB$DELETE_FILE continues processing files. If a failure status (bit 0 clear) is returned, then processing ceases immediately, and LIB$DELETE_FILE returns with the error status.

If the **user-error-procedure** argument is not specified, LIB$DELETE_FILE returns to its caller the most severe error status encountered while deleting the files. If the error routine is called for an error, the success status LIB$_ERRROUCAL is returned.

The error routine is not called for errors related to string copying.

**Call Format for a Confirm Routine**

The confirm routine is called only if the **user-confirm-procedure** argument was specified in the call to LIB$DELETE_FILE.

The calling format of the confirm routine is as follows:

```
user-confirm-procedure filespec ,fab [,user-specified-argument]
```

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

RMS resultant file specification of the file to be deleted. The **filespec** argument is the address of a descriptor pointing to the file specification.

If **resultant-name** was specified, it is used to pass the string to the confirm routine. Otherwise, a class S, type T string is passed. Any string class is supported.

On Alpha and I64 systems, the descriptor specified by each of the action routines for the **filespec** argument and the descriptor specified by the LIB$DELETE_FILE **resultant-name** argument, if any, must be of the same form. They must all be 32-bit descriptors or all 64-bit descriptors. If you do not specify a **resultant-name** argument, then the **filespec** argument must use a 32-bit descriptor.

**fab**
- OpenVMS usage: fab
- type: unspecified
- access: read only
- mechanism: by reference
RMS file access block (FAB) that describes the file being deleted. Your program may perform an RMS $OPEN on the FAB to obtain file attributes to determine whether the file should be deleted, but it must close the file with $CLOSE before returning to LIB$DELETE_FILE.

On Alpha and I64 systems, if the LIB$M_FIL_LONG_NAMES FLAGS is set, the FAB references a NAML block rather than a NAM block. The NAML block supports the use of long file names with a maximum length of NAML$C_MAXRSS. See the OpenVMS Record Management Services Reference Manual for information on NAML blocks.

user-specified-argument
OpenVMS usage: user_arg
type: unspecified
access: read only
mechanism: unspecified

The value of the user-specified-argument argument that LIB$DELETE_FILE passes to the confirm routine using the same passing mechanism that was used to pass it to LIB$DELETE_FILE. This is an optional argument.

If confirm routine returns a success status (bit 0 set), the file is then deleted; otherwise, the file is not deleted.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_ERRROUCAL Success, but an error routine was called. A file error was encountered, but the error routine was called to handle the condition.
LIB$INVARG Invalid argument. The flags argument has one or more undefined bits set.
LIB$_INVFILSPE Invalid file specification. Filespec or default-filespec is longer than 4095 characters.
LIB$_INVSTRDES Invalid string descriptor. The descriptor for a string argument was not a valid string descriptor.
LIB$_WRONUMARG Wrong number of arguments. An incorrect number of arguments was passed to LIB$DELETE_FILE.

Any condition value returned by LIB$SCOPY_xxx except those condition values specifying truncation errors.

Any condition value returned by RMS. If user-error-procedure is not specified, this is the most severe of the RMS errors encountered while deleting the files.

Example

PROGRAM DELETE_EXAMPLE(INPUT, OUTPUT);
{+}
{ Declare external function LIB$DELETE_FILE. Throughout this
{ example, the user-arg argument is not used.
{-}
FUNCTION LIB$DELETE_FILE(
  FILESPEC: VARYING [A] OF CHAR;
  DEFAULT_FILESPEC: VARYING [B] OF CHAR;
  REL_FILESPEC : VARYING [D] OF CHAR;
  %IMMED [UNBOUND] ROUTINE SUCCESS_ROUTINE
(FILESPEC : VARYING [A] OF CHAR) := %IMMED 0;
  %IMMED [UNBOUND] FUNCTION ERROR_ROUTINE
(FILESPEC : VARYING [A] OF CHAR; RMS_STS, RMS_STV : INTEGER)
 : BOOLEAN := %IMMED 0;
  %IMMED [UNBOUND] FUNCTION CONFIRM_ROUTINE
(FILESPEC: VARYING [A] OF CHAR): BOOLEAN := %IMMED 0;
  VAR USER_ARG : [UNSAFE] INTEGER := %IMMED 0;
  VAR RESULT_NAME : VARYING [C] OF CHAR := %IMMED 0
) : INTEGER; EXTERN;

{+} Declare a routine which will display the names of the files
   as they are deleted.
{+}

ROUTINE LOG_ROUTINE(FILESPEC : VARYING [A] OF CHAR);
BEGIN
  WRITELN('File ', FILESPEC, ' successfully deleted');
END;

{+} Declare a routine which will notify the user that an error
   occurred.
{+}

FUNCTION ERR_ROUTINE(FILESPEC: VARYING [A] OF CHAR;
  RMS_STS, RMS_STV: INTEGER): BOOLEAN;
BEGIN
  WRITELN('Delete of ', FILESPEC, ' failed ', HEX(RMS_STS));
  ERR_ROUTINE := TRUE;
END;

{+} Declare a routine which checks to see if the file should be
   deleted. If the filename contains the string ‘XYZ’, then it is
   deleted.
{+}

FUNCTION CONFIRM_ROUTINE( FILESPEC: VARYING [A] OF CHAR): BOOLEAN;
BEGIN
  IF INDEX(FILESPEC, 'XYZ') <> 0
  THEN
    CONFIRM_ROUTINE := TRUE
  ELSE
    CONFIRM_ROUTINE := FALSE;
  END;

{+} The main program begins here.
{+}

VAR
  FILES_TO_DELETE, RESULTANT_NAME : VARYING [255] OF CHAR;
  RET_STATUS : INTEGER;
BEGIN
  WRITE ('Files to delete: ');
  READLN(FILES_TO_DELETE);
  RET_STATUS := LIB$DELETE_FILE(
    FILES_TO_DELETE, '*', '', LOG_ROUTINE, ERR_ROUTINE,
    CONFIRM_ROUTINE, RESULTANT_NAME);
  IF NOT ODD(RET_STATUS)
    THEN
    WRITELN('Delete failed. The error was ', HEX(RET_STATUS));
END.

This Pascal program prompts the user for file specifications of files to be deleted. Of those, it deletes only files that contain the string XYZ somewhere in their resultant file specification. The names of deleted files are displayed.
LIB$DELETE_LOGICAL
Delete Logical Name

The Delete Logical Name routine requests the calling process' command language interpreter (CLI) to delete a supervisor-mode process logical name. LIB$DELETE_LOGICAL provides the same function as the DCL command DEASSIGN.

Format

LIB$DELETE_LOGICAL logical-name [,table-name]

Returns

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

Arguments

logical-name
OpenVMS usage: logical_name
type: character string
access: read only
mechanism: by descriptor
Logical name to be deleted. The logical-name argument is the address of a descriptor pointing to this logical name string. The maximum length of a logical name is 255 characters.

table-name
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor
Name of the table from which the logical name is to be deleted. The table-name argument is the address of a descriptor pointing to this name string. This is an optional argument. If the argument is omitted, the LNM$PROCESS table is used.

Description

LIB$DELETE_LOGICAL requests the calling process's command language interpreter (CLI) to delete a supervisor-mode process logical name. If the optional table-name argument is defined, the logical name is deleted from that table. Otherwise, the logical name is deleted from the LNM$PROCESS table.

Unlike the system service $DELLOG and $DELLNM, LIB$DELETE_LOGICAL does not require the caller to be executing in supervisor mode to delete a supervisor-mode logical name.

This routine is supported for use with the DCL and MCR command language interpreters.
LIB$ Routines
LIB$DELETE_LOGICAL

This routine does not support the DCL DEFINE and DEASSIGN commands’ special side effect of opening and closing a process-permanent file if the logical name “SYS$OUTPUT” is specified.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In that case, the error status LIB$_NOCLI is returned.

See the *HP OpenVMS DCL Dictionary* for a description of the DCL command DEASSIGN.

**Condition Values Returned**

- **SS$ _ACCVIO**
  Access violation. The logical name could not be read.

- **SS$ _IVLOGNAM**
  Invalid logical name. The logical name contained illegal characters or more than 255 characters.

- **SS$ _IVLOGTAB**
  Invalid logical name table

- **SS$ _NOLOGNAM**
  No logical name match. The logical name was not defined as a supervisor-mode process logical name.

- **SS$ _NOPRIV**
  No privilege for attempted operation.

- **SS$ _NORMAL**
  Routine successfully completed.

- **SS$ _TOOMANYLNAM**
  Logical name translation exceeded allowed depth.

- **LIB$ _INVSTRDES**
  Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.

- **LIB$ _NOCLI**
  No CLI present to perform function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.

- **LIB$ _UNECLIERR**
  Unexpected CLI error. The CLI returned an error status that was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL command language interpreter, please report the problem to your HP support representative.
LIB$DELETE_SYMBOL
Delete CLI Symbol

The Delete CLI Symbol routine requests the calling process’s command language interpreter (CLI) to delete an existing CLI symbol.

Format

LIB$DELETE_SYMBOL symbol [,table-type-indicator]

Returns

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

Arguments

symbol
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor
Name of the symbol to be deleted by LIB$DELETE_SYMBOL. The symbol argument is the address of a descriptor pointing to this symbol string. The symbol name is converted to uppercase, and trailing blanks are removed before use.

Symbol must begin with a letter, a digit, a dollar sign ($), a hyphen (-), or an underscore (_). The maximum length of symbol is 255 characters.

table-type-indicator
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference
Indicator of the table that contains the symbol to be deleted. The table-type-indicator argument is the address of a signed longword integer that is this table indicator.

If table-type-indicator is omitted, the local symbol table is used. The following are possible values for the table-type-indicator argument:

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Value</th>
<th>Table Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_CLI_LOCAL_SYM</td>
<td>1</td>
<td>Local symbol table</td>
</tr>
<tr>
<td>LIB$K_CLI_GLOBAL_SYM</td>
<td>2</td>
<td>Global symbol table</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$DELETE_SYMBOL

Description
LIB$DELETE_SYMBOL is supported for use with the DCL CLI. The error status 
LIB$_NOCLI is returned if LIB$DELETE_SYMBOL is used with the MCR CLI or called from 
an image run directly as a subprocess or as a detached process.

LIB$K_CLI_LOCAL_SYM and LIB$K_CLI_GLOBAL_SYM are defined in symbol libraries supplied by HP (macro or module name $LIBCLIDEF) and as global symbols.

Condition Values Returned

- **SS$_NORMAL**
  Routine successfully completed.

- **LIB$_FATERRLIB**
  Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to HP.

- **LIB$_INSVIRMEM**
  Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.

- **LIB$_INVAR**
  Invalid argument. The value of table-type-indicator was invalid.

- **LIB$_INVSTRDES**
  Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.

- **LIB$_INVSYMNAME**
  Invalid symbol name. The symbol name contained more than 255 characters or did not begin with a letter, a digit, a dollar sign, a hyphen, or an underscore.

- **LIB$_NOCLI**
  No CLI present to perform the function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.

- **LIB$_NOSUCHSYM**
  No such symbol. The symbol was not defined.

- **LIB$_UNECLIERR**
  Unexpected CLI error. The CLI returned an error status that was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL command language interpreter, please report the problem to your HP support representative.
LIB$DELETE_VM_ZONE
Delete Virtual Memory Zone

The Delete Virtual Memory Zone routine deletes a zone from the 32-bit virtual address space and returns all pages on VAX systems or pagelets on Alpha and I64 systems owned by the zone to the processwide 32-bit page pool. †

Format

LIB$DELETE_VM_ZONE  zone-id

Returns

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

Argument

zone-id
OpenVMS usage:  identifier
  type:    longword (unsigned)
  access:  read only
  mechanism:  by reference

Zone identifier. The zone-id is the address of a longword that contains the identifier of a zone created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE.

Description

LIB$DELETE_VM_ZONE deletes a zone and returns all pages on VAX systems or pagelets on Alpha and I64 systems owned by the zone to the processwide pool managed by LIB$GET_VM_PAGE. The pages or pagelets are then available for reallocation by later calls to LIB$GET_VM or LIB$GET_VM_PAGE.

It takes less time to free memory in a single operation by calling LIB$DELETE_VM_ZONE than to individually account for and free every block of memory that was allocated by calling LIB$GET_VM.

You must ensure that your program is no longer using any of the memory in the zone before you call LIB$DELETE_VM_ZONE. Your program must not do any further operations on the zone after you call LIB$DELETE_VM_ZONE.

If you specified deallocation filling when you created the zone, LIB$DELETE_VM_ZONE will fill all of the allocated blocks that are freed.

If the zone you are deleting was created using the LIB$CREATE_USER_VM_ZONE routine, then you must have an appropriate action routine for the delete operation. That is, in your call to LIB$CREATE_USER_VM_ZONE, you must have specified a user-delete-procedure.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$DELETE_VM_ZONE

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$BADBLOADR</td>
<td>An invalid zone-id argument or a corrupted zone.</td>
</tr>
</tbody>
</table>
LIB$DELETE_VM_ZONE_64 (Alpha and I64 Only)
Delete Virtual Memory Zone

The Delete Virtual Memory Zone routine deletes a zone from the 64-bit virtual address space and returns all Alpha and I64 system pagelets owned by the zone to the processwide 64-bit page pool.

Format

LIB$DELETE_VM_ZONE_64 zone-id

Returns

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

Argument

zone-id
OpenVMS usage: identifier
type: quadword (unsigned)
access: read only
mechanism: by reference

Zone identifier. The zone-id is the address of a quadword that contains the identifier of a zone created by a previous call to LIB$CREATE_VM_ZONE_64 or LIB$CREATE_USER_VM_ZONE_64.

Description

LIB$DELETE_VM_ZONE_64 deletes a zone and returns all pagelets on Alpha and I64 systems owned by the zone to the processwide pool managed by LIB$GET_VM_PAGE_64. The pagelets are then available for reallocation by later calls to LIB$GET_VM_64 or LIB$GET_VM_PAGE_64.

It takes less time to free memory in a single operation by calling LIB$DELETE_VM_ZONE_64 than to individually account for and free every block of memory that was allocated by calling LIB$GET_VM_64.

You must ensure that your program is no longer using any of the memory in the zone before you call LIB$DELETE_VM_ZONE_64. Your program must not do any further operations on the zone after you call LIB$DELETE_VM_ZONE_64.

If you specified deallocation filling when you created the zone, LIB$DELETE_VM_ZONE_64 will fill all of the allocated blocks that are freed.

If the zone you are deleting was created using the LIB$CREATE_USER_VM_ZONE_64 routine, then you must have an appropriate action routine for the delete operation. That is, in your call to LIB$CREATE_USER_VM_ZONE_64, you must have specified a user-delete-procedure.
LIB$ Routines
LIB$DELETE_VM_ZONE_64 (Alpha and I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADDLOADR</td>
<td>An invalid zone-id argument or a corrupted zone.</td>
</tr>
</tbody>
</table>
LIB$DIGIT_SEPARATOR Symbol

The Get Digit Separator Symbol routine returns the system's digit separator symbol.

Format

LIB$DIGIT_SEP  digit-separator-string [,resultant-length]

Returns

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

Arguments

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

Digit separator symbol returned by LIB$DIGIT_SEP. The digit-separator-string argument is the address of a descriptor pointing to the digit separator.

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

Number of characters written into digit-separator-string, not counting padding in the case of a fixed-length string. The resultant-length argument is the address of an unsigned word containing the length of the digit separator symbol. If the input string is truncated to the size specified in the digit-separator-string descriptor, resultant-length is set to this size. Therefore, resultant-length can always be used by the calling program to access a valid substring of digit-separator-string.

Description

LIB$DIGIT_SEP returns the symbol that is used to separate groups of three digits in the integer part of a number, for readability. A common digit separator is a comma (,) as in 3,006,854.

LIB$DIGIT_SEP attempts to translate the logical name SYS$DIGIT_SEP as a process, group, or system logical name. If the translation fails, LIB$DIGIT_SEP returns a comma (,), the United States digit separator. If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS$DIGIT_SEP as a systemwide logical name to provide a default for all users, and an individual user with a special need can define SYS$DIGIT_SEP as a process logical name to override the default symbol. For example, you may want to use the European digit separator, the period (.).
**LIB$ Routines**

**LIB$DIGIT_SEP**

BASIC implicitly uses LIB$DIGIT_SEP.

### Condition Values Returned

- **SS$_\text{NORMAL}$**
  - Routine successfully completed.

- **LIB$_\text{FATERRLIB}$**
  - Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to HP.

- **LIB$_\text{INSVIRMEM}$**
  - Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.

- **LIB$_\text{INVSTRDES}$**
  - Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.

- **LIB$_\text{STRTRU}$**
  - Successfully completed, but the digit separator string was truncated.

### Example

```pascal
PROGRAM DIGIT_SEP(INPUT, OUTPUT);
{
  This program uses LIB$DIGIT_SEP$ to return current value of SYS$DIGIT_SEP$.
}{
routine LIB$DIGIT_SEP$(%DESCR DIGIT_SEPSTR : VARYING [A] OF CHAR; %REF OUT_LEN : INTEGER); EXTERN;

VAR
  SEPARATOR : VARYING [256] OF CHAR;
  LENGTH : INTEGER;
BEGIN
  LIB$DIGIT_SEP$(SEPARATOR, LENGTH);
  WRITELN('104',SEPARATOR,'567',SEPARATOR,'934');
END.
```

This Pascal example demonstrates how to use LIB$DIGIT_SEP$. The output generated by this program is as follows:

```
104,567,934
```
LIB$DISABLE_CTRL

**Enable CLI Interception of Control Characters**

The Disable CLI Interception of Control Characters routine requests the calling process's command language interpreter (CLI) to not intercept the selected control characters when they are entered during an interactive terminal session. LIB$DISABLE_CTRL provides the same function as the DCL command SET NOCONTROL.

**Format**

LIB$DISABLE_CTRL  disable-mask [,old-mask]

**Returns**

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

**Arguments**

**disable-mask**

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

Bit mask indicating which control characters are not to be intercepted. The disable-mask argument is the address of an unsigned longword containing this bit mask.

Each of the 32 bits corresponds to one of the 32 possible control characters. If a bit is set, the corresponding control character is no longer intercepted by the CLI. Currently, only bits 20 and 25, corresponding to Ctrl/T and Ctrl/Y, are recognized.

The following mask is defined in symbol libraries supplied by HP to specify the value of disable-mask:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hex Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$M_CLI_CTRLT</td>
<td>%X'00100000'</td>
<td>Disables Ctrl/T</td>
</tr>
<tr>
<td>LIB$M_CLI_CTRLY</td>
<td>%X'02000000'</td>
<td>Disables Ctrl/Y</td>
</tr>
</tbody>
</table>

If a set bit does not correspond to a character that the CLI can intercept, LIB$DISABLE_CTRL returns an error.
**LIB$ Routines**

**LIB$DISABLE_CTRL**

**old-mask**

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

Previous bit mask. The old-mask argument is the address of an unsigned longword into which LIB$DISABLE_CTRL writes the old bit mask. The old bit mask is of the same form as disable-mask and indicates those control characters that were previously enabled. It may therefore be given to LIB$ENABLE_CTRL to reinstate the previous condition.

**Description**

The DCL and MCR CLIs can intercept the Ctrl/Y control character. The DCL CLI can intercept the Ctrl/T character. See the *HP OpenVMS DCL Dictionary* for information on how the DCL CLI processes control characters.

LIB$DISABLE_CTRL is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, LIB$DISABLE_CTRL returns the error status LIB$_NOCLI.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument. A bit in disable-mask was set that did not correspond to a control character supported by the CLI.</td>
</tr>
<tr>
<td>LIB$_NOCLI</td>
<td>No CLI present. Either the calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.</td>
</tr>
<tr>
<td>LIB$_UNECLIERR</td>
<td>Unexpected CLI error. The CLI returned an error status that was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to your HP support representative.</td>
</tr>
</tbody>
</table>
LIB$DO_COMMAND

Execute Command

The Execute Command routine stops program execution and directs the command language interpreter (CLI) to execute a command that you supply as the argument. If successful, LIB$DO_COMMAND does not return control to the calling program. Instead, LIB$DO_COMMAND begins execution of the specified command.

If you want control to return to the caller, use LIB$SPAWN instead.

Format

LIB$DO_COMMAND  command-string

Returns

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

Argument

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

Text of the command that LIB$DO_COMMAND executes. The command-string argument is the address of a descriptor pointing to the command text. The maximum length of the command is 255 characters.

Description

LIB$DO_COMMAND terminates your current image and then executes the contents of command-string as a command. The command is parsed using normal DCL rules.

LIB$DO_COMMAND is especially useful when you want to execute a CLI command after your program has finished executing. For example, you could use the routine to execute a SUBMIT or PRINT command to handle a file that your program has created.

Because of the following restrictions on LIB$DO_COMMAND, you should be careful when you incorporate it in your program:

• During the call to LIB$DO_COMMAND, the current image exits and control cannot return to it.

• The text of the command is passed to the current command language interpreter. Because you can define your own CLI in addition to DCL and MCR, you must make sure that the command will be handled by the intended CLI.
LIB$ Routines
LIB$DO_COMMAND

- If LIB$DO_COMMAND is called from an image run directly as a subprocess or detached process, it will not execute correctly, because no CLI is associated with a subprocess.

LIB$DO_COMMAND is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, the error status LIB$_NOCLI is returned. Note that the command can execute an indirect file using the at sign (@) feature of DCL.

Condition Values Returned

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument. command-string was more than 255 characters.</td>
</tr>
<tr>
<td>LIB$_NOCLI</td>
<td>No CLI present. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.</td>
</tr>
<tr>
<td>LIB$_UNECLIERR</td>
<td>Unexpected CLI error. The CLI returned an error status that was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to your HP support representative.</td>
</tr>
</tbody>
</table>

Example

```pascal
PROGRAM DO_COMMAND(INPUT, OUTPUT);
{
  This example uses LIB$DO_COMMAND to execute any DCL command that is entered by the user at the prompt.
}
PROCEDURE LIB$DO_COMMAND(CMDTXT : VARYING [A] OF CHAR);
  EXTERN;
VAR
  COMMAND : VARYING [256] OF CHAR;
BEGIN
  WRITELN('ENTER THE COMMAND YOU WANT TO EXECUTE: ');
  READLN(COMMAND);
  LIB$DO_COMMAND(COMMAND);
END.
```

This Pascal program shows how to call LIB$DO_COMMAND. An example of the output of this program is as follows:

```
$ RUN DO_COMMAND
ENTER THE COMMAND YOU WANT TO EXECUTE: SHOW TIME
30-MAY-2000 14:07:28
```
LIB$EDIV

Extended-Precision Divide

The Extended-Precision Divide routine performs extended-precision division. LIB$EDIV makes the VAX EDIV instruction available as a callable routine. \(^1\)

**Format**

```plaintext
LIB$EDIV longword-integer-divisor, quadword-integer-dividend, longword-integer-quotient, remainder
```

**Returns**

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

**Arguments**

**longword-integer-divisor**

OpenVMS usage: longword_signed  
type: longword integer (signed)  
access: read only  
mechanism: by reference

Divisor. The `longword-integer-divisor` argument is the address of a signed longword integer containing the divisor.

**quadword-integer-dividend**

OpenVMS usage: quadword_signed  
type: quadword integer (signed)  
access: read only  
mechanism: by reference

Dividend. The `quadword-integer-dividend` argument is the address of a signed quadword integer containing the dividend.

**longword-integer-quotient**

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

Quotient. The `longword-integer-quotient` argument is the address of a signed longword integer containing the quotient.

**remainder**

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

Remainder. The `remainder` argument is the address of a signed longword integer containing the remainder.

\(^1\) On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$ Routines
LIB$EDIV

Condition Values Returned

SS$_NORMAL Normal successful operation.

SS$_INTDIV Integer divide by zero. The quotient is replaced by bits 31:0 of the dividend, and the remainder is replaced by zero.

SS$_INTOVF Integer overflow. The quotient is replaced by bits 31:0 of the dividend, and the remainder is replaced by zero.

Example

```fortran
C+ This Fortran program demonstrates how to use LIB$EDIV.
C-
INTEGER DIVISOR,DIVIDEND(2),QUOTIENT,REMAINDER
C+
C Find the quotient and remainder of 4600387192 divided by 4096.
C Because 4600387192 is too large to store as a longword, use LIB$EDIV.
C-
DIVISOR = 4096
C+
C The dividend must be represented as a quadword. To do this use a vector
C of length 2. The first element is the low-order longword, and the second
C element is the high-order longword.
C Now, 4600387192 = '00000000112345678'x. So,
C-
DIVIDEND(1) = '12345678'X
DIVIDEND(2) = '000000001'X
C+
C Compute the quotient and remainder of 4600387192 divided by 4096.
C-
RETURN = LIB$EDIV(DIVISOR,DIVIDEND,QUOTIENT,REMAINDER)
TYPE *,'The longword integer quotient of 4600387192/4096 is:'
TYPE *,' ', QUOTIENT
TYPE *,'The longword integer remainder of 4600387192/4096 is:'
TYPE *,' ', REMAINDER
END
```

This Fortran example demonstrates how to call LIB$EDIV. The output generated by this program is as follows:

The longword integer quotient of 4600387192/4096 is:
1123141

The longword integer remainder of 4600387192/4096 is:
1656
LIB$EMODD
Extended Multiply and Integerize Routine for D-Floating-Point Values

The Extended Multiply and Integerize routine (D-Floating-Point Values) allows higher-level language users to perform accurate range reduction of D-floating arguments.

On Alpha and I64 systems, D-floating-point values are not supported in full precision in native OpenVMS Alpha and I64 programs. They are precise to 56 bits on VAX systems, 53 or 56 bits in translated VAX images, and 53 bits in native OpenVMS Alpha and I64 programs.

Format

LIB$EMODD  floating-point-multiplier ,multiplier-extension ,floating-point-multiplicand ,integer-portion ,fractional-portion

Returns

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

Arguments

floating-point-multiplier
OpenVMS usage: floating_point
type: D_floating
access: read only
mechanism: by reference

The multiplier. The floating-point-multiplier argument is a D-floating number.

multiplier-extension
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

The left-justified multiplier-extension bits. The multiplier-extension argument is an unsigned byte.

floating-point-multiplicand
OpenVMS usage: floating_point
type: D_floating
access: read only
mechanism: by reference

The multiplicand. The floating-point-multiplicand argument is a D-floating number.
LIB$ Routines
LIB$EMODD

integer-portion
OpenVMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by reference

The integer portion of the result. The integer-portion argument is the address of a signed longword integer containing the integer portion of the result.

fractional-portion
OpenVMS usage: floating_point
type: D_floating
access: write only
mechanism: by reference

The fractional portion of the result. The fractional-portion argument is a D-floating number.

Description

The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain \(x\) additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted, and a \(y\)-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of \(y\) bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of \(x\) and \(y\) are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>(x)</th>
<th>Bits</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$EMODD</td>
<td>8</td>
<td>7:0</td>
<td>64</td>
</tr>
</tbody>
</table>

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_FLTUND</td>
<td>Floating underflow. The integer and fraction operands are replaced by zero (0).</td>
</tr>
<tr>
<td>SS$_INTOVF</td>
<td>Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS$_INTOVF also.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand. The integer and fraction operands are unaffected.</td>
</tr>
</tbody>
</table>
LIB$EMODF

Extended Multiply and Integerize Routine for F-Floating-Point Values

The Extended Multiply and Integerize routine (F-Floating-Point Values) allows higher-level language users to perform accurate range reduction of F-floating arguments.

Format

LIB$EMODF floating-point-multiplier,multiplier-extension,floating-point-multiplicand,integer-portion,fractional-portion

Returns

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

Arguments

floating-point-multiplier
OpenVMS usage: floating_point
type: F_floating
access: read only
mechanism: by reference

The multiplier. The floating-point-multiplier argument is the address of an F-floating number containing the number.

multiplier-extension
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

The left-justified multiplier-extension bits. The multiplier-extension argument is the address of an unsigned byte containing these multiplier extension bits.

floating-point-multiplicand
OpenVMS usage: floating_point
type: F_floating
access: read only
mechanism: by reference

The multiplicand. The floating-point-multiplicand argument is an F-floating number.

integer-portion
OpenVMS usage: longword_signed
type: longword (signed)
access: write only
mechanism: by reference

The integer portion of the result. The integer-portion argument is the address of a signed longword integer containing the integer portion of the result.
fractional-portion
OpenVMS usage: floating_point
type: F_floating
access: write only
mechanism: by reference

The fractional portion of the result. The fractional-portion argument is the address of an F-floating number containing the fractional portion of the result.

Description
LIB$EMODF allows higher-level language users to perform accurate range reduction of F-floating arguments.

The floating-point multiplier-extension operand (second operand) is concatenated with the floating-point-multiplier (first operand) to gain x additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a y-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of y bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of x and y are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>x</th>
<th>Bits</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$EMODF</td>
<td>8</td>
<td>7:0</td>
<td>32</td>
</tr>
</tbody>
</table>

Condition Values Returned

SS$_NORMAL Routine successfully completed.
SS$_FLTUND Floating underflow. The integer and fraction operands are replaced by zero.
SS$_INTOVF Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS$_INTOVF also.
SS$_ROPRAND Reserved operand. The integer and fraction operands are unaffected.
Extended Multiply and Integerize Routine for G-Floating-Point Values

The Extended Multiply and Integerize routine (G-Floating-Point Values) allows higher-level language users to perform accurate range reduction of G-floating arguments.

Format

```
LIB$EMODG floating-point-multiplier,multiplier-extension,floating-point-multiplicand,integer-portion,fractional-portion
```

Returns

OpenVMS usage: cond_value

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

Arguments

**floating-point-multiplier**

OpenVMS usage: floating_point

- **type:** G_floating
- **access:** read only
- **mechanism:** by reference

The multiplier. The **floating-point-multiplier** argument is a G-floating number.

**multiplier-extension**

OpenVMS usage: word_unsigned

- **type:** word (unsigned)
- **access:** read only
- **mechanism:** by reference

The left-justified multiplier-extension bits. The **multiplier-extension** argument is an unsigned word.

**floating-point-multiplicand**

OpenVMS usage: floating_point

- **type:** G_floating
- **access:** read only
- **mechanism:** by reference

The multiplicand. The **floating-point-multiplicand** argument is a G-floating number.

**integer-portion**

OpenVMS usage: longword_signed

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

The integer portion of the result. The **integer-portion** argument is the address of a signed longword integer containing the integer portion of the result.
fractional-portion
OpenVMS usage: floating_point
type: G_floating
access: write only
mechanism: by reference

The fractional portion of the result. The fractional-portion argument is a G-floating number.

Description

The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain x additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a y-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of y bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of x and y are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>x</th>
<th>Bits</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$EMODG</td>
<td>11</td>
<td>15:5</td>
<td>64</td>
</tr>
</tbody>
</table>

Condition Values Returned

- SS$_NORMAL Routine successfully completed.
- SS$_FLTUND Floating underflow. The integer and fraction operands are replaced by zero.
- SS$_INTOVF Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS$_INTOVF also.
- SS$_ROPRAND Reserved operand. The integer and fraction operands are unaffected.
LIB$EMODH
Extended Multiply and Integerize Routine for H-Floating-Point Values

On OpenVMS VAX systems, the Extended Multiply and Integerize routine (H-Floating-Point Values) allows higher-level language users to perform accurate range reduction of H-floating arguments.

This routine is not available to native OpenVMS Alpha programs but is available to translated VAX images.

Format

```
LIB$EMODH floating-point-multiplier,multiplier-extension,floating-point-multiplicand,integer-portion,fractional-portion
```

Returns

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

Arguments

`floating-point-multiplier`
OpenVMS usage: floating_point
type: H_floating
access: read only
mechanism: by reference

The multiplier. The `floating-point-multiplier` argument is an H-floating number.

`multiplier-extension`
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

The left-justified multiplier-extension bits. The `multiplier-extension` argument is an unsigned word.

`floating-point-multiplicand`
OpenVMS usage: floating_point
type: H_floating
access: read only
mechanism: by reference

The multiplicand. The `floating-point-multiplicand` argument is an H-floating number.
integer-portion
OpenVMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by reference

The integer portion of the result. The integer-portion argument is the address of a signed longword integer containing the integer portion of the result.

fractional-portion
OpenVMS usage: floating_point
type: H_floating
access: write only
mechanism: by reference

The fractional portion of the result. The fractional-portion argument is an H-floating number.

Description
The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain x additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a y-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of y bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of x and y are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>x</th>
<th>Bits</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$EMODH</td>
<td>15</td>
<td>15:1</td>
<td>128</td>
</tr>
</tbody>
</table>

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_FLTUND</td>
<td>Floating underflow. The integer and fraction operands are replaced by zero.</td>
</tr>
<tr>
<td>SS$_INTOVF</td>
<td>Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS$_INTOVF also.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand. The integer and fraction operands are unaffected.</td>
</tr>
</tbody>
</table>
LIB$EMODS (Alpha and I64 Only)

Extended Multiply and Integerize Routine for S-Floating-Point Values

The Extended Multiply and Integerize routine (IEEE S-Floating-Point Values) allows higher-level language users to perform accurate range reduction of IEEE S-float arguments.

Format

LIB$EMODS floating-point-multiplier , multiplier-extension , floating-point-multiplicand , integer-portion , fractional-portion

Returns

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

Arguments

floating-point-multiplier
OpenVMS usage: floating_point  
type: IEEE S_floating  
access: read only  
mechanism: by reference

The multiplier. The floating-point-multiplier argument is the address of an IEEE S-floating number containing the number.

multiplier-extension
OpenVMS usage: byte_unsigned  
type: byte (unsigned)  
access: read only  
mechanism: by reference

The left-justified multiplier-extension bits. The multiplier-extension argument is the address of an unsigned byte containing these multiplier extension bits.

floating-point-multiplicand
OpenVMS usage: floating_point  
type: IEEE S_floating  
access: read only  
mechanism: by reference

The multiplicand. The floating-point-multiplicand argument is an IEEE S-float number.

integer-portion
OpenVMS usage: longword_signed  
type: longword (signed)  
access: write only  
mechanism: by reference

The integer portion of the result. The integer-portion argument is the address of a signed longword integer containing the integer portion of the result.
LIB$ Routines
LIB$EMODS (Alpha and I64 Only)

fractional-portion
OpenVMS usage: floating_point
type: IEEE S_floating
access: write only
mechanism: by reference

The fractional portion of the result. The fractional-portion argument is the address of an IEEE S-floating number containing the fractional portion of the result.

Description
LIB$EMODS allows higher-level language users to perform accurate range reduction of IEEE S-floating arguments.

The floating-point multiplier-extension operand (second operand) is concatenated with the floating-point-multiplier (first operand) to gain \( x \) additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a \( y \)-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of \( y \) bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of \( x \) and \( y \) are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>( x )</th>
<th>Bits</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$EMODS</td>
<td>8</td>
<td>7:0</td>
<td>32</td>
</tr>
</tbody>
</table>

Condition Values Returned

- **SS$_NORMAL**
  - Routine successfully completed.
- **SS$_FLTUND**
  - Floating underflow. The integer and fraction operands are replaced by zero.
- **SS$_INTOVF**
  - Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS$_INTOVF also.
- **SS$_ROPRAND**
  - Reserved operand. The integer and fraction operands are unaffected.
LIB$EMODT (Alpha and I64 Only)
Extended Multiply and Integerize Routine for T-Floating-Point Values

The Extended Multiply and Integerize routine (IEEE T-Floating-Point Values) allows higher-level language users to perform accurate range reduction of IEEE T-floating arguments.

Format

LIB$EMODT floating-point-multiplier ,multiplier-extension ,floating-point-multiplicand ,integer-portion ,fractional-portion

Returns

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

Arguments

floating-point-multiplier
OpenVMS usage: floating_point
type: IEEE T_floating
access: read only
mechanism: by reference

The multiplier. The floating-point-multiplier argument is the address of an IEEE T-floating number containing the number.

multiplier-extension
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

The left-justified multiplier-extension bits. The multiplier-extension argument is the address of an unsigned byte containing these multiplier extension bits.

floating-point-multiplicand
OpenVMS usage: floating_point
type: IEEE T_floating
access: read only
mechanism: by reference

The multiplicand. The floating-point-multiplicand argument is an IEEE T-floating number.

integer-portion
OpenVMS usage: longword_signed
type: longword (signed)
access: write only
mechanism: by reference

The integer portion of the result. The integer-portion argument is the address of a signed longword integer containing the integer portion of the result.
LIB$ Routines
LIB$EMODT (Alpha and I64 Only)

fractional-portion
OpenVMS usage: floating_point
type: IEEE T_floating
access: write only
mechanism: by reference

The fractional portion of the result. The fractional-portion argument is the address of an IEEE T-floating number containing the fractional portion of the result.

Description
LIB$EMODT allows higher-level language users to perform accurate range reduction of IEEE T-floating arguments.

The floating-point multiplier-extension operand (second operand) is concatenated with the floating-point-multiplier (first operand) to gain \( x \) additional low-order fraction bits. The multiplicand is multiplied by the extended multiplier. After multiplication, the integer portion is extracted and a \( y \)-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of \( y \) bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

The values of \( x \) and \( y \) are as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>( x )</th>
<th>Bits</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$EMODT</td>
<td>11</td>
<td>11:0</td>
<td>64</td>
</tr>
</tbody>
</table>

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_FLTUND</td>
<td>Floating underflow. The integer and fraction operands are replaced by zero.</td>
</tr>
<tr>
<td>SS$_INTOVF</td>
<td>Integer overflow. The integer operand is replaced by the low-order bits of the true result. Floating overflow is indicated by SS$_INTOVF also.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand. The integer and fraction operands are unaffected.</td>
</tr>
</tbody>
</table>
LIB$EMUL

Extended-Precision Multiply

The Extended-Precision Multiply routine performs extended-precision multiplication. LIB$EMUL makes the VAX EMUL instruction available as a callable routine.¹

Format

LIB$EMUL longword-integer-multiplier,longword-integer-multiplicand,addend,product

Returns

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

Arguments

longword-integer-multiplier
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Multiplier used by LIB$EMUL in the extended-precision multiplication. The longword-integer-multiplier argument is the address of a signed longword integer containing the multiplier.

longword-integer-multiplicand
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Multiplicand used by LIB$EMUL in the extended-precision multiplication. The longword-integer-multiplicand argument is the address of a signed longword integer containing the multiplicand.

addend
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Addend used by LIB$EMUL in the extended-precision multiplication. The addend argument is the address of a signed longword integer containing the addend.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
**LIB$ Routines**

**LIB$EMUL**

**product**
OpenVMS usage: quadword\_signed  
type: quadword integer (signed)  
access: write only  
mechanism: by reference  

Product of the extended-precision multiplication. The `product` argument is the address of a signed quadword integer into which LIB$EMUL writes the product.

**Description**

The multiplicand argument is multiplied by the multiplier argument giving a double-length result. The addend argument is sign-extended to double-length and added to the result. LIB$EMUL then writes the result into the `product` argument.

**Condition Values Returned**

SS\$_NORMAL Routine successfully completed.

**Example**

```fortran
INTEGER MULT1, MULT2, ADDEND, PRODUCT(2)
C+
C Find the extended precision multiplication of 268435456 times 4096.  
C That is, find the extended precision product of 2**28 times 2**12.  
C Since 268435456 times 4096 is 2**40, a quadword value is needed for  
C the calculation: use LIB$EMUL.  
C-
   MULT1 = 4096
   MULT2 = 268435456
   APPEND = 0
C+
C Compute 268435456*4096.  
C Note that product will be stored as a quadword. This value will be stored  
C in the 2 dimensional vector PRODUCT. The first element of PRODUCT will  
C contain the low order bits, while the second element will contain the high  
C order bits.  
C-
   RETURN = LIB$EMUL(MULT1,MULT2,APPEND,PRODUCT)  
   TYPE *,'PRODUCT(2) =',PRODUCT(2),' and PRODUCT(1) = ',PRODUCT(1)  
   TYPE *,' Note that 256 and 0 represent the hexadecimal value'  
   type *,14H'10000000000'x,' which in turn, represents 2**40.'  
END
```

This Fortran program demonstrates how to use LIB$EMUL. The output generated by this program is as follows:

```
PRODUCT(2) = 256 and PRODUCT(1) = 0
```

Note that 256 and 0 represent the hexadecimal value `10000000000`x, which in turn represents $2^{40}$. 

LIB$ENABLE_CTRL
Enable CLI Interception of Control Characters

The Enable CLI Interception of Control Characters routine requests the calling
process's command language interpreter (CLI) to resume interception of the
selected control characters when they are typed during an interactive terminal
session. LIB$ENABLE_CTRL provides the same function as the DCL command
SET CONTROL.

Format

LIB$ENABLE_CTRL enable-mask [,old-mask]

Returns

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

Arguments

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

Bit mask indicating for which control characters LIB$ENABLE_CTRL is to
enable interception. The enable-mask argument is the address of an unsigned
longword containing this bit mask. Each of the 32 bits corresponds to one of the
32 possible control characters. If a bit is set, the corresponding control character
is intercepted by the CLI. Currently, only bits 20 and 25, corresponding to Ctrl/T
and Ctrl/Y, are recognized.

The following mask is defined in symbol libraries supplied by HP to specify the
value of enable-mask:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hex Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$M_CLI_CTRLT</td>
<td>%X'00100000'</td>
<td>Enables Ctrl/T</td>
</tr>
<tr>
<td>LIB$M_CLI_CTRLY</td>
<td>%X'02000000'</td>
<td>Enables Ctrl/Y</td>
</tr>
</tbody>
</table>

If a set bit does not correspond to a character that the CLI can intercept, an error
is returned.

old-mask
OpenVMS usage: mask_longword
type: longword (unsigned)
access: write only
mechanism: by reference

Previous bit mask. The old-mask argument is the address of an unsigned
longword containing the old bit mask. The old bit mask is of the same form as
enable-mask.
LIB$ ROUTINES
LIB$ENABLE_CTRL

Description

LIB$ENABLE_CTRL provides the functions of the DCL command SET CONTROL. Normally, Ctrl/Y interrupts the current command, command procedure, or image. After a call to LIB$DISABLE_CTRL, Ctrl/Y is treated like Ctrl/U followed by a carriage return. LIB$ENABLE_CTRL restores the normal operation of Ctrl/Y or Ctrl/T.

Both the DCL and MCR CLIs can intercept control characters. See the HP OpenVMS DCL Dictionary for information on how the CLI processes control characters.

LIB$ENABLE_CTRL is supported for use with the DCL or MCR CLIs.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, the error status LIB$_NOCLI is returned.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument. A bit in enable-mask was set which did not correspond to a control character supported by the CLI.</td>
</tr>
<tr>
<td>LIB$_NOCLI</td>
<td>No CLI present. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.</td>
</tr>
<tr>
<td>LIB$_UNECLIERR</td>
<td>Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to your HP support representative.</td>
</tr>
</tbody>
</table>
LIB$ESTABLISH
Establish a Condition Handler

The Establish a Condition Handler routine moves the address of a condition handling routine (which can be a user-written or a library routine) to longword 0 of the stack frame of the caller of LIB$ESTABLISH. †

This routine is not available to native OpenVMS Alpha and I64 programs but is recognized and handled appropriately by most HP high-level language compilers.

Format

LIB$ESTABLISH new-handler

Returns

OpenVMS usage: routine
type: procedure value
access: write only
mechanism: by reference

Previous contents of SF$A_HANDLER (longword 0) of the caller’s stack frame; zero if no handler existed.

Argument

new-handler
OpenVMS usage: procedure
type: procedure value
access: read only
mechanism: by value

Routine to be set up as the condition handler. The new-handler argument is the address of the procedure value to this routine.

Description

LIB$ESTABLISH moves the address of a condition-handling routine to longword 0 of the stack frame of the caller of LIB$ESTABLISH. This condition-handling routine then becomes the caller’s condition handler. LIB$ESTABLISH returns the previous contents of longword 0. This can either be the address of the caller’s previous condition handler or zero if no handler existed.

The new condition handler remains in effect for your routine until you call LIB$REVERT or until control returns to the caller of the routine that called LIB$ESTABLISH. Once this happens, you must call LIB$ESTABLISH again if the same (or a new) condition handler is to be associated with the routine that called LIB$ESTABLISH.

LIB$ESTABLISH modifies the caller’s stack frame.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$ESTABLISH

LIB$ESTABLISH is provided primarily for use with languages without built-in error handling facilities. Do not use LIB$ESTABLISH with languages that provide error handling, such as BASIC, COBOL, Pascal, and PL/I. The language-support library for these languages depends on predefined language-specific handlers, and use of LIB$ESTABLISH with these languages may adversely affect the behavior of your program. See the language documentation for more information about how each language handles errors.

In VAX MACRO, use the following instruction instead of calling LIB$ESTABLISH:

```fortran
MOVAB HANDLER, (FP) ; set handler address
; in current stack frame
```

**Condition Values Returned**

None.

**Example**

```fortran
C+ C This Fortran program demonstrates the use of LIB$ESTABLISH.
C C This is the main program.
C- EXTERNAL LOG_HANDL
   CHARACTER TIMBUF
   OPEN (UNIT=99, FILE = 'ERRLOG', STATUS = 'NEW')
   CALL LIB$ESTABLISH (LOG_HANDL)
   CALL SYS$BINTIM (TIMBUF, TIMADR)
C+ C The rest of the main program would go here.
C- END
   INTEGER*4 FUNCTION LOG_HANDL (SIGARGS, MECHARGS)
   INTEGER*4 SIGARGS (*), MECHARGS (5)
C+ C This is the handler to journal any signaled error messages.
C- INCLUDE '($SSDEF)'
   EXTERNAL PUT_LINE
   LOG HANDL = $S$ RESIGNAL
   CALL SYS$PUTMSG (SIGARGS, PUT_LINE, )
   RETURN
END

C+ C This is the action subroutine.
C- LOGICAL*4 FUNCTION PUT_LINE (LINE)
   CHARACTER*(*)LINE
   PUT LINE = .FALSE.
100 WRITE (99,200)LINE
200 FORMAT (A)
   RETURN
END
```

In this Fortran example, the function `log_handl` is the condition handler for the program, and thus receives control when an error occurs.
LIB$EXPAND_NODENAME
Expand a Node Name to Its Full Name Equivalent

The Expand a Node Name to Its Full Name Equivalent routine expands a node name to its full name equivalent. †

Format

LIB$EXPAND_NODENAME nodename, fullname [,resultant-length]

Returns

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

Arguments

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

Node name to be expanded. The nodename argument contains the address of a descriptor pointing to this node-name string.

The error LIB$_INVARG is returned if nodename contains an invalid node name, points to a null string, or contains more than 1024 characters. The error LIB$_INVSTRDES is returned if nodename is an invalid descriptor.

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

Expanded node name. The fullname argument contains the address of a descriptor pointing to the expanded node-name string. LIB$EXPAND_NODENAME writes the expanded node-name string into the buffer pointed to by the fullname descriptor.

The error LIB$_INVSTRDES is returned if fullname is an invalid descriptor.

The length field of the fullname descriptor is not updated unless fullname is a dynamic descriptor with a length less than the resulting expanded full name. Refer to the OpenVMS RTL String Manipulation (STR$) Manual for dynamic string descriptor usage.

The fullname argument contains an unusable result when LIB$EXPAND_NODENAME returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$EXPAND_NODENAME

resultant-length
OpenVMS usage: word_unsigned
Type: word (unsigned)
Access: write only
Mechanism: by reference

Length of the expanded node name. The resultant-length argument is the address of an unsigned word that contains this length in bytes.

The resultant-length argument contains an unusable result when LIB$EXPAND_NODENAME returns in error.

Description
This routine expands the input node name to its full name equivalent. Input is validated against the supported form of node names. The error LIB$_INVARG is returned if the input node name is invalid.

If the returned full name overflows the buffer pointed to by fullname, the returned full name is truncated, and the alternate successful status LIB$_STRTRU is returned. The resultant-length argument is set to the value of the length field of the fullname descriptor if this argument is supplied.

If the length of the returned full name is less than or equal to the output buffer, the expanded full name is returned in fullname. Resultant-length is set to the actual length of the expanded full name if this argument is supplied.

In a DECnet environment, expanding a DECnet-Plus node name results in the error condition LIB$_INVARG.

LIB$EXPAND_NODENAME uses the underlying network directory services to look up the full name. In a DECnet-Plus for OpenVMS environment, LIB$EXPAND_NODENAME verifies the existence of the expanded full name in the naming environment. If the expanded full name does not exist in the naming environment, an error condition is returned from the underlying network services and is propagated back to the caller of LIB$EXPAND_NODENAME.

It is recommended that applications use full names instead of the short form of full names whenever possible. Because the short form of a full name is intended to be used only in a specific naming environment, make sure the short form of a full name is expanded in the right naming environment to avoid ambiguity. See LIB$COMPRESS_NODENAME for more information about where and when to use the short form of a full name.

Any error resulting from calling the underlying network services is propagated and returned as condition values in this routine.

LIB$EXPAND_NODENAME supports any string class for the nodename and fullname string arguments.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. Characters are truncated in the output buffer pointed to by the fullname descriptor.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$EXPAND_NODENAME

LIB$_INVARG Invalid argument:
  • **nodename** is invalid.
  • **nodename** points to a null string.
  • The length of the node name is more than 1024 characters.
  • The expanded DECnet Phase V node name is invalid in a DECnet for OpenVMS environment.

LIB$_INVSTRDES Invalid string descriptor.
LIB$_WRONUMARG Wrong number of arguments.

Any condition value returned by RTL routine LIB$SCOPY_R_DX or DECnet service $IPC.
LIB$ Routines
LIB$EXTV

LIB$EXTV
Extract a Field and Sign-Extend

The Extract a Field and Sign-Extend routine returns a sign-extended longword field that has been extracted from the specified variable bit field. LIB$EXTV makes the VAX EXTV instruction available as a callable routine.  

Format

```
LIB$EXTV position, size, base-address
```

Returns

OpenVMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by value

Field extracted by LIB$EXTV, sign-extended to a longword.

Arguments

position

OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Position (relative to the base address) of the first bit in the field that LIB$EXTV extracts. The position argument is the address of a signed longword integer containing the position.

size

OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

Size of the bit field LIB$EXTV extracts. The size argument is the address of an unsigned byte containing the size. The maximum size is 32 bits.

base-address

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

Base address of the bit field LIB$EXTV extracts from the specified variable bit field. The base-address argument is an unsigned longword containing the base address.

1 On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Description

The variable-length bit field is an OpenVMS data type used to store small integers packed together in a larger data structure. It is often used to store single flag bits.

Three scalar attributes define a variable bit field:

- The base address is the address of a byte in memory that serves as a reference point for locating the bit field.
- The bit position is a signed longword containing the displacement of the least significant bit of the field with respect to bit 0 of the base address.
- The size is a byte integer indicating the size of the bit field in bits (in the range $0 \leq \text{size} \leq 32$). That is, a bit field can be no more than one longword in length.

A variable-length bit field has the following format. The area containing asterisks indicates the field.

```
P+S−1 P 0
************
```

- **S** = Size of Field in Bits
- **P** = Bit Displacement of Field from Bit Zero of Address A

Bit fields are zero-origin, which means that the routine regards the first bit in the field as being the zero position.

Condition Value Signaled

SS$_{ROPRAND}$  
A reserved operand fault occurs if a size greater than 32 is specified.

Example

```
SIGN_EXTEND: ROUTINE OPTIONS (MAIN);
DECLARE LIB$EXTV ENTRY
(FIXED BINARY (31), /* Address of longword containing */
  /* beginning bit position */
FIXED BINARY (7), /* Address of byte containing size */
  /* of field */
FIXED BINARY (31)) /* Address of field */
RETURNS (FIXED BINARY (31)); /* Return value */
DECLARE (VALUE, SMALL_INT) FIXED BINARY (31);
ON ENDFILE (SYSIN) STOP;
```
LIB$ Routines
LIB$EXTV

DO WHILE ('1'b);
   /* Loop continuously, until end of file */
   PUT SKIP(2);
   GET LIST (VALUE) OPTIONS (PROMPT ('Value: '));
   SMALL_INT = LIB$EXTV (0, 4, VALUE); /* Extract and sign-extend */
   /* first 4 bits */
   PUT SKIP LIST (VALUE, SMALL_INT);
END;

END SIGN_EXTEND;

This PL/I program extracts a field and returns it sign-extended into a longword.
LIB$EXTZV
Extract a Zero-Extended Field

The Extract a Zero-Extended Field routine returns a longword zero-extended field that has been extracted from the specified variable bit field. LIB$EXTZV makes the VAX EXTZV instruction available as a callable routine. ¹

Format

LIB$EXTZV position ,size ,base-address

Returns

OpenVMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by value

Field extracted by LIB$EXTZV, zero-extended to a longword.

Arguments

position
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Position (relative to the base address) of the first bit in the field LIB$EXTZV extracts. The position argument is the address of a signed longword integer containing the position.

size
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

Size of the bit field LIB$EXTZV extracts. The size argument is the address of an unsigned byte containing the size. The maximum size is 32 bits.

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

Base address of the bit field LIB$EXTZV extracts. The base-address argument is an unsigned longword containing the base address.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Description

The variable-length bit field is an OpenVMS data type used to store small integers packed together in a larger data structure. It is often used to store single flag bits.

Three scalar attributes define a variable bit field:

- The base address is the address of the byte in memory that serves as a reference point for locating the bit field.
- The bit position is a signed longword containing the displacement of the least significant bit of the field with respect to bit 0 of the base address.
- The size is a byte integer indicating the size of the bit field in bits (in the range $0 \leq \text{size} \leq 32$). That is, a bit field can be no more than one longword in length.

A variable-length bit field has the following format. The area containing asterisks indicates the field.

```
P+S−1  P  0
**************
```

\[ S = \text{Size of Field in Bits} \]
\[ P = \text{Bit Displacement of Field from Bit Zero of Address A} \]

Bit fields are zero-origin fields, which means that the routine regards the first bit in the field as being the zero position.

Condition Value Signaled

SS$_$ROPRAND

A reserved operand fault occurs if a size greater than 32 is specified.
LIB$FFx
Find First Clear or Set Bit

The Find First Clear or Set Bit routines search the field specified by the start position, size, and base for the first clear or set bit. LIB$FFC and LIB$FFS make the VAX FFC and VAX FFS instructions available as callable routines. ¹

Format

LIB$FFC position ,size ,base ,find-position
LIB$FFS position ,size ,base ,find-position

Returns

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

Arguments

position
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Starting position, relative to the base address, of the bit field to be searched by LIB$FFx. The position argument is the address of a signed longword integer containing the starting position.

size
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

Number of bits to be searched by LIB$FFx. The size argument is the address of an unsigned byte containing the size of the bit field to be searched. The maximum size is 32 bits.

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

The base argument is the address of the bit field that LIB$FFx searches.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$ Routines
LIB$FFx

**find-position**
- OpenVMS usage: longword_signed
- type: longword integer (signed)
- access: write only
- mechanism: by reference

Bit position of the first bit in the specified state (clear or set), relative to the base address. The **find-position** argument is the address of a signed longword integer into which LIB$FFC writes the position of the first clear bit and into which LIB$FFS writes the position of the first set bit.

**Description**
LIB$FFC searches the field specified by the start position, size, and base for the first clear bit. LIB$FFS searches the field for the first set bit.

If a bit in the specified state is found, LIB$FFx writes the position (relative to the base) of that bit into **find-position** and returns a success status. If no bits are in the specified state or if **size** is zero, LIB$FFx returns LIB$_NOTFOU and sets **find-position** to the starting position plus the size.

LIB$FFx regards the first bit in the field as being the zero position.

**Condition Values Returned**
- SS$_NORMAL Routine successfully completed. A bit in the specified state was found.
- LIB$_NOTFOU A bit in the specified state was not found.

**Condition Value Signaled**
- SS$_ROPRAND Reserved operand fault. A size greater than 32 was specified.
LIB$FID_TO_NAME
Convert Device and File ID to File Specification

The Convert Device and File ID to File Specification routine converts a disk device name and file identifier to a file specification.

Format

LIB$FID_TO_NAME device-name ,file-id ,filespec [,filespec-length] [,directory-id] [,acp-status]

Returns

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

Arguments

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

Device name to be converted. The device-name argument is the address of a descriptor pointing to the device name. It must reference a disk device, and must contain 64 characters or less. LIB$FID_TO_NAME obtains device-name from the NAM$T_DVI field of an OpenVMS RMS name block.

file-id
OpenVMS usage: vector_word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference, array reference

Specifies the file identifier. The file-id argument is the address of an array of three words containing the file identification. LIB$FID_TO_NAME obtains file-id from the NAM$W_FID field of an OpenVMS RMS name block. The $FIDDEF macro defines the structure of file-id.

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

Receives the file specification. The filespec argument is the address of a descriptor pointing to the file specification string. As of OpenVMS Version 7.2, the maximum file specification string that can be returned is 4095 bytes on Alpha and I64 systems, and 510 bytes on VAX systems. On versions prior to Version 7.2, the maximum is 510 bytes on both platforms. Refer to the Description section for more information about the file specification returned.
filespec-length
OpenVMS usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Receives the number of characters written into filespec, excluding padding in the case of a fixed-length string. The optional filespec-length argument is the address of an unsigned word containing the number of characters.

If the output string is truncated to the number of characters specified in filespec, then filespec-length is set to that truncated size. Therefore, you can always use filespec-length to access a valid substring of filespec.

directory-id
OpenVMS usage: vector_word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference, array reference

Specifies a directory file identifier. The directory-id argument is the address of an array of three words containing the directory file identifier. LIB$FID_TO_NAME obtains this array from the NAM$W_DID field of an OpenVMS RMS name block. The $FIDDEF macro defines the structure of directory-id.

This parameter is relevant only for a structure level 1 disk on OpenVMS VAX systems. This parameter is ignored on OpenVMS Alpha and I64 systems because level-1 disks are not supported on OpenVMS Alpha and I64 systems.

acp-status
OpenVMS usage: longword_unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

The status resulting from traversing the backward links. The optional acp-status argument is the address of an unsigned longword containing the status.

Description
LIB$FID_TO_NAME converts a disk device name and file identifier to a file specification by requesting the ACP file specification attribute.

On OpenVMS Alpha and I64 systems, if the file specification is longer than can be accommodated by the filespec buffer, a directory in the path may be replaced by a DID abbreviation (see the Guide to OpenVMS File Applications). If the file specification, even after DID abbreviation, is longer than can be accommodated by the buffer, the file specification is truncated, and LIB$STRTRU is returned as an alternate success status.

On OpenVMS VAX systems, if you use the LIB$FID_TO_NAME routine on a structure level 1 disk, specify the directory-id argument to ensure proper operation of the routine.

LIB$FID_TO_NAME uses the directory backpointer stored in the file header. With files in SYS$COMMON, the directory structure is duplicated because of some SET FILE/ENTERs of directory names. If directory names have been renamed or the tree structure modified (which the OpenVMS operating system
does with the [SYCOMMON] tree), the file specification returned by this routine may not be useful.

LIB$FID_TO_NAME stores the output arguments (filespec, filespec-length, and acp-status) only if the routine successfully finishes.

_________________________ Note ____________________________

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.

________________________

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$STRTRU</td>
<td>Output string truncated (qualified success).</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Required argument omitted, or <strong>device-name</strong> is longer than 64 characters.</td>
</tr>
<tr>
<td>LIB$_INVFILSPE</td>
<td>The <strong>device-name</strong> argument does not reference a disk.</td>
</tr>
</tbody>
</table>

Any condition value returned by RTL routine LIB$ANALYZE_SDESC, or the $ASSIGN, $QIO, or $DASSGN system services.
LIB$FILE_SCAN

File Scan

The File Scan routine searches an area, such as a directory, for all files matching the file specification given and transfers program execution to the specified user-written routine. Wildcards are acceptable. An action routine is called for each file and/or error found. LIB$FILE_SCAN allows the search sequence to continue even if an error occurs while processing a particular file.

Format

LIB$FILE_SCAN  fab ,user-success-procedure ,user-error-procedure [,context]

Returns

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

Arguments

fab
OpenVMS usage: fab
type: unspecified
access: read only
mechanism: by reference

File Access Block (FAB) referencing a valid NAM block or NAML block. The fab argument is the address of the FAB that contains the address and length of the file specification being searched for by LIB$FILE_SCAN. On Alpha and I64 systems, NAML blocks support the use of file specifications with a maximum length of NAML$C_MAXRSS. See the OpenVMS Record Management Services Reference Manual for information on NAML blocks.

user-success-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied success routine that LIB$FILE_SCAN calls when a file is found. The success routine is invoked with the FAB address that was passed to LIB$FILE_SCAN. The user context may be passed to this routine using the FAB$L_CTX field in the FAB.

user-error-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied error routine that LIB$FILE_SCAN calls when it encounters an error. The error routine is called with the FAB argument that was passed to LIB$FILE_SCAN.
context
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference

Default file context used in processing file specifications for multiple input files. The context argument is the address of a longword, which must be initialized to zero by your program before the first call to LIB$FILE_SCAN. After the first call, LIB$FILE_SCAN maintains this longword. You must not change the value of context in subsequent calls to LIB$FILE_SCAN.

Name blocks and file specification strings are allocated by LIB$FILE_SCAN, and context is used to retain their addresses so they may be deallocated later. If the context argument is not passed, unspecified portions of the file specification will be inherited from the previous file specification processed, rather than from multiple input file specifications.

Description
LIB$FILE_SCAN is called with the address of a File Access Block (FAB) and calls an action routine for each file found and/or error returned. LIB$FILE_SCAN allows the search sequence to continue even if an error occurs while processing a particular file.

If this routine is called once for each file specification argument in a command line, portions of the file specifications which are not specified by the user are inherited from the last files processed.

On Alpha and I64 systems, support for a file specification greater than 255 characters is provided by the use of NAML blocks rather than NAM blocks. See the OpenVMS Record Management Services Reference Manual for information on NAML blocks.

You must call LIB$FILE_SCAN_END before initiating a new sequence of calls to LIB$FILE_SCAN.

Condition Values Returned

Any condition value returned by the RMS Parse service.
LIB$ Routines
LIB$FILE_SCAN_END

LIB$FILE_SCAN_END
End-of-File Scan

The End-of-File Scan routine is called after each sequence of calls to LIB$FILE_SCAN. LIB$FILE_SCAN_END deallocates any saved OpenVMS RMS context and/or deallocates the virtual memory that had been allocated for holding the related file specification information.

Format

LIB$FILE_SCAN_END [fab] [,context]

Returns

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

Arguments

fab
OpenVMS usage: fab
type: unspecified
access: modify
mechanism: by reference

File access block (FAB) used with LIB$FILE_SCAN. The optional fab argument is the address of the FAB that contains the address and length of the file specification.

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

Temporary default context used in LIB$FILE_SCAN. The optional context argument is the address of a longword containing this temporary default context.

Description

Your program should call LIB$FILE_SCAN_END after each sequence of calls to LIB$FILE_SCAN. The function that LIB$FILE_SCAN_END performs depends upon the arguments you specify. If you specify fab, LIB$FILE_SCAN_END parses the null string to deallocate any saved RMS context. If you specify context, LIB$FILE_SCAN_END deallocates any virtual memory that was allocated for holding the related file specification information. If you specify both fab and context, LIB$FILE_SCAN_END performs both functions. However, if you do not specify either argument, LIB$FILE_SCAN_END does nothing.
If LIB$FILE_SCAN is directed to process the specifications for multiple input files, LIB$FILE_SCAN_END is used to deallocate those saved file specifications. If LIB$FILE_SCAN_END is called by your program after each sequence of calls to LIB$FILE_SCAN, it will prevent the defaults from the previous call from affecting context value in the next call to LIB$FILE_SCAN. LIB$FILE_SCAN_END does this by replacing the context value passed to it with a temporary context value that your program passes to LIB$FILE_SCAN the next time it is called.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>RMS$_FAB</td>
<td>The <strong>fab</strong> argument is not the address of a valid FAB.</td>
</tr>
</tbody>
</table>
LIB$ FIND_FILE
Find File

The Find File routine is called with a file specification for which it searches. LIB$ FIND_FILE returns one file specification for each call. The file specification may contain wildcards.

Format

LIB$FIND_FILE "filenamespec", "resultant-filenamespec", "context", 
["default-filenamespec"], 
["related-filenamespec"], 
["status-value"], 
[, "flags"]

Returns

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

Arguments

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

File specification, which may contain wildcards, that LIB$FIND_FILE uses to search for the desired file. The filenamespec argument is the address of a descriptor pointing to the file specification. If running on Alpha or I64 and flag LIB$M_FILE_LONG NAMES is set, the maximum length of a file specification is specified by NAML$C_MAXRSS, otherwise the maximum length of a file specification is 255 bytes.

The file specification used may also contain a search list logical name. If present, the search list logical name elements can be used as accumulative to related file specifications, so that portions of file specifications not specified by the user are inherited from previous file specifications.

resultant-filenamespec
OpenVMS usage: char_string
type: character string
access: modify
mechanism: by descriptor

Resultant file specification that LIB$FIND_FILE returns when it finds a file that matches the specification in the filenamespec argument. The resultant-filenamespec argument is the address of a descriptor pointing to the resultant file specification.

ccontext
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference
A longword integer variable into which the routine stores a context value for use by future calls to LIB$FIND_FILE or LIB$FIND_FILE_END. The context argument is an unsigned longword integer containing the address of the context. This variable must be set to zero before the first call to LIB$FIND_FILE. You can use the same context argument from one LIB$FIND_FILE call to another provided you have not called LIB$FIND_FILE_END for that context first. LIB$FIND_FILE uses this argument to retain the context when processing multiple input files. Portions of file specifications that the user does not specify may be inherited from the last files processed because the file contexts are retained in this argument. You must not change the value of context in subsequent calls to LIB$FIND_FILE.

default-filespec
OpenVMS usage: char_string

type: character string

access: read only

mechanism: by descriptor

Default file specification. The default-filespec argument is the address of a descriptor pointing to the default file specification. See the OpenVMS Record Management Services Reference Manual for information about default file specifications.

related-filespec
OpenVMS usage: char_string

type: character string

access: read only

mechanism: by descriptor

Related file specification containing the context of the last file processed. The related-filespec argument is the address of a descriptor pointing to the related file specification.

The related file specification is useful when you are processing lists of file specifications. Unspecified portions of the file specification are inherited from the last file processed. For more information on related file specifications, see the OpenVMS Record Management Services Reference Manual.

status-value
OpenVMS usage: longword_unsigned

type: longword (unsigned)

access: write only

mechanism: by reference

RMS secondary status value from a failing RMS operation. The status-value argument is an unsigned longword containing the address of a longword-length buffer to receive the RMS secondary status value (usually returned in the file access block field, FAB$L_STV).

flags
OpenVMS usage: mask_longword

type: longword (unsigned)

access: read only

mechanism: by reference

User flags. The flags argument is the address of an unsigned longword containing the user flags.
The flag bits and their corresponding symbols are described in the following table:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIB$M_FIL_NOWILD</td>
<td>If set, LIB$FIND_FILE returns an error if a wildcard character is input.</td>
</tr>
<tr>
<td>1</td>
<td>LIB$M_FIL_MULTIPLE</td>
<td>If set, this performs temporary defaulting for multiple input files and the related-filespec argument is ignored. See description of context in LIB$FILE_SCAN. Each time LIB$FIND_FILE is called with a different file specification, the specification from the previous call is automatically used as a related file specification. This allows parsing of the elements of a search-list logical name such as DISK2:[SMITH] FIL1.TYP,FIL2.TYP, and so on. Use of this feature is required to get the desired defaulting with search list logical name. LIB$FIND_FILE_END must be called between each command line in interactive use or the defaults from the previous command line affect the current file specification.</td>
</tr>
<tr>
<td>2</td>
<td>LIB$M_FIL_LONG_NAMES</td>
<td>(Alpha and I64 only) If set, LIB$FIND_FILE can process file specifications with a maximum length of NAML$C_MAXRSS. If clear, LIB$FIND_FILE can process file specifications with a maximum length of 255 (default).</td>
</tr>
</tbody>
</table>

**Description**

LIB$FIND_FILE returns one file specification per call unless it fails to find the target file specification. In this case, the routine returns the condition value RMS$NMF (no more files). Each successful call to LIB$FIND_FILE results in a new resultant-filespec.

When you call LIB$FIND_FILE repeatedly using the same context, filespec is saved only if you set the MULTIPLE bit. If you specify a different filespec on your next call and set the MULTIPLE bit, the file specification from the previous call defaults as the related file specification.

For each LIB$FIND_FILE call, RMS first applies the defaults from default-filespec and then uses the defaults from related-filespec, if relevant. Default file specifications are used only if components are missing from the filespec argument and the needed components are found in default-filespec. The related-filespec argument is used when you process lists of file specifications. Unspecified portions of the file specification are inherited from the last file processed. This provides an extra level of file specification defaults. For additional information on related file specifications and input file parsing, see the Guide to OpenVMS File Applications.
The `filespec` argument can contain wildcard characters. `LIB$FIND_FILE` can be called repeatedly using the same `context` argument until the error `RMS$_NMF` (no more files) is returned.

`LIB$FIND_FILE` searches for a certain wildcard file specification and returns all file specifications that satisfy that wildcard file specification.

If you make multiple calls to `LIB$FIND_FILE`, be aware of the following behavior:

- If the `NOWILD` bit is not set and the file specification does not contain any wildcard characters, `LIB$FIND_FILE` returns the appropriate file name on the first call and the condition value `RMS$_NMF` on the next call.
- If the `NOWILD` bit is set and you use the same nonwildcard file specification, `LIB$FIND_FILE` returns the file name on the first call as well as each subsequent call.

On Alpha and I64 systems, support for file specifications longer than 255 characters is provided only when the `LIB$M_FIL_LONG_NAMES` flag is set in the `flags` argument. When this flag is set, a NAML block (rather than a NAM block) is part of the context, and file specifications can have a maximum length of NAML$C_MAXRSS. See the *OpenVMS Record Management Services Reference Manual* for information on NAML blocks.

You must call `LIB$FIND_FILE_END` before initiating a new sequence of calls to `LIB$FIND_FILE` to properly deallocate all of the internal data structures that were allocated in the calls to `LIB$FIND_FILE`. After you call `LIB$FIND_FILE_END`, the context value is no longer valid and cannot be used on any subsequent `LIB$FIND_FILE` calls.

If the error `RMS$_CHN` is returned, RMS has no more channels to assign. There are two possible reasons for this:

- You did not call `LIB$FIND_FILE_END` before initiating a new call with a context variable to `LIB$FIND_FILE`. (This is the most common reason.)
- The system parameter `CHANNELCNT` is too low.

**Condition Values Returned**

- `RMS$_NORMAL` Routine successfully completed.
- `LIB$_NOWILD` A wildcard character was present in the file specification parsed, and the wildcard flag bit was set to no wildcard. (This is actually the SHR$_NOWILD error message after application of the `LIB$` facility code.)
- `RMS$_CHN` No more channels.
- `RMS$_NMF` No more files.

Any condition value returned by RMS Parse and Search services, `LIB$GET_VM`, `LIB$GET_VM_64`, `LIB$FREE_VM`, `LIB$FREE_VM_64`, `LIB$SCOPY_R_DX`, or `LIB$SCOPY_R_DX_64`. 
LIB$ FIND_FILE_END
End of Find File

The End of Find File routine is called once after each sequence of calls to LIB$ FIND_FILE. LIB$ FIND_FILE_END deallocates any saved OpenVMS RMS context and deallocates the virtual memory used to hold the allocated context block.

Format

LIB$ FIND_FILE_END context

Returns

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

Argument

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

Zero or the address of a FAB/NAM buffer from a previous call to LIB$ FIND_FILE. The context argument is the address of a longword that contains this context.

Description

LIB$ FIND_FILE_END should be called by your program after each sequence of calls to LIB$ FIND_FILE. This will prevent the default values from the previous call from affecting the next file specification.

LIB$ FIND_FILE_END deallocates the context used in the last call to LIB$ FIND_FILE so that the context retained will not be used in subsequent calls to LIB$ FIND_FILE. If LIB$ FIND_FILE was directed to process file specifications for multiple input files, the saved file specifications are also deallocated.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
RMS$_FAB File access block argument is not the address of a valid FAB.
LIB$FIND_IMAGE_SYMBOL
Find Universal Symbol in Shareable Image File

The Find Universal Symbol in Shareable Image File routine reads universal symbols from the shareable image file. This routine then dynamically activates a shareable image into the P0 address space of a process.

Format

LIB$FIND_IMAGE_SYMBOL filename ,symbol ,symbol-value [,image-name] [,flags]

Returns

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

Arguments

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

Name of the file for which LIB$FIND_IMAGE_SYMBOL is searching. The filename argument is the address of a descriptor pointing to this file name string. This argument may contain only the file name. File type cannot be indicated. If any file specification punctuation characters (:, [ < ; .) are present, the error SS$IVLOGNAM is returned.

You can specify a file specification for the image name with the optional image-name argument. If you do not specify image-name, a default file specification of SYS$SHARE:.EXE is applied to the file name. If the file is not in SYS$SHARE:.EXE, a logical name must be used to direct this routine to locate the correct file. Only logical names defined in the system logical name table with the /EXEC attribute will be considered while the image activator is processing a request from an image that was installed with privileges. If the calling image was installed with privileges, the image being activated and any shareable images or message sections it references must be installed as a known image with the INSTALL utility. Running an image to which you have only Execute (not Read) access results in the same restrictions on logical names and shareable images as does running a privileged image.

On VAX systems, the filename descriptor must be class D, S, or Z.

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

Symbol for which LIB$FIND_IMAGE_SYMBOL is searching in the filename file. The symbol argument is the address of a descriptor pointing to the symbol name
string. The symbol name string can be input in uppercase, lowercase, or mixed case letters.

**symbol-value**

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

Symbol value that LIB$FIND_IMAGE_SYMBOL has located. The `symbol-value` argument is the address of a signed longword integer into which LIB$FIND_IMAGE_SYMBOL returns the symbol value. If the symbol is relocatable, the starting virtual address of the shareable image in memory is added to the symbol value.

**image-name**

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

Default file specification applied to the image name. The optional `image-name` argument is a string used as the RMS default file specification when parsing `filename` as the primary filename. If `image-name` is not supplied, then a default file specification of SYS$SHARE:.EXE is applied to the image name.

On VAX systems, the `image-name` descriptor must be class D, S, or Z.

**flags**

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

Control flags. The `flags` argument is the address of a longword integer that contains the control flags.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved to HP</td>
<td>Reserved to HP</td>
</tr>
<tr>
<td>1</td>
<td>Reserved to HP</td>
<td>Reserved to HP</td>
</tr>
<tr>
<td>2</td>
<td>Reserved to HP</td>
<td>Reserved to HP</td>
</tr>
<tr>
<td>3</td>
<td>Reserved to HP</td>
<td>Reserved to HP</td>
</tr>
<tr>
<td>4</td>
<td><code>LIB$M_FIS_MIXEDCASE</code></td>
<td>Causes <code>LIB$FIND_IMAGE_SYMBOL</code> to look for the symbol without converting it to uppercase.</td>
</tr>
</tbody>
</table>

This is an optional argument. If omitted, the default is 0. If omitted, or if `LIB$M_FIS_MIXEDCASE` (bit 4) is 0, `LIB$FIND_IMAGE_SYMBOL` converts the specified symbol to uppercase before it is used.
Description

The shareable image that LIB$FIND_IMAGE_SYMBOL activates must have been already linked and must be position independent. You must have read access to the shareable image file to use this routine.

LIB$FIND_IMAGE_SYMBOL writes the symbol value that it has located into the symbol-value argument.

After the first call to LIB$FIND_IMAGE_SYMBOL for a particular image, successive calls for that image are processed quickly. The image is activated only once and an in-memory database is maintained. There is no way to deallocate this database, nor is there any supported method to remove an activated image from the address space. All images are activated into P0 space.

LIB$FIND_IMAGE_SYMBOL locates the universal symbol in its database qualified by the file name exactly as given in the filename argument. Therefore, a reference to a lexically different but equivalent file name causes a new copy of the same shareable image to be loaded and searched. To avoid this situation, always specify the desired file name in the same form.

To work properly with translated VAX images on Alpha and I64 systems, LIB$FIND_IMAGE_SYMBOL may modify the name of the file being searched and may retry the search if the first search failed. If called from a translated image, LIB$FIND_IMAGE_SYMBOL appends “_TV” to the file name before searching. This locates the translated version of the image being searched. If the search fails to find the file or the file does not define the symbol, LIB$FIND_IMAGE_SYMBOL tries again with the unmodified original file name. This locates the native Alpha or I64 version of the image. If the second search also fails, an error is returned. If LIB$FIND_IMAGE_SYMBOL is called from a native Alpha or I64 program, the order of the searches is reversed. The first search is done with the unmodified original file name. If that fails, the second search is done with “_TV” appended to the file name. If the second search fails, an error is returned.

LIB$FIND_IMAGE_SYMBOL disables AST recognition while it is executing. AST recognition is reenabled before returning to the caller only if AST recognition was previously enabled.

LIB$FIND_IMAGE_SYMBOL signals all errors and returns the status in R0.

LIB$FIND_IMAGE_SYMBOL may signal a warning (LIB$EOMWARN) to indicate that the image being activated contains modules that had compilation warnings. A condition handler used with LIB$FIND_IMAGE_SYMBOL should probably handle this as a special case.

To allow LIB$FIND_IMAGE_SYMBOL to continue executing after signaling LIB$EOMWARN, the condition handler should exit with SS$CONTINUE. For this reason, you may choose not to use LIB$SIG_TO_RET as a condition handler for LIB$FIND_IMAGE_SYMBOL.
LIB$ Routines
LIB$FIND_IMAGE_SYMBOL

Condition Values Returned

LIB$$_BADCCC Illegal compilation code.
LIB$$_EOMERROR Compilation errors.
LIB$$_EOMFATAL Fatal compilation errors.
LIB$$_EOMWARN Compilation warnings.
LIB$$_GSOTYP Illegal universal symbol directory record type.
LIB$$_ILLFMLCNT Maximum argument count exceeds maximum for routine.
LIB$$_ILLMODNAM Illegal module name length.
LIB$$_ILLPSCLLEN Illegal program section length.
LIB$$_ILLRECLLEN Illegal record length in module.
LIB$$_ILLRECLN2 Illegal record length.
LIB$$_ILLRECTYPE Illegal record type in module.
LIB$$_ILLRECTYPE2 Illegal record type.
LIB$$_ILLSYMLEN Illegal symbol length.
LIB$$_NOEOM No end of module record contained in the module.
LIB$$_RECTOOSML Record too small; data overflows object record in module.
LIB$$_SEQUENCE Illegal record sequence in module.
LIB$$_SEQUENCE2 Illegal record sequence.
LIB$$_STRVL Illegal object language structure level in module.

Note that all of the above error messages indicate a format error in the shareable image.
LIB$$_INSVIRMEM Insufficient virtual memory.
SS$$_IVLOGNAM The filename argument contained more than just a file name; a device or directory specification was found in the string.

Any condition values returned by RTL routines LIB$INSERT_TREE or LIB$LOOKUP_TREE.

Any condition values returned by RMS.
LIB$FIND_VM_ZONE
Return the Next Valid Zone Identifier

The Return the Next Valid Zone Identifier routine returns the zone identifier of the next valid zone in the heap management 32-bit database. †

Format

LIB$FIND_VM_ZONE context,zone-id

Returns

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

Arguments

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

Context specifier. The context argument is the address of an unsigned longword used to keep the scan context for finding the next valid zone. The context argument must be 0 to initialize the scan and to start with the first returnable zone identifier.

zone-id
OpenVMS usage: identifier
type: longword (unsigned)
access: write only
mechanism: by reference

Zone identifier. The zone-id argument is the address of an unsigned longword that receives the zone identifier for the next zone.

Description

At each call, LIB$FIND_VM_ZONE scans the heap management 32-bit zone database and returns the zone-id of the next valid zone. (The first and second calls to LIB$FIND_VM_ZONE return the zone-id of the 32-bit default zone and the 32-bit string zone, respectively.) This capability allows a program to deal with each 32-bit VM zone created during the invocation, including those created outside of the program.

Note

LIB$FIND_VM_ZONE finds only 32-bit zones. You must use LIB$FIND_VM_ZONE_64 to loop through all VM zones.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
The \texttt{context} argument controls the state of the scan. It determines what zone to return (the first, the next, and so forth). On the initial call, specified by \texttt{context}=0, \texttt{LIB$\text{VERIFY_VM_ZONE}} is called to verify the heap management zone database. If the database is corrupt, further calls to this routine will produce no additional useful output.

When no more zones can be found, the routine returns the condition value \texttt{LIB$\_NOTFOU}.

If a zone has been corrupted in some major way (for example, if the validity code has been changed), then this routine may not be able to locate it in the zone database.

Note that ASTs may be disabled while \texttt{LIB$\text{FIND_VM_ZONE}} is executing code that depends on the stability of the heap management zone database. In general it is the caller's responsibility to ensure that the calling program has exclusive access to the zone database while scanning for multiple zones with this routine. Results are unpredictable if another thread of control modifies the zone database or the associated areas during the scanning.

\textbf{Condition Values Returned}

- \texttt{SS$\_NORMAL} Routine successfully completed.
- \texttt{LIB$\_BADZONE} Invalid zone.
- \texttt{LIB$\_NOTFOU} Zone identifier not found (alternate success status).
- \texttt{LIB$\_WRONUMARG} Wrong number of arguments.

\textbf{Example}

```fortran
IMPLICIT NONE
INTEGER*4 status,context,zone_id
INTEGER*4 lib$find_vm_zone,lib$show_vm_zone
context = 0
status = lib$find_vm_zone (context, zone_id)
DO WHILE (status)
   print *
   status = lib$show_vm_zone (zone_id, 0)
   status = lib$find_vm_zone (context, zone_id)
END DO
END
```

Sample output for this Fortran program is shown below:

```
Zone Id = 00020020, Zone name = "DEFAULT_ZONE"
Zone Id = 000200B0, Zone name = "STRING_ZONE"
```
LIB$FIND_VM_ZONE_64 (Alpha and I64 Only)
Return the Next Valid Zone Identifier

The Return the Next Valid Zone Identifier routine returns the zone identifier of the next valid zone in the heap management 64-bit database.

Format

LIB$FIND_VM_ZONE_64 context ,zone-id

Returns

OpenVMS usage: cond_value

type: longword (unsigned)

access: write only

mechanism: by value

Arguments

context

OpenVMS usage: context

type: quadword (unsigned)

access: modify

mechanism: by reference

Context specifier. The context argument is the address of an unsigned quadword used to keep the scan context for finding the next valid zone. The context argument must be 0 to initialize the scan and to start with the first returnable zone identifier.

zone-id

OpenVMS usage: identifier

type: quadword (unsigned)

access: write only

mechanism: by reference

Zone identifier. The zone-id argument is the address of an unsigned quadword that receives the zone identifier for the next zone.

Description

At each call, LIB$FIND_VM_ZONE_64 scans the heap management 64-bit zone database and returns the zone-id of the next valid zone. (The first and second calls to LIB$FIND_VM_ZONE_64 return the zone-id of the 64-bit default zone and the 64-bit string zone, respectively.) This capability allows a program to deal with each VM 64-bit zone created during the invocation, including those created outside of the program.

Note

LIB$FIND_VM_ZONE_64 finds only 64-bit zones. You must use LIB$FIND_VM_ZONE_64 to loop through all VM zones.
LIB$ Routines
LIB$FIND_VM_ZONE_64 (Alpha and I64 Only)

The context argument controls the state of the scan. It determines what zone to return (the first, the next, and so forth). On the initial call, specified by context=0, LIB$VERIFY_VM_ZONE_64 is called to verify the heap management zone database. If the database is corrupt, further calls to this routine will produce no additional useful output.

When no more zones can be found, the routine returns the condition value LIB$ _NOTFOU.

If a zone has been corrupted in some major way (for example, if the validity code has been changed), then this routine may not be able to locate it in the zone database.

Note that ASTs may be disabled while LIB$FIND_VM_ZONE_64 is executing code that depends on the stability of the heap management zone database. In general it is the caller’s responsibility to ensure that the calling program has exclusive access to the zone database while scanning for multiple zones with this routine. Results are unpredictable if another thread of control modifies the zone database or the associated areas during the scanning.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _BADZONE</td>
<td>Invalid zone.</td>
</tr>
<tr>
<td>LIB$ _NOTFOU</td>
<td>Zone identifier not found (alternate success status).</td>
</tr>
<tr>
<td>LIB$ _WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>

Example

```fortran
IMPLICIT NONE
INTEGER*4 status
INTEGER*8 context, zone_id
INTEGER*4 lib$find_vm_zone_64, lib$show_vm_zone_64
context = 0
status = lib$find_vm_zone_64 (context, zone_id)
DO WHILE (status)
   print *
   status = lib$show_vm_zone_64 (zone_id, 0)
   status = lib$find_vm_zone_64 (context, zone_id)
END DO

Sample output for this Fortran program is as follows:

Zone Id = 0000000000020040, Zone name = "DEFAULT_ZONE"
Zone Id = 0000000000020140, Zone name = "STRING_ZONE"
```
LIB$FIT_NODENAME
Fit a Node Name into an Output Field

The Fit a Node Name Into an Output Field routine fits a node name into an output field. It attempts to compress the node name to fit the output field. If this fails, it trims the node name. †

Format

LIB$FIT_NODENAME nodename, output-buffer [,output-width][,resultant-length]

Returns

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

Arguments

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

Node name to be fitted into the desired output field. The nodename argument contains the address of a descriptor pointing to this node-name string.

The error LIB$_INVARG is returned if nodename contains an invalid node name, points to a null string, or contains more than 1024 characters. The error LIB$_INVSTRDES is returned if nodename is an invalid descriptor.

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

The output buffer. The output-buffer argument contains the address of a descriptor pointing to the output buffer. LIB$FIT_NODENAME writes the final output node name into the buffer pointed to by output-buffer.

The error LIB$_INVSTRDES is returned if output-buffer is an invalid descriptor.

The length field of the output-buffer descriptor is not updated unless output-buffer is a dynamic descriptor with a length less than the resulting fitted node name. Refer to the OpenVMS RTL String Manipulation (STR$) Manual for dynamic string descriptor usage.

The output-buffer argument contains an unusable result when LIB$FIT_NODENAME returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$FIT_NODENAME

**output-width**

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

Field width desired for the fit operation. The **output-width** argument is the address of an unsigned word that contains this field width in bytes.

If **output-width** is omitted, the current length of **output-buffer** is used. If **output-buffer** is not a fixed-length string, specify **output-width** to ensure that the desired width is used.

If the lengths of both **output-buffer** and **output-width** are specified, the length in **output-width** is used. In this case, if the current length of **output-buffer** is smaller than the length of **output-width**, the output node name is truncated at the end, and the alternate successful status LIB$_STRTRU is returned.

**resultant-length**

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

Length of the output node name. The **resultant-length** argument is the address of an unsigned word that contains this length in bytes.

The **resultant-length** argument contains an unusable result when LIB$FIT_NODENAME returns in error.

**Description**

This routine fits the input node name into the desired output field for display purposes. It first attempts to get the usable short form of the input node name by calling LIB$COMPRESS_NODENAME. If that fails, the input node name is expanded by LIB$EXPAND_NODENAME and then trimmed by LIB$TRIM_FULLNAME to fit the desired output width.

The input is validated against the supported form of input node names. The error LIB$_INVARG is returned if the input node name is invalid.

Node-name compression is always attempted even if the length of the input node name is less than or equal to the desired output width. This is to ensure that the short form of a full name is always chosen for display purposes.

When the compressed node name is too long to fit the desired output width, the input node name is expanded using LIB$EXPAND_NODENAME and trimmed using LIB$TRIM_FULLNAME. In this case, the alternate success status LIB$_STRTRU is returned.

When LIB$FIT_NODENAME encounters errors from the underlying network services, it tries to return the string-truncated compressed node name. If it is the compression operation that fails, LIB$FIT_NODENAME returns the string-truncated input node name. The alternate successful status LIB$_STRTRU is returned.
Note that the returned node name can be either a compressed usable short form of the input node name or an unusable trimmed or truncated node name. The caller should always assume an unusable node name is returned when it finds the alternate success return status LIB$_STRTRU. On the other hand, the SS$_NORMAL return status means that a usable form of a node name is returned.

LIB$FIT_NODENAME adds padding spaces to the end of the output buffer if the output node name is shorter than the size of the output buffer. The argument resultant-length, if supplied, is set to the length of the output node name, excluding any padding spaces.

**Condition Values Returned**

- **SS$_NORMAL**  
  Routine successfully completed.

- **LIB$_STRTRU**  
  Routine successfully completed. Characters are truncated in the output buffer pointed to by output-buffer.

- **LIB$_INVARG**  
  Invalid argument:
  - **nodename** is invalid.
  - **nodename** points to a null string.
  - The length of the node name is more than 1024 characters.

- **LIB$_INVSTRDES**  
  Invalid string descriptor.

- **LIB$_WRONUMARG**  
  Wrong number of arguments.

Any condition value returned by LIB$SCOPY_R_DX.
LIB$ Routines
LIB$FIXUP_FLT

LIB$FIXUP_FLT
Fix Floating Reserved Operand

The Fix Floating Reserved Operand routine finds the reserved operand of any F-floating, D-floating, G-floating, or H-floating instruction (with some exceptions) after a reserved operand fault has been signaled.† LIB$FIXUP_FLT changes the reserved operand from –0.0 to the value of the new-operand argument, if present; or to +0.0 if new-operand is absent.

This routine is available on OpenVMS Alpha and I64 systems in translated form and is applicable to translated VAX images only.

Format

LIB$FIXUP_FLT signal-arguments,mechanism-arguments [,new-operand]

Returns

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

Arguments

signal-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: read only
mechanism: by reference, array reference

Signal argument vector. The signal-arguments argument is the address of an array of unsigned longwords containing the signal argument vector.

mechanism-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: read only
mechanism: by reference, array reference

Mechanism argument vector. The mechanism-arguments argument is the address of an array of unsigned longwords containing the mechanism argument vector.

new-operand
OpenVMS usage: floating-point
type: F_floating
access: read only
mechanism: by reference

An F-floating value to replace the reserved operand. The new-operand argument is the address of an F-floating number containing the new operand. This is an optional argument. If omitted, the default value is +0.0.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$FIXUP_FLT

Description
LIB$FIXUP_FLT finds the reserved operand of any F-floating, D-floating, G-floating, or H-floating instruction (with some exceptions) after a reserved operand fault has been signaled. LIB$FIXUP_FLT changes the reserved operand from –0.0 to the value of the *new-operand* argument, if present; or to +0.0 if *new-operand* is absent. LIB$FIXUP_FLT cannot handle the following cases and will return a status of SS$_RESIGNAL if any of them occur:

- The currently active signaled condition is not SS$_ROPRAND.
- The reserved operand's data type is not F-floating, D-floating, G-floating, or H-floating.
- The reserved operand is an element in the coefficient table for one of the VAX POLYx instructions.

If the status value returned from LIB$FIXUP_FLT is seen by the condition handling facility (as would be the case if LIB$FIXUP_FLT was the handler), any success value is equivalent to SS$_CONTINUE, which causes the instruction to be restarted. Any failure value is equivalent to SS$_RESIGNAL, which causes the condition to be ressignaled to the next handler. This ressignal status is because the condition handler (LIB$FIXUP_FLT) was unable to handle the condition correctly.

LIB$FIXUP_FLT can be enabled directly as a condition handler. The *signal-arguments* and *mechanism-arguments* arguments are passed to the condition handler by OpenVMS exception dispatching.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The reserved operand was found and has been fixed.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation. An argument to LIB$FIXUP_FLT or an operand of the faulting instruction could not be read or written.</td>
</tr>
<tr>
<td>SS$_RESIGNAL</td>
<td>The signaled condition was not SS$_ROPRAND, or the reserved operand was not a floating-point value or was an element in a POLYx table.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. The optional argument <em>new-operand</em> was supplied but was itself an F-floating reserved operand.</td>
</tr>
<tr>
<td>LIB$_BADSTA</td>
<td>Bad stack. The stack frame linkage has been corrupted since the time of the reserved operand exception.</td>
</tr>
</tbody>
</table>
LIB$ FLT_UNDER
Floating-Point Underflow Detection

The Floating-Point Underflow Detection routine enables or disables floating-point underflow detection for the calling routine activation. The previous setting is returned as a function value. †

This routine is available on OpenVMS Alpha and I64 systems in translated form and is applicable to translated VAX images only.

Format

LIB$FLT_UNDER new-setting

Returns

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

The old floating-point underflow enable setting (the previous contents of the SF$W_PSW[PSW$V_FU] in the caller’s frame).

Argument

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

New floating-point underflow enable setting. The new-setting argument is the address of an unsigned byte containing the new setting. Bit 0 set to 1 means enable; bit 0 set to 0 means disable.

Description

LIB$FLT_UNDER affects only the current routine activation and does not affect any of its callers or any routines that it may call. However, the setting does remain in effect for any routines entered through a JSB entry point.

The caller’s stack frame will be modified by this routine.

Condition Values Returned

None.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
Example

This Fortran example program shows the use of LIB$FLT_UNDER.

```
INTEGER*4 NEW_SETTING
REAL*4 X, Y, Z
NEW_SETTING = 0
X = 1E-20
Y = 1E20
CALL LIB$FLT_UNDER( NEW_SETTING )
TYPE *, 'First Case: This should not have an underflow exception'
Z = X / Y
TYPE *, 'If this lines prints then the underflow exception 1 was disabled.'
NEW_SETTING = 1
X = 1E-20
Y = 1E20
CALL LIB$FLT_UNDER( NEW_SETTING )
TYPE *, 'Second Case: This should have an underflow exception 1 and then stop.'
Z = X / Y
TYPE *, 'If this line prints, then the underflow exception 1 was disabled.'
END
```

In this Fortran example, floating-point underflow detection is disabled the first time X is divided by Y. The second time, underflow detection is enabled, and the program stops because of the error generated.
LIB$ Routines
LIB$FORMAT_DATE_TIME

LIB$FORMAT_DATE_TIME
Format Date and/or Time

The Format Date and/or Time routine allows the user to select at run time a specific output language and format for a date or time, or both.

Format

LIB$FORMAT_DATE_TIME date-string [,date] [,.user-context] [,date-length] [,flags]

Returns

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

Arguments

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

Receives the requested date or time, or both, that has been formatted for output according to the currently selected format and language. The date-string argument is the address of a descriptor pointing to this string.

date
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

The date or time, or both, to be formatted for output. The date argument is the address of an unsigned quadword that contains the absolute date or time, or both to be formatted. If you omit this argument, or if you supply a zero passed by value, then the current system time is used. Note that the date argument must represent an absolute time, not a delta time.

user-context
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference

User context that retains the translation context over multiple calls to this routine. The user-context argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.
The user-context parameter is optional. However, if a context cell is not passed, the routine LIB$FORMAT_DATE_TIME may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, HP recommends that users specify a different context cell for each calling thread.

**date-length**

OpenVMS usage: longword_signed  
Type: longword (signed)  
Access: write only  
Mechanism: by reference

Number of bytes of text written to the date-string argument. The date-length argument is the address of a signed longword that receives this string length. Note that date-length specifies the number of bytes of text, not the number of characters, written to date-string.

**flags**

OpenVMS usage: mask_longword  
Type: longword (unsigned)  
Access: read only  
Mechanism: by reference

Bit mask that allows the user to specify whether the date, time, or both are output. The flags argument is the address of an unsigned bit mask containing the specified values. Valid values are LIB$M_DATE_FIELDS and LIB$M_TIME_FIELDS.

Default values are determined as follows:

- If the flags argument is omitted, LIB$FORMAT_DATE_TIME determines which fields to format according to the current definition of LIB$DT_FORMAT.
- If the flags argument is specified, LIB$FORMAT_DATE_TIME uses the flags value to determine which fields to format. That is, the flags argument can be used to override the definition of LIB$DT_FORMAT when specifying which fields should be formatted for output. If the field specified by flags was not assigned a format through the definition of LIB$DT_FORMAT, the standard OpenVMS format is used.

**Description**

The LIB$FORMAT_DATE_TIME routine formats an OpenVMS internal format date-time quadword into a textual string of some predefined format. The language to be used and the format in which to output the information are programmable using either of the following methods.

- The language and format are programmable at compile time through the use of the routine LIB$INIT_DATE_TIME_CONTEXT.
- The language and format are determined at run time through the translation of the logical names SYS$LANGUAGE and LIB$DT_FORMAT.

In general, if an application is formatting text for internal storage or transmission, the language and format should be specified at compile time. If this is the case, use the routine LIB$INIT_DATE_TIME_CONTEXT to specify the language and format of your choice.
If an application is formatting text for presentation to a user, the logical name method of specifying language and format should be used. In this method, the user assigns equivalence names to the logical names SYS$LANGUAGE and LIB$DT_FORMAT, thereby selecting the language and format of the date and time at run time.

If the logical name method is used, the translations of the logical names SYS$LANGUAGE and LIB$DT_FORMAT specify one or more executive mode logicals, which in turn must be translated to determine the actual format string. These additional logicals supply such things as the names of the days of the week and the months in the selected language (determined by SYS$LANGUAGE). All of these logicals are predefined, so that a non-privileged user can select any one of these languages and formats. A user can create his or her own languages and formats; however, the CMEXEC, SYSNAME, and SYSPRV privileges are required.

With the exception of SYS$LANGUAGE and LIB$DT_FORMAT, all logical names used by this routine must be defined from the executive mode.

See the *HP OpenVMS Programming Concepts Manual* for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.

### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ABSTIMREQ</td>
<td>Absolute time required.</td>
</tr>
<tr>
<td>LIB$DEFFORUSE</td>
<td>Default format used; unable to determine the desired format.</td>
</tr>
<tr>
<td>LIB$ENGLUSED</td>
<td>English used; unable to determine or use the specified language.</td>
</tr>
<tr>
<td>LIB$REENTRANCY</td>
<td>Reentrant invocation with same context variable.</td>
</tr>
<tr>
<td>LIB$STRTRU</td>
<td>Output string truncated.</td>
</tr>
<tr>
<td>LIB$UNRFORCOD</td>
<td>Unrecognized format code.</td>
</tr>
</tbody>
</table>

Any condition values returned by the $NUMTIM system service, or RTL routines LIB$GET_VM, LIB$GET_VM_64, LIB$ANALYZE_SDESC, or LIB$ANALYZE_SDESC_64.
LIB$FORMAT_SOGW_PROT
Format Protection Mask

The Format Protection Mask routine translates a protection mask into a formatted string.

Format

LIB$FORMAT_SOGW_PROT  protection-mask, [access-names], [ownership-names],
  [ownership-separator], [list-separator], protection-string, [protection-length]

Returns

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

Arguments

protection-mask
OpenVMS usage: protection
type: word (unsigned)
access: read only
mechanism: by reference
The address of a word that holds a 16-bit protection mask to be translated.

access-names
OpenVMS usage: access_names
type: array [0..31] of quadword string descriptor
access: read only
mechanism: by reference
The address of the access name table for the associated object class. For example, it is the value returned in accnam by LIB$GET_ACCNAM. This parameter defaults to the access name table for the FILE object class.

ownership-names
OpenVMS usage: char_string
type: array [0..3] of quadword string descriptor
access: read only
mechanism: by reference
The address of a vector of 4 quadword descriptors that points to the ownership name. The default value is the full ownership category names (System, Owner, Group, World).

ownership-separator
OpenVMS usage: char_string
type: character-coded text string
access: read only
mechanism: by descriptor
LIB$ Routines
LIB$FORMAT_SOGW_PROT

The address of a descriptor that points to the ownership separator string. The separator string is inserted after the ownership name to introduce a nonempty set of access names. By default, the value is "; " (the colon and space characters).

**list-separator**

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

The address of a descriptor that points to the list separator string. The list separator string is inserted between ownership-access type pairs. By default, the value is " , " (the comma and space characters).

**protection-string**

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

The address of a character-string descriptor that receives the output of the routine call. The protection-string argument points to the formatted protection string at the end of a call. The protection string has the following components repeated for each of: System, Owner, Group, World:

ownership-name[ownership-separator][access-types][list-separator]

An example of a formatted protection string is

System: RWED, Owner: RWED, Group: RW, World: R

**protection-length**

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

The address of a word that receives the length of the string returned in the protection-string argument.

Description

LIB$FORMAT_SOGW_PROT translates a 16-bit protection mask into a formatted string. This routine works for any protected object class by specifying the correct access name table. The address of the access name table can be obtained from the LIB$GET_ACCNAM routine.

Several formatting options are available. The caller can specify ownership names, ownership separators, or list separators.

Condition Values Returned

- **SS$_NORMAL** Routine successfully completed.
- **LIB$_INVARG** Required parameter missing.
- **LIB$_WRONGNUMARG** Wrong number of arguments.
- **STR$_TRU** String truncation warning.
LIB$FREE_DATE_TIME_CONTEXT
Free the Context Area Used When Formatting Dates and Times for Input or Output

The Free the Context Area Used When Formatting Dates and Times for Input or Output routine frees the virtual memory associated with the context area used by the date/time input and output formatting routines.

Format

LIB$FREE_DATE_TIME_CONTEXT [user-context]

Returns

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

Argument

user-context
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference

User context that retains the translation context over multiple calls to the date/time input and output formatting routines. The user-context argument is the address of an unsigned longword that contains this context. If the user-context argument was not specified in the call to LIB$FORMAT_DATE_TIME, LIB$CONVERT_DATE_STRING, or LIB$GET_MAXIMUM_DATE_LENGTH, then no argument should be supplied when calling this routine.

Description

The LIB$FREE_DATE_TIME_CONTEXT routine frees the virtual memory associated with the context area used by the date/time input and output formatting routines. A call to this routine is optional, since the same functions are performed at image exit.

Condition Values Returned

SS$_NORMAL Routine successfully completed.

Any condition value returned by LIB$FREE_VM. If one of these condition values is returned, it indicates either an internal coding error or that memory was corrupted by the user's program.
LIB$ Routines
LIB$FREE_EF

LIB$FREE_EF
Free Event Flag

The Free Event Flag routine frees a local event flag previously allocated by
LIB$GET_EF or by LIB$RESERVE_EF. LIB$FREE_EF is the complement of
LIB$GET_EF.

Format

LIB$FREE_EF  event-flag-number

Returns

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

Argument

event-flag-number
OpenVMS usage:  ef_number
type:  longword integer (unsigned)
access:  read only
mechanism:  by reference

Event flag number to be deallocated by LIB$FREE_EF. The event-flag-number
argument is the address of a signed longword integer that contains the event
flag number, which is the value allocated to the user by LIB$GET_EF or
LIB$RESERVE_EF.

Description

When a local event flag allocated by calling LIB$GET_EF or LIB$RESERVE_EF
is no longer needed, LIB$FREE_EF should be called to free the event flag for use
by other routines.

See the HP OpenVMS Programming Concepts Manual for more information.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_EF_ALRFRE</td>
<td>Event flag already free.</td>
</tr>
<tr>
<td>LIB$_EF_RESSYS</td>
<td>Event flag reserved to system. This error occurs if the event flag number is outside the ranges of 1 to 23 and 32 to 63.</td>
</tr>
</tbody>
</table>
LIB$FREE_LUN
Free Logical Unit Number

The Free Logical Unit Number routine releases a logical unit number allocated by LIB$GET_LUN to the pool of available numbers. LIB$FREE_LUN is the complement of LIB$GET_LUN.

Format

LIB$FREE_LUN logical-unit-number

Returns

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

Argument

logical-unit-number
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Logical unit number to be deallocated. The logical-unit-number argument is the address of a signed longword integer that contains this logical unit number, which is the value previously returned by LIB$GET_LUN.

Description

When a logical unit number allocated by calling LIB$GET_LUN is no longer needed, it should be released for use by other routines.

This routine is useful only in BASIC or FORTRAN programs.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_LUNALRFRE</td>
<td>Logical unit number is already free.</td>
</tr>
<tr>
<td>LIB$_LUNRESSYS</td>
<td>Logical unit number reserved to system. This occurs if the specified logical unit number is outside the range of 100 through 299.</td>
</tr>
</tbody>
</table>
LIB$FREE_TIMER
Free Timer Storage

The Free Timer Storage routine frees the storage allocated by LIB$INIT_TIMER.

Format

LIB$FREE_TIMER handle-address

Returns

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

Argument

handle-address
OpenVMS usage: address
type: longword (unsigned)
access: modify
mechanism: by reference

Pointer to a block of storage containing the value returned by a previous call to LIB$INIT_TIMER; this is the storage that LIB$FREE_TIMER deallocates. The handle-address argument is the address of an unsigned longword containing that value.

Description

LIB$FREE_TIMER frees a block of storage previously allocated by LIB$INIT_TIMER. LIB$FREE_TIMER assumes that handle-address was returned by a previous call to LIB$INIT_TIMER. If the block referred to by handle-address was not allocated by LIB$INIT_TIMER, LIB$FREE_TIMER returns an error. If the routine completes successfully, LIB$FREE_TIMER sets handle-address to zero.

Condition Values Returned

SS$NORMAL Routine successfully completed.
LIB$_BADBLOADR Bad block address; LIB$FREE_TIMER could not deallocate the block to which handle-address points.
LIB$_INVAR G Invalid argument; handle-address was not supplied or did not point to a timer block.
LIB$FREE_VM
Free Virtual Memory from Program Region

The Free Virtual Memory from Program Region routine deallocates an entire block of contiguous bytes that was allocated by a previous call to LIB$GET_VM. The arguments passed are the same as for LIB$GET_VM. †

Format
LIB$FREE_VM number-of-bytes ,base-address [,zone-id]

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

Arguments

number-of-bytes
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Number of contiguous bytes to be deallocated by LIB$FREE_VM. The number-of-bytes argument is the address of a signed longword integer that contains this number. The value of number-of-bytes must be greater than zero. Byte counts are rounded in the same manner as in LIB$GET_VM.

Note
You may omit the number-of-bytes argument if you are using boundary tags (LIB$M_VM_BOUNDARY_TAGS).

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

Address of the first byte to be deallocated by LIB$FREE_VM. The base-address argument contains the address of an unsigned longword that is this address. The value of base-address must be the address of a block of memory that was allocated by a previous call to LIB$GET_VM.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$FREE_VM

**zone-id**

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

The address of a longword that contains a zone identifier created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE.

You must specify the same **zone-id** value as when you called LIB$GET_VM to allocate the block. An error status will be returned if you specify an incorrect **zone-id**. The **zone-id** argument is optional. If **zone-id** is omitted or if the longword contains the value 0, the 32-bit default zone is used.

**Description**

LIB$FREE_VM returns the block of memory to a free list associated with the zone, so the block is available on a subsequent call to LIB$GET_VM for the zone.

The **base-address** argument must contain the address of the first byte of memory that was allocated by a previous call to LIB$GET_VM. LIB$FREE_VM rounds up the value of **number-of-bytes** to a multiple of the block size for the zone.

**Note**

You cannot free part of a block that was allocated by a call to LIB$GET_VM. The whole block must be freed by a single call to LIB$FREE_VM.

Neither can you combine contiguous blocks of memory that were allocated by several calls to LIB$GET_VM into one larger block that is freed by a single call to LIB$FREE_VM.

If you specified deallocation filling when you created the zone, LIB$FREE_VM will fill each byte freed. Note that part of a free block is used to store control information, so some bytes will not contain the fill value.

LIB$FREE_VM is fully reentrant, so it can be called by routines executing at AST-level or in an Ada multitasking environment.

If the zone you are freeing was created using the LIB$CREATE_USER_VM_ZONE routine, then you must have an appropriate action routine for the free operation. That is, in your call to LIB$CREATE_USER_VM_ZONE, you must have specified a user deallocation procedure.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>The <strong>base-address</strong> argument contained a bad block address. Either an address was outside of the area allocated by LIB$GET_VM, the contents of <strong>base-address</strong> were not properly aligned, part of the space being deallocated was previously deallocated, or a zone was found to be corrupt.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$FREE_VM

LIB$_BADBLOSIZ  The **number-of-bytes** argument is less than or equal to 0, or the **number-of-bytes** argument is incorrect for a zone containing fixed size blocks.

LIB$_BADTAGVAL  For a zone that uses boundary tags, the tag field was corrupted.
LIB$ Routines
LIB$FREE_VM_64 (Alpha and I64 Only)

LIB$FREE_VM_64 (Alpha and I64 Only)
Free Virtual Memory from Program Region

The Free Virtual Memory from Program Region routine deallocates an entire block of contiguous bytes that was allocated by a previous call to LIB$GET_VM_64. The arguments passed are the same as for LIB$GET_VM_64.

Format

LIB$FREE_VM_64 number-of-bytes , base-address [,zone-id]

Returns

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

Arguments

number-of-bytes
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Number of contiguous bytes to be deallocated by LIB$FREE_VM_64. The number-of-bytes argument is the address of a signed quadword integer that contains this number. The value of number-of-bytes must be greater than zero.

Byte counts are rounded in the same manner as in LIB$GET_VM_64.

Note

You may omit the number-of-bytes argument if you are using boundary tags (LIB$M_VM_BOUNDARY_TAGS).

base-address
OpenVMS usage: address
type: quadword (unsigned)
access: read only
mechanism: by reference

Address of the first byte to be deallocated by LIB$FREE_VM_64. The base-address argument contains the address of an unsigned quadword that is this address. The value of base-address must be the address of a block of memory that was allocated by a previous call to LIB$GET_VM_64.
LIB$ Routines
LIB$FREE_VM_64 (Alpha and I64 Only)

**zone-id**
OpenVMS usage: identifier
type: quadword (unsigned)
access: read only
mechanism: by reference

The address of a quadword that contains a zone identifier created by a previous call to LIB$CREATE_VM_ZONE_64 or LIB$CREATE_USER_VM_ZONE_64.

You must specify the same **zone-id** value as when you called LIB$GET_VM_64 to allocate the block. An error status will be returned if you specify an incorrect **zone-id**. The **zone-id** argument is optional. If **zone-id** is omitted or if the quadword contains the value 0, the 64-bit default zone is used.

**Description**

LIB$FREE_VM_64 returns the block of memory to a free list associated with the zone, so the block is available on a subsequent call to LIB$GET_VM_64 for the zone.

The **base-address** argument must contain the address of the first byte of memory that was allocated by a previous call to LIB$GET_VM_64. LIB$FREE_VM_64 rounds up the value of **number-of-bytes** to a multiple of the block size for the zone.

**Note**

You cannot free part of a block that was allocated by a call to LIB$GET_VM_64. The whole block must be freed by a single call to LIB$FREE_VM_64.

Neither can you combine contiguous blocks of memory that were allocated by several calls to LIB$GET_VM_64 into one larger block that is freed by a single call to LIB$FREE_VM_64.

If you specified deallocation filling when you created the zone, LIB$FREE_VM_64 will fill each byte freed. Note that part of a free block is used to store control information, so some bytes will not contain the fill value.

LIB$FREE_VM_64 is fully reentrant, so it can be called by routines executing at AST-level or in an Ada multitasking environment.

If the zone you are freeing was created using the LIB$CREATE_USER_VM_ZONE_64 routine, then you must have an appropriate action routine for the free operation. That is, in your call to LIB$CREATE_USER_VM_ZONE_64, you must have specified a user deallocation procedure.
LIB$ Routines
LIB$FREE_VM_64 (Alpha and I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>The base-address argument contained a bad block address. Either an address was outside of the area allocated by LIB$GET_VM_64, the contents of base-address were not properly aligned, part of the space being deallocated was previously deallocated, or a zone was found to be corrupt.</td>
</tr>
<tr>
<td>LIB$_BADBLOSIZ</td>
<td>The number-of-bytes argument is less than or equal to 0, or the number-of-bytes argument is incorrect for a zone containing fixed size blocks.</td>
</tr>
<tr>
<td>LIB$_BADTAGVAL</td>
<td>For a zone that uses boundary tags, the tag field was corrupted.</td>
</tr>
</tbody>
</table>
LIB$FREE_VM_PAGE
Free Virtual Memory Page

The Free Virtual Memory Page routine deallocates a block of contiguous pages on VAX systems or pagelets on Alpha and I64 systems that were allocated by previous calls to LIB$GET_VM_PAGE. †

Format

LIB$FREE_VM_PAGE number-of-pages ,base-address

Returns

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

Arguments

number-of-pages
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

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

Number of pages on VAX systems or pagelets on Alpha and I64 systems. The number-of-pages argument is the address of a longword integer that specifies the number of contiguous pages on VAX systems or pagelets on Alpha and I64 systems to be deallocated. The value of number-of-pages must be greater than zero.

base-address
Block address. The base-address argument is the address of a longword that contains the address of the first byte of the first VAX page or Alpha or I64 pagelet to be deallocated.

Description

LIB$FREE_VM_PAGE deallocates a block of contiguous 512-byte pages starting at base-address. Each of the pages or pagelets specified by number-of-pages and base-address must have been allocated by previous calls to LIB$GET_VM_PAGE. The pages or pagelets are returned to the processwide pool and are available to satisfy subsequent calls to LIB$GET_VM_PAGE.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
You can free a smaller group of pages or pagelets than you allocated. That is, if you allocated a group of contiguous pages or pagelets by a single call to LIB$GET_VM_PAGE, you can deallocate them in several calls to LIB$FREE_VM_PAGE. You can also combine contiguous groups of pages or pagelets that were allocated in several calls to LIB$GET_VM_PAGE into one large group that is freed by a single call to LIB$FREE_VM_PAGE.

LIB$FREE_VM_PAGE is fully reentrant, so it may be called by routines executing at AST level or in an Ada multitasking environment.

**Condition Values Returned**

- **SS$_NORMAL** Routine successfully completed.
- **LIB$_BADBLOADR** Pages on VAX systems or pagelets on Alpha and I64 systems not allocated by LIB$GET_VM_PAGE, the value of **base-address** is not a page boundary, or the pages were previously freed.
- **LIB$_BADBLOSIZ** The **number-of-pages** argument is less than or equal to zero.
LIB$FREE_VM_PAGE_64 (Alpha and I64 Only)
Free Virtual Memory Page

The Free Virtual Memory Page routine deallocates a block of contiguous Alpha or I64 pagelets that was allocated by previous calls to LIB$GET_VM_PAGE_64.

Format

LIB$FREE_VM_PAGE_64 number-of-pages ,base-address

Returns

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

Arguments

number-of-pages
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Number of Alpha or I64 pagelets. The address of a quadword integer that specifies the number of contiguous Alpha or I64 pagelets to be deallocated. The value of number-of-pages must be greater than zero.

base-address
OpenVMS usage: address
type: quadword (unsigned)
access: read only
mechanism: by reference

Block address. The base-address argument is the address of a quadword that contains the address of the first byte of the first Alpha or I64 pagelet to be deallocated.

Description

LIB$FREE_VM_PAGE_64 deallocates a block of contiguous Alpha or I64 pagelets starting at base-address. Each of the pagelets specified by number-of-pages and base-address must have been allocated by previous calls to LIB$GET_VM_PAGE_64. The pagelets are returned to the processwide pool and are available to satisfy subsequent calls to LIB$GET_VM_PAGE_64.

You can free a smaller group of pagelets than you allocated. That is, if you allocated a group of contiguous pagelets by a single call to LIB$GET_VM_PAGE_64, you can deallocate them in several calls to LIB$FREE_VM_PAGE_64. You can also combine contiguous groups of pagelets that were allocated in several calls to LIB$GET_VM_PAGE_64 into one large group that is freed by a single call to LIB$FREE_VM_PAGE_64.

LIB$FREE_VM_PAGE_64 is fully reentrant, so it may be called by routines executing at AST level or in an Ada multitasking environment.
LIB$ Routines
LIB$FREE_VM_PAGE_64 (Alpha and I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>Alpha pagelets not allocated by LIB$GET_VM_PAGE_64, the value of <strong>base-address</strong> is not a pagelet boundary, or the pagelets were previously freed.</td>
</tr>
<tr>
<td>LIB$_BADBLOSIZ</td>
<td>The <strong>number-of-pages</strong> argument is less than or equal to zero.</td>
</tr>
</tbody>
</table>
LIB$GETDVI
Get Device/Volume Information

The Get Device/Volume Information routine provides a simplified interface to the $GETDVI system service. It returns information about the primary and secondary device characteristics of an I/O device. The calling process need not have a channel assigned to the device about which it wants information.

Format


Returns

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

Arguments

item-code
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Code specifying the item of information you are requesting. The item-code argument is the address of a signed longword containing the item code. All valid $GETDVI item codes whose names begin with DVI$_ are accepted. See the Description section for more information on item codes.

channel
OpenVMS usage: channel
type: word (unsigned)
access: read only
mechanism: by reference

OpenVMS I/O channel assigned to the device for which LIB$GETDVI returns information. The channel argument is the address of an unsigned word containing the channel specification. If channel is not specified, device-name is used instead. You must specify either channel or device-name, but not both. If neither is specified, the error status SS$_IVDEVNAM is returned.

device-name
OpenVMS usage: device_name
type: character string
access: read only
mechanism: by descriptor

Name of the device for which LIB$GETDVI returns information. The device-name argument is the address of a descriptor pointing to the device name string. If this string contains a colon, the colon and the characters that follow it are ignored.
The **device-name** may be either a physical device name or a logical name. If the first character in the string is an underscore character ( underscore ), the name is considered a physical device name. Otherwise, the name is considered a logical name, and logical name translation is performed until either a physical device name is found or the system default number of translations has been performed.

If **device-name** is not specified, **channel** is used instead. You must specify either **channel** or **device-name**, but not both. If neither is specified, the error status **SS$_IVDEVNAM** is returned. The device name must not be longer than 255 characters.

**longword-integer-value**

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

Numeric value of the information requested. The **longword-integer-value** argument is the address of a signed longword containing the numeric value. If an item is listed as only returning a string value, this argument is ignored.

**resultant-string**

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

String representation of the information requested. The **resultant-string** argument is the address of a descriptor pointing to this information. If **resultant-string** is not specified and if the value returned has only a string representation, the error status **LIB$_INVARG** is returned.

Refer to Table lib–4 for a description of the string representation used for each item.

**resultant-length**

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

Number of significant characters written to **resultant-string** by LIB$GETDVI. The **resultant-length** argument is the address of an unsigned word containing this length.

**pathname**

OpenVMS usage: path_name  
**type**: character text string  
**access**: read only  
**mechanism**: by descriptor

(164 and Alpha only) The name of the path about which $GETDVI is to return information. The **pathname** argument is the address of a character string descriptor pointing to this name string. The path name may be used with either the **channel** or **device-name** arguments.
Check the definitions of the item codes to see if the **pathname** argument is used. In general, item codes that return information that may vary by path will make use of the **pathname** argument. The paths for a multipath device can be seen with the SHOW DEVICE /FULL command, the SYS$DEVICE_PATH_SCAN system service, or the F$MULTIPATH DCL lexical function.

If the **pathname** argument is used, it will be validated against the existing paths for the device specified. If the path does not exist, the error SS$_NOSUCHPATH will be returned, even if the item codes(s) used do not make use of the **pathname** argument.

**Description**

LIB$GETDVI returns two categories of information:

- Primary device characteristics
- Secondary device characteristics

LIB$GETDVI does not allow you to get more than one item of information in a single call.

LIB$GETDVI provides the following features in addition to those provided by the $GETDVI system service.

- Instead of a list of item descriptors, which may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.
- For items which return numeric values, LIB$GETDVI can optionally provide a formatted string interpretation of the value. For example, if the device owner UIC is requested, LIB$GETDVI can return the UIC formatted as [identifier].
- For string arguments, LIB$GETDVI understands all string classes supported by the Run-Time Library.
- Calls to LIB$GETDVI are synchronous; LIB$GETDVI calls LIB$GET_EF to allocate a local event flag number for synchronization.

See the description of the $GETDVI system service in the *HP OpenVMS System Services Reference Manual: A–GETUAI* for more detailed information.

**Item Codes**

All item codes that can be used with the $GETDVI system service may be used as the **item-code** argument to LIB$GETDVI. These codes have symbolic names beginning with DVI$_._

The use of a DVI$_._ code by itself will return the primary device characteristic associated with that code. To obtain the secondary device characteristics, add 1 to the code. See the description of the $GETDVI system service for a list of the defined item codes. The symbolic names for these items are defined in HP supplied symbol libraries in module $DVifdef (where appropriate).
Value Formats

By using the `longword-integer-value` and `resultant-string` arguments to LIB$GETDVI, the information requested can be returned in two different fashions.

- For each item described as a “string” in the table of Item Codes for the $GETDVI service, the value is returned in `resultant-string`.

- For all other items—those that have numeric values—the numeric representation is returned in `longword-integer-value` (if specified), and a formatted string interpretation of the value is returned in `resultant-string`.

Each formatted item is written left-justified; `resultant-length`, if specified, gives the number of characters used. Table lib–4 lists the formats used for the string interpretations.
Table 1. Formats Used for LIB$GETDVI Strings

<table>
<thead>
<tr>
<th>Item or Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVI$_ACPPID</td>
<td>The string value is returned as an 8-digit hexadecimal number.</td>
</tr>
<tr>
<td>DVI$_PID</td>
<td>The string value is returned as an 8-digit hexadecimal number.</td>
</tr>
<tr>
<td>DVI$_ACPTYPE</td>
<td>The ACP type string is one of the following:</td>
</tr>
<tr>
<td></td>
<td>NONE  No ACP</td>
</tr>
<tr>
<td></td>
<td>F11V1  Files-11 Level 1</td>
</tr>
<tr>
<td></td>
<td>F11V2  Files-11 Level 2</td>
</tr>
<tr>
<td></td>
<td>F11V3  Files-11 presentation of ISO 9660</td>
</tr>
<tr>
<td></td>
<td>F11V4  Files-11 presentation of High Sierra</td>
</tr>
<tr>
<td></td>
<td>F11V5  Files-11 structure level 5 (ODS-5)</td>
</tr>
<tr>
<td></td>
<td>F11V6  Files-11 structure level 5 (ODS-6)</td>
</tr>
<tr>
<td></td>
<td>F64 Files 64 support for Spiralog</td>
</tr>
<tr>
<td></td>
<td>HBS  Not currently defined</td>
</tr>
<tr>
<td></td>
<td>HBVS  ACP for Host Based Volume Shadowing</td>
</tr>
<tr>
<td></td>
<td>MTA  Magnetic Tape</td>
</tr>
<tr>
<td></td>
<td>NET  Networks</td>
</tr>
<tr>
<td></td>
<td>REM  Remote I/O</td>
</tr>
<tr>
<td></td>
<td>UCX  ACP for TCP/IP Services for OpenVMS</td>
</tr>
<tr>
<td>DVI$_OWNUIC</td>
<td>The standard UIC format ([\text{group,member}]) is used. If the format of a UIC includes identifiers from the access rights database in place of the octal group and member numbers, the UIC string returned will have these identifiers, if available.</td>
</tr>
<tr>
<td>DVI$_VPROT</td>
<td>The volume protection string is in the following form:</td>
</tr>
<tr>
<td></td>
<td>SYSTEM=RWLP,OWNER=RWLP,GROUP=RWLP,WORLD=RWLP</td>
</tr>
<tr>
<td></td>
<td>If a category has no access, the equal sign is omitted. The string will not contain any embedded spaces.</td>
</tr>
<tr>
<td>Boolean</td>
<td>The value string returned is TRUE if the low bit of the value is set, or FALSE if the low bit is clear.</td>
</tr>
<tr>
<td>All others</td>
<td>The value string is returned in the form of an unsigned decimal integer.</td>
</tr>
</tbody>
</table>

**Note**

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.
## Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>String truncated. This is an alternate success return status. The <code>resultant-string</code> argument could not contain all the characters of the returned item.</td>
</tr>
<tr>
<td>SS$_BADPARAM</td>
<td>Unrecognized item code. The <code>item-code</code> argument was not recognized as valid by <code>$GETDVI</code>.</td>
</tr>
<tr>
<td>SS$_IVDEVNAM</td>
<td>The device name string contains invalid characters, or neither the <code>channel</code> nor <code>device-name</code> arguments were specified.</td>
</tr>
<tr>
<td>LIB$_INSEF</td>
<td>Insufficient event flags. A local event flag number could not be allocated by a call to LIB$GET_EF.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid arguments. The <code>$GETDVI</code> Item Code describes the item as a “string”, and no <code>resultant-string</code> argument was specified.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. The descriptor of the <code>resultant-string</code> argument is not a valid descriptor.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments. An incorrect number of arguments was passed to LIB$GETDVI.</td>
</tr>
</tbody>
</table>

Any condition values returned by LIB$SCOPY_xxx, or the `$GETDVI` system service.
LIB$GETJPI
Get Job/Process Information

The Get Job/Process Information routine provides a simplified interface to the $GETJPI system service. It provides accounting, status, and identification information about a specified process.

LIB$GETJPI obtains only one item of information in a single call.

Format


Returns

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

Arguments

item-code
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Item identifier code specifying the item of information you are requesting. The item-code argument is the address of a signed longword containing the item code. You may request only one item in each call to LIB$GETJPI.

LIB$GETJPI accepts all $GETJPI item codes. These names begin with JPI$ and are defined in symbol libraries in module $JPIDEF supplied by HP.

process-id
OpenVMS usage: process_id
type: longword (unsigned)
access: modify
mechanism: by reference

Process identifier of the process for which you are requesting information. The process-id argument is the address of an unsigned longword containing the process identifier. If you do not specify process-id, process-name is used.

The process-id is updated to contain the process identifier actually used, which may be different from what you originally requested if you specified process-name or used wildcard process searching.
process-name
OpenVMS usage: process_name
type: character string
access: read only
mechanism: by descriptor

A 1- to 15-character string specifying the name of the process for which you are requesting information. The process-name argument is the address of a descriptor pointing to the process name string. The name must correspond exactly to the name of the process for which you are requesting information; LIB$GETJPI does not allow trailing blanks or abbreviations.

If you do not specify process-name, process-id is used. If you specify neither process-name nor process-id, the caller’s process is used. Also, if you do not specify process-name and you specify zero for process-id, the caller’s process is used. In this way, you can fetch the item you want and the caller’s PID in a single call to LIB$GETJPI.

resultant-value
OpenVMS usage: varying_arg
type: unspecified
access: write only
mechanism: by reference

Numeric value of the information you request. The resultant-value argument is the address of a longword or quadword into which LIB$GETJPI writes the numeric value of this information. Refer to Table lib–5 for information on which items return longword values and which return quadword values. If the item you request returns only a string value, this argument is ignored.

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

String representation of the information you request. The resultant-string argument is the address of the descriptor for a character string into which LIB$GETJPI writes the string representation. Table lib–5 describes the string representation used for each item.

If you do not include resultant-string, but the item you request has only a string representation, the error status LIB$_INVARG is returned.

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

Number of significant characters written to resultant-string by LIB$GETJPI. The resultant-length argument is the address of an unsigned word integer into which LIB$GETJPI writes the number of characters.
Description

LIB$GETJPI provides the following features in addition to those provided by the $GETJPI system service:

- Instead of a list of item descriptors, which may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.

- For items which return numeric values, LIB$GETJPI can optionally provide a formatted string interpretation of the value. For example, if the process UIC is requested, LIB$GETJPI can return the UIC formatted as [g,m].

- For string arguments, all string classes supported by the Run-Time Library are understood.

- Calls to LIB$GETJPI are synchronous. LIB$GETJPI calls LIB$GET_EF to allocate a local event flag number for synchronization.

See the description of the $GETJPI system service in the HP OpenVMS System Services Reference Manual: A–GETUAI for more information.

By using the resultant-value and resultant-string arguments to LIB$GETJPI, you can request that the information be returned in two ways. For each item described as a “string” in the table of Item Codes for the $GETJPI service, the value is returned in resultant-string. For all other items—those which have numeric values—the numeric representation is returned in resultant-value (if specified), and a formatted string interpretation of the value is returned in resultant-string.

Each formatted item is written left-justified; resultant-length, if specified, gives the number of characters used.

Table lib–5 lists the formats used for the string interpretations.

Table lib–5  Item Code Formats for LIB$GETJPI

<table>
<thead>
<tr>
<th>Item or Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPI$_AUTHPRIV</td>
<td>The string representation of these quadword privilege masks is a list of each privilege that is enabled. The privilege names are in uppercase, and are separated by commas.</td>
</tr>
<tr>
<td>JPI$_CURPRIV</td>
<td>Same as for JPI$_AUTHPRIV.</td>
</tr>
<tr>
<td>JPI$_IMAGPRIV</td>
<td>Same as for JPI$_AUTHPRIV.</td>
</tr>
<tr>
<td>JPI$_PROCPRIV</td>
<td>Same as for JPI$_AUTHPRIV.</td>
</tr>
<tr>
<td>JPI$_LOGINTIM</td>
<td>The string representation of the quadword time is a standard absolute date-time string.</td>
</tr>
<tr>
<td>JPI$_PID</td>
<td>The process identification string is an 8-digit hexadecimal number.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Item Code Formats for LIB$GETJPI

<table>
<thead>
<tr>
<th>Item or Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPI$._STATE</td>
<td>The process state string is one of the following:</td>
</tr>
<tr>
<td></td>
<td>CEF</td>
</tr>
<tr>
<td></td>
<td>COM</td>
</tr>
<tr>
<td></td>
<td>COMO</td>
</tr>
<tr>
<td></td>
<td>CUR</td>
</tr>
<tr>
<td></td>
<td>COLPG</td>
</tr>
<tr>
<td></td>
<td>FPG</td>
</tr>
<tr>
<td></td>
<td>HIB</td>
</tr>
<tr>
<td></td>
<td>HIBO</td>
</tr>
<tr>
<td></td>
<td>LEF</td>
</tr>
<tr>
<td></td>
<td>LEFO</td>
</tr>
<tr>
<td></td>
<td>MWAIT</td>
</tr>
<tr>
<td></td>
<td>PFW</td>
</tr>
<tr>
<td></td>
<td>SUSP</td>
</tr>
<tr>
<td></td>
<td>SUSPO</td>
</tr>
<tr>
<td>JPI$._UIC</td>
<td>The standard UIC format [group/member] is used. If the format of a UIC includes identifiers from the access rights database in place of the octal group and member numbers, the UIC string returned will have these identifiers, if available.</td>
</tr>
<tr>
<td>JPI$._MODE</td>
<td>The current mode string is one of the following: BATCH, INTERACTIVE or NETWORK.</td>
</tr>
<tr>
<td>All others</td>
<td>The string value is returned as an unsigned decimal integer.</td>
</tr>
</tbody>
</table>

---

**Note**

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.

### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$._NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$._STRTRU</td>
<td>String truncated. This is an alternate success return status. The resultant-string argument could not contain all the characters of the returned item.</td>
</tr>
<tr>
<td>SS$._BADPARAM</td>
<td>Unrecognized item code. The item-code argument was not recognized as valid by $GETJPI.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>LIB$_INSEF</td>
<td>Insufficient event flags. A local event flag number could not be allocated by a call to LIB$GET_EF.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid arguments. The $GETJPI Item Code describes the item as a “string”, and no resultant-string argument was specified.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. The descriptor for a string argument was not a valid string descriptor.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments. An incorrect number of arguments was passed to LIB$GETJPI.</td>
</tr>
</tbody>
</table>

Any condition value returned by LIB$SCOPY_xxx, or the $GETJPI system service.
LIB$ GETQUI
Get Queue Information

The Get Queue Information routine provides a simplified interface to the $GETQUI system service. It provides queue, job, file, characteristic, and form information about a specified process.

LIB$GETQUI obtains only one item of information in a single call.

Format


Returns

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

Arguments

function-code
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Function code specifying the function that LIB$GETQUI is to perform. The function-code argument is the address of a signed longword containing the function code.

LIB$GETQUI accepts all $GETQUI function codes. These names begin with QUI$_ and are defined in symbol libraries in module $QUIDEF supplied by HP.

item-code
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Item identifier code specifying the item of information you are requesting. The item-code argument is the address of a signed longword containing the item code. You may request only one item in each call to LIB$GETQUI.

LIB$GETQUI accepts all $GETQUI item codes. These names begin with QUI$_ and are defined in symbol libraries in module $QUIDEF supplied by HP.
**search-number**
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Numeric value used to process your request. The **search-number** argument is the address of a signed longword integer containing the number needed to process your request. The **search-number** argument corresponds directly to QUI$_SEARCH_NUMBER as described by the $GETQUI system service.

**search-name**
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Character string used to process your request. The **search-name** argument is the address of a string descriptor that provides the name needed to process your request. The **search-name** argument corresponds directly to QUI$_SEARCH_NAME as described by the $GETQUI system service.

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

Optional bit mask indicating request to be performed. The **search-flags** argument is the address of an unsigned longword integer containing the bit mask. The **search-flags** argument directly corresponds to $QUI_SEARCH_FLAGS as described by the $GETQUI system service.

**resultant-value**
OpenVMS usage: varying_arg
type: unspecified
access: write only
mechanism: by reference

Numeric value of the information you requested. The **resultant-value** argument is the address of a longword, quadword or octaword into which LIB$GETQUI writes the numeric value of this information. Refer to Table lib–6 for information on which items return values other than longwords.

If the item you requested returns only a string value, this argument is ignored.

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

String representation of the information you requested. The **resultant-string** argument is the address of the descriptor for a character string into which LIB$GETQUI writes the string representation. Table lib–6 describes the string representation used for each item.
LIB$ Routines
LIB$getqui

If you do not include resultant-string, but the item you request has only a string representation, the error status LIB$_INVARG is returned.

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

Number of significant characters written to resultant-string by LIB$getqui. The resultant-length argument is the address of a signed word integer into which LIB$getqui writes the number of characters.

Description
LIB$getqui provides a simplified interface to the $GETQUI system service. It provides queue, job, file, characteristic, and form information about a specified process. This routine obtains only one item of information in a single call.

LIB$getqui provides the following features in addition to those provided by the $GETQUI system service.
• Instead of a list of item descriptors that may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.
• For items that return numeric values, LIB$getqui optionally can provide a formatted string interpretation of the value. For example, if you request the characteristics of a queue, LIB$getqui can return the list of characteristics as “23,42,76,98,125”.
• For string arguments, all string classes supported by the Run-Time Library are understood.
• Calls to LIB$getqui are synchronous. LIB$getqui calls $GETQUIW to force the synchronization.

LIB$getqui retains context. This means that previous calls to LIB$getqui affect current calls to LIB$getqui.

See the description of the $GETQUI system service in the HP OpenVMS System Services Reference Manual: A–GETUI for more information.

By using the resultant-value and resultant-string arguments to LIB$getqui, you can request that the information be returned in two ways. For items that have numeric values, the numeric representation is returned in resultant-value (if specified), and a formatted string interpretation of the value is returned in resultant-string. For each item described as a “string” in the table of Item Codes for the $GETQUI service, the value is returned in resultant-string.

Each formatted item is written left-justified; resultant-length, if specified, gives the number of characters used.

The $GETQUI system service requires some item codes. LIB$getqui provides those item codes for you by corresponding your input to LIB$getqui directly to the required input codes.
The following table describes all of the required and optional input needed to perform your task with LIB$GETQUI:

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUI$_CANCEL</td>
<td>Accepts no input.</td>
</tr>
<tr>
<td>QUI$_DISPLAY_CHARACTERISTIC</td>
<td>A characteristic name or number, or both. Optionally, a search flags number.</td>
</tr>
<tr>
<td>QUI$_DISPLAY_ENTRY</td>
<td>Optionally, an entry number, user name, and search flags number. The default user name is that of the calling process.</td>
</tr>
<tr>
<td>QUI$_DISPLAY_FILE</td>
<td>Optionally, a search flags number.</td>
</tr>
<tr>
<td>QUI$_DISPLAY_FORM</td>
<td>A form name or number, or both. Optionally, a search flags number.</td>
</tr>
<tr>
<td>QUI$_DISPLAY_JOB</td>
<td>Optionally, a search flags number.</td>
</tr>
<tr>
<td>QUI$_DISPLAY_QUEUE</td>
<td>A queue name. Optionally, a search flags number.</td>
</tr>
<tr>
<td>QUI$_TRANSLATE_QUEUE</td>
<td>A queue name.</td>
</tr>
</tbody>
</table>

Table lib–6 lists the formats used for the string interpretations.

**Table lib–6 Item Code Formats for LIB$GETQUI**

<table>
<thead>
<tr>
<th>Item or Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUI$_AFTER_TIME</td>
<td>Returns a quadword resultant-value as well as a resultant-string.</td>
</tr>
<tr>
<td>QUI$_CHARACTERISTICS</td>
<td>Returns an octaword resultant-value as well as a comma-separated list that lists all the characteristic numbers, output as a resultant-string.</td>
</tr>
<tr>
<td>QUI$_SUBMISSION_TIME</td>
<td>Returns a quadword resultant-value as well as a resultant-string.</td>
</tr>
<tr>
<td>QUI$_UIC</td>
<td>Returns a formatted resultant-string as well as a longword.</td>
</tr>
</tbody>
</table>

**Note**

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.
LIB$ Routines
LIB$GETQUI

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_STRTRU String truncated. This is an alternate success return status. The resultant-string argument could not contain all the characters of the returned item.
SS$_BADPARAM Unrecognized item code. The item-code argument was not recognized as valid by $GETQUI.
LIB$_INSEF Insufficient event flags. A local event flag number could not be allocated by a call to LIB$GET_EF.
LIB$_INVARG Invalid arguments. The $GETQUI Item Code describes the item as a “string”, and no resultant-string argument was specified.
LIB$_INVSTRDES Invalid string descriptor. The descriptor for a string argument was not a valid string descriptor.
LIB$_WRONUMARG Wrong number of arguments. An incorrect number of arguments was passed to LIB$GETQUI.

Any condition value returned by LIB$SCOPY_xxx, or the $GETQUI system service.
LIB$GETSYI
Get Systemwide Information

The Get Systemwide Information routine provides a simplified interface to the $GETSYI system service. The $GETSYI system service obtains status and identification information about the system. LIB$GETSYI returns only one item of information in a single call.

Format


Returns

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

Arguments

item-code
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Item code specifying the desired item of information. The item-code argument is the address of a signed longword containing this item code. All valid $GETSYI item codes are accepted.

resultant-value
OpenVMS usage: varying_arg
type: unspecified
access: write only
mechanism: by reference

Numeric value returned by LIB$GETSYI. The resultant-value argument is the address of a longword or quadword containing this value. If an item is listed as returning only a string value, this argument is ignored.

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

Information returned by LIB$GETSYI. The resultant-string argument is the address of a descriptor pointing to the character string that will receive this information.

See the Description section for more information about value formats. If resultant-string is not specified and if the returned value has only a string representation, the error status LIB$_INVARG is returned.
resultant-length
OpenVMS usage: word_unsigned
Type: word (unsigned)
Access: write only
Mechanism: by reference

Number of significant characters written to resultant-string, not including blank padding or truncated characters. The resultant-length argument is the address of an unsigned word into which LIB$GETSYI returns this number.

class-system-id
OpenVMS usage: identifier
Type: longword (unsigned)
Access: modify
Mechanism: by reference

Cluster system identification (CSID) of the node for which information is to be returned. The class-system-id argument is the address of this CSID. If class-system-id is specified and is nonzero, node-name is not used. If class-system-id is specified as zero, LIB$GETSYI uses node-name and writes into the class-system-id argument the CSID corresponding to the node identified by node-name.

The class-system-id of an OpenVMS node is assigned by the cluster-connection software and may be obtained by the DCL command SHOW CLUSTER. The value of the class-system-id for an OpenVMS node is not permanent; a new value is assigned to an OpenVMS node whenever it joins or rejoins the OpenVMS Cluster.

If class-system-id is specified as –1, LIB$GETSYI assumes a wildcard operation and returns the requested information for each OpenVMS node in the cluster, one node per call.

If class-system-id is not specified, node-name is used.

node-name
OpenVMS usage: char_string
Type: character string
Access: read only
Mechanism: by descriptor

Name of the node for which information is to be returned. The node-name argument is the address of a descriptor pointing to the node name string. If class-system-id is not specified or is specified as zero, node-name is used. If neither node-name nor class-system-id is specified, the caller’s node is used. See the class-system-id argument for more information.

The node name string must contain from 1 to 15 characters and must correspond exactly to the OpenVMS node name; no trailing blanks nor abbreviations are permitted.
Description

LIB$GETSYI provides the following features in addition to those provided by the $GETSYI system service:

- Instead of a list of item descriptors, which may be difficult to construct in high-level languages, the single item desired is specified as an integer code which is passed by reference. Results are written to separate arguments.
- For items which return numeric values, LIB$GETSYI can optionally provide a formatted string interpretation of the value.
- For string arguments, all string classes supported by the Run-Time Library are understood.
- Calls to LIB$GETSYI are synchronous. LIB$GETSYI calls LIB$GET_EF to allocate a local event flag number for synchronization.

All item codes that can be used with the $GETSYI system service may be used as the item-code argument to LIB$GETSYI. See the description of the $GETSYI system service for a list of the defined item codes. Note that the symbolic names for these items are defined in symbol libraries in module $SYIDEF (where appropriate) supplied by HP.

Value Formats

By using the resultant-value and resultant-string arguments to LIB$GETSYI, you can request that the information be returned in two ways. For each item described as a “string” in the table of Item Codes for the $GETSYI service, the value is returned in resultant-string. For all other items—those which have numeric values—the numeric representation is returned in resultant-value (if specified), and an unsigned decimal integer representation is stored in resultant-string.

Each formatted item is written left-justified; resultant-length, if specified, gives the number of characters used.

See the HP OpenVMS System Services Reference Manual: A–GETUAI for a description of the $GETSYI system service.

Note

This routine calls LIB$GET_EF. Please read the note in the Description section of that routine.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$ _BADPARAM</td>
<td>Unrecognized item code. The item-code argument was not recognized as valid by $GETSYI.</td>
</tr>
<tr>
<td>LIB$ _INSEF</td>
<td>Insufficient event flags. A local event flag number could not be allocated by a call to LIB$GET_EF.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$GETSYI

LIB$_INVARG  Invalid arguments. The $GETSYI Item Code describes the item as a “string”, and no resultant-string argument was specified.

LIB$_INVSTRDES Invalid string descriptor. The descriptor of the resultant-string argument is not a valid descriptor.

LIB$_STRTRU String truncated. This is an alternate success return status. The resultant-string argument could not contain all the characters of the returned item.

LIB$_WRONUMARG Wrong number of arguments. An incorrect number of arguments was passed to LIB$GETSYI.

Any condition values returned by LIB$SCOPY_xxx, or the $GETSYI system service.
**LIB$GET_ACCNAM**

*Get Access Name Table for Protected Object Class (by Name)*

The Get Access Name Table for Protected Object Class (by Name) routine is a simplified interface to the $GET_SECURITY system service, and returns a pointer to the access name table for a protected object class that is specified by name.

**Format**

```
LIB$GET_ACCNAM [clsnam], [objnam], accnam
```

**Returns**

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

**Arguments**

### clsnam

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

The address of a character-string descriptor pointing to the name of a protected object class. This argument is optional and defaults to FILE.

### objnam

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

The address of a character-string descriptor pointing to the name of a protected object. This argument is optional. If it is omitted, the access name table returned is that used for objects of the class specified by the clsnam argument.

### accnam

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

The address of a longword into which this routine writes the address of the access name table.
LIB$ Routines
LIB$GET_ACCNAM

Description

LIB$GET_ACCNAM returns the address of the access name table for the specified protected object. The format of the table is a vector of 32 quadword string descriptors. Each table entry points to the name of an access type. The index into the vector is the bit position in an access-desired mask. Undefined access types have zero-length names. The table can be used as input to the LIB$PARSE_SOGW_PROT, LIB$FORMAT_SOGW_PROT, LIB$PARSE_ACCESS_CODE, $PARSE_ACL, and $FORMAT_ACL routines.

The semantics of this routine are as follows:

1. If the clsnam parameter is omitted, clsnam defaults to “FILE.”
2. If clsnam is not the name of an object class, then the routine returns an error status (SS$_NOCLASS), and the value of accnam is undefined.
3. If the objnam parameter is omitted, then accnam points to the table corresponding to clsnam, and the routine returns a success status (SS$_NORMAL). The table returned is the table of access names for a new object of class clsnam.
4. Otherwise, if clsnam and objnam do in fact name a protected object, then accnam points to the table corresponding to the protected object class, and the routine returns a success status (SS$_NORMAL).
5. Otherwise, if clsnam and objnam do not name a protected object, then the routine returns an error status (the exact status value depends on the security class), and the value of accnam is undefined.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_NOCLASS</td>
<td>No matching object class was found.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>The accnam argument was omitted.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>

In addition, any completion status may be returned from $GET_SECURITY.
The Get Access Name Table for Protected Object Class (by Context) routine is a simplified interface to the $GET_SECURITY system service, and returns a pointer to the access name table for a protected object class that is specified by a context longword returned from $GET_SECURITY or $SET_SECURITY.

Format

LIB$GET_ACCNAM_BY_CONTEXT  contxt,accnam

Returns

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

Arguments

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

The address of a nonzero longword context value returned by $GET_SECURITY or $SET_SECURITY.

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

The address of a longword into which this routine writes the address of the access name table.

Description

LIB$GET_ACCNAM_BY_CONTEXT returns the address of the access name table for the specified protected object class. The format of the table is a vector of 32 quadword string descriptors. Each table entry points to the name of an access type. The index into the vector is the bit position in an access-desired mask. Undefined access types have zero-length names. The table can be used as input to the LIB$PARSE_SOGW_PROT, LIB$FORMAT_SOGW_PROT, LIB$PARSE_ACCESS_CODE, $PARSE_ACL, and $FORMAT_ACL routines.
The semantics of this routine are as follows:

- If the `contxt` argument is valid, then the `accnam` argument points to the table corresponding to the protected object class, and the routine returns a success status (SS$\_NORMAL).

- If the `contxt` argument is not valid, then the routine returns an error status, and the value of `accnam` is undefined.

**Condition Values Returned**

- **SS$\_NORMAL**  
  Routine successfully completed.

- **LIB$\_WRONUMARG**  
  Wrong number of arguments.

In addition, error status may be returned from $GET\_SECURITY.
LIB$GET_COMMAND
Get Line from SYS$COMMAND

The Get Line from SYS$COMMAND routine gets one record of ASCII text from the current controlling input device, specified by the logical name SYS$COMMAND.

Format

LIB$GET_COMMAND resultant-string [,prompt-string] [,resultant-length]

Returns

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

Arguments

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

String that LIB$GET_COMMAND gets from SYS$COMMAND. The resultant-string argument is the address of a descriptor pointing to this string.

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

Prompt message that LIB$GET_COMMAND displays on the controlling terminal. The prompt-string argument is the address of a descriptor pointing to the prompt. Any string can be a valid prompt. By convention however, a prompt string consists of text followed by a colon ( : ), a space, and no carriage-return/line-feed combination. The maximum size of the prompt message is 255 characters. If the controlling input device is not a terminal, this argument is ignored.

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

Number of bytes written into resultant-string by LIB$GET_COMMAND, not counting padding in the case of a fixed string. The resultant-length argument is the address of an unsigned word containing this length. If the input string is truncated to the size specified in the resultant-string descriptor, resultant-length is set to this size. Therefore, resultant-length can always be used by the calling program to access a valid substring of resultant-string.
LIB$ Routines
LIB$GET_COMMAND

Description
LIB$GET_COMMAND uses the OpenVMS RMS $GET service (see the OpenVMS Record Management Services Reference Manual) to get one record of ASCII text from the current controlling input device, specified by SYS$COMMAND.

When you log in, the OpenVMS operating system creates three files as default I/O control streams for your process.

- SYS$INPUT, your default input device
- SYS$OUTPUT, your default output device
- SYS$COMMAND, the device that supplies the commands to your process

These files remain open until you log out. They are the interface between your interactive input and output or your batch commands and the OpenVMS software. Initially, all three files are equated with the terminal. However, with the DCL command ASSIGN, you can change these assignments to obtain information from a file or put information into a file. SYS$INPUT and SYS$COMMAND are usually identical, but the input and command streams can be different. For example, during the execution of an indirect command file from an interactive terminal, SYS$COMMAND refers to the terminal and SYS$INPUT refers to the command file.

LIB$GET_COMMAND opens file SYS$COMMAND on the first call. The RMS internal stream identifier (ISI) is stored in the routine's static storage for subsequent calls. Hence, this routine is not AST reentrant.

If prompt-string is provided and if the SYS$COMMAND device is a terminal, LIB$GET_COMMAND displays the prompt message. If the device is not a terminal, the prompt-string is ignored.

LIB$GET_COMMAND is used when a program needs input from some source other than the current input stream. Usually, it is used to input from the terminal rather than from an indirect command file. For example, a program may ask a question which cannot be answered by an indirect command file entry. In this case the program would call LIB$GET_COMMAND to get one record of ASCII text from SYS$COMMAND, the terminal.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. RMS completion status.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>An internal consistency check on Run-Time Library data structures has failed.</td>
</tr>
<tr>
<td></td>
<td>This may indicate a programming error in the Run-Time Library, or that your</td>
</tr>
<tr>
<td></td>
<td>program may have overwritten those data structures.</td>
</tr>
<tr>
<td>LIB$_INPSTRTRU</td>
<td>The input string has been truncated to the size specified in the resultant-</td>
</tr>
<tr>
<td></td>
<td>string descriptor (fixed-length strings only). The resultant-length argument is</td>
</tr>
<tr>
<td></td>
<td>also set to this size. This is an error (as opposed to LIB$_STRTRU which is</td>
</tr>
<tr>
<td></td>
<td>a success) because the truncation is not under program control.</td>
</tr>
</tbody>
</table>
LIB$_INSVIRMEM  Insufficient virtual memory to allocate the dynamic string.

LIB$_INVARG  Invalid arguments. The descriptor class field is not a recognized code or is zero.

Any valid RMS status code.

Any code returned by LIB$GET_VM, LIB$GET_VM_64, LIB$SCOPY_R_DX, or LIB$SCOPY_R_DX_64.
LIB$ Get Common
Get String from Common

The Get String from Common routine copies a string in the common area to the destination string. (The common area is an area of storage that remains defined across multiple image activations in a process.) The string length is taken from the first longword of the common area.

Format

LIB$GET_COMMON resultant-string [resultant-length]

Returns

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

Arguments

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

Destination string into which LIB$GET_COMMON writes the string copied from the common area. The resultant-string argument is the address of a descriptor pointing to the destination string.

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

Number of characters written into resultant-string by LIB$GET_COMMON, not counting padding in the case of a fixed-length string. The resultant-length argument is the address of an unsigned word integer containing the number of characters copied. If the input string is truncated to the size specified in the resultant-string descriptor, resultant-length is set to this size. Therefore, resultant-length can always be used by the calling program to access a valid substring of resultant-string.

Description

LIB$PUT_COMMON allows a program to copy a string into the process's common storage area. This area remains defined during multiple image activations. LIB$GET_COMMON allows a program to copy a string from the common area into a destination string. The programs reading and writing the data in the common area must agree upon its amount and format.

The maximum number of characters that can be copied is 252. The actual number of characters copied is returned by the optional argument, resultant-length (if given).
You can use LIB$PUT_COMMON and LIB$GET_COMMON to pass information between images run successively, such as chained images run by LIB$RUN_PROGRAM. Since the common area is unique to each process, do not use LIB$GET_COMMON and LIB$PUT_COMMON to share information across processes.

**Condition Values Returned**

- **SS$_NORMAL**  Routine successfully completed.
- **LIB$_FATERRLIB**  Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to HP.
- **LIB$_INSVIRMEM**  Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.
- **LIB$_INVSTRDES**  Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.
- **LIB$_STRTRU**  Successfully completed. The string was longer than the buffer and was truncated.
LIB$ Routines
LIB$GET_CURR_INVO_CONTEXT (Alpha and I64 Only)

LIB$GET_CURR_INVO_CONTEXT (Alpha and I64 Only)
Get Current Invocation Context

The Get Current Invocation Context routine gets the current invocation context of any active procedure.

A thread can obtain the invocation context of a current procedure using the following function format:

Format

LIB$GET_CURR_INVO_CONTEXT invo_context

Returns

None.

Argument

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: write only
mechanism: by reference

Address of an invocation context block into which the procedure context of the caller will be written.

Description

LIB$GET_CURR_INVO_CONTEXT gets the current invocation context of any active procedure.

See the HP OpenVMS Calling Standard manual for additional information.

Condition Values Returned

None.
LIB$GET_DATE_FORMAT
Get the User's Date Input Format

The Get the User's Date Input Format routine returns information about the user's choice of a date/time input format.

Format

LIB$GET_DATE_FORMAT  format-string [, user-context]

Returns

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

Arguments

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

Receives the translation of LIB$DT_INPUT_FORMAT. The format-string argument is the address of a descriptor pointing to this format string.

user-context
OpenVMS usage:  context
type:  longword (unsigned)
access:  modify
mechanism:  by reference

Context variable that retains the translation context over multiple calls to this routine. The user-context argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

The user-context argument is optional. However, if a context cell is not passed, LIB$GET_DATE_FORMAT may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, HP recommends that users specify a different context cell for each calling thread.

Description

Depending on which method was used to specify the input formats, LIB$GET_DATE_FORMAT either translates the logicals LIB$DT_INPUT_FORMAT and LIB$FORMAT_MNEMONICS, or uses the preinitialized context components LIB$K_FORMAT_MNEMONICS and LIB$K_INPUT_FORMAT to return the user's specified date/time input format in a "legible" form. This format string can then be used as a guideline for entering date/time strings.
The string returned by LIB$GET_DATE_FORMAT parallels the currently defined input format string, consisting of the format punctuation (with most whitespace compressed) and “legible” mnemonics representing the various format fields. The English (default) versions of these mnemonics are as follows:

<table>
<thead>
<tr>
<th>Format Field</th>
<th>Legible Mnemonic (Default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>YYYY&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Numeric month</td>
<td>MM</td>
</tr>
<tr>
<td>Alphabetic month</td>
<td>MONTH</td>
</tr>
<tr>
<td>Numeric day</td>
<td>DD</td>
</tr>
<tr>
<td>Hours (12- or 24-hour)</td>
<td>HH</td>
</tr>
<tr>
<td>Minutes</td>
<td>MM</td>
</tr>
<tr>
<td>Seconds</td>
<td>SS</td>
</tr>
<tr>
<td>Fractional seconds</td>
<td>CC&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Meridiem indicator</td>
<td>AM/PM</td>
</tr>
</tbody>
</table>

<sup>1</sup>This variable-length field mnemonic has a numeric suffix representing the number of digits allowed or required in the field. For instance, YYYY4 indicates a four-digit year field.

For example, consider the following input format string:

```$ DEFINE LIB$DT_INPUT_FORMAT - _$ "$'MAAU D0, Y2 Y2!H02:!M0:!S0.!C4 AM/PM"
```

If LIB$GET_DATE_FORMAT were called for this format string, the format string returned would be as follows:

MONTH DD, YYYY2 HH:MM:SS.CC4 AM/PM

See the *HP OpenVMS Programming Concepts Manual* for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.

**Condition Values Returned**

- **SS$_$NORMAL**: Routine successfully completed.
- **LIB$_$DEFFORUSE**: Default format used; unable to determine desired format.
- **LIB$_$ENGLUSED**: English used; unable to determine or use desired language.
- **LIB$_$ILLFORMAT**: Illegal format string.
- **LIB$_$INVARG**: Invalid argument; a required argument was not specified.
- **LIB$_$INVSTRDES**: Invalid input string descriptor.
- **LIB$_$REENTRANCY**: Reentrancy detected.
- **LIB$_$STRTRU**: String truncated.
- **LIB$_$UNRFORCOD**: Unrecognized format code.
- **LIB$_$WRONUMARG**: Wrong number of arguments.

Any condition value returned by LIB$GET_VM, LIB$SCOPY_R_DX, and LIB$SFREE1_DD.
LIB$GET_EF
Get Event Flag

The Get Event Flag routine allocates one local event flag from a processwide pool and returns the number of the allocated flag to the caller. If no flags are available, LIB$GET_EF returns an error as its function value.

Format

LIB$GET_EF event-flag-number

Returns

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

Argument

event-flag-number
OpenVMS usage: ef_number
type: longword (unsigned)
access: write only
mechanism: by reference

Number of the local event flag that LIB$GET_EF allocated, or –1 if no local event flag was available. The event-flag-number argument is the address of a signed longword integer into which LIB$GET_EF writes the number of the local event flag that it allocates.

Description

LIB$GET_EF and LIB$FREE_EF cause local event flags to be allocated and deallocated at run time, so that your routine remains independent of other routines executing in the same process.

LIB$GET_EF provides your program with an arbitrary event flag number. You can obtain a specific event flag number by calling LIB$RESERVE_EF.

Note

Beware of running multiple images linked with /NOSYSSHR in the same process and having more than one image make calls to LIB$GET_EF. Each image contains its own copy of the event flag bit array that is designed to be process-wide and synchronize ownership of event flags. Multiple calls to LIB$GET_EF could cause the same event flag to be allocated more than once.

See the HP OpenVMS Programming Concepts Manual for more information.
## Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INSEF</td>
<td>Insufficient event flags. There were no more event flags available for allocation.</td>
</tr>
</tbody>
</table>
LIB$GET_FOREIGN
Get Foreign Command Line

The Get Foreign Command Line routine requests the calling image’s command language interpreter (CLI) to return the contents of the “foreign command” line that activated the current image.

Format

LIB$GET_FOREIGN resultant-string [,prompt-string] [,resultant-length] [,flags]

Returns

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

Arguments

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

String that LIB$GET_FOREIGN uses to receive the foreign command line. The resultant-string argument is the address of a descriptor pointing to this string. If the foreign command text returned was obtained by a prompt to SYS$INPUT (see the description of flags), the text is translated to uppercase so as to be more consistent with text returned from the CLI.

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

Optional user-supplied prompt for text that LIB$GET_FOREIGN uses if no command-line text is available. The prompt-string argument is the address of a descriptor pointing to the user prompt. If omitted, no prompting is performed. It is recommended that prompt-string be specified. If prompt-string is omitted and if no command-line text is available, a zero-length string will be returned.

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

Number of bytes written into resultant-string by LIB$GET_FOREIGN, not counting padding in the case of a fixed-length resultant-string. The resultant-length argument is the address of an unsigned word into which LIB$GET_FOREIGN writes the number of bytes.
LIB$ GET_FOREIGN

flags

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

Value that LIB$GET_FOREIGN uses to control whether or not prompting is
to be performed. The flags argument is the address of an unsigned longword
integer containing this value. If the low bit of flags is zero, or if flags is omitted,
prompting is done only if the CLI does not return a command line. If the low
bit is 1, prompting is done unconditionally. If specified, flags is set to 1 before
returning to the caller.

The primary use of flags is to allow a utility program to be invoked once
with subcommand text on the command line, and then to repeatedly prompt
for further subcommands from SYS$INPUT. This is accomplished by calling
LIB$GET_FOREIGN repeatedly, specifying in the call a prompt-string string
and a flags variable that is initialized to zero at the beginning of the program.
The first call gets the subcommand text from the command line, after which flags
will be set to 1, causing further subcommands to be requested through prompts
to SYS$INPUT.

Description

LIB$GET_FOREIGN returns the contents of the command line that you use to
activate an image. It can be used to give your program access to the qualifiers of
a foreign command or to prompt for further command line text.

A foreign command is a command that you can define and then use as if it were
a DCL or MCR command in order to run a program. When you use the foreign
command at command level, the CLI parses the foreign command only and
activates the image. It ignores any options or qualifiers that you have defined for
the foreign command. Once the CLI has activated the image, the program can
call LIB$GET_FOREIGN to obtain and parse the remainder of the command line
(after the command itself) for whatever options it may contain. See the OpenVMS
User’s Manual for information on how to define a foreign command.

If no command line is available, LIB$GET_FOREIGN can optionally call
LIB$GET_INPUT to prompt the user for command text. If desired, LIB$GET_-
FOREIGN can be called repetitively, returning the command line on the first call,
but prompting for further text on subsequent calls.

LIB$GET_FOREIGN can also be used for images invoked by the RUN command,
for which further text must be obtained by prompting. Such an image can also be
invoked by the DCL command MCR or by the MCR CLI. The text following the
image name will be returned to the executing image.

The action of LIB$GET_FOREIGN depends on the environment in which the
image is activated.

- If you use a foreign command to invoke the image, you can call LIB$GET_
FOREIGN to obtain the command qualifiers following the foreign command.
You can also use LIB$GET_FOREIGN to prompt repeatedly for more
qualifiers after the command. This technique is shown in the example.

- If the image is in the SYS$SYSTEM: directory, the image can be invoked
by the DCL command MCR or by the MCR CLI. In this case, LIB$GET_
FOREIGN returns the command line text following the image name.
If the image is invoked by a DCL command RUN, LIB$GET_FOREIGN can be used to prompt for additional text.

If the image is not invoked by a foreign command or MCR, or if there is no information remaining on the command line, and the user-supplied prompt is present, LIB$GET_INPUT is called to prompt for a command line. If the prompt is not present, LIB$GET_FOREIGN returns a zero length string.

**Condition Values Returned**

- **SS$_NORMAL** Routine successfully completed.
- **LIB$_FATERRLIB** A fatal internal error was detected.
- **LIB$_INPSTRTRU** The input string was truncated. The *resultant-string* argument could not contain all of the characters. The *resultant-length* argument reflects the truncated length.
- **LIB$_INSVIRMEM** Insufficient virtual memory.
- **LIB$_INVSTRDES** Invalid string descriptor.

A condition value returned by OpenVMS RMS. SYS$INPUT was prompted for command text and RMS returned an error. The most typical error will be RMS$$_EOF, end-of-file.

**Example**

```pli
EXAMPLE: ROUTINE OPTIONS (MAIN);
%INCLUDE $STSDEF; /* Status-testing definitions */
DECLARE COMMAND LINE CHARACTER(80) VARYING,
   PROMPT FLAG FIXED BINARY(31) INIT(0),
   LIB$GET_FOREIGN ENTRY (CHARACTER(*) VARYING,
                              CHARACTER(*) VARYING,
                              FIXED BINARY(15),
                              FIXED BINARY(31))
                     OPTIONS(VARIABLE) RETURNS (FIXED BINARY(31)),
      RMS$_EOF GLOBALREF FIXED BINARY(31) VALUE;
/* Repeat forever calling LIB$GET_FOREIGN to obtain subcommand text and print the text. Exit when an end-of-file is found. */
DO WHILE ('1'B); /* Do while TRUE */
   STS$VALUE = LIB$GET_FOREIGN
      (COMMAND LINE,'Input: ','
      PROMPT FLAG);
   IF STS$SUCCESS THEN
      PUT LIST (' Command was ','COMMAND_LINE);
   ELSE DO;
      IF STS$VALUE ^= RMS$_EOF THEN
         PUT LIST ('Error encountered');
         RETURN;
      END;
      PUT SKIP; /* Skip to next line */
   END; /* End of DO WHILE loop */
END;
```

This PL/I example shows the use of the optional *flags* argument to permit repeated calls to LIB$GET_FOREIGN. The command line text is retrieved on the first pass only; after the first pass, the program prompts from SYS$INPUT.
LIB$GET_FULLNAME_OFFSET
Get the Offset to the Starting Position of the Most Significant Part of a Full Name

The Get the Offset to the Starting Position of the Most Significant Part of a Full Name routine returns the offset to the starting position of the most significant part of a full name.† The most significant part of a full name is determined by the underlying network services.

Format

LIB$GET_FULLNAME_OFFSET   fullname, offset

Returns

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

Arguments

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

Full name. The fullname argument contains the address of the descriptor pointing to this full name string.

The error LIB$_INVARG is returned if fullname contains an invalid full name, points to a null string, or contains more than 1024 characters. The error LIB$_INVSTRDES is returned if fullname is an invalid descriptor.

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

The offset in bytes of the starting position of the most significant part of fullname. The offset argument is the address of an unsigned word that contains this offset.

The offset argument contains an unusable result when LIB$GET_FULLNAME_OFFSET returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
Description

This routine returns the byte offset of the starting position of the most significant part of the input full name. The returned offset can be used to position the display of a full name in a fixed-size output region, for example, scroll regions in DECwindows applications. The most significant part of a full name is determined by the underlying network services.

You must validate **fullname** by expanding it with LIB$EXPAND_NO DENAME before calling LIB$GET_FULLNAME_OFFSET. LIB$GET_FULLNAME_OFFSET returns the error LIB$_INVARG if **fullname** is invalid.

In a DECnet for OpenVMS environment, processing a DECnet-Plus for OpenVMS full name using LIB$GET_FULLNAME_OFFSET results in the error condition LIB$_INVARG.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument:</td>
</tr>
<tr>
<td></td>
<td>• <strong>fullname</strong> is invalid.</td>
</tr>
<tr>
<td></td>
<td>• <strong>fullname</strong> points to a null string.</td>
</tr>
<tr>
<td></td>
<td>• The length of the full name is more than 1024 characters.</td>
</tr>
<tr>
<td></td>
<td>• Processing a DECnet-Plus for OpenVMS node name in a DECnet for OpenVMS</td>
</tr>
<tr>
<td></td>
<td>environment is invalid.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>

Any condition value returned by the $IPC DECnet service.

Examples

The following table gives some examples of the results of LIB$GET_FULLNAME_OFFSET:

<table>
<thead>
<tr>
<th>Full Name</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>0</td>
</tr>
<tr>
<td>DEC:.FOO.NODE</td>
<td>9</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$GET_HOSTNAME

LIB$GET_HOSTNAME
Get Host Node Name

The Get Host Node Name routine returns the host node name of the local system.
†

Format

LIB$GET_HOSTNAME  hostname [,resultant-length] [,flags]

Returns

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

Arguments

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

The host node name. The hostname argument contains the address of a
descriptor pointing to the host node name. LIB$GET_HOSTNAME writes the
host node-name string into the buffer pointed to by the hostname descriptor.

The error LIB$_INVSTRDES is returned if hostname is an invalid descriptor.

The length field of the hostname descriptor is not updated unless hostname is
dynamic descriptor with a length less than the host node name to be returned.
Refer to the OpenVMS RTL String Manipulation (STR$) Manual for dynamic
string descriptor usage.

The hostname argument contains an unusable result when LIB$GET_HOSTNAME returns in error.

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

Length of the host node name. The resultant-length argument is the address of
an unsigned word that contains this length in bytes.

The resultant-length argument contains an unusable result when LIB$GET_HOSTNAME returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit
descriptors, if applicable, is planned for this routine.
flags
OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by value

The value LIB$GET_HOSTNAME uses to control the form of the host node name that it returns in the output descriptor hostname. If flags is equal to 0, or if flags is omitted, the host node name returned is in the network usable form. If flags is equal to 1, the host node name returned is in the parsable form.

Unused bits in flags must be 0. Nonzero unused bits result in the error condition LIB$_INVARG.

Description

This routine returns the host node name. The routine searches for the first host node name using the following order:

1. Get host node name from $GETSYI system service.
2. Translate the executive mode logical SYS$NODE_FULLNAME once.
3. Translate the executive mode logical SYS$NODE once.

The error LIB$_NOHOSNAM is returned if no host node name is found.

LIB$GET_HOSTNAME can return the host node name in the following two forms:

- Network usable form — The form that can be passed directly to the network. This form does not contain unnecessary double quotation marks (double quotation marks ["] that are not part of the node name) and also does not contain trailing double colons, for example: DEC:.FOO."simple name with spaces".
- Parsable form — The form that can be passed directly to the part of the system that does node-name syntax parsing, for example, $FILESAN and DCL command parsing. This form contains trailing double colons and is fully quoted if there are special characters. Individual double quotation marks (") that are part of a simple name are doubled (""), for example: "DEC:.FOO.""simple name with spaces""" captive.

You must use double quotation marks for a node name with special characters to facilitate correct parsing.

If the returned node name overflows the buffer pointed to by hostname, the host node name is truncated at the end, and the alternate success status LIB$_STRTRU is returned.

The resultant-length argument, if supplied, is set to the length of the node-name string copied to the output buffer pointed to by hostname.
LIB$ Routines
LIB$GET_HOSTNAME

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. Characters are truncated in the output buffer pointed to by <code>hostname</code>.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid input argument. Unused bits in <code>flags</code> are not set to 0.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
<tr>
<td>LIB$_NOHOSNAM</td>
<td>No host node name found.</td>
</tr>
</tbody>
</table>

Any condition value returned by LIB$SCOPY_R_DX, or the $FILESCAN system service.
LIB$GET_INPUT
Get Line from SYS$INPUT

The Get Line from SYS$INPUT routine gets one record of ASCII text from the current controlling input device, specified by SYS$INPUT.

Format

LIB$GET_INPUT resultant-string [,prompt-string] [,resultant-length]

Returns

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

Arguments

resultant-string
OpenVMS usage: char_string
type: character string
access: write only
mechanism: by descriptor
String that LIB$GET_INPUT gets from the input device. The resultant-string argument is the address of a descriptor pointing to the character string into which LIB$GET_INPUT writes the text received from the current input device.

prompt-string
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor
Prompt message that is displayed on the controlling terminal. The prompt-string argument is the address of a descriptor containing the prompt. Any string can be a valid prompt. By convention however, a prompt consists of text followed by a colon (:) a space, and no carriage-return/line-feed combination. The maximum size of the prompt message is 255 characters. If the controlling input device is not a terminal, this argument is ignored.

resultant-length
OpenVMS usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference
Number of bytes written into resultant-string by LIB$GET_INPUT, not counting padding in the case of a fixed string. The resultant-length argument is the address of an unsigned word containing this number. If the input string is truncated to the size specified in the resultant-string descriptor, resultant-length is set to this size. Therefore, resultant-length can always be used by the calling program to access a valid substring of resultant-string.
LIB$ Routines
LIB$GET_INPUT

Description
LIB$GET_INPUT uses the OpenVMS RMS $GET service to get one record of ASCII text from the current controlling input device, specified by SYS$INPUT. (For more information about the RMS $GET service, see the OpenVMS Record Management Services Reference Manual.)

When you log in, the OpenVMS operating system creates three files as default I/O control streams for your process.

- SYS$INPUT, your default input device
- SYS$OUTPUT, your default output device
- SYS$COMMAND, the device that supplies the commands to your process

These files remain open until you log out. They are the interface between your interactive input and output or your batch commands and the OpenVMS software. Initially, all three names are equated with the terminal. However, with the DCL command ASSIGN, you can change these assignments to obtain information from a file or put information into a file. SYS$INPUT and SYS$COMMAND are usually identical, but the input and command streams can be different. For example, during the execution of an indirect command file from an interactive terminal, SYS$COMMAND refers to the terminal and SYS$INPUT refers to the command file.

LIB$GET_INPUT opens file SYS$INPUT on the first call. The RMS internal stream identifier (ISI) is stored in the routine's static storage for subsequent calls.

If prompt-string is provided and the SYS$INPUT device is a terminal, LIB$GET_INPUT displays the prompt message. If the device is not a terminal, the prompt-string argument is ignored.

If you want to get input from some source other than the current input stream, use LIB$GET_COMMAND.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. RMS completion status.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>An internal consistency check on Run-Time Library data structures has failed. This may indicate a programming error in the Run-Time Library, or that your program may have overwritten those data structures.</td>
</tr>
<tr>
<td>LIB$_INPSTRTRU</td>
<td>The input string has been truncated to the size specified in the resultant-string descriptor (fixed-length strings only). The resultant-length argument is also set to this size. This is an error (as opposed to LIB$_STRTRU, which is a success) because the truncation is not under program control.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory to allocate the dynamic string.</td>
</tr>
</tbody>
</table>
LIB$_INVARG  Invalid arguments. The descriptor class field is not a recognized code or is zero.

Any RMS condition value returned by $GET.

Any condition value returned by LIB$GET_VM, LIB$GET_VM_64, LIB$SCOPY_R_DX, or LIB$SCOPY_R_DX_64.
LIB$ GET_INVO_CONTEXT (Alpha and I64 Only)
Get Invocation Context

The Get Invocation Context routine gets the invocation context of any active procedure.

Format

LIB$GET_INVO_CONTEXT invo_handle, invo_context

Returns

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

Arguments

**invo_handle**
OpenVMS usage: invo_handle
type: longword (unsigned)
access: read only
mechanism: by value
Handle for the desired invocation. Returned by LIB$GET_INVO_HANDLE.

**invo_context**
OpenVMS usage: invo_context_blk
type: structure
access: write only
mechanism: by reference
Address of an invocation context block into which the procedure context of the frame specified by **invo_handle** will be written.

Description

LIB$GET_INVO_CONTEXT gets the invocation context of any active procedure.

_________________________ Note __________________________

If **invo_handle** does not represent any procedure context in the active call chain, the new contents of the invocation context block are unpredictable.

_____________________________________________________

See the *HP OpenVMS Calling Standard* manual for additional information.

Condition Values Returned

0 Indicates failure.
0 Indicates success.
LIB$GET_INVO.Handle (Alpha and I64 Only)
Get Invocation Handle

The Get Invocation Handle routine gets an invocation handle of any active procedure.

A thread can obtain an invocation handle corresponding to any invocation context block by using the following function format.

Format

LIB$GET_INVO_HANDLE invo_context

Returns

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

Invocation handle of the invocation context that was passed. If the returned value is LIB$K_INVO_HANDLE_NULL, the invocation context that was passed was invalid.

Argument

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: read only
mechanism: by reference

Address of an invocation context block. Here, only the frame pointer and stack pointer fields of an invocation context block must be defined.

Description

LIB$GET_INVO_HANDLE gets an invocation handle of any active procedure.

See the HP OpenVMS Calling Standard manual for additional information.

Condition Values Returned

None.
LIB$ Routines
LIB$GET_LOGICAL

LIB$GET_LOGICAL
Get Logical Name

The Get Logical Name routine calls the system service routine $TRNLNM to return information about a logical name.

Format


Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

Arguments

logical-name
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor
Logical name for which LIB$GET_LOGICAL searches. The logical-name argument is the address of a descriptor pointing to the logical name string.

resultant-string
OpenVMS usage: char_string
type: character string
access: write only
mechanism: by descriptor
Logical name equivalent returned. The resultant-string argument is the address of a descriptor pointing to a character string into which LIB$GET_LOGICAL writes the equivalence name of the logical.

resultant-length
OpenVMS usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference
Length of the equivalence name string returned by LIB$GET_LOGICAL. The resultant-length argument is the address of an unsigned word integer into which LIB$GET_LOGICAL writes the length.

table-name
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor
Name of the table in which to search for the logical name. The **table-name** argument contains the address of a descriptor pointing to a character string which contains the table name. If no table is specified, LNMS$FILE_DEV is used.

**max-index**

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

Largest equivalence name index. Each equivalence name for the logical name has an index associated with it. The **max-index** argument is the address of a signed longword integer into which LIB$GET_LOGICAL write the value. If no equivalence names (and, therefore, no index values) exist, LIB$GET_LOGICAL returns a value of -1.

**index**

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

Equivalence name index value. LIB$GET_LOGICAL will return the equivalence name string that has the specified index value. The **index** argument is the address of an unsigned longword integer specifying the index value.

**acmode**

OpenVMS usage: access_mode  
type: byte (unsigned)  
access: read only  
mechanism: by reference  

Access mode to be used in the translation. The **acmode** argument is the address of a byte specifying the access mode. The $PSLDEF macro defines symbolic names for the four access modes.

When you specify the **acmode** argument, all names at access modes which are less privileged than the specified access mode are ignored.

If you do not specify **acmode**, the translation is performed without regard to access mode; however, the translation process proceeds from the outermost to the innermost access modes. Thus, if two logical names with the same name, but at different access modes, exist in the same table, the name with the outermost access mode is translated.

**flags**

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

Flags controlling the search for the logical name. The **flags** argument is the address of a longword integer that contains the control flags. The $LNMDEF macro defines these flags. Currently only bit 0 of this argument is used.
LIB$ Routines
LIB$GET_LOGICAL

Bit Value Description

| 0 | LNM$M_CASE_BLIND | If set, LIB$GET_LOGICAL does not distinguish between uppercase and lowercase letters in the logical name to be translated. |

This is an optional argument. If omitted the default is 0.

Description

LIB$GET_LOGICAL provides a simplified interface to the $TRNLNM system service. It provides most of the features found in $TRNLNM with some additional benefits. For string arguments, all string classes supported by the Run-Time Library are understood. The list of item descriptors, which may be difficult to construct in high-level languages, is handled internally by LIB$GET_LOGICAL.

See the description of the $TRNLNM system service in the HP OpenVMS System Services Reference Manual for more information.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation. Cannot access the location specified.</td>
</tr>
<tr>
<td>SS$_BADPARAM</td>
<td>Bad parameter value.</td>
</tr>
<tr>
<td>SS$_IVLOGNAM</td>
<td>Invalid logical name. The logical name or its value contained more than 255 characters.</td>
</tr>
<tr>
<td>SS$_IVLOGTAB</td>
<td>Invalid logical name table.</td>
</tr>
<tr>
<td>SS$_NOLOGNAM</td>
<td>The logical name was not found in the specified table.</td>
</tr>
<tr>
<td>SS$_NOPRIV</td>
<td>No privileges for attempted operation.</td>
</tr>
<tr>
<td>SS$_TOOMANYNAM</td>
<td>Logical name translation exceeded allowed depth.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Required argument is missing.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Success, but source string truncated.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>
LIB$GET_LUN
Get Logical Unit Number

The Get Logical Unit Number routine allocates one logical unit number from a processwide pool. If a unit is available, its number is returned to the caller. Otherwise, an error is returned as the function value.

Format

LIB$GET_LUN  logical-unit-number

Returns

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

Argument

logical-unit-number
OpenVMS usage:  longword_signed
type:  longword integer (signed)
access:  write only
mechanism:  by reference

Allocated logical unit number or –1 if none was available. The logical-unit-number argument is the address of a longword into which LIB$GET_LUN returns the value of the allocated logical unit. LIB$GET_LUN can allocate logical unit numbers 100 through 119 on VAX, and 100 through 299 on Alpha and I64.

Description

LIB$GET_LUN allocates one logical unit number from a processwide pool. If a unit is available, its number is returned to the caller. Otherwise, an error is returned as the function value.

On VAX systems, LIB$GET_LUN reserves logical unit numbers starting at 119 and continues in descending order through 100.

On Alpha and I64 systems, LIB$GET_LUN reserves logical unit numbers 100 through 299. To maintain compatibility with VAX systems, LIB$GET_LUN reserves logical unit numbers starting at 119 and continues in descending order through 100. When these are exhausted, LIB$GET_LUN reserves logical unit numbers starting at 299 and continues in descending order through 120.

LIB$GET_LUN assumes that the logical unit numbers in the range 0 through 99 may be in use by your program, but it cannot determine which logical unit numbers are actually in use by your program.
LIB$ Routines
LIB$GET_LUN

Call LIB$GET_LUN only from Fortran or BASIC programs. Those languages and
LIB$GET_LUN share the concept of unit numbers and a similar number space.

Note

Beware of running multiple images linked with /NOSYSSHR in the same
process and having more than one image make calls to LIB$GET_LUN.
Each image contains its own copy of the event flag bit array that is
designed to be process-wide and synchronize ownership of event flags.
Multiple calls to LIB$GET_EF could cause the same event flag to be
allocated more than once.

Condition Values Returned

SS$_NORMAL              Routine successfully completed.
LIB$_INSLUN             Insufficient logical unit numbers. No logical unit
                        numbers were available.
LIB$GET_MAXIMUM_DATE_LENGTH
Retrieve the Maximum Length of a Date/Time String

Given an output format and language, the Retrieve the Maximum Length of a Date/Time String routine determines the maximum possible length for the `date-string` string returned by LIB$FORMAT_DATE_TIME.

**Format**

```
LIB$GET_MAXIMUM_DATE_LENGTH date-length [,user-context] [,flags]
```

**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>

**Arguments**

**date-length**

OpenVMS usage: longword_signed

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

Receives the maximum possible length of the `date-string` argument returned to LIB$FORMAT_DATE_TIME. The `date-length` argument is the address of a signed longword that receives this maximum length. The length written to `date-length` reflects the greatest possible length of an output date/time string for the currently selected output format and natural language.

For example, if the selected output date/time format includes the alphabetic, unabbreviated month name (assuming English as the natural language), the longest month name (September) would have to be taken into consideration when determining the maximum possible length of `date-string`.

**user-context**

OpenVMS usage: context

<table>
<thead>
<tr>
<th>type</th>
<th>longword (unsigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>modify</td>
</tr>
<tr>
<td>mechanism</td>
<td>by reference</td>
</tr>
</tbody>
</table>

Context variable that retains the translation context over multiple calls to this routine. The `user-context` argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

The `user-context` parameter is optional. However, if a context cell is not passed, the routine LIB$GET_MAXIMUM_DATE_LENGTH may abort if two threads of execution attempt to manipulate the context area concurrently. Therefore, when calling this routine in situations where reentrancy might occur, such as from AST level, HP recommends that users specify a different context cell for each calling thread.
LIB$ Routines
LIB$GET_MAXIMUM_DATE_LENGTH

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

Bit mask that allows the user to specify whether the date, time, or both are to be included in the calculation of the maximum date length. The flags argument is the address of an unsigned bit mask containing the specified values. Valid values are LIB$M_DATE_FIELDS and LIB$M_TIME_FIELDS. The values specified for flags must correspond to the flags argument passed to LIB$FORMAT_DATE_TIME.

Description

The LIB$GET_MAXIMUM_DATE_LENGTH routine determines the maximum possible length for a formatted date/time string as returned by LIB$FORMAT_DATE_TIME. The maximum length returned takes into account the currently specified output format and natural language; date-length represents the maximum possible length of the string written to the date-string argument of LIB$FORMAT_DATE_TIME.

Consider the following example, in which the output format is defined as follows.

DEFINE LIB$DT_FORMAT LIB$DATE_FORMAT_012, LIB$TIME_FORMAT_012

This date/time format would appear as follows:

!MAU !DD, !Y4 !HH2:!M0 !MIU

Given this format, the maximum possible length for this date/time string is calculated using the longest possible date string followed by a space and the longest possible time string. One example that meets these requirements is as follows (assuming English as the selected language):

SEPTEMBER 21, 2000 11:24 PM

The maximum possible length of this date-string would then be 28.

See the HP OpenVMS Programming Concepts Manual for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_ABSTIMREQ Absolute time required.
LIB$_DEFFORUSE Default format used; unable to determine desired format.
LIB$_ENGLUSED English used by default; unable to translate SYS$LANGUAGE.
LIB$_REENTRANCY Reentrant invocation with same context variable.
LIB$_STRTRU Output string truncated.
LIB$_UNRFORCOD Unrecognized format code.

Any condition value returned by LIB$GET_VM.
LIB$GET_PREV_INVO_CONTEXT (Alpha and I64 Only)

Get Previous Invocation Context

The Get Previous Invocation Context routine gets the previous invocation context of any active procedure.

A thread can obtain the invocation context of the procedure context preceding any other procedure context using the following function format.

**Format**

\[
\text{LIB$GET_PREV_INVO_CONTEXT } \text{invo\_context}
\]

**Returns**

- OpenVMS usage: longword\_unsigned
- type: longword\_(unsigned)
- access: write only
- mechanism: by value

**Argument**

\[
\text{invo\_context}
\]

- OpenVMS usage: invo\_context\_blk
- type: structure
- access: modify
- mechanism: by reference

Address of an invocation context block. The given context block is updated to represent the context of the previous (calling) frame.

For the purposes of this function, the minimum fields of an invocation block that must be defined are those IREG and FREG fields corresponding to registers used by a context whether the registers are preserved or not. Note that the invocation context blocks written by the routines specified in these sections define all possible fields in a context block. Such context blocks satisfy this minimum requirement.

**Description**

LIB$GET_PREV_INVO_CONTEXT gets the previous invocation context of any active procedure.

See the *HP OpenVMS Calling Standard* manual for more information.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The initial context represents the bottom of the call chain.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$GET_PREV_INVO_HANDLE (Alpha and I64 Only)

LIB$GET_PREV_INVO_HANDLE (Alpha and I64 Only)
Get Previous Invocation Handle

The Get Previous Invocation Handle routine gets the previous invocation handle of any active procedure.

A thread can obtain an invocation handle of the procedure context preceding that of a specified procedure context by using the following function format.

Format

LIB$GET_PREV_INVO_HANDLE invo_handle

Returns

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

An invocation handle for the invocation context that is previous to that which was specified as the target.

Argument

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

An invocation handle that represents a target invocation context.

Description

LIB$GET_PREV_INVO_HANDLE gets the previous invocation handle of any active procedure.

See the HP OpenVMS Calling Standard manual for more information.

Condition Values Returned

None.
LIB$GET_SYMBOL
Get Value of CLI Symbol

The Get Value of CLI Symbol routine requests the calling process’s command
language interpreter (CLI) to return the value of a CLI symbol as a string.
LIB$GET_SYMBOL then returns the string to the caller. Optionally, LIB$GET_-
SYMBOL can return the length of the returned value and the table in which the
symbol was found.

Format
LIB$GET_SYMBOL  symbol ,resultant-string [,resultant-length] [,table-type-indicator]

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

Arguments

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

Name of the symbol for which LIB$GET_SYMBOL searches. The symbol
argument is the address of a descriptor pointing to the name of the symbol.
LIB$GET_SYMBOL converts the symbol name to uppercase and removes trailing
blanks before the search. The symbol argument must begin with a letter, a digit,
a dollar sign ($), a hyphen (-), or an underscore (_). The maximum length of
symbol is 255 characters.

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

Value of the returned symbol. The resultant-string argument is the address of
a descriptor pointing to a character string into which LIB$GET_SYMBOL writes
the value of the symbol.

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

Length of the symbol value returned by LIB$GET_SYMBOL. The resultant-
length argument is the address of an unsigned word integer into which
LIB$GET_SYMBOL writes the length.
LIB$ Routines
LIB$GET_SYMBOL

**table-type-indicator**
- OpenVMS usage: longword_signed
- type: longword integer (signed)
- access: write only
- mechanism: by reference

Indicator of which table contained the symbol. The **table-type-indicator** argument is the address of a signed longword integer into which LIB$GET_SYMBOL writes the table indicator.

Possible values of the table indicator are listed below.

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Value</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_CLI_LOCAL_SYM</td>
<td>1</td>
<td>Local symbol table</td>
</tr>
<tr>
<td>LIB$K_CLI_GLOBAL_SYM</td>
<td>2</td>
<td>Global symbol table</td>
</tr>
</tbody>
</table>

LIB$K_CLI_LOCAL_SYM and LIB$K_CLI_GLOBAL_SYM are defined in symbol libraries supplied by HP (macro or module name $LIBCLIDEF) and as global symbols.

**Description**

LIB$GET_SYMBOL first searches the local symbol table for the symbol name, then searches the global symbol table. Numeric values are converted to an ASCII representation of a signed decimal number before being returned.

LIB$GET_SYMBOL is supported for use with the DCL command language interpreter. If used with the MCR CLI, the error status LIB$_NOCLI will be returned.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to get the symbol. In that case, LIB$GET_SYMBOL returns the error status LIB$_NOCLI.

**Condition Values Returned**

- **SS$_NORMAL** Routine successfully completed.
- **LIB$_STRTRU** Routine successfully completed; string truncated. The destination string could not contain all the characters in the symbol value.
- **LIB$_FATERRLIB** Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.
- **LIB$_INSCLIMEM** Insufficient CLI memory. The CLI could not obtain enough virtual memory to perform the function. This may be caused by having too many symbols defined. Deleting some symbol definitions may relieve the situation.
- **LIB$_INSVIRMEM** Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.
<table>
<thead>
<tr>
<th><strong>LIB$ Routines</strong></th>
<th><strong>LIB$GET_SYMBOL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LIB$ _INVSTRDES</strong></td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
<tr>
<td><strong>LIB$ _INVSYMNAM</strong></td>
<td>Invalid symbol name. The symbol name contained more than 255 characters or did not begin with a letter or dollar sign ($).</td>
</tr>
<tr>
<td><strong>LIB$ _NOCLI</strong></td>
<td>No CLI present. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.</td>
</tr>
<tr>
<td><strong>LIB$ _NOSUCHSYM</strong></td>
<td>No such symbol. The symbol was not defined in either the local or global symbol table.</td>
</tr>
<tr>
<td><strong>LIB$ _UNECLIERR</strong></td>
<td>Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL command language interpreter, please report the problem to your HP support representative.</td>
</tr>
</tbody>
</table>
LIB$ GET_UIB_INFO
Unwind Routine

Returns information from the unwind information block (UIB).

Format


Returns

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

Arguments

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

Address of a quadword that contains the virtual address of an unwind information block (UIB).

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

Address of a quadword that contains the GP value that must be added to the UIB condition handler value. Must be specified if handler_fv is specified.

uw_desc_va
OpenVMS usage: address
type: quadword (unsigned)
access: write
mechanism: by reference

Address of a quadword to store the virtual address of the unwind descriptor area. If none is present, then zero is returned. This is an optional argument.

un_desc_len
OpenVMS usage: address
type: quadword (unsigned)
access: write
mechanism: by reference

Address of a quadword to store the length (in bytes) of the unwind descriptor area. If none are present, then zero is returned. This is an optional argument.
**handler_fv**
OpenVMS usage: address
- type: quadword (unsigned)
- access: write
- mechanism: by reference

Address of a quadword to store the function value of the condition handler. If none is present, then zero is returned. This is an optional argument.

**ossd_va**
OpenVMS usage: address
- type: quadword (unsigned)
- access: write
- mechanism: by reference

Address of a quadword to store the address of the operating system-specific data area. If none is present, then zero is returned. This is an optional argument.

**lsda_va**
OpenVMS usage: address
- type: quadword (unsigned)
- access: write
- mechanism: by reference

Address of a quadword to store the address of the language-specific data area (LSDA). If none is present, then zero is returned. This is an optional argument.

**Description**
Takes in the address of an unwind information block (UIB) and the GP value for a routine and returns the addresses of the start of the unwind descriptors (if any), the handler function descriptor (if any), and the operating system-specific data area (if any). The size in bytes of the unwind descriptors is also returned.

**Related Services**
SYS$SET_UNWIND_TABLE, SYS$CLEAR_UNWIND_TABLE, SYS$GET_UNWIND_ENTRY_INFO,

**Condition Values Returned**
- SS$_NORMAL: Routine completed successfully.
- LIB$_INVARG: Bad UIB virtual address.
LIB$GET_USERS_LANGUAGE

Return the User’s Language

The Return the User’s Language routine determines the user’s choice of a natural language. The choice is determined by translating the logical SYS$LANGUAGE.

Format

LIB$GET_USERS_LANGUAGE language

Returns

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

Argument

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

Receives the translation of SYS$LANGUAGE. The language argument is the address of a descriptor pointing to this language name.

Description

The LIB$GET_USERS_LANGUAGE routine translates the logical SYS$LANGUAGE and returns the user’s choice of a natural language. If the logical SYS$LANGUAGE does not translate for some reason, then the language defaults to English and the status LIB$_ENGLUSED is returned.

If a failure or truncation occurs while copying the language name to the language string argument, that error status overrides the LIB$_ENGLUSED or SS$_NORMAL status.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_ENGLUSED English used by default; unable to translate SYS$LANGUAGE.

Any condition value returned by LIB$SCOPY_R_DX.
LIB$GET_VM
Allocate Virtual Memory

The Allocate Virtual Memory routine allocates a specified number of contiguous bytes in the program region and returns the 32-bit virtual address of the first byte allocated. †

Format

LIB$GET_VM  number-of-bytes, base-address [,zone-id]

Returns

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

Arguments

number-of-bytes
OpenVMS usage:  longword_signed
type:  longword integer (signed)
access:  read only
mechanism:  by reference

Number of contiguous bytes that LIB$GET_VM allocates. The number-of-bytes argument is the address of a longword integer containing the number of bytes. LIB$GET_VM allocates enough memory to satisfy the request. Your program should not reference an address before the first byte address allocated (base-address) or beyond the last byte allocated (base-address + number-of-bytes—1) since that space may be assigned to another routine. The value of number-of-bytes must be greater than zero.

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

First virtual address of the contiguous block of bytes allocated by LIB$GET_VM. The base-address argument is the address of an unsigned longword containing this base address.

zone-id
OpenVMS usage:  identifier
type:  longword (unsigned)
access:  read only
mechanism:  by reference

The zone-id argument is the address of a longword that contains a zone identifier created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_†

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
VM_ZONE. This argument is optional. If zone-id is omitted or if the longword contains the value 0, the 32-bit default zone is used.

**Description**

LIB$GET_VM satisfies an allocation request by reusing free memory in the zone, or by obtaining additional memory from the processwide 32-bit page pool managed by LIB$GET_VM_PAGE.

LIB$GET_VM rounds up the value of number-of-bytes to a multiple of the block-size specified for the zone. The first byte allocated is aligned on the boundary specified by the alignment value for the zone.

If you specified allocation filling when you created the zone, LIB$GET_VM will fill each byte allocated. Otherwise, LIB$GET_VM does not initialize the memory and its contents are unpredictable.

All memory allocated by LIB$GET_VM has user mode read/write access, even if the call to LIB$GET_VM was made from a more privileged access mode.

The space allocated by successive calls to LIB$GET_VM may be noncontiguous because another routine can call LIB$GET_VM between your calls. If AST interrupts occur, LIB$GET_VM may allocate space to another routine between execution of any two statements in your program. Even if successive calls to LIB$GET_VM return two contiguous blocks, you must not combine them into one large block that is freed by a single call to LIB$FREE_VM.

LIB$GET_VM is fully reentrant, so it may be called by routines executing at AST level or in an Ada multitasking environment.

Your program must retain the address of the allocated area. This allows you to access or deallocate the space later.

If the zone you are getting was created using the LIB$CREATE_USER_VM_ZONE routine, then you must have an appropriate action routine for the get operation. That is, in your call to LIB$CREATE_USER_VM_ZONE, you must have specified a user-get-routine.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>Invalid zone-id or a corrupt zone.</td>
</tr>
<tr>
<td>LIB$_BADBLOSIZ</td>
<td>Bad block size. The value of number-of-bytes was less than or equal to 0.</td>
</tr>
<tr>
<td></td>
<td>For the fixed-size blocks algorithm, LIB$_BADBLOSIZ can also be generated</td>
</tr>
<tr>
<td></td>
<td>if the value of algorithm-argument specified in the call to LIB$CREATE_VM_ZONE is less than number-of-bytes.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial allocation is made in this case.</td>
</tr>
<tr>
<td>LIB$_PAGLIMEXC</td>
<td>Allocation exceeds the page-limit, set when the zone was create.</td>
</tr>
</tbody>
</table>
LIB$GET_VM_64 (Alpha and I64 Only)

Allocate Virtual Memory

The Allocate Virtual Memory routine allocates a specified number of contiguous bytes in the program region and returns the 64-bit virtual address of the first byte allocated.

Format

LIB$GET_VM_64 number-of-bytes, base-address [,zone-id]

Returns

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

Arguments

number-of-bytes
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Number of contiguous bytes that LIB$GET_VM_64 allocates. The number-of-bytes argument is the address of a quadword integer containing the number of bytes. LIB$GET_VM_64 allocates enough memory to satisfy the request. Your program should not reference an address before the first byte address allocated (base-address) or beyond the last byte allocated (base-address + number-of-bytes minus 1) since that space may be assigned to another routine. The value of number-of-bytes must be greater than zero.

base-address
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference

First virtual address of the contiguous block of bytes allocated by LIB$GET_VM_64. The base-address argument is the address of an unsigned quadword containing this base address.

zone-id
OpenVMS usage: identifier
type: quadword (unsigned)
access: read only
mechanism: by reference

The zone-id argument is the address of a quadword that contains a zone identifier created by a previous call to LIB$CREATE_VM_ZONE_64 or LIB$CREATE_USER_VM_ZONE_64. This argument is optional. If zone-id is omitted or if the quadword contains the value 0, the 64-bit default zone is used.
LIB$ Routines
LIB$GET_VM_64 (Alpha and I64 Only)

Description
LIB$GET_VM_64 satisfies an allocation request by reusing free memory in the zone, or by obtaining additional memory from the processwide 64-bit page pool managed by LIB$GET_VM_PAGE_64.

LIB$GET_VM_64 rounds up the value of number-of-bytes to a multiple of the block-size specified for the zone. The first byte allocated is aligned on the boundary specified by the alignment value for the zone.

If you specified allocation filling when you created the zone, LIB$GET_VM_64 will fill each byte allocated. Otherwise, LIB$GET_VM_64 does not initialize the memory and its contents are unpredictable.

All memory allocated by LIB$GET_VM_64 has user mode read/write access, even if the call to LIB$GET_VM_64 has user mode read/write access, even if the call to LIB$GET_VM_64 was made from a more privileged access mode.

The space allocated by successive calls to LIB$GET_VM_64 may be noncontiguous because another routine can call LIB$GET_VM_64 between your calls. If AST interrupts occur, LIB$GET_VM_64 may allocate space to another routine between execution of any two statements in your program. Even if successive calls to LIB$GET_VM_64 return two contiguous blocks, you must not combine them into one large block that is freed by a single call to LIB$FREE_VM_64.

LIB$GET_VM_64 is fully reentrant, so it may be called by routines executing at AST level or in an Ada multitasking environment.

Your program must retain the address of the allocated area. This allows you to access or deallocate the space later.

If the zone you are getting was created using the LIB$CREATE_USER_VM_ZONE_64 routine, then you must have an appropriate action routine for the get operation. That is, in your call to LIB$CREATE_USER_VM_ZONE_64, you must have specified a user-get-routine.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>Invalid zone-id or a corrupt zone.</td>
</tr>
<tr>
<td>LIB$_BADBLOSIZ</td>
<td>Bad block size. The value of number-of-bytes was less than or equal to 0.</td>
</tr>
<tr>
<td></td>
<td>For the fixed-size blocks algorithm, LIB$_BADBLOSIZ can also be generated if</td>
</tr>
<tr>
<td></td>
<td>the value of algorithm-argument specified in the call to LIB$CREATE_VM_ZONE_</td>
</tr>
<tr>
<td></td>
<td>64 is less than number-of-bytes.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. The request required more dynamic memory than</td>
</tr>
<tr>
<td></td>
<td>was available from the operating system. No partial allocation is made in</td>
</tr>
<tr>
<td></td>
<td>this case.</td>
</tr>
<tr>
<td>LIB$_PAGLIMEXC</td>
<td>Allocation exceeds the page-limit, set when the zone was create.</td>
</tr>
</tbody>
</table>
LIB$GET_VM_PAGE
Get Virtual Memory Page

The Get Virtual Memory Page routine allocates a specified number of contiguous pages on VAX systems or pagelets on Alpha and I64 systems of memory in the program region and returns the virtual address of the first allocated page on VAX or pagelet on Alpha or I64.

Format

LIB$GET_VM_PAGE number-of-pages ,base-address

Returns

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

Arguments

number-of-pages
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Number of pages on VAX systems or pagelets on Alpha and I64 systems. The number-of-pages argument is the address of a longword integer that specifies the number of contiguous pages on VAX systems or pagelets on Alpha and I64 systems to be allocated. The value of number-of-pages must be greater than 0.

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

Block address. The base-address argument is the address of a longword that is set to the address of the first byte of the newly allocated block of pages on VAX systems or pagelets on Alpha and I64 systems.

Description

LIB$GET_VM_PAGE allocates blocks of contiguous (512 byte) pages on VAX systems and pagelets on Alpha and I64 systems in the program region. LIB$GET_VM_PAGE manages a processwide pool of free pages. If there are not enough contiguous free pages or pagelets to satisfy an allocation request, additional pages are created by calling the system service $EXPREG. All memory allocated by LIB$GET_VM_PAGE is pagelet aligned; that is, the low-order nine bits of the base address are zero.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
All memory allocated by LIB$GET_VM_PAGE has user-mode read/write access, even if the call to LIB$GET_VM_PAGE is made from a more privileged access mode.

The contents of memory allocated by LIB$GET_VM_PAGE are unpredictable. Your program must assign values to all locations that it uses.

LIB$GET_VM_PAGE is designed for request sizes ranging from one page or pagelet to a few hundred pages or pagelets. For very large request sizes (over 1000 pages or pagelets in a single request), you should call the system service $EXPREG.

LIB$GET_VM_PAGE is fully reentrant, so it can be called by routines executing at AST level or in an Ada multitasking environment.

### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADDLOSIZ</td>
<td>The value of the <strong>number-of-pages</strong> argument is less than or equal to 0.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial allocation is made in this case.</td>
</tr>
</tbody>
</table>
LIB$GET_VM_PAGE_64 (Alpha and I64 Only)

Get Virtual Memory Page

The Get Virtual Memory Page routine allocates a specified number of contiguous Alpha or I64 pagelets of memory in the program region and returns the virtual address of the first allocated pagelet.

Format

LIB$GET_VM_PAGE_64 number-of-pages, base-address

Returns

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

Arguments

number-of-pages
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Number of Alpha or I64 pagelets. The number-of-pages argument is the address of a quadword integer that specifies the number of contiguous Alpha or I64 pagelets to be allocated. The value of number-of-pages must be greater than 0.

base-address
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference

Block address. The base-address argument is the address of a quadword that is set to the address of the first byte of the newly allocated block of Alpha or I64 pagelets.

Description

LIB$GET_VM_PAGE_64 allocates blocks of contiguous Alpha or I64 pagelets in the program region. LIB$GET_VM_PAGE_64 manages a processwide pool of free pagelets. If there are not enough contiguous free pagelets to satisfy an allocation request, additional pagelets are created by calling the system service $EXPREG_64. All memory allocated by LIB$GET_VM_PAGE_64 is aligned to physical page size.

All memory allocated by LIB$GET_VM_PAGE_64 has user-mode read/write access, even if the call to LIB$GET_VM_PAGE_64 is made from a more privileged access mode.

The contents of memory allocated by LIB$GET_VM_PAGE_64 are unpredictable. Your program must assign values to all locations that it uses.
LIB$ Routines
LIB$GET_VM_PAGE_64 (Alpha and I64 Only)

LIB$GET_VM_PAGE_64 is fully reentrant, so it can be called by routines executing at AST level or in an Ada multitasking environment.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_BADBLOSIZ The value of the argument number-of-pages is less than or equal to 0.
LIB$_INSVIRMEM Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial allocation is made in this case.
LIB$Routines

LIB$I64_CREATE_INVO_CONTEXT (I64 Only)

Create Invocation Context

The Create Invocation Context routine allocates an invocation context block from heap storage and initializes it.

Format

LIB$I64_CREATE_INVO_CONTEXT [malloc] [,free] [,ident]

Returns

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

Arguments

malloc
OpenVMS usage: function_value
type: procedure
access: read
mechanism: by value

A procedure reference for a user callback routine that allocates memory. This is an optional argument. The default is to use an implementation of the C RTL routine malloc. If specified, this routine is used to allocate the invocation context block field LIBICB$PH_UO_MALLOC for use during the stack walk.

free
OpenVMS usage: function_value
type: procedure
access: read
mechanism: by value

A procedure reference for a user callback routine that deallocates memory. This value is placed in the invocation context block field LIBICB$PH_UO_FREE. This is an optional argument; however, it must be specified if malloc is specified. The default is to use an implementation of the C RTL routine free.

ident
OpenVMS usage: user_value
type: quadword
access: read
mechanism: by value

Specifies a user ident value to be placed in the invocation context block LIBICB$IH_UO_IDENT field. In turn, this value is passed to the malloc and free routines. This is an optional argument; the default value is zero.
LIB$ Routines
LIB$I64_CREATE_INVO_CONTEXT (I64 Only)

Description
LIB$I64_CREATE_INVO_CONTEXT simplifies creating and properly initializing an invocation context block. The routine allocates an invocation context block from heap storage and initializes it. Users of this routine should call LIB$I64_FREE_INVO_CONTEXT when the invocation context block is no longer required.

This routine sets the cache unwind flag LIBICB$V_UO_FLAG_CACHE_UNWIND in the invocation context block to speed up the stack walk. Do not use this routine in conjunction with LIB$I64_INIT_INVO_CONTEXT, as the same initialization is performed by both routines.

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>any non-zero value</td>
<td>Represents the address of the allocated invocation context block.</td>
</tr>
</tbody>
</table>
LIB$164_FREE_INVO_CONTEXT (I64 Only)
Deallocate Invocation Context Block

The Free Invocation Context Block routine deallocates an invocation context block that was previously allocated.

Format

LIB$164_FREE_INVO_CONTEXT invo_context

Returns

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

Argument

invo_context  
OpenVMS usage: invo_context_blk  
type: structure  
access: modify only  
mechanism: by reference

Address of an invocation context block.

Description

LIB$164_FREE_INVO_CONTEXT deallocates an invocation context block that was previously allocated using LIB$164_CREATE_INVO_CONTEXT. This routine calls LIB$164_PREV_INVO_END as a convenience.

Condition Values Returned

None.
LIB$ Routines

LIB$I64_GET_CURR_INVO_CONTEXT (I64 Only)

LIB$I64_GET_CURR_INVO_CONTEXT (I64 Only)
Get Current Invocation Context

The Get Current Invocation Context routine gets the invocation context of a current procedure.

Format

LIB$I64_GET_CURR_INVO_CONTEXT invo_context

Returns

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

Argument

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: modify only
mechanism: by reference

Address of an invocation context block into which the procedure context of the caller will be written.

Description

LIB$I64_GET_CURR_INVO_CONTEXT gets the invocation context of a current procedure. The invocation context block must be properly initialized as described in the HP OpenVMS Calling Standard manual before calling this routine.

Condition Value Returned

0 Facilitates use in the implementation of the C language unwind setjmp or longjmp function. Check the LIBICB$L_ALERT_CODE field of the invocation context block for further status indication.
LIB$64_GET_CURR_INVO_HANDLE (I64 Only)
Get Current Invocation Handle

The Get Current Invocation Handle routine gets the invocation handle for the current procedure.

Format

LIB$64_GET_CURR_INVO_HANDLE invo_handle

Returns

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

Argument

invo_handle
OpenVMS usage: invo_handle
type: quadword
access: write only
mechanism: by reference

Address of a quadword into which the invocation handle of the caller will be written.

Description

LIB$64_GET_CURR_INVO_HANDLE gets the invocation handle for the current procedure.

Condition Values Returned

0 The initial context represents the bottom of the call stack.
1 Indicates success.
3 The current operation completed without error, but a stack corruption was detected at the next level down.
LIB$ Routines
LIB$I64_GET_FR (I64 Only)

LIB$I64_GET_FR (I64 Only)
Get Floating-Point Register

The Get Floating-Point Register routine copies the value of the floating-point register.

Format

LIB$I64_GET_FR invo_context, index, fr_copy

Returns

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

Argument

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: read
mechanism: by reference
Address of a valid invocation context block.

index
OpenVMS usage: index
type: longword
access: read
mechanism: by value
Floating point register index.

fr_copy
OpenVMS usage: floating-point value
type: octaword
access: write
mechanism: by value
Address of an octaword to receive the contents of the specified floating-point register.

Description

Given an invocation context block and floating-point register index such that 0 <= index < 128, LIB$I64_GET_FR copies the register value to fr_copy. For example, an index value of 4 fetches the value, which represents the contents of F4 for the context.

LIB$I64_GET_FR returns failure status if the index represents a scratch register whose contents have not been realized.
Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_GET_GR (I64 Only)

LIB$I64_GET_GR (I64 Only)
Get Invocation Context Block Value


Format

LIB$I64_GET_GR invo_context, index, gr_copy

Returns

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

Argument

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: read
mechanism: by reference

Address of a valid invocation context block.

index
OpenVMS usage: index
type: longword
access: read
mechanism: by value

Index into the IREG array of the invocation context block.

gr_copy
OpenVMS usage: floating-point value
type: octaword
access: write
mechanism: by value

Address of an octaword to receive the value from the invocation context block.

Description

Given an invocation context block and general register index such that 0 <= index < 128, LIB$I64_GET_GR copies the register value to gr_copy, for example, index 4 fetches the invocation context block IREG[4] value, which represents the contents of R4 for the context.

If the register represented by index has its corresponding NaT bit set, the read succeeds and the return status is set to 3. If the register represented by index lies beyond the allocated general registers, the read fails and gr_copy is unchanged. That is, the highest allowed index is 32 + ICB.CFM.SOF - 1.

LIB$I64_GET_GR fails if the index represents a scratch register whose contents have not been realized.
LIB$ Routines
LIB$I64_GET_GR (I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success, and that the NaT bit was clear.</td>
</tr>
<tr>
<td>3</td>
<td>Indicates success, and that the NaT bit was set.</td>
</tr>
</tbody>
</table>
The Get Invocation Context routine gets the invocation context of any active procedure.

**Format**

```
LIB$I64_GET_INVO_CONTEXT invo_handle, invo_context
```

**Returns**

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

**Arguments**

invo_handle  
OpenVMS usage: invo_handle  
type: quadword  
access: modify only  
mechanism: by reference

Address of an invocation context block into which the procedure context of the frame specified by invo_handle will be written.

invo_context  
OpenVMS usage: invo_context_blk  
type: structure  
access: write only  
mechanism: by reference

Address of an invocation context block into which the procedure context of the frame specified by invo_handle will be written.

**Description**

LIB$I64_GET_INVO_CONTEXT gets the invocation context of any active procedure.

--- Note ---

The invocation context block must be properly initialized as described in the *HP OpenVMS Calling Standard* manual before calling this routine.
LIB$ Routines
LIB$I64_GET_INVO_CONTEXT (I64 Only)

Condition Value Returned

0 Facilitates use in the implementation of the C language unwind setjmp or longjmp function. Check the LIBICB$L_ALERT_CODE field of the invocation context block for further status indication.
LIB$ Routines
LIB$I64_GET_INVO_HANDLE (I64 Only)

LIB$I64_GET_INVO_HANDLE (I64 Only)
Get Invocation Handle

The Get Invocation Handle routine obtains the invocation handle corresponding to any invocation context block.

Format

LIB$I64_GET_INVO_HANDLE invo_context, invo_handle

Returns

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

Arguments

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: read only
mechanism: by reference
Address of a valid invocation context block.

invo_handle
OpenVMS usage: invo_handle
type: quadword (unsigned)
access: write only
mechanism: by reference
Address of the location into which the invocation context handle is to be written.
If the call fails, the value of the invocation context handle is LIB$K_INVO_HANDLE_NULL.

Description

LIB$GET_INVO_HANDLE gets the invocation context of any active procedure.

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$164_GET_PREV_INVO_CONTEXT (I64 Only)
Get Previous Invocation Context

The Get Current Invocation Context routine obtains the invocation context of the procedure context preceding any other procedure context.

Format

LIB$164_GET_PREV_INVO_CONTEXT invo_context

Returns

OpenVMS usage: cond_value
Type: longword (unsigned)
Access: write only
Mechanism: by value

Argument

invo_context
OpenVMS usage: invo_context_blk
Type: structure
Access: modify only
Mechanism: by reference

Address of a valid invocation context block. The given invocation context block is updated to represent the context of the previous (calling) frame.

The LIBICB$V_BOTTOM_OF_STACK flag of the invocation context block is set if the target frame represents the end of the invocation call chain or if stack corruption is detected.

Description

The LIB$164_GET_PREV_INVO_CONTEXT routine obtains the invocation context of the procedure context preceding any other procedure context.

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The initial context represents the bottom of the call stack.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>3</td>
<td>The current operation completed without error, but a stack corruption was detected at the next level down.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_GET_PREV_INVO_HANDLE (I64 Only)

LIB$I64_GET_PREV_INVO_HANDLE (I64 Only)
Get Previous Invocation Handle

The Get Previous Invocation Handle routine gets an invocation handle of the
procedure context preceding that of a specified procedure context.

Format

LIB$I64_GET_PREV_INVO_HANDLE invo_handle_in, invo_handle_out

Returns

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

Argument

invo_handle_in
OpenVMS usage: invo_handle
type: quadword
access: read only
mechanism: by reference

The address of an invocation handle that represents a target invocation context.

invo_handle_out
OpenVMS usage: invo_handle
type: quadword
access: write only
mechanism: by reference

Address of the location into which the invocation context handle of the previous
context is to be written. If the call fails, the value of the previous invocation
context handle is LIB$K_INVO_HANDLE_NULL.

Description

LIB$I64_GET_PREV_INVO_HANDLE gets an invocation handle of the procedure
context preceding that of a specified procedure context.

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$I64_GET_UNWIND_HANDLER_FV (I64 Only)

Get Function Value For Condition Handler

The Get Function Value For Condition Handler routine finds the function value (address of the procedure descriptor) for the condition handler.

Format

LIB$I64_GET_UNWIND_HANDLER_FV pc_value, handler_fv

Returns

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

Arguments

pc_value
OpenVMS usage: PC value
type: quadword
access: read
mechanism: by reference

Address of a location that contains the PC value.

pc_value is used to find the unwind information block and the unwind information block condition handler pointer.

handler_fv
OpenVMS usage: address
type: quadword
access: write
mechanism: by reference

A quadword to receive the function value of the procedure descriptor for the condition handler, if there is one.

Description

Given a pc_value, LIB$I64_GET_UNWIND_HANDLER_FV finds the function value (address of the procedure descriptor) for the condition handler, if present, and writes it to handler_fv. If not present, then it writes 0 to handler_fv.

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$I64_GET_UNWIND_LSDA (I64 Only)

Find Address of Unwind Information Block Language-Specific Data

The Find Address of Unwind Information Block Language-Specific Data routine finds the address of the unwind information block language-specific data area.

Format

```
LIB$I64_GET_UNWIND_LSDA pc_value, unwind_lsda_p
```

Returns

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

Arguments

**pc_value**
OpenVMS usage: PC value
type: quadword
access: read
mechanism: by reference

Address of a quadword to receive the address of the language-specific data area, if there is one.

**unwind_lsda_p**
OpenVMS usage: address
type: quadword
access: write
mechanism: by reference

Address of a location that contains the PC value. **pc_value** is used to find the unwind information block and the unwind information block language-specific data area address.

Description

Given a **pc_value**, LIB$I64_GET_UNWIND_LSDA finds the address of the unwind information block language-specific data area (LSDA), and writes it to **unwind_lsda_p**. If not present, it then writes 0 to **unwind_lsda_p**.

Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$164_GET_UNWIND_OSSD (I64 Only)
Find Address of the Unwind Information Block Operating System-Specific Data Area

The Find Address of the Unwind Information Block Operating System-Specific Data Area routine finds the address of the unwind information block operating system-specific data area.

Format

LIB$164_GET_UNWIND_OSSD pc_value, unwind_ossd_p

Returns

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

Argument

pc_value
OpenVMS usage: PC value
type: quadword
access: read
mechanism: by reference

Address of a location that contains the PC value. pc_value is used to find the unwind information block and the unwind information block operating system-specific data area address.

unwind_ossd_p
OpenVMS usage: address
type: quadword
access: write
mechanism: by reference

Address of a quadword to receive the address of the operating system-specific data area.

Description

Given a pc_value, LIB$164_GET_UNWIND_OSSD finds the address of the unwind information block operating system-specific data area, if present, and writes it to unwind_ossd_p. If not present, then it writes 0 to unwind_ossd_p.

Condition Values Returned

0 Indicates failure.
1 Indicates success.
LIB$ Routines
LIB$I64_INIT_INVO_CONTEXT (I64 Only)

LIB$I64_INIT_INVO_CONTEXT (I64 Only)
Initialize an Invocation Context Block

The Initialize Invocation Context routine initializes an invocation context block that has already been allocated by the user.

Format

LIB$I64_INIT_INVO_CONTEXT invo_context, invo_version [,cache_unwind_flag]

Returns

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

Arguments

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: modify only
mechanism: by reference
Address of an invocation context block.

invo_version
OpenVMS usage: version_number
type: byte
access: read only
mechanism: by value

The value LIBICB$K_INVO_CONTEXT_VERSION. This is used to verify the operating environment.

cache_unwind_flag
OpenVMS usage: flag
type: longword
access: read only
mechanism: by value

A flag indicating if the cache unwind flag, LIBICB$V_UO_FLAG_CACHE_UNWIND, should be set in the invocation context block. A value of zero clears the flag; a value of one sets the flag. This is an optional argument. The default is zero.

Description

LIB$I64_INIT_INVO_CONTEXT initializes an invocation context block that the user has already allocated (on the stack, or from heap, or other storage). Use this routine as an alternative to LIB$I64_CREATE_INVO_CONTEXT, which both allocates and initializes an invocation context block.
Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates a version number mismatch.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_IS_AST_DISPATCH_FRAME (I64 Only)

LIB$I64_IS_AST_DISPATCH_FRAME (I64 Only)
Determine AST Exception Frame Dispatch

The Determine AST Exception Frame Dispatch routine determines whether a given PC value represents an AST dispatch frame.

Format

LIB$I64_IS_AST_DISPATCH_FRAME pc_value

Returns

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

Argument

pc_value
OpenVMS usage: PC value
type: quadword
access: read
mechanism: by reference

Address of a quadword that contains the PC value.
The pc_value is used to find the operating system-specific data area in the unwind information for this routine.

Description

LIB$I64_IS_AST_DISPATCH_FRAME determines whether a given PC value represents an AST dispatch frame.

Condition Values Returned

0
The operating system-specific data area is present and the EXCEPTION_FRAME flag is clear. Returns 0 if the operating system-specific data area is not present.

1
The operating system-specific data area is present and the EXCEPTION_FRAME flag is set.
LIB$I64_IS_EXC_DISPATCH_FRAME (I64 Only)
Determine Exception Frame Dispatch

The Determine Exception Frame Dispatch routine determines whether a given PC value represents an exception dispatch frame.

Format

LIB$I64_IS_EXC_DISPATCH_FRAME pc_value

Returns

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

Argument

**pc_value**
OpenVMS usage: PC value
type: quadword
access: read
mechanism: by reference

Address of a quadword that contains the PC value.

The pc_value is used to find the operating system-specific data area in the unwind information for this routine.

Description

LIB$I64_IS_EXC_DISPATCH_FRAME determines whether a given PC value represents an exception dispatch frame.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The operating system-specific data area is present and the EXCEPTION_FRAME flag is clear. Returns 0 if the operating system-specific data area is not present.</td>
</tr>
<tr>
<td>1</td>
<td>The operating system-specific data area is present and the EXCEPTION_FRAME flag is set.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_PREV_INVO_END (I64 Only)

LIB$I64_PREV_INVO_END (I64 Only)
End Call Tracing Operations

The End Call Tracing Operations routine should be called at the conclusion of call tracing operations to free the memory used to process unwind descriptors.

Format

LIB$I64_PREV_INVO_END (invo_context)

Returns

OpenVMS usage: cond_value

type: longword (unsigned)

access: write only

mechanism: by value

Argument

invo_context
OpenVMS usage: invo_context_blk

type: structure

access: modify only

mechanism: by reference

Address of a valid invocation context block previously used for call tracing.

Description

LIB$I64_PREV_INVO_END should be called at the conclusion of call tracing operations to free the memory used to process unwind descriptors. The call tracing routines are LIB$I64_GET_INVO_CONTEXT, LIB$I64_GET_PREV_INVO_CONTEXT, and LIB$I64_GET_CURR_INVO_CONTEXT.

To provide efficient call tracing, some unwind information is tracked in heap storage from one call to the next. This heap storage should be freed before you release or reuse the invocation context block.

Calling this routine is necessary if the LIBICB$V_UO_FLAG_CACHE_UNWIND flag is set in the LIBICB$Q_UO_FLAGS field of the invocation context block. If this flag is not set, unwind information is released and re-created at each call, and calling this routine is not required.

Condition Values Returned

0 Indicates failure.

1 Indicates success.
LIB$64_PUT_INVO_REGISTERS (I64 Only)

Put Invocation Registers

The Put Invocation Registers routine updates the fields of a given procedure invocation context.

Note that if user override routines are specified in the invocation context block, then they are used to find and modify the invocation context.

Format


Returns

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

Arguments

invo_handle
OpenVMS usage: invo_handle
type: quadword (unsigned)
access: read only
mechanism: by reference

Handle for the invocation to be updated.

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: read only
mechanism: by reference

Address of a valid invocation context block that contains new register contents.

Each register that is set in the *xx_mask* argument (along with its NaT bit, if any) is updated using the value found in the corresponding IREG[n], FREG[n], BRANCH[n], or PRED[n] field. GP, TP, and AI can also be updated in this way.

No other fields of the invocation context block are used.

gr_mask
OpenVMS usage: mask_octaword
type: 128-bit vector
access: read only
mechanism: by reference

Address of a 128-bit bit vector, where each bit corresponds to a register field in the *invo_context* argument. Bits 0 through 127 correspond to IREG[0] through IREG[127].

Bit 0 corresponds to R0, which cannot be written, and is ignored.

Bit 1 corresponds to the global data pointer (GP).
Bit 13 corresponds to the thread pointer (TP).
Bit 25 corresponds to the argument information register (AI).
If bit 12, which corresponds to SP, is set, then no changes are made.

fr_mask
OpenVMS usage: mask_octaword
type: 128-bit vector
access: read only
mechanism: by reference

Address of a 128-bit bit vector, where each bit corresponds to a register field in the passed invo_context.

To update floating-point registers F32-F127, provide a pointer to an array of 96 octawords in LIBICB$PH_F32_F127.

Bits 0 through 127 correspond to FREG[0] through FREG[127].
Bit 0 corresponds to F0, which cannot be written, and is ignored.
Bit 1 corresponds to F1, which cannot be written, and is ignored.

br_mask
OpenVMS usage: mask_byte
type: 8-bit vector
access: read only
mechanism: by reference

Address of a 8-bit bit vector, where each bit corresponds to a register field in the passed invo_context. Bits 0 through 7 correspond to BRANCH[0] through BRANCH[7].

pr_mask
OpenVMS usage: mask_quadword
type: 64-bit vector
access: read only
mechanism: by reference

Address of a 64-bit bit vector, where each bit corresponds to a register field in the passed invo_context. Bits 0 through 63 correspond to PRED[0] through PRED[63].

misc_mask
OpenVMS usage: mask_quadword
type: 64-bit vector
access: read only
mechanism: by reference

Address of a 64-bit bit vector, where each bit corresponds to a register field in the passed invo_context as follows:

- Bit 0=PC.
- Bit 1=FPSR.
- Bits 2–63 are reserved.
Description

LIB$164_PUT_INVO_REGISTERS updates the fields of a given procedure invocation context.

____________________________ Caution ______________________________

Great care must be taken to ensure that a valid stack frame and execution environment result; otherwise, execution may become unpredictable.

________________________________________________________________________

Condition Values Returned

<table>
<thead>
<tr>
<th>0</th>
<th>In the following circumstances:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• When the invocation handle does not represent an active invocation context.</td>
</tr>
<tr>
<td></td>
<td>• When bit 12 of the gr_mask argument is set</td>
</tr>
<tr>
<td></td>
<td>• When a scratch register has not been saved, or a register's save location or status cannot be determined (valid bit clear).</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_SET_FR (I64 Only)

LIB$I64_SET_FR (I64 Only)
Set Floating-Point Register

The Set Floating-Point Register routine writes the invocation context block floating-point registry entry corresponding to a floating-point register value.

Format

LIB$I64_SET_FR invo_context, index, fr_copy

Returns

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

Arguments

invocation_context
OpenVMS usage: invo_context_blk
type: structure
access: modify
mechanism: by reference
Address of a valid invocation context block.

index
OpenVMS usage: index
type: longword
access: read
mechanism: by value
Index into the FREG array of the invocation context block.

fr_copy
OpenVMS usage: floating-point value
type: octaword
access: write
mechanism: by value
Address of an octaword that contains the floating-point value to be written to the invocation context block.

Description

Given an invocation context block, a floating-point register index, and a floating-point register value in fr_copy, writes the corresponding invocation context block FREG entry, and calls LIB$I64_PUT_INVO_REGISTERS to write the actual context. The invocation context block remains unchanged if the routine fails.

LIB$I64_SET_FR fails if LIB$I64_PUT_INVO_REGISTERS fails.
### Condition Values Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_SET_GR (I64 Only)

LIB$I64_SET_GR (I64 Only)
Copy Invocation Block General Register

The Copy Invocation Block General Register routine writes the invocation context block general register.

Format

LIB$I64_SET_GR invo_context, index, fr_copy

Returns

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

Arguments

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: modify
mechanism: by reference
Address of a valid invocation context block.

index
OpenVMS usage: index
type: longword
access: read
mechanism: by value
Index into the IREG array of the invocation context block.

gr_copy
OpenVMS usage: integer value
type: quadword
access: write
mechanism: by value
Address of a quadword that contains the value to be written to the invocation context block.

Description

Given an invocation context block, a general register index such that 1 <= index < 128, and a quadword value gr_copy, LIB$I64_SET_GR writes the corresponding invocation context block general register, clears the corresponding NaT bit and uses LIB$I64_PUT_INVO_REGISTERS to write to the actual context. The invocation context block remains unchanged if the routine fails. LIB$I64_SET_GR fails if LIB$I64_PUT_INVO_REGISTERS fails.
**Condition Values Returned**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$I64_SET_PC (I64 Only)

LIB$I64_SET_PC (I64 Only)
Write Context Block and Quadword PC Value

The Write Context Block and Quadword PC Value routine writes invocation context block PC.

Format

LIB$I64_SET_PC invo_context, pc_copy

Returns

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

Arguments

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: modify
mechanism: by reference

Address of a valid invocation context block.

pc_copy
OpenVMS usage: PC value
type: quadword
access: read
mechanism: by reference

Address of a quadword that contains the PC value to be written to the invocation context block.

Description

Given an invocation context block and a quadword PC value in pc_copy, LIB$I64_SET_PC writes the pc_copy value to the invocation context block PC and then uses LIB$I64_PUT_INVO_REGISTERS to write to the actual context. The invocation context block remains unchanged if the routine fails.

LIB$I64_SET_PC fails if LIB$I64_PUT_INVO_REGISTERS fails.

Condition Values Returned

0 Indicates failure.
1 Indicates success.
LIB$ICHAR
Convert First Character of String to Integer

The Convert First Character of String to Integer routine converts the first character of a source string to an 8-bit ASCII integer extended to a longword.

Format

LIB$ICHAR source-string

Returns

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

First character of the source string. This character is returned by LIB$ICHAR as an 8-bit ASCII value extended to a longword. If the source string has zero length, LIB$ICHAR returns a zero.

Argument

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

Source string whose first character is converted to an integer by LIB$ICHAR. The source-string argument is the address of a descriptor pointing to this source string.

Description

Although Fortran users can call LIB$ICHAR, it is more efficient to use the Fortran intrinsic function ICHAR, which generates equivalent code in line.

Condition Values Returned

None.

Example

PROGRAM ICHAR(INPUT, OUTPUT);
{+}
{ This program demonstrates how to call LIB$ICHAR
{ to convert the first character of string to an
{ integer value.
{ -}
FUNCTION LIB$ICHAR(SRCSTR : VARYING [A] OF CHAR) : INTEGER;
EXTERN;
{+}
{ Declare the variables to be used.
{-}
VAR
   CHARSTR : VARYING [256] OF CHAR;
   RET_STATUS : INTEGER;

{+}
{ Begin the main program. Read the character string,
  call LIB$ICHAR, and print the result.
{-}
BEGIN
   WRITELN('Enter string: ');
   READLN(CHARSTR);
   RET_STATUS := LIB$ICHAR(CHARSTR);
   WRITELN(RET_STATUS);
END.

The output generated by this Pascal program is as follows:

$ RUN ICHAR
Enter string:
Pencil sharpener
   80
$ RUN ICHAR
Enter string:
pencil sharpener
   112

Notice that this routine changes any uppercase characters to lowercase.
The Index to Relative Position of Substring routine returns an index, which is the relative position of the first occurrence of a substring in the source string.

**Format**

LIB$INDEX source-string ,sub-string

**Returns**

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

The relative position of the first character of the substring if found, or zero if not found.

On Alpha and I64 systems, if the relative position of the substring can exceed $2^{32} - 1$, assign the return value to a quadword to ensure that you retrieve the correct relative position.

**Arguments**

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

Source string to be searched by LIB$INDEX. The **source-string** argument is the address of a descriptor pointing to this source string.

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

Substring to be found. The **sub-string** argument is the address of a descriptor pointing to this substring.

**Description**

The relative character positions returned by LIB$INDEX are numbered 1, 2, ..., \( n \). Zero means that the substring was not found.

If the substring has a zero length, LIB$INDEX returns the value 1, indicating success, no matter how long the source string is. If the source string has a zero length and the substring has a nonzero length, zero is returned, indicating that the substring was not found.

Fortran users may use the built-in INDEX function rather than calling LIB$INDEX directly.
LIB$ Routines
LIB$INDEX

Condition Values Returned

None.
LIB$INIT_DATE_TIME_CONTEXT
Initialize the Context Area Used in Formatting Dates and Times for Input or Output

The Initialize the Context Area Used in Formatting Dates and Times for Input or Output routine allows the user to initialize the context area used by LIB$FORMAT_DATE_TIME or LIB$CONVERT_DATE_STRING with specific strings, instead of through logical name translation.

Format

LIB$INIT_DATE_TIME_CONTEXT user-context ,component ,init-string

Returns

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

Arguments

user-context
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference

User context that retains the translation context over multiple calls to this routine. The user-context argument is the address of an unsigned longword that contains this context. The initial value of the context variable must be zero. Thereafter, the user program must not write to the cell.

component
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

The component of the context that is being initialized. The component argument is the address of a signed longword that indicates this component. Only one component can be initialized per call to LIB$INIT_DATE_TIME; these component codes are shown in the following list.

- LIB$K_MONTH_NAME
- LIB$K_MONTH_NAME_ABB
- LIB$K_FORMAT_MNEMONICS
- LIB$K_WEEKDAY_NAME
- LIB$K_WEEKDAY_NAME_ABB
- LIB$K_RELATIVE_DAY_NAME
LIB$ Routines
LIB$INIT_DATE_TIME_CONTEXT

- LIB$K_MERIDIEM_INDICATOR
- LIB$K_OUTPUT_FORMAT
- LIB$K_INPUT_FORMAT
- LIB$K_LANGUAGE

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

The characters that are to be used in formatting dates and times for input or output. The **init-string** argument is the address of a descriptor pointing to this string.

**Description**

The LIB$INIT_DATE_TIME_CONTEXT routine allows the user to initialize the context area used by either LIB$CONVERT_DATE_STRING or LIB$FORMAT_DATE_TIME with specific strings instead of through logical name translations. This routine is therefore useful when the application is formatting either input or output strings that are used only by other computer applications and are not intended for presentation to users.

When the text will be parsed by another program, you must specify all of the context (including spellings). For applications where the context specifies a user’s preferred format style, spellings can be looked up from the logical name tables.

Therefore, when the text will be parsed by another program, the minimum effort required to initialize the necessary format strings would be a call to LIB$INIT_DATE_TIME_CONTEXT specifying the input or output format strings to be used. If the specified format requires spelled items, such as month names or day names, then additional calls to LIB$INIT_DATE_TIME_CONTEXT are required to provide the spellings of these items. Applications where the context specifies a user’s preferred format style can specify only the language name, and allow the strings to be looked up from logical name tables.

The format of the strings used by this routine is as follows:

```
[delim][string-1][delim] [string-2][delim] ... [delim][string-n][delim]
```

In this format, [delim] is any character that is not in any of the strings, and [string-x] is the spelling of that instance of the component.

For example, a string passed to this routine to specify the English spellings of the month names might be as follows:

```
| JAN | FEB | MAR | APR | MAY | JUN |
| JUL | AUG | SEP | OCT | NOV | DEC |
```

Note that the string starts and ends with a delimiter. Thus, there is one more delimiter than there are string elements. Each type of component has a natural number of elements associated. The string must contain exactly that number of elements.
Month names (full or abbreviated) 12
Format mnemonics 9
Day names (full or abbreviated) 7
Relative day names 3
Meridiem indicators 2
Output format strings 2
Input format string 1
Language 1

In order to specify the input format mnemonics using LIB$INIT_DATE_TIME_CONTEXT, the user must initialize the component LIB$K_FORMAT_MNEMONICS with the appropriate values. The following table lists in order the 9 fields that must be initialized, along with their default (English) values.

<table>
<thead>
<tr>
<th>Order</th>
<th>Format Field</th>
<th>Legible Mnemonic (Defaults)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Year</td>
<td>YYYY</td>
</tr>
<tr>
<td>2</td>
<td>Numeric month</td>
<td>MM</td>
</tr>
<tr>
<td>3</td>
<td>Numeric day</td>
<td>DD</td>
</tr>
<tr>
<td>4</td>
<td>Hours (12- or 24-hour)</td>
<td>HH</td>
</tr>
<tr>
<td>5</td>
<td>Minutes</td>
<td>MM</td>
</tr>
<tr>
<td>6</td>
<td>Seconds</td>
<td>SS</td>
</tr>
<tr>
<td>7</td>
<td>Fractional seconds</td>
<td>CC</td>
</tr>
<tr>
<td>8</td>
<td>Meridiem indicator</td>
<td>AM/PM</td>
</tr>
<tr>
<td>9</td>
<td>Alphabetic month</td>
<td>MONTH</td>
</tr>
</tbody>
</table>

For example, the following would be a valid definition of LIB$K_FORMAT_MNEMONICS using Austrian as the natural language:

|JJJJ|MM|TT|SS|MM|SS|HH|MONAT|

To specify an output format using LIB$INIT_DATE_TIME_CONTEXT, the user must initialize the variable LIB$K_OUTPUT_FORMAT. There are two elements associated with this output format string. One describes the date format fields, the other the time format fields. The order in which they appear in the string determines the order in which they are output. A single space is inserted into the output stream between the two elements, if the call to LIB$FORMAT_DATE_TIME specifies that both be output. In the following example, the two elements associated with the output format string are delimited by vertical bars.

| !DB-!MAAU-!Y4 | !H04:!M0:!S0.!C2 | $ASCTIM$ service for outputting times. Note that the middle delimiter is replaced by a space in the resultant output.

See the HP OpenVMS Programming Concepts Manual for a description of system date and time operations as well as a detailed description of the format mnemonics used in these routines.
LIB$ Routines
LIB$INIT_DATE_TIME_CONTEXT

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_ILLCOMPONENT</td>
<td>Illegal value for the component.</td>
</tr>
<tr>
<td>LIB$_ILLINISTR</td>
<td>Illegally formed init-string.</td>
</tr>
<tr>
<td>LIB$_NUMELEMENTS</td>
<td>Incorrect number of elements for the component.</td>
</tr>
<tr>
<td>LIB$_UNRFORCOD</td>
<td>Unrecognized format code.</td>
</tr>
</tbody>
</table>

Any condition value returned by LIB$GET_VM or LIB$ANALYZE_SDESC.
LIB$INIT_TIMER
Initialize Times and Counts

The Initialize Times and Counts routine stores the current values of specified times and counts for use by LIB$SHOW_TIMER or LIB$STAT_TIMER.

Format

LIB$INIT_TIMER [context]

Returns

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

Argument

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

Context variable that retains the values of the times and counts. The context argument contains the address of an unsigned longword that is this context. When you call LIB$INIT_TIMER, you must use the optional context argument only if you want to maintain several sets of statistics simultaneously.

- If context is omitted, the control block is allocated in static storage. This method is not AST reentrant.
- If context is zero, a control block is allocated in dynamic heap storage. The times and counts will be stored in that block and the address of the block returned in context. This method is fully reentrant and modular.
- If context is nonzero, it is considered to be the address of a control block previously allocated by a call to LIB$INIT_TIMER. If so, the control block is reused, and fresh times and counts are stored in it.

When LIB$INIT_TIMER returns, the block of storage referred to by context will contain the times and counts.

Description

LIB$INIT_TIMER stores the current values of specified times and counts in one of three places, depending on the value of the optional context argument.

You need to call LIB$FREE_TIMER only if you have specified context in LIB$INIT_TIMER and you want to deallocate all heap storage resources.
Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>The <em>context</em> argument is zero, and there is insufficient virtual memory to allocate a storage block.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument; <em>context</em> is nonzero and the block to which it refers was not initialized on a previous call to LIB$INIT_TIMER.</td>
</tr>
</tbody>
</table>
Insert Entry in a Balanced Binary Tree

The Insert Entry in a Balanced Binary Tree routine inserts a node in a balanced binary tree. †

Format

```
LIB$INSERT_TREE  treehead,symbol,flags,user-compare-routine,user-allocation-procedure,new-node
[,user-data]
```

Returns

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

Arguments

**treehead**

OpenVMS usage: address
type: address
access: modify
mechanism: by reference

Tree head for the binary tree. The *treehead* argument is the address of a longword that is this tree head. The initial value of *treehead* is 0.

**symbol**

OpenVMS usage: user_arg
type: longword (unsigned)
access: unspecified
mechanism: unspecified

Key to be inserted.

**flags**

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

Control flags. The *flags* argument is the address of the control flags. Currently only bit 0 is used.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Action if Set</th>
<th>Action if Clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Duplicate entries are</td>
<td>The address of the existing duplicate entry is returned to the <em>new-node</em> argument.</td>
</tr>
<tr>
<td></td>
<td>inserted.</td>
<td></td>
</tr>
</tbody>
</table>

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$INSERT_TREE

**user-compare-routine**

OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied compare routine that LIB$INSERT_TREE calls to compare a symbol with a node. The **user-compare-routine** argument is required; LIB$INSERT_TREE calls the compare routine for every node except the first node in the tree. The value returned by the compare routine indicates the relationship between the symbol key and the node.

For more information on the compare routine, see Call Format for a Compare Routine in the Description section.

**user-allocation-procedure**

OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied allocate routine that LIB$INSERT_TREE calls to allocate virtual memory for a node. The **user-allocation-procedure** argument is required; LIB$INSERT_TREE always calls the allocate routine.

For more information on the allocate routine, see Call Format for an Allocate Routine in the Description section.

**new-node**

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

Location where the new key is inserted. The **new-node** argument is the address of an unsigned longword that is the address of the new node.

**user-data**

OpenVMS usage: user_arg
type: unspecified
access: unspecified
mechanism: by value

User data that LIB$INSERT_TREE passes to the compare and allocate routines. The **user-data** argument is optional.

**Description**

This Description section contains three parts:

- Guidelines for Using LIB$INSERT_TREE
- Call Format for a Compare Routine
- Call Format for an Allocate Routine
Guidelines for Using LIB$INSERT_TREE
LIB$INSERT_TREE inserts a node in a balanced binary tree. You supply two routines: compare and allocate. The compare routine compares the symbol key to the node, and the allocate routine allocates virtual memory for the node to be inserted. LIB$INSERT_TREE first calls the compare routine to find the location at which the new node is inserted. Then LIB$INSERT_TREE calls the allocate routine to allocate memory for the new node. Most programmers insert data in the new node by initializing it within the allocate routine as soon as memory has been allocated.

You may pass the data to be inserted into the tree using either the symbol argument alone or both the symbol and user-data arguments. The symbol argument is required. It may contain all of the data, just the name of the node, or the address of the data. If you decide to use symbol to pass just the name of the node, you must use the user-data argument to pass the rest of the data to be inserted in the new node.

Call Format for a Compare Routine
The call format of a compare routine is as follows:

```c
user-compare-routine symbol ,comparison-node [user-data]
```

LIB$INSERT_TREE passes both the symbol and comparison-node arguments to the compare routine, using the same passing mechanism that was used to pass them to LIB$INSERT_TREE. The user-data argument is passed in the same way, but its use is optional.

The user-compare-routine argument in the call to LIB$INSERT_TREE specifies the compare routine. This argument is required. LIB$INSERT_TREE calls the compare routine for every node except the first node in the tree.

The value returned by the compare routine is the result of comparing the symbol key with the current node. The following table interprets the possible values returned by the compare routine:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>The symbol argument is less than the current node.</td>
</tr>
<tr>
<td>Zero</td>
<td>The symbol argument is equal to the current node.</td>
</tr>
<tr>
<td>Positive</td>
<td>The symbol argument is greater than the current node.</td>
</tr>
</tbody>
</table>

This is an example of a user-supplied compare routine, written in C.

```c
struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    char Text[80];
};

static long Compare_node(char* Key_string,
                          struct Full_node* Node,
                          void* Dummy)
```
/* 
** This function compares the string described by Key_string with 
** the string contained in the data node Node, and returns 0 
** if the strings are equal, -1 if Key_string is < Node, and 
** 1 if Key_string > Node. 
*/
{
    int result;
    result = strcmp(Key_string, Node->Text);
    if (result < 0)
        return -1;
    else if (result == 0)
        return 0;
    else
        return 1;
}

Call Format for an Allocate Routine
LIB$INSERT_TREE calls the allocate routine to allocate virtual memory for a 
node. The allocate routine then stores the value of user-data in the field of the 
allocated node.

The format of the call is as follows:
user-allocation-procedure symbol ,new-node [,user-data]

LIB$INSERT_TREE passes the symbol, new-node, and user-data arguments 
to your allocate routine, using the same passing mechanisms that were used to 
pass them to LIB$INSERT_TREE. Use of user data is optional.

A node header appears at the beginning of each node. The following figure shows 
the structure of a node header.

<table>
<thead>
<tr>
<th>Left Link</th>
<th>Right Link</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 Bytes)</td>
<td>(4 Bytes)</td>
<td>(2 Bytes)</td>
</tr>
</tbody>
</table>

Therefore, any node you declare that LIB$INSERT_TREE manipulates must 
contain 10 bytes of reserved data at the beginning for the node header.

How a node is structured depends on how you allocate your user data. You can 
allocate data in one of two ways:

1. Your data immediately follows the node header. In this case, your allocation 
routine must allocate a block equal in size to the sum of your data plus 10 
bytes for the node header, as shown in the following figure.
2. The node contains the 10 bytes of header information and a longword pointer to the user data, as shown in the following figure.

The **user-allocation-procedure** argument in the call to LIB$INSERT_TREE specifies the allocate routine. This argument is required. LIB$INSERT_TREE always calls the allocate routine.

Following is an example of a user-supplied allocate routine written in C.

```c
struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    char Text[80];
};

static long Alloc_node(char* Key_string,
                     struct Full_node** Ret_addr, void* Dummy)
{
    /*
    ** Allocate virtual memory for a new node. Key_string is the string to be entered into the newly allocated node. RET_ADDR will contain the address of the allocated memory.
    */
    long Status_code;
    long Alloc_size = sizeof(struct Full_node);
    extern long lib$get_vm();

    /* Allocate node: size of header, plus the length of our data. */
    Status_code = lib$get_vm (&Alloc_size, Ret_addr);
    if (!(Status_code & 1))
        lib$stop(Status_code);
```
LIB$ Routines
LIB$INSERT_TREE

```c
/*
 ** Store the data in the newly allocated virtual memory.
 */
strcpy((*Ret_addr)->Text, Key_string);
return (Status_code);
}
```

Condition Values Returned

- **LIB$_NORMAL** Routine successfully completed.
- **LIB$_INSVIRMEM** Insufficient virtual memory. The user-supplied allocation routine returned an error.
- **LIB$_KEYALRINS** Routine successfully completed, but a key was found in the tree. No new key was inserted.

Any other failure status reported by the user allocation procedure.

Example

The following C program shows the use of LIB$INSERT_TREE, LIB$LOOKUP_TREE, and LIB$TRAVERSE_TREE.

```c
/*
 ** This program asks the user to enter a series of strings, one per line. The user can then query the program to find strings that were previously entered. At the end, the entire tree is displayed, along with sequence numbers that indicate the order in which each element was entered.
 ** This program serves as an example of the use of LIB$INSERT_TREE, LIB$LOOKUP_TREE and LIB$TRAVERSE_TREE.
 */
#include <stdio.h>
#include <string.h>
#include <libdef.h>

char Text_line[80];

struct Tree_element /* Define the structure of our record. This record could contain many useful data items. */
{
    long Seq_num; /* record. This record could */
    char Text[80]; /* contain many useful data items. */
};

struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    struct Tree_element my_node;
};

struct Tree_element Rec; /* Declare an instance of the record */
extern long lib$insert_tree(); /* Function to insert node */
extern long lib$traverse_tree(); /* Function to walk tree */
extern long lib$lookup_tree(); /* Function to find a node */
extern void lib$stop(); /* Routine to signal fatal error */
```
static long Compare_node_2(); /* Compare entry (2 arg) */
static long Compare_node_3(); /* Compare entry (3 arg) */
static long Alloc_node(); /* Allocation entry */
static long Print_node(); /* Print entry for walk */
static void Display_Node();

main ()
{
    struct Full_node* Tree_head; /* Head for the tree */
    struct Full_node* New_node; /* New node after insert */
    long Status_code; /* Return status code */
    long Counter; /* Sequence number */
    long flags = 1;
    
    /* ** Initialize the tree to null */
    Tree_head = NULL;
    printf("Enter one word per line, ^Z to begin searching the tree\n");
    /* ** Loop, reading lines of text until the end of the file. */
    Counter = 0;
    printf("> ");
    while (gets(Text_line))
    {
        Counter++;
        Rec.Seq_num = Counter;
        strcpy(Rec.Text, Text_line);
        Status_code = lib$insert_tree ( /* Insert the entry into the tree */
            &Tree_head, &Rec, &flags,
            Compare_node_3, Alloc_node, &New_node);
        if (!(Status_code & 1))
            lib$stop(Status_code);
        printf("> ");
    }
    
    /* ** End of file encountered. Begin searching the tree. */
    printf("\nYou will now be prompted for words to find. ");
    printf("Enter one per line.\n");
    Rec.Seq_num = -1;
    printf("Word to find? ");
    while (gets(Text_line))
    {
        strcpy(Rec.Text, Text_line);
        Status_code = lib$lookup_tree (&Tree_head, &Rec,
            Compare_node_2, &New_node);
        if (Status_code == LIB$KEYNOTFOU)
            printf("The word you entered does not appear in the tree.\n");
        else
            Display_Node(New_node);
        printf("Word to find? ");
    }
    
    /* ** The user has finished searching the tree for specific items. It */
    /* is now time to traverse the entire tree. */
    printf("\n");
    printf("The following is a dump of the tree. Notice that the words\n");
    printf("are in alphabetical order\n");
}
LIB$ Routines
LIB$INSERT_TREE

Status_code = lib$straverse_tree(&Tree_head, Print_node, 0);
return(Status_code);
}

static long Print_node(struct Full_node* Node, void* Dummy)
{
    /*
     * Print the string contained in the current node.
     */
    printf("%d	%s\n", Node->my_node.Seq_num, Node->my_node.Text);
    return(LIB$_NORMAL);
}

static long Alloc_node(struct Tree_element* Rec, struct Full_node** Ret_addr, void* Dummy)
{
    /*
     * Allocate virtual memory for a new node. Rec is the
     * data record to be entered into the newly
     * allocated node. RET_ADDR will contain the address
     * of the allocated memory.
     */
    long Status_code;
    long Alloc_size = sizeof(struct Full_node);
    extern long lib$get_vm();
    /*
     * Allocate node: size of header, plus the length of our data.
     */
    Status_code = lib$get_vm (&Alloc_size, Ret_addr);
    if (!(Status_code & 1))
        lib$stop(Status_code);
    /*
     * Store the data in the newly allocated virtual memory.
     */
    (*Ret_addr)->my_node.Seq_num = Rec->Seq_num;
    strcpy((*Ret_addr)->my_node.Text, Rec->Text);
    return (Status_code);
}

static long Compare_node_3(struct Tree_element* Rec, struct Full_node* Node, void* Dummy)
{
    /*
     * Call the 2 argument version of the compare routine
     */
    return(Compare_node_2 ( Rec, Node ));
}

static long Compare_node_2(struct Tree_element* Rec, struct Full_node* Node)
{
    /*
     * This function compares the string described by Key_string with
     * the string contained in the data node Node, and returns 0
     * if the strings are equal, -1 if Key_string is < Node, and
     * 1 if Key_string > Node.
     */
    int result;
LIB$ Routines
LIB$INSERT_TREE

.parseFloat the result of the comparison.
*/
result = strcmp(Rec->Text, Node->my_node.Text);
if (result < 0)
    return -1;
else if (result == 0)
    return 0;
else
    return 1;
}

static void Display_Node(struct Full_node* Node)
{
    /*
    ** This routine prints the data into the node of the tree
    ** once LIB$LOOKUP_TREE has been called to find the node.
    */
    printf("The sequence number for \"%s\" is %d\n",
            Node->my_node.Text, Node->my_node.Seq_num);
}

The output generated by this program is as follows:

$ run tree
Enter one word per line, ^Z to begin searching the tree
> apple
> orange
> peach
> pear
> grapefruit
> lemon
> <CRZ>

You will now be prompted for words to find. Enter one per line.

Word to find? lime
The word you entered does not appear in the tree

Word to find? orange
The sequence number for "orange" is 2

Word to find? <CRZ>

The following is a dump of the tree. Notice that the words
are in alphabetical order
1 apple
5 grapefruit
6 lemon
2 orange
3 peach
4 pear
$
LIB$ Routines
LIB$INSERT_TREE_64 (Alpha and I64 Only)

LIB$INSERT_TREE_64 (Alpha and I64 Only)
Insert Entry in a Balanced Binary Tree

The Insert Entry in a Balanced Binary Tree routine inserts a node in a balanced binary tree.

Format

LIB$INSERT_TREE_64 treehead ,symbol ,flags ,user-compare-routine ,user-allocation-procedure ,new-node [,user-data]

Returns

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

Arguments

treehead
OpenVMS usage: address
type: address
access: modify
mechanism: by reference

Tree head for the binary tree. The treehead argument is the address of a quadword that is this tree head. The initial value of treehead is 0.

symbol
OpenVMS usage: user_arg
type: quadword (unsigned)
access: unspecified
mechanism: unspecified

Key to be inserted.

flags
OpenVMS usage: mask_quadword
type: quadword (unsigned)
access: read only
mechanism: by reference

Control flags. The flags argument is the address of the control flags. Currently only bit 0 is used.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>If clear, the address of the existing duplicate entry is returned to the new-node argument. If set, duplicate entries are inserted.</td>
</tr>
</tbody>
</table>
**user-compare-routine**
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied compare routine that LIB$INSERT_TREE_64 calls to compare a symbol with a node. The **user-compare-routine** argument is required; LIB$INSERT_TREE_64 calls the compare routine for every node except the first node in the tree. The value returned by the compare routine indicates the relationship between the symbol key and the node.

For more information on the compare routine, see Call Format for a Compare Routine in the Description section.

**user-allocation-procedure**
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied allocate routine that LIB$INSERT_TREE_64 calls to allocate virtual memory for a node. The **user-allocation-procedure** argument is required; LIB$INSERT_TREE_64 always calls the allocate routine.

For more information on the allocate routine, see Call Format for an Allocate Routine in the Description section.

**new-node**
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference

Location where the new key is inserted. The **new-node** argument is the address of an unsigned quadword that is the address of the new node.

**user-data**
OpenVMS usage: user_arg
type: unspecified
access: unspecified
mechanism: by value

User data that LIB$INSERT_TREE_64 passes to the compare and allocate routines. The **user-data** argument is optional.

**Description**
This Description section contains three parts:
- Guidelines for Using LIB$INSERT_TREE_64
- Call Format for a Compare Routine
- Call Format for an Allocate Routine
Guidelines for Using LIB$INSERT_TREE_64
LIB$INSERT_TREE_64 inserts a node in a balanced binary tree. You supply two
routines: compare and allocate. The compare routine compares the symbol key
to the node, and the allocate routine allocates virtual memory for the node to
be inserted. LIB$INSERT_TREE_64 first calls the compare routine to find the
location at which the new node is inserted. Then LIB$INSERT_TREE_64 calls
the allocate routine to allocate memory for the new node. Most programmers
insert data in the new node by initializing it within the allocate routine as soon
as memory has been allocated.

You may pass the data to be inserted into the tree using either the symbol
argument alone or both the symbol and user-data arguments. The symbol
argument is required. It may contain all of the data, just the name of the node,
or the address of the data. If you decide to use symbol to pass just the name of
the node, you must use the user-data argument to pass the rest of the data to be
inserted in the new node.

Call Format for a Compare Routine
The call format of a compare routine is as follows:
user-compare-routine symbol ,comparison-node [,user-data]

LIB$INSERT_TREE_64 passes both the symbol and comparison-node
arguments to the compare routine, using the same passing mechanism that
was used to pass them to LIB$INSERT_TREE_64. The user-data argument is
passed in the same way, but its use is optional.

The user-compare-routine argument in the call to LIB$INSERT_TREE_64
specifies the compare routine. This argument is required. LIB$INSERT_TREE_64
calls the compare routine for every node except the first node in the tree.

The value returned by the compare routine is the result of comparing the symbol
key with the current node. Following are the possible values returned by the
compare routine:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>The symbol argument is less than the current node.</td>
</tr>
<tr>
<td>Zero</td>
<td>The symbol argument is equal to the current node.</td>
</tr>
<tr>
<td>Positive</td>
<td>The symbol argument is greater than the current node.</td>
</tr>
</tbody>
</table>

This is an example of a user-supplied compare routine, written in C.

```c
struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    char Text[80];
};

static long Compare_node(char* Key_string,
                        struct Full_node* Node,
                        void* Dummy)
```
LIB$ Routines
LIB$INSERT_TREE_64 (Alpha and I64 Only)

/*
** This function compares the string described by Key_string with
** the string contained in the data node Node, and returns 0
** if the strings are equal, -1 if Key_string is < Node, and
** 1 if Key_string > Node.
*/
{
    int result;
    result = strcmp(Key_string, Node->Text);
    if (result < 0)
        return -1;
    else if (result == 0)
        return 0;
    else
        return 1;
}

Call Format for an Allocate Routine

LIB$INSERT_TREE_64 calls the allocate routine to allocate virtual memory for a
node. The allocate routine then stores the value of user-data in the field of the
allocated node.

The format of the call is as follows:
user-allocation-procedure symbol,new-node [,user-data]

LIB$INSERT_TREE_64 passes the symbol, new-node, and user-data
arguments to your allocate routine, using the same passing mechanisms that
were used to pass them to LIB$INSERT_TREE_64. Use of user data is optional.

A node header appears at the beginning of each node. The following figure shows
the structure of a node header.

| Left Link | (8 Bytes) |
| Right Link | (8 Bytes) |
| Reserved | (2 Bytes) |

ZK-8082A-GE

Therefore, any node you declare that LIB$INSERT_TREE_64 manipulates must
contain 18 bytes of reserved data at the beginning for the node header.

How a node is structured depends on how you allocate your user data. You can
allocate data in one of two ways:

1. Your data immediately follows the node header. In this case, your allocation
   routine must allocate a block equal in size to the sum of your data plus 18
   bytes for the node header, as shown in the following figure.
2. The node contains the 18 bytes of header information and a quadword pointer to the user data as shown in the following figure.

The **user-allocation-procedure** argument in the call to LIB$INSERT_TREE_64 specifies the allocate routine. This argument is required. LIB$INSERT_TREE_64 always calls the allocate routine.

This is an example of a user-supplied allocate routine written in C.

```c
struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    char Text[80];
};

static long Alloc_node(char* Key_string,
                       struct Full_node** Ret_addr, void* Dummy)
{
    /*
    ** Allocate virtual memory for a new node. Key_string
    ** is the string to be entered into the newly
    ** allocated node. RET_ADDR will contain the address
    ** of the allocated memory.
    */
    long Status_code;
    __int64 Alloc_size = sizeof(struct Full_node);
    extern long LIB$GET_VM_64();
    /*
    ** Allocate node: size of header, plus the length of our data.
    */
```
LIB$ Routines

LIB$INSERT_TREE_64 (Alpha and I64 Only)

```c
Status_code = LIB$GET_VM_64 (&Alloc_size, Ret_addr);
if (!((Status_code & 1)))
    lib$stop(Status_code);
/*
** Store the data in the newly allocated virtual memory.
*/
strcpy((*Ret_addr)->Text, Key_string);
return (Status_code);
```

Condition Values Returned

- **LIB$_NORMAL** Routine successfully completed.
- **LIB$_INSVIRMEM** Insufficient virtual memory. The user-supplied allocation procedure returned an error.
- **LIB$_KEYALRINS** Routine successfully completed, but a key was found in the tree. No new key was inserted.

Any other failure status reported by the user allocation procedure.

Example

The following C program shows the use of LIB$INSERT_TREE_64, LIB$LOOKUP_TREE_64, and LIB$TRAVERSE_TREE_64.

```c
/*
** This program asks the user to enter a series of strings,
** one per line. The user can then query the program to find
** strings that were previously entered. At the end, the entire
** tree is displayed, along with sequence numbers that
** indicate the order in which each element was entered.
**
** This program serves as an example of the use of LIB$INSERT_TREE_64,
** LIB$LOOKUP_TREE_64 and LIB$TRAVERSE_TREE_64.
*/

#pragma pointer_size long
#include <stdio.h>
#include <string.h>
#include <libdef.h>

char Text_line[80];

struct Tree_element /* Define the structure of our record. This record could
{ /* contain many useful data items. */
    long Seq_num;
    char Text[80];
};

struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    struct Tree_element my_node;
};

struct Tree_element Rec; /* Declare an instance of the record */

extern long lib$insert_tree_64(); /* Function to insert node */
extern long lib$traverse_tree_64(); /* Function to walk tree */
extern long lib$lookup_tree_64(); /* Function to find a node */
extern void lib$stop(); /* Routine to signal fatal error */
```
LIB$ Routines
LIB$INSERT_TREE_64 (Alpha and I64 Only)

static long Compare_node_2(); /* Compare entry (2 arg) */
static long Compare_node_3(); /* Compare entry (3 arg) */
static long Alloc_node(); /* Allocation entry */
static long Print_node(); /* Print entry for walk */
static void Display_Node();

main () {
    struct Full_node* Tree_head; /* Head for the tree */
    struct Full_node* New_node; /* New node after insert */
    long Status_code; /* Return status code */
    long Counter; /* Sequence number */
    long flags = 1;
    
    /* Initialize the tree to null */
    Tree_head = NULL;
    printf("Enter one word per line, ^Z to begin searching the tree\n");
    /*
    ** Loop, reading lines of text until the end of the file.
    */
    Counter = 0;
    printf("> ");
    while (gets(Text_line)) {
        Counter++;
        Rec.Seq_num = Counter;
        strcpy(Rec.Text, Text_line);
        Status_code = lib$insert_tree_64 (/* Insert the entry into the tree */
            &Tree_head, &Rec, &flags,
            Compare_node_3, Alloc_node, &New_node);
        if (!(Status_code & 1))
            lib$stop(Status_code);
        printf("> ");
    }
    /*
    ** End of file encountered. Begin searching the tree.
    */
    printf("\nYou will now be prompted for words to find. ");
    printf("Enter one per line.\n");
    Rec.Seq_num = -1;
    printf("Word to find? ");
    while (gets(Text_line)) {
        strcpy(Rec.Text, Text_line);
        Status_code = lib$lookup_tree_64 (&Tree_head, &Rec,
            Compare_node_2, &New_node);
        if (Status_code == LIB$_KEYNOTFOU)
            printf("The word you entered does not appear in the tree.\n");
        else
            Display_Node(New_node);
        printf("Word to find? ");
    }
    /*
    ** The user has finished searching the tree for specific items. It
    ** is now time to traverse the entire tree.
    */
    printf("\n");
    printf("The following is a dump of the tree. Notice that the words\n");
    printf("are in alphabetical order\n");
Status_code = lib$straverse_tree_64(&Tree_head, Print_node, 0);
return(Status_code);
}

static long Print_node(struct Full_node* Node, void* Dummy)
{
    /*
    ** Print the string contained in the current node.
    */
    printf("%d\t%s\n", Node->my_node.Seq_num, Node->my_node.Text);
    return(LIB$_NORMAL);
}

static long Alloc_node(struct Tree_element* Rec, struct Full_node** Ret_addr, void* Dummy)
{
    /*
    ** Allocate virtual memory for a new node. Rec is the
    ** data record to be entered into the newly
    ** allocated node. RET_ADDR will contain the address
    ** of the allocated memory.
    */
    long Status_code;
    __int64 Alloc_size = sizeof(struct Full_node);
    extern long lib$get_vm_64();
    /*
    ** Allocate node: size of header, plus the length of our data.
    */
    Status_code = lib$get_vm_64 (&Alloc_size, Ret_addr);
    if (!(Status_code & 1))
        lib$stop(Status_code);
    /*
    ** Store the data in the newly allocated virtual memory.
    */
    (*Ret_addr)->my_node.Seq_num = Rec->Seq_num;
    strcpy((*Ret_addr)->my_node.Text, Rec->Text);
    return (Status_code);
}

static long Compare_node_3(struct Tree_element* Rec, struct Full_node* Node, void* Dummy)
{
    /*
    ** Call the 2 argument version of the compare routine
    */
    return(Compare_node_2 ( Rec, Node ));
}

static long Compare_node_2(struct Tree_element* Rec, struct Full_node* Node)
{ /*
    ** This function compares the string described by Key_string with
    ** the string contained in the data node Node, and returns 0
    ** if the strings are equal, -1 if Key_string is < Node, and
    ** 1 if Key_string > Node.
    */
    int result;
LIB$ Routines
LIB$INSERT_TREE_64 (Alpha and I64 Only)

/*
** Return the result of the comparison.
*/
result = strcmp(Rec->Text, Node->my_node.Text);
if (result < 0)
    return -1;
else if (result == 0)
    return 0;
else
    return 1;
}

static void Display_Node(struct Full_node* Node)
{
    /*
    ** This routine prints the data into the node of the tree
    ** once LIB$LOOKUP_TREE has been called to find the node.
    */
    printf("The sequence number for " Node->my_node.Text, Node->my_node.Seq_num);

The output generated by this program is as follows:

$ run tree
Enter one word per line, ^Z to begin searching the tree
> apple
> orange
> peach
> pear
> grapefruit
> lemon
> ^Z

You will now be prompted for words to find. Enter one per line.

Word to find? lime
The word you entered does not appear in the tree

Word to find? orange
The sequence number for "orange" is 2

Word to find? ^Z

The following is a dump of the tree. Notice that the words are in alphabetical order
1 apple
5 grapefruit
6 lemon
2 orange
3 peach
4 pear
$
LIB$INSQHI
Insert Entry at Head of Queue

The Insert Entry at Head of Queue routine inserts a queue entry at the head of the specified self-relative longword interlocked queue. LIB$INSQHI makes the INSQHI instruction available as a callable routine.

Format

LIB$INSQHI entry,header [,retry-count]

Returns

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

Arguments

entry
OpenVMS usage: unspecified
type: unspecified
access: modify
mechanism: by reference, array reference

Entry to be inserted by LIB$INSQHI. The entry argument contains the address of this signed quadword-aligned array that must be at least 8 bytes long. Bytes following the first 8 bytes can be used for any purpose by the calling program. For Alpha and I64 systems, the entry argument must contain a 32-bit sign-extended address. An illegal operand exception occurs for any other form of address.

header
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue into which entry is to be inserted. The header argument contains the address of this signed aligned quadword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue. For Alpha systems, the header argument must contain a 32-bit sign-extended address. An illegal operand exception occurs for any other form of address.

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

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$INSQHI

The number of times the insertion is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of an unsigned longword that contains the retry count value. A value of 1 causes no retries. The default value is 10.

Description

The queue into which LIB$INSQHI inserts an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

Self-Relative Queues

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry’s predecessor and successor. If these links are longwords, the queue is referred to as a self-relative longword queue.

You can use the LIB$INSQHI, LIB$INSQTI, LIB$REMQHI, and LIB$REMQTI routines to manage your self-relative longword queue on a VAX or an Alpha or I64 system. These routines implement the INSQHI, INSQTI, REMQHI, and REMQTI instructions that allow you to insert and remove an entry at the head or tail of a self-relative longword queue.

Synchronization

When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

Alignment

Use of the self-relative longword queue routines requires that the queue header and each of the queue entries be quadword aligned. You can use the Run-Time Library routine LIB$GET_VM on a VAX, Alpha, or I64 system to allocate quadword-aligned virtual memory for a queue.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Values Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was added to the front of the queue, and the resulting queue contains more than one entry.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not quadword aligned, or the header address equals the entry address.</td>
</tr>
</tbody>
</table>
LIB$ ONEENTQUE  Routine successfully completed. The entry was added to the front of the queue, and the resulting queue contains one entry.

LIB$ SECINTFAI  A secondary interlock failure occurred; the insertion was attempted the number of times specified by retry-count. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

Examples

1. INTEGER*4 FUNCTION INSERT_Q (QENTRY)
   COMMON/QUEUES/QHEADER
   INTEGER*4 QENTRY(10), QHEADER(2)
   INSERT_Q = LIB$INSQHI (QENTRY, QHEADER)
   RETURN
END

   This is a Fortran application using processor-shared memory.

2. COM (QUEUES) QENTRY%(9), QHEADER%(1)
   EXTERNAL INTEGER FUNCTION LIB$INSQHI
   IF LIB$INSQHI (QENTRY%() BY REF, QHEADER%() BY REF) AND 1%
     THEN GOTO 1000
     .
     .
1000 REM INSERTED OK

   In BASIC and Fortran, queues can be quadword aligned in a named COMMON block by using a linker option file to specify PSECT alignment. For instance, to create a COMMON block called QUEUES, use the LINK command with the FILE/OPTIONS qualifier, where FILE.OPT is a linker option file containing the following line:

   PSECT = QUEUES, QUAD
LIB$ Routines
LIB$INSQHIQ (Alpha and I64 Only)

LIB$INSQHIQ (Alpha and I64 Only)
Insert Entry at Head of Queue

The Insert Entry at Head of Queue routine inserts a queue entry at the head of the specified self-relative quadword interlocked queue. LIB$INSQHIQ makes the INSQHIQ instruction available as a callable routine.

Format

LIB$INSQHIQ entry ,header [,retry-count]

Returns

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

Arguments

entry
OpenVMS usage: unspecified
type: unspecified
access: modify
mechanism: by reference, array reference

Entry to be inserted by LIB$INSQHIQ. The entry argument contains the address of this signed octaword-aligned array that must be at least 16 bytes long. Bytes following the first 16 bytes can be used for any purpose by the calling program.

header
OpenVMS usage: octaword_signed
type: octaword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue into which entry is to be inserted. The header argument contains the address of this signed aligned octaword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

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

The number of times the insertion is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of an unsigned longword that contains the retry count value. A value of 1 causes no retries. The default value is 10.
**Description**

The queue into which LIB$INSQHIQ inserts an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or cross-processor queues.

**Self-Relative Queues**

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry's predecessor and successor. If these links are quadwords, the queue is referred to as a self-relative quadword queue.

You can use the LIB$INSQHIQ, LIB$INSQTIQ, LIB$REMQHIQ, and LIB$REMQTIQ routines to manage your self-relative quadword queue on an Alpha or I64 system. These routines implement the INSQHIQ, INSQTIQ, REMQHIQ, and REMQTIQ instructions that allow you to insert and remove an entry at the head or tail of a self-relative quadword queue.

**Synchronization**

When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

**Alignment**

Use of the self-relative quadword queue routines requires that the queue header and each of the queue entries be octaword aligned. You can use the Run-Time Library routine LIB$GET_VM_64 to allocate octaword aligned virtual memory for a queue.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was added to the front of the queue, and the resulting queue contains more than one entry.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not octaword aligned, or the header address equals the entry address.</td>
</tr>
<tr>
<td>LIB$_ONEENTQUE</td>
<td>Routine successfully completed. The entry was added to the front of the queue, and the resulting queue contains one entry.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$INSQHIQ (Alpha and I64 Only)

LIB$SECINTFAI  A secondary interlock failure occurred; the insertion was attempted the number of times specified by \texttt{retry-count}. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

Example

The following Fortran application uses processor-shared memory:

\begin{verbatim}
INTEGER*4 FUNCTION INSERT_Q (QENTRY)
COMMON/QUEUES/QHEADER
INTEGER*8 QENTRY(10), QHEADER(2)
INSERT_Q = LIB$INSQHIQ (QENTRY, QHEADER)
RETURN
END
\end{verbatim}

In Fortran, queues can be octaword aligned in a named COMMON block by using a linker option file to specify PSECT alignment. For instance, to create a COMMON block called QUEUES, use the LINK command with the FILE/OPTIONS qualifier, where FILE.OPT is a linker option file containing the following line:

\begin{verbatim}
PSECT = QUEUES, OCTA
\end{verbatim}
LIB$INSQTI
Insert Entry at Tail of Queue

The Insert Entry at Tail of Queue routine inserts a queue entry at the tail of the specified self-relative longword interlocked queue. † LIB$INSQTI makes the INSQTI instruction available as a callable routine.

Format

LIB$INSQTI entry ,header [,retry-count]

Returns

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

Arguments

entry
OpenVMS usage: unspecified
type: unspecified
access: modify
mechanism: by reference, array reference

Entry to be inserted at the tail of the queue by LIB$INSQTI. The entry argument contains the address of this signed quadword-aligned array that must be at least 8 bytes long. Bytes following the first 8 bytes can be used for any purpose by the calling program.

For Alpha and I64 systems, the entry argument must contain a 32-bit sign-extended address. An illegal operand exception occurs for any other form of address.

header
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue into which the queue entry is to be inserted. The header argument contains the address of this signed aligned quadword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

For Alpha and I64 systems, the header argument must contain a 32-bit sign-extended address. An illegal operand exception occurs for any other form of address.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
retry-count
OpenVMS usage: longword_unsigned
type: longword (unsigned)
access: read only
mechanism: by reference

The number of times the insertion is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of a longword which contains the retry count value. The default value is 10.

Description
The queue into which LIB$INSQTI inserts an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

Self-Relative Queues
A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry’s predecessor and successor. If these links are longwords, the queue is referred to as a self-relative longword queue.

You can use the LIB$INSQHI, LIB$INSQTI, LIB$REMQHI, and LIB$REMQTI routines to manage your self-relative longword queue on a VAX, Alpha, or I64 system. These routines implement the INSQHI, INSQTI, REMQHI, and REMQTI instructions that allow you to insert and remove an entry at the head or tail of a self-relative longword queue.

Synchronization
When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

Alignment
Use of the self-relative longword queue routines requires that the queue header and each of the queue entries be quadword aligned. You can use the Run-Time Library routine LIB$GET_VM on a VAX, Alpha, or I64 system to allocate quadword-aligned virtual memory for a queue.
## Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ NORMAL</td>
<td>Routine successfully completed. The entry was added to the tail of the queue: the resulting queue contains more than one entry.</td>
</tr>
<tr>
<td>SS$ ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not quadword aligned, or the header address equals the entry address.</td>
</tr>
<tr>
<td>LIB$ ONEENTQUE</td>
<td>Routine successfully completed. The entry was added to the tail of the queue: the resulting queue contains one entry.</td>
</tr>
<tr>
<td>LIB$ SECINTFAI</td>
<td>A secondary interlock failure occurred; the insertion was attempted the number of times specified by <code>retry-count</code>. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$INSQTIQ (Alpha and I64 Only)

LIB$INSQTIQ (Alpha and I64 Only)
Insert Entry at Tail of Queue

The Insert Entry at Tail of Queue routine inserts a queue entry at the tail of the specified self-relative quadword interlocked queue. LIB$INSQTIQ makes the INSQTIQ instruction available as a callable routine.

Format

LIB$INSQTIQ entry,header [,retry-count]

Returns

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

Arguments

entry
OpenVMS usage: unspecified
type: unspecified
access: modify
mechanism: by reference, array reference

Entry to be inserted at the tail of the queue by LIB$INSQTIQ. The entry argument contains the address of this signed octaword-aligned array that must be at least 16 bytes long. Bytes following the first 16 bytes can be used for any purpose by the calling program.

header
OpenVMS usage: octaword_signed
type: octaword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue into which the queue entry is to be inserted. The header argument contains the address of this signed aligned octaword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

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

The number of times the insertion is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of a longword that contains the retry count value. The default value is 10.
Description

The queue into which LIB$INSQTIQ inserts an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

Self-Relative Queues

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry's predecessor and successor. If these links are quadwords, the queue is referred to as a self-relative quadword queue.

You can use the LIB$INSQHIQ, LIB$INSQTIQ, LIB$REMQHIQ, and LIB$REMQTIQ routines to manage your self-relative quadword queue on an Alpha or I64 system. These routines implement the INSQHIQ, INSQTIQ, REMQHIQ, and REMQTIQ instructions that allow you to insert and remove an entry at the head or tail of a self-relative quadword queue.

Synchronization

When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

Alignment

Use of the self-relative quadword queue routines requires that the queue header and each of the queue entries be octaword aligned. You can use the Run-Time Library routine LIB$GET_VM_64 to allocate octaword aligned virtual memory for a queue.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was added to the tail of the queue: the resulting queue contains more than one entry.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not octaword aligned, or the header address equals the entry address.</td>
</tr>
<tr>
<td>LIB$ ONEENTQUE</td>
<td>Routine successfully completed. The entry was added to the tail of the queue: the resulting queue contains one entry.</td>
</tr>
</tbody>
</table>
A secondary interlock failure occurred; the insertion was attempted the number of times specified by **retry-count**. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.
LIB$INSV
Insert a Variable Bit Field

The Insert a Variable Bit Field routine replaces the variable bit field specified by
the base, position, and size arguments with bits 0 through \((\text{size} - 1)\) of the source
field. If the size of the bit field is zero, nothing is inserted. LIB$INSV makes the
VAX INSV instruction available as a callable routine. ¹

Format

LIB$INSV longword-integer-source,position,size,base-address

Returns

None.

Arguments

longword-integer-source
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Source field to be inserted by LIB$INSV. The \text{longword-integer-source} argument is the address of a signed longword integer that contains this source field.

position
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Bit position relative to the base address where insertion of \text{longword-integer-source} is to begin. The \text{position} argument is the address of a longword integer that contains this relative bit position.

size
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

Size of the bit field to be inserted by LIB$INSV. The \text{size} argument is the address of an unsigned byte that contains the size of this bit field. The maximum size is 32 bits.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
base-address
OpenVMS usage: address  
type: address  
access: read only  
mechanism: by value  

Field into which LIB$INSV writes the source field. The base-address argument is an unsigned longword containing the base address of this aligned bit string.

Condition Value Signaled

SS$_ROPRAND  
A reserved operand fault is signaled if a size greater than 32 is specified.

Examples

1.  INTEGER*4  COND_VALUE  
    CALL LIB$INSV (4, 0, 3, COND_VALUE)  
    This example shows how to set bits 0 through 2 of longword COND_VALUE to the value 4 in Fortran.

2.  DECLARE INTEGER COND_VALUE  
    CALL LIB$INSV (4%, 0%, 3%, COND_VALUE)  
    This example uses BASIC to set bits 0 through 2 of longword COND_VALUE to the value 4.


LIB$INT_OVER  

Integer Overflow Detection

The Integer Overflow Detection routine enables or disables integer overflow detection for the calling routine activation. The previous integer overflow enable setting is returned. †

This routine is available on OpenVMS Alpha and I64 systems in translated form and is applicable to translated VAX images only.

Format

LIB$INT_OVER new-setting

Returns

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

Old integer overflow enable setting (the previous contents of SF$W_PSW[PSW$V_IV] in the caller’s frame).

Argument

new-setting

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

New integer overflow enable setting. The new-setting argument is the address of an unsigned longword that contains the new integer overflow enable setting. Bit 0 set to 1 means enable, bit 0 set to 0 means disable.

Description

The caller’s stack frame will be modified by this routine.

LIB$INT_OVER affects only the current routine activation and does not affect any of its callers or any routines that it may call. However, the setting remains in effect for any routines which are subsequently entered through a JSB entry point.

Condition Values Returned

None.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$INT_OVER

Example

INTOVF: ROUTINE OPTIONS (MAIN);
DECLARE LIB$INT_OVER ENTRY (FIXED BINARY (7)) /* Address of byte for
/* enable/disable
/* setting */
RETURNS (FIXED BINARY (31)); /* Old setting */
DECLARE DISABLE FIXED BINARY (7) INITIAL (0) STATIC READONLY;
DECLARE (A,B) FIXED BINARY (7);
ON FIXEDOVERFLOW PUT SKIP LIST ('Overflow');
A = 127;
B = A + 2;
PUT LIST ('In MAIN');
BEGIN;
DECLARE RESULT FIXED BINARY (31);
/* Disable recognition of integer overflow in this block */
RESULT = LIB$INT_OVER (DISABLE);
B = A + 2;
PUT SKIP LIST ('In BEGIN block');
CALL Q;
  Q: routine;
  B = A + 2;
  PUT LIST ('In Q');
END Q;
END /* Begin */;
END INTOVF;

This PL/I routine shows how to use LIB$INT_OVER to enable or disable the
detection of integer overflow. Note that in PL/I, integer overflow is always
enabled unless explicitly overridden by a call to this routine. However, disabling
integer overflow is only effective for the block which calls this routine; descendent
blocks are unaffected. The output generated by this PL/I program is as follows:

In MAIN
In BEGIN block
Overflow In Q
LIB$LEN
Length of String Returned as Longword Value

The Length of String Returned as Longword Value routine returns the length of a string.

Format

LIB$LEN source-string

Returns

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

Length of the source string, extracted and zero-extended to 32 bits.

Argument

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

Source string whose length is returned by LIB$LEN. The source-string argument contains the address of a descriptor pointing to this source string.

Description

The BASIC and Fortran intrinsic function LEN generates equivalent in-line code at run time. Therefore, it is more efficient for BASIC and Fortran users to use the intrinsic function LEN than to call LIB$LEN.

If you need both the length of the string and the address of its first byte, you should use LIB$ANALYZE_SDESC or LIB$ANALYZE_SDESC_64.

Condition Values Returned

None.
LIB$ LOCC
Locate a Character

The Locate a Character routine locates a character in a string by comparing successive bytes in the string with the character specified. The search continues until the character is found or the string has no more characters. LIB$LOCC makes the VAX LOCC instruction available as a callable routine. ¹

Format

LIB$LOCC character-string ,source-string

Returns

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

The relative position from the start of source-string to the first equal character or zero if no match is found.

Arguments

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

String whose initial character is used by LIB$LOCC in the search. The character-string argument contains the address of a descriptor pointing to this string. Only the first character of character-string is used, and its length is not checked.

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

String to be searched by LIB$LOCC. The source-string argument is the address of a descriptor pointing to this character string.

Description

LIB$LOCC returns the position of the first equal character relative to the start of the source string as an index. An index is the relative position of the first occurrence of a substring in the source string. If no character matches or if the string has a length of zero, then a zero is returned, indicating that the character was not found.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Condition Values Returned

None.

Examples

1. IDENTIFICATION DIVISION.
   PROGRAM-ID. LIBLOC.
   ENVIRONMENT DIVISION.
   DATA DIVISION.
   WORKING-Storage SECTION.
   01 SEARCH-STRING PIC X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
   01 SEARCH-CHAR PIC X.
   01 IND-POS PIC 9(9) USAGE IS COMP.
   01 DISP-IND PIC 9(9).
   ROUTINE DIVISION.
   001-MAIN.
   MOVE SPACE TO SEARCH-CHAR.
   DISPLAY " ".
   DISPLAY "ENTER SEARCH CHARACTER: " WITH NO ADVANCING.
   ACCEPT SEARCH-CHAR.
   CALL "LIB$LOCC" USING BY DESCRIPTOR SEARCH-CHAR, SEARCH-STRING GIVING IND-POS.
   IF IND-POS = ZERO
   DISPLAY "CHAR ENTERED (" SEARCH-CHAR ") NOT A VALID SEARCH CHAR"
   STOP RUN.
   MOVE IND-POS TO DISP-IND.
   DISPLAY "SEARCH CHAR (" SEARCH-CHAR ") WAS FOUND IN POSITION " DISP-IND.
   GO TO 001-MAIN.

This COBOL program accepts a character as input and returns as output the
character's position in a search string. The output generated by this COBOL
program is as follows:

$ RUN LIBLOC
ENTER SEARCH CHARACTER: X
SEARCH CHAR (X) WAS FOUND IN POSITION 000000024

ENTER SEARCH CHARACTER: Y
SEARCH CHAR (Y) WAS FOUND IN POSITION 000000025

ENTER SEARCH CHARACTER: B
SEARCH CHAR (B) WAS FOUND IN POSITION 000000002

ENTER SEARCH CHARACTER: b
CHAR ENTERED (b) NOT A VALID SEARCH CHAR

Notice that uppercase and lowercase letters are not considered equal.
This BASIC example also shows the use of LIB$LOCC. The output generated by this BASIC program is “5”. 

EXTERNAL INTEGER FUNCTION LIB$LOCC
I% = 0
CHARSTR$ = ‘DAY’
SRCSTR$ = ‘ONE DAY AT A TIME’
I% = LIB$LOCC(CHARSTR$, SRCSTR$)
PRINT I% 
90 END
LIB$LOCK_IMAGE
Lock an Image in the Process Working Set (Alpha and I64 Only)

Locks the specified image in the process’s working set.

Format

LIB$LOCK_IMAGE address

Returns

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

Argument

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

Address of a byte within the image to be locked in the working set. If the address argument is 0, the current image (which contains the call to LIB$LOCK_IMAGE) is locked in the working set.

Description

LIB$LOCK_IMAGE locks the specified image in the process’s working set.

This routine is typically used by a privileged user before the program, executing in kernel mode, raises IPL above IPL 2. Above IPL 2, paging is not allowed by the system. The program must access only pages valid in the process’s working set.

Condition Values Returned

SS$_WASSET The specified image is locked in the working set and had previously been locked in the working set.

SS$_WASCLR The specified image is locked in the working set and had previously not been locked in the working set.

Other status codes returned by sys$lkwset_64.

Examples

New example TBS?
LIB$ LOOKUP_KEY
Look Up Keyword in Table

The Look Up Keyword in Table routine scans a table of keywords to find one that matches the keyword or keyword abbreviation specified by search-string.

Format

LIB$LOOKUP_KEY search-string ,key-table-array [,key-value] [,keyword-string] [,resultant-length]

Returns

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

Arguments

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

String for which LIB$LOOKUP_KEY will search in the keyword table. The search-string argument is the address of a descriptor pointing to this string.

key-table-array
OpenVMS usage: unspecified
type: unspecified
access: read only
mechanism: by reference, array reference

Keyword table. The key-table-array argument contains the address of an array that is this keyword table.

key-value
OpenVMS usage: longword_unsigned
type: longword (unsigned)
access: write only
mechanism: by reference

Associated value of the keyword found by LIB$LOOKUP_KEY. The key-value argument contains the address of an unsigned longword into which LIB$LOOKUP_KEY writes the associated value of the matched keyword.

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

Full keyword string matched. The keyword-string argument contains the address of a character-string descriptor. LIB$LOOKUP_KEY writes the complete text of the matched keyword into the character string.
resultant-length
OpenVMS usage: word unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Number of characters copied into the character-string pointed to by keyword-string, not counting padding in the case of a fixed-length string. The resultant-length argument is the address of an unsigned word integer that contains the number of characters in the matched keyword that were copied into the character-string.

Description
LIB$LOOKUP_KEY is intended to help programmers to write utilities that have command qualifiers with values.

LIB$LOOKUP_KEY locates a matching keyword or keyword abbreviation by comparing the first n characters of each keyword in the keyword table with the supplied string, where n is the length of the supplied string.

When a keyword match is found, the following information is optionally returned to the caller:
• The longword value associated with the matched keyword
• The full keyword string (any descriptor type)

An exact match is found if the length of the keyword found is equal to the length of the supplied string.

If an exact keyword match is found, no further processing is performed, and a normal return status is returned to the caller. Otherwise, after a match has been found, the rest of the keyword table is scanned. If an additional match is found, a “not enough characters” return status is returned to the caller. If the keyword table contains a keyword that is an abbreviation of another keyword in the table, an exact match can occur for short abbreviations.

Figure lib–5 shows the structure of the keyword table, which the calling program creates for this routine.

Figure lib–5  Keyword Table

Vector

Vector–Count

Address of Keyword String

Associated Keyword Value

\[ \ldots \]

Counted–ASCII String

Keyword String

ZK–1976–GE

Vector-count is the number of longwords that follow, and counted-ASCII-string starts with a byte that is the unsigned count of the number of ASCII characters that follow.
Because of the format of the keyword table, this routine cannot be called easily from high-level languages. The examples that follow show how to use a macro, \$LIB_KEY_TABLE, to construct a keyword table from MACRO or BLISS. A separate example shows how a table could be constructed in Fortran.

Use of the \$LIB_KEY_TABLE macro results in data that is not position-independent code (PIC). If your application requires PIC data, you must fill in the address of the keyword strings at execution time. See the Fortran example (example 3) for a demonstration of this technique.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. A unique keyword match was found.</td>
</tr>
<tr>
<td>LIB$_AMBKEY</td>
<td>Multiple keyword match found. Not enough characters were specified to allow a unique match.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory to return keyword string. This is only possible if keyword-string is a dynamic string.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid arguments, not enough arguments, and/or bad keyword table.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>String truncated.</td>
</tr>
<tr>
<td>LIB$_UNRKEY</td>
<td>The keyword you specified does not appear in the keyword table you specified.</td>
</tr>
</tbody>
</table>

**Examples**

1. **KEYTABLE:**

   $LIB_KEY_TABLE < -
   <ADD, 1>, -
   <DELETE, 2>, -
   <EXIT, 3>>

   This VAX MACRO fragment defines a keyword table named KEYTABLE containing the three keywords ADD, DELETE, and EXIT with associated keyword values of 1, 2, and 3, respectively.

   The $LIB_KEY_TABLE macro is supplied in the default macro library SYS$LIBRARY:STARLET.MLB. Because this library is automatically searched by the assembler, you do not have to specify it in the DCL command MACRO.

2. **LIBRARY 'SYS$LIBRARY:STARLET.L32';**

   OWN
   KEYTABLE: $LIB_KEY_TABLE (  
   (ADD, 1),
   (DELETE, 2),
   (EXIT, 3));

   This BLISS code fragment specifies that SYS$LIBRARY:STARLET.L32 is to be searched to resolve references. It defines a keyword table named KEYTABLE containing the three keywords ADD, DELETE, and EXIT with associated keyword values of 1, 2, and 3, respectively.
The $LIB_KEY_TABLE macro is supplied in the BLISS library
SYS$LIBRARY:STARLET.L32 and in the BLISS require file
SYS$LIBRARY:STARLET.REQ. BLISS does not automatically search either of
these files, so you must explicitly cause them to be searched by including the
appropriate LIBRARY or REQUIRE statement in your module. You should
use the precompiled library because it is more efficient for the compiler.

3. PARAMETER ( 
   1 MAXKEYSIZE = 6, ! Maximum keyword size
   2 NKEYS = 3) ! Number of keywords
BYTE KEYS (MAXKEYSIZE+1, NKEYS)
INTEGER*4 KEYTABLE (0:NKEYS*2)
DATA KEYWORDS /
   1 3,‘A’,‘D’,‘D’,‘ ‘,‘ ‘,‘ ‘,‘ ‘, ! Counted ASCII ‘ADD’
   2 6,‘D’,‘E’,‘L’,‘E’,‘T’,‘E’, ! Counted ASCII ‘DELETE’
   3 4,‘E’,‘X’,‘I’,‘T’,‘ ‘,‘ ‘ / ! Counted ASCII ‘EXIT’

   KEYTABLE(0) = NKEYS*2 ! Number of longwords to follow
   KEYTABLE(1) = %LOC(KEYWORDS(1,1)) ! Address of keyword string
   KEYTABLE(2) = 1 ! Keyword value for ‘ADD’
   KEYTABLE(3) = %LOC(KEYWORDS(1,2)) ! Address of keyword string
   KEYTABLE(4) = 2 ! Keyword value for ‘DELETE’
   KEYTABLE(5) = %LOC(KEYWORDS(1,3)) ! Address of keyword string
   KEYTABLE(6) = 3 ! Keyword value for ‘EXIT’

This Fortran code fragment constructs a keyword table named KEYTABLE
containing the three keywords ADD, DELETE, and EXIT with associated
keyword values of 1, 2, and 3, respectively. This construction method results
in position-independent coded data, although the generated code for the
typical Fortran module contains other non-PIC values.
LIB$ LOOKUP_TREE
Look Up an Entry in a Balanced Binary Tree

The Look Up an Entry in a Balanced Binary Tree routine looks up an entry in a balanced binary tree. †

Format

LIB$LOOKUP_TREE  treehead ,symbol ,user-compare-routine ,new-node

Returns

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

Arguments

treehead  
OpenVMS usage:  address  
type:  address  
access:  read only  
mechanism:  by reference

Tree head for the binary tree. The treehead argument is the address of an unsigned longword that is this tree head.

symbol  
OpenVMS usage:  user_arg  
type:  longword (unsigned)  
access:  unspecified  
mechanism:  unspecified

Key to be looked up in the binary tree.

user-compare-routine  
OpenVMS usage:  procedure  
type:  procedure value  
access:  function call (before return)  
mechanism:  by value

User-supplied compare routine that LIB$LOOKUP_TREE calls to compare a symbol with a node. The value returned by the compare routine indicates the relationship between the symbol key and the current node.

For more information on the compare routine, see Call Format for a Compare Routine in the Description section.

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

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
Location where the new symbol was found. The new-node argument is the address of an unsigned longword that is the new node location.

Description

Call Format for a Compare Routine
The call format of a compare routine is as follows:

\texttt{user-compare-routine} symbol, comparison-node [,user-data]

\texttt{LIB$LOOKUP_TREE} passes both the \texttt{symbol} and \texttt{comparison-node} arguments to the compare routine, using the same passing mechanism that was used to pass them to \texttt{LIB$LOOKUP_TREE}. The \texttt{user-data} argument is passed in the same way, but its use is optional.

The \texttt{user-compare-routine} argument in the call to \texttt{LIB$LOOKUP_TREE} specifies the compare routine. This argument is required. \texttt{LIB$LOOKUP_TREE} calls the compare routine for every node except the first node in the tree.

The value returned by the compare routine is the result of comparing the symbol key with the current node. The table below lists the possible values returned by the compare routine:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>The \texttt{symbol} argument is less than the current node.</td>
</tr>
<tr>
<td>Zero</td>
<td>The \texttt{symbol} argument is equal to the current node.</td>
</tr>
<tr>
<td>Positive</td>
<td>The \texttt{symbol} argument is greater than the current node.</td>
</tr>
</tbody>
</table>

For an example of a user-supplied compare routine written in C, see the description of \texttt{LIB$INSERT_TREE}.

Condition Values Returned

\texttt{LIB$_NORMAL} Routine successfully completed. The key was found.

\texttt{LIB$_KEYNOTFOU} Error. The key was not found.

Example

The C example provided in the description of \texttt{LIB$INSERT_TREE} also demonstrates how to use \texttt{LIB$LOOKUP_TREE}. Refer to that example for assistance in using this routine.
Look Up an Entry in a Balanced Binary Tree

The Look Up an Entry in a Balanced Binary Tree routine looks up an entry in a balanced binary tree.

Format

LIB$LOOKUP_TREE_64  treehead, symbol, user-compare-routine, new-node

Returns

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

Arguments

treehead
OpenVMS usage: address
type: address
access: read only
mechanism: by reference

Tree head for the binary tree. The treehead argument is the address of an unsigned quadword that is this tree head.

symbol
OpenVMS usage: user_arg
type: quadword (unsigned)
access: unspecified
mechanism: unspecified

Key to be looked up in the binary tree.

user-compare-routine
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied compare routine that LIB$LOOKUP_TREE_64 calls to compare a symbol with a node. The value returned by the compare routine indicates the relationship between the symbol key and the current node.

For more information on the compare routine, see Call Format for a Compare Routine in the Description section.

new-node
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference

Location where the new symbol was found. The new-node argument is the address of an unsigned quadword that is the new node location.
Description

Call Format for a Compare Routine
The call format of a compare routine is as follows:

```
user-compare-routine symbol ,comparison-node [,user-data]
```

LIB$LOOKUP_TREE_64 passes both the `symbol` and `comparison-node` arguments to the compare routine, using the same passing mechanism that was used to pass them to LIB$LOOKUP_TREE_64. The `user-data` argument is passed in the same way, but its use is optional.

The `user-compare-routine` argument in the call to LIB$LOOKUP_TREE_64 specifies the compare routine. This argument is required. LIB$LOOKUP_TREE_64 calls the compare routine for every node except the first node in the tree.

The value returned by the compare routine is the result of comparing the symbol key with the current node. The following table lists the possible values returned by the compare routine:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>The <code>symbol</code> argument is less than the current node.</td>
</tr>
<tr>
<td>Zero</td>
<td>The <code>symbol</code> argument is equal to the current node.</td>
</tr>
<tr>
<td>Positive</td>
<td>The <code>symbol</code> argument is greater than the current node.</td>
</tr>
</tbody>
</table>

For an example of a user-supplied compare routine written in C, see the description of LIB$INSERT_TREE_64.

Condition Values Returned

- `LIB$_NORMAL` Routine successfully completed. The key was found.
- `LIB$_KEYNOTFOU` Error. The key was not found.

Example

The C example provided in the description of LIB$INSERT_TREE_64 also demonstrates how to use LIB$LOOKUP_TREE_64. Refer to that example for assistance in using this routine.
LIB$LP_LINES
Lines on Each Printer Page

The Lines on Each Printer Page routine computes the default number of lines on a printer page. This routine can be used by native-mode OpenVMS utilities that produce listing files and paginate files.

Format
LIB$LP_LINES

Returns
OpenVMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by value

The default number of lines on a physical printer page. If the logical name translation or conversion to binary fails, a default value of 66 is returned.

Arguments
None.

Description
LIB$LP_LINES computes the default number of lines on a printer page. This routine can be used by native-mode OpenVMS utilities that produce listing files and paginate files. The algorithm used by LIB$LP_LINES is:

1. Translate the logical name SYS$LP_LINES.
2. Convert the ASCII value obtained to a binary integer.
3. Verify that the resulting value is in the range [30:255].
4. If any of the prior steps fail, return the default paper size of 66 lines.

You can use LIB$LP_LINES to monitor the current default length of the line printer page. You can also supply your own default length for the current process. United States standard paper stock permits 66 lines on each physical page.

If you are writing programs for a utility that formats a listing file to be printed on a line printer, you can use LIB$LP_LINES to make your utility independent of the default page length. Your program can use LIB$LP_LINES to obtain the current length of the page. It can then calculate the number of lines of text on each page by subtracting the lines used for margins and headings.
The following is one suggested format:

- Three lines for the top margin
- Three lines for the bottom margin
- Three lines for listing heading information, consisting of:
  - A language-processor identification line
  - A source-program identification line
  - One blank line

**Condition Values Returned**

None.

**Examples**

1. ```
   lplines = LIB$LP_LINES()
   PRINT 10, lplines
   10 Format (' Line printer page = ',I5,' lines.')
   end
   ```

   This Fortran program displays the current default length of the line printer page.

2. ```
   100 EXTERNAL INTEGER FUNCTION LIB$LP_LINES
   200 DECLARE INTEGER LPLINES
   300 LPLINES = LIB$LP_LINES
   400 PRINT "Line printer page = "; LPLINES32767 END
   ```

   This BASIC program displays the current default length of the line printer page.

3. ```
   PROGRAM LINES(OUTPUT);
   FUNCTION LIB$LP_LINES : INTEGER;
   EXTERN;
   BEGIN
   WRITELN('Line printer page = ',LIB$LP_LINES,' lines.');
   END.
   ```

   This Pascal program displays the current default length of the line printer page.
LIB$MATCHC
Match Characters, Return Relative Position

The Match Characters, Return Relative Position routine searches a source string for a specified substring and returns an index, which is the relative position of the first occurrence of a substring in the source string. The relative character positions returned by LIB$MATCHC are numbered 1, 2, . . . , n. Thus, zero means that the substring was not found.

Format

LIB$MATCHC  sub-string ,source-string

Returns

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

The relative position of the first character of the substring if found, or zero if not found.

Arguments

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

Substring to be found. The sub-string argument is the address of a descriptor pointing to this substring.

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

Source string to be searched by LIB$MATCHC. The source-string argument is the address of a descriptor pointing to this source string.

Description

LIB$MATCHC searches a source string for a specified substring and returns an index, which is the relative position of the first occurrence of a substring in the source string.

The relative character positions returned by LIB$MATCHC are numbered 1, 2, . . . , n. Thus, zero means that the substring was not found.
If the substring has a zero length, LIB$MATCHC returns the value 1, indicating success, no matter how long the source string is. If the source string has a zero length and the substring has a nonzero length, zero is returned, indicating that the substring was not found.

**Condition Values Returned**

None.
LIB$ MATCH_COND
Match Condition Values

The Match Condition Values routine checks to see if a given condition value matches a list of condition values that you supply.

Format

LIB$MATCH_COND match-condition-value,compare-condition-value,...

Returns

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

A zero, if the input condition value did not match any condition value in the list, or i – 1, for a match between the first argument and the ith argument.

Arguments

match-condition-value
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by reference

Condition value to be matched. The match-condition-value argument is the address of an unsigned longword that contains this condition value.

compare-condition-value
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by reference

The condition values to be compared to match-condition-value. The compare-condition-value arguments are the addresses of the unsigned longwords that contain these condition values.

Description

LIB$MATCH_COND checks for a match between the condition value addressed by match-condition-value and the condition values addressed by the subsequent arguments. Each argument is the address of a longword containing a condition value.

LIB$MATCH_COND is provided for programmers who want to match a list of one or more condition values. It is designed to be used in multipath branch statements available in most higher-level languages.

LIB$MATCH_COND compares the portion (STS$V_COND_ID) of the condition value referenced by the first argument to the same portion of the condition value referenced by the second through Nth arguments. If the facility-specific bit (STS$V_FAC_SP = bit 15) is clear in match-condition-value (meaning that the condition value is systemwide rather than facility specific), the facility code field
(STS$V_FAC_NO = bits 27:17) is ignored and only the STS$V_MSG_ID fields (bits 15:3) are compared.

The routine returns a 0 if a match is not found, a 1 if the condition value matches the first condition value in the list (the second argument), a 2 if it matches the second condition value (the third argument), and so on. LIB$MATCH_COND checks for null argument entries in the argument list.

When LIB$MATCH_COND is called with only two arguments, the possible values for the value returned are true (1) or false (0).

Each condition handler must examine the signal argument vector to determine which condition is being signaled. If the condition is not one that the handler knows about, the handler should resignal. A handler should not assume that only one kind of condition can occur in the routine which established it or in any routines it calls. However, because a condition value may be modified by an intervening handler, each handler should only compare that part of the condition value that distinguishes it from another.

**Condition Values Returned**

None.

**Example**

```c
C+  C This Fortran program demonstrates the use of
C LIB$MATCH_COND.
C Declare handler routine as external.
C-
EXTERNAL HANDLER
C+  C Declare the handler that will be used.
C-
TYPE *, 'Establishing handler...' CALL LIB$ESTABLISH ( HANDLER )
OPEN ( UNIT = 1, NAME = 'MATCH.DAT', STATUS = 'OLD')

C+  C Revert to normal error processing.
C-
CALL LIB$REVERT
CLOSE ( UNIT = 1 ) CALL EXIT
END

C+  C This is the handler routine.
C-
INTEGER*4 FUNCTION HANDLER ( SIGARGS, MECHARGS )
INTEGER*4 SIGARGS(*), STATUS INCLUDE '(SSDEF)' INCLUDE '(FORDEF)' INCLUDE '(CHFDEF)' RECORD /CHFDEF2/ MECHARGS HANDLER = SS$CONTINUE
C+  C This handler will type out an error message. In this case the
C message is regarding a file open status.
C-
TYPE *, 'Entering handler...' STATUS = LIB$MATCH_COND ( SIGARGS (2) , FOR$_FILNOTFOU,
1 FOR$_NO_SUCDEV, FOR$_FILNAMSPE, FOR$_OPEFAI )
```
This Fortran program uses a computed GOTO to alter the program execution sequence on a condition value.

If the file called MATCH.DAT does not exist, the following output is returned:

- Establishing handler...
- Entering handler...
- ERROR -- File not found
- Returning from handler...

If the file MATCH.DAT does exist, the output returned is as follows:

- Establishing handler...
LIB$MOVC3
Move Characters

The Move Characters routine makes the VAX MOVC3 instruction available as a callable routine. ¹ The source item is moved to the destination item. Overlap of the source and destination items does not affect the result.

Format

LIB$MOVC3 word-integer-length ,source ,destination

Returns

None.

Arguments

word-integer-length
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

Number of bytes to be moved from source to destination by LIB$MOVC3. The word-integer-length argument is the address of an unsigned word that contains this number of bytes. The maximum transfer is 65,535 bytes.

source
OpenVMS usage: unspecified
type: unspecified
access: read only
mechanism: by reference

Item to be moved. The source argument is the address of this item.

destination
OpenVMS usage: unspecified
type: unspecified
access: write only
mechanism: by reference

Item into which source will be moved. The destination argument is the address of this item.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Description

LIB$MOVC3 is useful for moving large blocks of data, such as arrays, when such an operation would otherwise have to be performed by a programmed loop.

For more information, see the VAX Architecture Reference Manual or the Alpha Architecture Reference Manual. See also OTS$MOVE3.

Condition Values Returned

None.
LIB$MOVC5
Move Characters with Fill

The Move Characters with Fill routine makes the VAX MOVC5 instruction available as a callable routine. 1 The source item is moved to the destination item. Overlap of the source and destination items does not affect the result.

Format

LIB$MOVC5  word-integer-source-length ,source [,fill] ,word-integer-destination-length ,destination

Returns

None.

Arguments

word-integer-source-length
OpenVMS usage:  word_unsigned
type:  word (unsigned)
access:  read only
mechanism:  by reference

Number of bytes in the source item. The word-integer-source-length argument is the address of an unsigned word that contains this number of bytes. The maximum length of source is 65,535 bytes.

source
OpenVMS usage:  unspecified
type:  unspecified
access:  read only
mechanism:  by reference

Item to be moved by LIB$MOVC5. The source argument is the address of this item. If word-integer-source-length is zero, indicating that destination is to be entirely filled by the fill character, then source is ignored by LIB$MOVC5.

fill
OpenVMS usage:  byte_signed
type:  byte integer (signed)
access:  read only
mechanism:  by reference

Character used to pad source to the length of destination. The fill argument is the address of a signed byte integer that contains this fill character. If word-integer-destination-length is less than or equal to word-integer-source-length, fill is unused and may be omitted.

1 On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
**LIB$ Routines**

**LIB$MOVC5**

**word-integer-destination-length**
OpenVMS usage: word_unsigned
Type: word (unsigned)
Access: read only
Mechanism: by reference

Length of **destination** in bytes. The **word-integer-destination-length** argument is the address of an unsigned word that contains this number of bytes. The maximum value of **word-integer-destination-length** is 65,535 bytes.

**destination**
OpenVMS usage: unspecified
Type: unspecified
Access: write only
Mechanism: by reference

Item into which **source** will be moved. The **destination** argument is the address of this item.

**Description**

If the destination item is shorter than the source item, the highest-addressed bytes of the source are not moved.

For more information, see the *VAX Architecture Reference Manual*. See also OTS$MOVE5.

**Condition Values Returned**

None.
LIB$ MOVTC
Move Translated Characters

The Move Translated Characters routine moves the source string, character by character, to the destination string after translating each character using the specified translation table. LIB$MOVTC makes the VAX MOVTC instruction available as a callable routine.1

Format

LIB$MOVTC source-string,fill-character,translation-table,destination-string

Returns

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

Arguments

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

Source string to be translated and moved by LIB$MOVTC. The source-string argument is the address of a descriptor pointing to this source string.

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

Fill character used to pad source-string to the length of destination-string. The fill-character argument is the address of a descriptor pointing to a string. The first character of this string is used as the fill character. The length of this string is not checked and fill-character is not translated.

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

Translation table used by LIB$MOVTC. The translation-table argument is the address of a descriptor pointing to the translation table string. The translation table string is assumed to be 256 characters long.

1 On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
You can use any one of the translation tables included in the Description section that follows, supplied by HP, or you can create your own. Translation tables supplied by HP have names in the format LIB$AB_xxx_yyy, which represent the addresses of the 256-byte translation tables and can be accessed as external (string) variables. If a particular language cannot generate descriptors for external strings, then you must create them manually. The example following the Description section shows the creation of a string descriptor for a translation table using VAX BASIC.

**destination-string**

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

Destination string into which LIB$MOVTC writes the translated **source-string**. The **destination-string** argument is the address of a descriptor pointing to this destination string.

**Description**

Each character in the source string is used as an index into the translation table. The byte found is then placed into the destination string. The fill character is used if the destination string is longer than the source string. If the source string is longer than the destination string, the source string is truncated. Overlap of the source and destination strings does not affect execution.

The translation tables used by LIB$MOVTC and LIB$MOVTUC follow. Each table is preceded by explanatory text.
ASCII to EBCDIC Translation Table

- The numbers on the left represent the low-order bits of the ASCII characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the ASCII characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent EBCDIC characters in hexadecimal notation.

Figure lib–6 is the ASCII to EBCDIC translation table.

**Figure lib–6  LIB$AB_ASC_EBC**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 10 40 F0 7C D7 79 97 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 11 4F F1 C1 D8 81 98 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02 12 7F F2 C2 D9 82 99 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>03 13 7B F3 C3 E2 83 A2 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>37 3C 5B F4 C4 E3 84 A3 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2D 3D 6C F5 C5 E4 85 A4 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2E 32 50 F6 C6 E5 86 A5 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2F 26 7D F7 C7 E6 87 A6 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>16 18 4D F8 C8 E7 88 A7 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>05 19 5D F9 C9 E8 89 A8 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>25 3F 5C 7A D1 E9 91 A9 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0B 27 4E 5E D2 4A 92 C0 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0C 1C 6B 4C D3 E0 93 6A 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0D 1D 60 7E D4 5A 94 D0 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0E 1E 4B 6E D5 5F 95 A1 3F 3F 3F 3F 3F 3F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0F 1F 61 6F D6 6D 96 07 3F 3F 3F 3F 3F FF</td>
<td></td>
</tr>
</tbody>
</table>
ASCII to EBCDIC Reversible Translation Table

- The numbers on the left represent the low-order bits of the ASCII characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the ASCII characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent EBCDIC characters in hexadecimal notation.

Figure lib–7 is the ASCII to EBCDIC reversible translation table.

**Figure lib–7 LIB$AB_ASC_EBC_REV**

<table>
<thead>
<tr>
<th>Row Bits 0 − 3</th>
<th>Column</th>
<th>Bits 4 − 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 10 40 F0 7C D7 79 97 20 30 41 58 76 9F B8 DC</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 11 4F F1 C1 D8 81 98 21 31 42 59 77 A0 B9 DD</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02 12 7F F2 C2 D9 82 99 22 32 43 5A 78 AA BA DE</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>03 13 7B F3 C3 E2 83 A2 23 33 44 5D 7C AA BA DE</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>37 3C 5B F4 C4 E8 84 A3 24 34 45 56 7D AF BA DE</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>32 33 5A F6 C6 E5 86 A5 36 47 56 7C AA BA DE</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2F 26 7D F7 C7 E6 87 A6 17 08 48 67 8D AF BF ED</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16 18 4D F8 C8 E7 88 A7 28 38 49 68 8E B0 CA EE</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>05 19 5D F9 C9 E8 89 A8 29 39 51 69 8F B1 CB EF</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>25 3F 5C 7A D1 E9 91 A9 2A 3A 52 70 90 B2 CC FA</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0B 27 4E 5E D2 4A 92 C0 2B 3B 53 71 9A B3 CD FB</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0C 1C 6B 4C D3 E0 93 6A 2C 04 54 72 9B B4 CE FC</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0D 1D 60 7E D4 5A 94 D0 09 14 55 73 9C B5 CF FD</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0E 1E 4B 6E D5 5F 95 A1 0A 3E 56 74 9D B6 DA FE</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0F 1F 61 6F D6 6D 96 07 1B E1 57 75 9E B7 DB FF</td>
<td></td>
</tr>
</tbody>
</table>

ZK–4248–GE
**EBCDIC to ASCII Translation Table**

- The numbers on the left represent the low-order bits of the EBCDIC characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the EBCDIC characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent ASCII characters in hexadecimal notation.

Figure lib–8 is the EBCDIC to ASCII translation table.

**Figure lib–8 LIB$AB_EBC_ASC**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>00 10 5C 5C 20 26 2D 5C 5C 5C 5C 5C 5C 7B 7D 5C 30</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01 11 5C 5C 5C 5C 5C 2F 5C 5C 61 6A 7E 5C 41 4A 5C 31</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>02 12 5C 16 5C 5C 5C 5C 62 6B 73 5C 42 4B 53 32</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>03 13 5C 5C 5C 5C 5C 63 6C 74 5C 43 4C 54 33</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>5C 5C 5C 5C 5C 5C 5C 64 6D 75 5C 44 4D 55 34</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>05 05 0A 5C 5C 5C 5C 5C 65 6E 76 5C 45 4E 56 35</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>5C 08 17 5C 5C 5C 5C 5C 66 6F 77 5C 46 4F 57 36</td>
</tr>
<tr>
<td>7</td>
<td>7F</td>
<td>5C 1B 04 5C 5C 5C 5C 5C 67 70 78 5C 47 50 58 37</td>
</tr>
<tr>
<td>8</td>
<td>5C 18 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5C 19 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>5C 5C 5C 5C 5C 5B 5D 7C 3A 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>5C 5C 5C 5C 2E 24 2C 23 5C 5C 5C 5C 5C 5C 5C 5C</td>
</tr>
<tr>
<td>C</td>
<td>0C 1C 5C 14 3C 2A 25 40 5C 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0D 1D 05 15 28 29 5F 27 5C 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0E 1E 06 5C 2B 3B 3E 3D 5C 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0F 1F 07 1A 21 5E 3F 22 5C 5C 5C 5C 5C 5C 5C 5C</td>
<td></td>
</tr>
</tbody>
</table>

ZK–4249–GE
**EBCDIC to ASCII Reversible Translation Table**

- The numbers on the left represent the low-order bits of the EBCDIC characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the EBCDIC characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent ASCII characters in hexadecimal notation.

Figure lib–9 is the EBCDIC to ASCII reversible translation table.

**Figure lib–9  LIB$AB_EBC_ASC_REV**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 10 80 90 20 26 2D BA C3 CA D1 D8 7B 7D 5C 30</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 11 81 91 A0 A9 2F BB 61 6A 7E D9 41 4A 9F 31</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02 12 82 16 A1 AA B2 BC 62 6B 73 DA 42 4B 53 32</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>03 13 83 93 A2 AB B3 BD 63 6C 74 DB 43 4C 54 33</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9C 9D 84 94 A3 AC B4 BE 64 6D 75 DC 44 4D 55 34</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>09 85 0A 95 A4 AD B5 BF 65 6E 76 DD 45 4E 56 35</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>86 08 17 96 A5 AE B6 C0 66 6F 77 DE 46 4F 57 36</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7F 87 1B 04 A6 AF B7 C1 67 70 78 DF 47 50 58 37</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>97 18 88 98 A7 B0 B8 C2 68 71 79 E0 48 51 59 38</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8D 19 89 99 A8 B1 B9 60 69 72 7A E1 49 52 5A 39</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>8E 92 8A 9A 5B 5D 7C 3A C4 CB D2 E2 E8 EE F4 FA</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>08 8F 8B 9B 2E 24 2C 23 C5 CC D3 E3 E9 EF F5 FB</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0C 1C 8C 14 3C 2A 25 40 C6 CD D4 E4 EA F0 F6 FC</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0D 1D 05 15 28 29 5F 27 C7 CE D5 E5 EB F1 F7 FD</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0E 1E 06 9E 2B 3B 3E 3D C8 CF D6 E6 EC F2 F8 FE</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0F 1F 07 1A 21 5E 3F 22 C9 D0 D7 E7 ED F3 F9 FF</td>
<td></td>
</tr>
</tbody>
</table>

ZK–4250–GE
Packed Decimal to Trailing Overpunch Numeric Value Translation Table

- The numbers on the left represent the low-order bits of the packed decimal values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the packed decimal values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent trailing overpunch numeric values in hexadecimal notation.

Figure lib–10 is the packed decimal to trailing overpunch numeric value translation table.

**Figure lib–10  LIB$AB_CVTPT_O**

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 0 – 3</td>
<td>Bits 4 – 7</td>
</tr>
<tr>
<td>0</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>1</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>2</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>3</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>4</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>5</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>6</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>7</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>8</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>9</td>
<td>7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>A</td>
<td>7B 41 42 43 44 45 46 47 48 49 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>B</td>
<td>7D 4A 4B 4C 4D 4E 4F 50 51 52 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>C</td>
<td>7B 41 42 43 44 45 46 47 48 49 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>D</td>
<td>7D 4A 4B 4C 4D 4E 4F 50 51 52 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>E</td>
<td>7B 41 42 43 44 45 46 47 48 49 7B 7B 7B 7B 7B 7B</td>
</tr>
<tr>
<td>F</td>
<td>7B 41 42 43 44 45 46 47 48 49 7B 7B 7B 7B 7B 7B</td>
</tr>
</tbody>
</table>
Packed Decimal to Unsigned Trailing Numeric Value Translation Table

- The numbers on the left represent the low-order bits of the packed decimal values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the packed decimal values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent unsigned trailing numeric values in hexadecimal notation.

Figure lib–11 is the packed decimal to unsigned trailing numeric value translation table.

**Figure lib–11 LIB$AB_CVTPT_U**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 A B C D E F</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>
| A             | 30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 | ZK–4252–GE
| B             | 30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 |
| C             | 30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 |
| D             | 30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 |
| E             | 30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 |
| F             | 30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 |
Trailing Overpunch Numeric to Packed Decimal Value Translation Table

- The numbers on the left represent the low-order bits of the trailing overpunch numeric values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the trailing overpunch numeric values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent packed decimal values in hexadecimal notation.

Figure lib–12 is the trailing overpunch numeric to packed decimal value translation table.

### Figure lib–12  LIB$AB_CVTTP_O

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 C 0 0 7 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
</tr>
<tr>
<td>1 0 0 0 0 D 1 C 1 C 8 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>2 0 0 0 0 0 2 C 2 C 9 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>3 0 0 0 0 0 3 C 3 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>4 0 0 0 0 0 4 C 4 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>5 0 0 0 0 0 5 C 5 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>6 0 0 0 0 0 6 C 6 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>7 0 0 0 0 0 7 C 7 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>8 0 0 0 0 0 8 C 8 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>9 0 0 0 0 0 9 C 9 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>A 0 0 0 0 0 0 D 1 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>B 0 0 0 0 0 0 2 D 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>C 0 0 0 0 0 0 3 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>D 0 0 0 0 0 0 4 D 0 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>E 0 0 0 0 0 0 5 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>F 0 0 0 0 0 0 6 D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

ZK–4253–GE
Unsigned Numeric to Packed Decimal Value Translation Table

- The numbers on the left represent the low-order bits of the unsigned numeric values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the unsigned numeric values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent packed decimal values in hexadecimal notation.

Figure lib–13 is the unsigned numeric to packed decimal value translation table.

**Figure lib–13 LIB$AB_CVTTT_U**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 00 1C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>00 00 00 2C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>00 00 00 3C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>00 00 00 4C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>00 00 00 5C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>00 00 00 6C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>00 00 00 7C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>00 00 00 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>00 00 00 9C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

ZK–4254–GE
Trailing Overpunch Numeric to Unsigned Numeric Value Translation Table

- The numbers on the left represent the low-order bits of the trailing overpunch numeric values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the trailing overpunch numeric values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent unsigned numeric values in hexadecimal notation.

Figure lib–14 is the trailing overpunch numeric to unsigned numeric value translation table.

**Figure lib–14 LIB$AB_CVT_O_U**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10 20 30 40 37 60 70 80 90 A0 B0 C0 D0 E0 F0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 01 11 30 31 38 61 71 81 91 A1 B1 C1 D1 E1 F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 02 12 22 32 39 62 72 82 92 A2 B2 C2 D2 E2 F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 03 13 23 33 33 63 73 83 93 A3 B3 C3 D3 E3 F3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 04 14 24 34 54 64 74 84 94 A4 B4 C4 D4 E4 F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 05 15 25 35 55 65 75 85 95 A5 B5 C5 D5 E5 F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 06 16 26 36 56 66 76 86 96 A6 B6 C6 D6 E6 F6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 07 17 27 37 57 67 77 87 97 A7 B7 C7 D7 E7 F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 08 18 28 38 58 68 78 88 98 A8 B8 C8 D8 E8 F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 09 19 29 39 59 69 79 89 99 A9 B9 C9 D9 E9 F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A 0A 1A 2A 3A 5A 6A 7A 8A 9A AA BA CA DA EA FA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B 0B 1B 2B 3B 5B 6B 7B 8B 9B AB BB CB DB EB FB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C 0C 1C 2C 3C 5C 6C 7C 8C 9C AC BC CC DC EC FC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 0D 1D 2D 3D 5D 6D 7D 8D 9D AD BD CD DD ED FD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E 0E 1E 2E 3E 5E 6E 7E 8E 9E AE BE CE DE EE FE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F 0F 1F 2F 3F 5F 6F 7F 8F 9F AF BF CF DF EF FF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ZK–4255–GE
**Unsigned Numeric to Trailing Overpunch Translation Table**

Figure lib–15 is indexed by 0 through 9 for the positive overpunches and 10 through 19 for the negative overpunches.

The unsigned binary representation of the least significant digit is moved into R2. Then, if you require a positive result, the following code results:

```assembly
MOVC3   LIB$AB_CVT_U_O[R2], #1,R0
```

If you require a negative result, the following code is generated:

```assembly
MOVC3   LIB$AV_CVT_U_O + 10[R2], #1,R0
```

The result is the overpunch representation for the last byte of the negative number.

Figure lib–15 is the unsigned numeric to trailing overpunch translation table.

**Figure lib–15  LIB$AB_CVT_U_O**

<table>
<thead>
<tr>
<th>0 – 9</th>
<th>10 – 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>7B</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>44</td>
<td>45</td>
</tr>
<tr>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>7D</td>
<td>4A</td>
</tr>
<tr>
<td>4B</td>
<td>4C</td>
</tr>
<tr>
<td>4D</td>
<td>4E</td>
</tr>
<tr>
<td>4F</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
</tr>
</tbody>
</table>

ZK–4256–GE
Packed Decimal to Zone Numeric Translation Table

- The numbers on the left represent the low-order bits of the packed decimal values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the packed decimal values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent zoned numeric values in hexadecimal notation.

Figure lib–16 is the packed decimal to zone numeric translation table.

**Figure lib–16 LIBSAB_CVTPT_Z**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>A</td>
<td>30</td>
</tr>
<tr>
<td>B</td>
<td>70</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
</tr>
<tr>
<td>D</td>
<td>70</td>
</tr>
<tr>
<td>E</td>
<td>30</td>
</tr>
<tr>
<td>F</td>
<td>30</td>
</tr>
</tbody>
</table>
**Zone to Packed Decimal Translation Table**

- The numbers on the left represent the low-order bits of the zoned numeric values in hexadecimal notation.
- The numbers across the top represent the high-order bits of the zoned numeric values in hexadecimal notation.
- The numbers in the body of the table represent the equivalent packed decimal values in hexadecimal notation.

Figure lib–17 is the zone to packed decimal translation table.

**Figure lib–17 LIB$AB_CVTTP_Z**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 00 00 0C 00 00 00 0D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00 00 00 1C 00 00 00 1D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00 00 00 2C 00 00 00 2D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00 00 00 3C 00 00 00 3D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00 00 00 4C 00 00 00 4D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00 00 00 5C 00 00 00 5D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00 00 00 6C 00 00 00 6D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00 00 00 7C 00 00 00 7D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00 00 00 8C 00 00 00 8D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>00 00 00 9C 00 00 00 9D 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

ZK–6415–GE
ASCII Uppercase Translation Table

- The numbers on the left represent the low-order bits of the ASCII characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the ASCII characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent uppercase ASCII characters in hexadecimal notation.

Figure lib–18 is the ASCII uppercase translation table.

Figure lib–18  LIBSAB_UPCASE

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 10 20 30 40 50 60 50 80 90 A0 B0 C0 D0 C0 F0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 11 21 31 41 51 61 81 91 A1 B1 C1 D1 D1 F1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02 12 22 32 42 52 62 82 92 A2 B2 C2 D2 D2 F2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>03 13 23 33 43 53 63 83 93 A3 B3 C3 D3 D3 F3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>04 14 24 34 44 54 64 84 94 A4 B4 C4 D4 D4 F4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>05 15 25 35 45 55 65 85 95 A5 B5 C5 D5 D5 F5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>06 16 26 36 46 56 66 86 96 A6 B6 C6 D6 D6 F6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>07 17 27 37 47 57 67 87 97 A7 B7 C7 D7 D7 F7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>08 18 28 38 48 58 68 88 98 A8 B8 C8 D8 D8 F8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>09 19 29 39 49 59 69 89 99 A9 B9 C9 D9 D9 F9</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0A 1A 2A 3A 4A 5A 6A 8A 9A AA BA CA DA CA DA</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0B 1B 2B 3B 4B 5B 6B 7B 9B AB BB CB DB DB DB</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0C 1C 2C 3C 4C 5C 6C 7C 8C 9C AC BC CC DC DC DC</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0D 1D 2D 3D 4D 5D 6D 7D 8D 9D AD BD CD DD DD DD</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0E 1E 2E 3E 4E 5E 6E 7E 8E 9E AE BE CE DE FE FE</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0F 1F 2F 3F 4F 5F 6F 7F 8F 9F AF BF CF DF FF</td>
<td></td>
</tr>
</tbody>
</table>

ZK-6416-GE
ASCII Lowercase Translation Table

- The numbers on the left represent the low-order bits of the ASCII characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the ASCII characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent lowercase ASCII characters in hexadecimal notation.

Figure lib–19 is the ASCII lowercase translation table.

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column</th>
<th>Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>10 20 30</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11 21 31</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>12 22 32</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>13 23 33</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>14 24 34</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>15 25 35</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>16 26 36</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>17 27 37</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>18 28 38</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>19 29 39</td>
</tr>
<tr>
<td>A</td>
<td>0A</td>
<td>1A 2A 3A</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>1B 2B 3B</td>
</tr>
<tr>
<td>C</td>
<td>0C</td>
<td>1C 2C 3C</td>
</tr>
<tr>
<td>D</td>
<td>0D</td>
<td>1D 2D 3D</td>
</tr>
<tr>
<td>E</td>
<td>0E</td>
<td>1E 2E 3E</td>
</tr>
<tr>
<td>F</td>
<td>0F</td>
<td>1F 2F 3F</td>
</tr>
</tbody>
</table>

Condition Values Returned

SS$_NORMAL Routine successfully completed.

LIB$_STRTRU Routine successfully completed; string truncated. The destination string could not contain all the characters.

LIB$_FATERRLIB Fatal internal error.

LIB$_INSVIRMEM Insufficient virtual memory.

LIB$_INVSTRDES Invalid string descriptor.

Example

```basic
1 !+
!This BASIC program shows the method
!of creating a descriptor for the appropriate
!translation table in order to call LIB$MOVTC.
!-

OPTION TYPE = EXPLICIT
```
Declare the translation table as an EXTERNAL LONG variable.

EXTERNAL LONG LIB$AB_ASC_EBC
EXTERNAL LONG FUNCTION LIB$MOVTCEXTERNAL SUB LIB$STOP
EXTERNAL LONG CONSTANT DSC$K_CLASS_S, DSC$K_DTYPE_T

Define a record which models the required translation table descriptor.

RECORD STR_TYPE
  WORD DSC$W_LENGTH
  BYTE DSC$B_DTYPE
  BYTE DSC$B_CLASS
  LONG DSC$A_POINTER
END RECORD STR_TYPE

DECLARE LONG I, RET_STS
DECLARE STR_TYPE STR_VAR

MAP (FOO) STRING DST = 3%
MAP (FOO) BYTE DST_ARRAY(2)

Fill the translation table descriptor record.
Note that the length of the translation table string is set to 256, and the pointer receives the address of the HP translation table LIB$AB_ASC_EBC.

STR_VAR::DSC$W_LENGTH = 256
STR_VAR::DSC$B_DTYPE = DSC$K_DTYPE_T
STR_VAR::DSC$B_CLASS = DSC$K_CLASS_S
STR_VAR::DSC$A_POINTER = LOC(LIB$AB_ASC_EBC)

RET_STS = LIB$MOVTC( "ABC", ",", STR_VAR BY REF, DST )
IF (RET_STS AND 1%) = 0%
  THEN
    CALL LIB$STOP( RET_STS BY VALUE )
END IF

Add 256 to the translated value in order to return an unsigned value.

PRINT (256 + DST_ARRAY(I)) FOR I = 0% TO 2%

The output generated by this BASIC program is as follows:

193
194
195
LIB$ MOVTUC
Move Translated Until Character

The Move Translated Until Character routine moves the source string, character by character, to the destination string after translating each character using the specified translation table until the stop character is encountered. LIB$MOVTUC makes the VAX MOVTUC instruction available as a callable routine. ¹

Format

LIB$MOVTUC source-string ,stop-character ,translation-table ,destination-string [,fill-character]

Returns

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

The relative position in the source string of the character that is translated to the stop character. Zero is returned if the stop character is not found. This value is set to –1 if destination-string cannot be allocated.

Arguments

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

Source string to be translated and moved by LIB$MOVTUC. The source-string argument is the address of a descriptor pointing to this source string.

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

Stop character that causes LIB$MOVTUC to stop translating the source string. The stop-character argument is the address of a descriptor pointing to a string. The first character of this string is used as the stop character. The length of this string is not checked. During the translation, LIB$MOVTUC accesses each character in the source string and uses it as an index into the translation table. If this translated character is the specified stop character, translation stops, and stop-character is not translated.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
**translation-table**
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Translation table used by LIB$MOVTUC. The *translation-table* argument is the address of a descriptor pointing to the translation table string. The translation table string is assumed to be 256 characters long.

You can use any of the translation tables included in the Description section of LIB$MOVTC, or you can create your own. When using a translation table supplied by HP, the names LIB$AB_xxx_yyy represent the addresses of the 256-byte translation tables, and can be accessed as external (string) variables. If a particular language cannot generate descriptors for external strings, then they must be created manually. The example for the routine LIB$MOVTC shows the creation of a string descriptor for a translation table using VAX BASIC.

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

Destination string into which LIB$MOVTUC writes the translated *source-string*. The *destination-string* argument is the address of a descriptor pointing to this destination string.

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

Character used to pad *source-string* to the length of *destination-string*. The *fill-character* argument is the address of a descriptor pointing to a string. The first character of this string is used as the fill character. The length of this string is not checked and *fill-character* is not translated.

If the fill character is included, the remainder of the destination string (after the stop character) is filled with the specified fill character. If it is not included, the remainder of the destination string remains unchanged.

**Description**

During the translation, LIB$MOVTUC accesses each character in the source string and uses it as an index into the translation table. If the table entry contains the specified stop character, the routine is terminated and the relative position of the source character is returned.

If the source string is longer than the destination string, then the source string is truncated. If the optional fill character is present, any remaining positions in the destination string are filled with the fill character. If the source or destination string is exhausted (before the stop character is found), a zero index is returned.

The results are unpredictable if the source and destination strings overlap and have different starting addresses.
LIB$ Routines
LIB$MOVTUC

See the description of LIB$MOVTC for the translation tables used by LIB$MOVTC and LIB$MOVTUC. Each translation table is preceded by explanatory text.

**Condition Values Returned**

None.
LIB$MULT_DELTA_TIME
Multiply Delta Time by Scalar

The Multiply Delta Time by Scalar routine multiplies a delta time by a longword integer scalar.

Format

LIB$MULT_DELTA_TIME  multiplier , delta-time

Returns

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

Arguments

multiplier
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

The value by which LIB$MULT_DELTA_TIME multiplies the delta time. The multiplier argument is the address of a signed longword containing the integer scalar. If multiplier is negative, the absolute value of multiplier is used.

delta-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: modify
mechanism: by reference

The delta time to be multiplied. The delta-time argument is the address of an unsigned quadword containing the number to be multiplied. The initial delta-time argument must be greater than 0. After LIB$MULT_DELTA_TIME performs the multiplication, the result is returned to delta-time. (The original delta-time value is overwritten.)

Description

LIB$MULT_DELTA_TIME multiplies a delta time by a longword integer scalar. The result of the multiplication is returned to the delta-time argument.

Condition Values Returned

LIB$_NORMAL Routine successfully completed.
LIB$_IVTIME Invalid time.
LIB$_WRONUMARG Incorrect number of arguments.
LIB$ Routines
LIB$MULTF_DELTA_TIME

LIB$MULTF_DELTA_TIME
Multiply Delta Time by an F-Floating Scalar

The Multiply Delta Time by an F-Floating Scalar routine multiplies a delta time by an F-floating scalar.

Format

LIB$MULTF_DELTA_TIME multiplier, delta-time

Returns

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

Arguments

multiplier
OpenVMS usage: floating_point
type: F_floating
access: read only
mechanism: by reference

The value by which LIB$MULTF_DELTA_TIME multiplies the delta time. The multiplier argument is the address of an F-floating value containing the scalar. If multiplier is negative, the absolute value of multiplier is used.

delta-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: modify
mechanism: by reference

The delta time to be multiplied. The delta-time argument is the address of an unsigned quadword containing the number to be multiplied. The initial delta-time argument must be greater than 0. After LIB$MULTF_DELTA_TIME performs the multiplication, the result is returned to delta-time. (The original delta-time value is overwritten.)

Description

LIB$MULTF_DELTA_TIME multiplies a delta time by an F-floating scalar. The result of the multiplication is returned to the delta-time argument.

Condition Values Returned

LIB$_NORMAL Routine successfully completed.
LIB$_IVTIME Invalid time.
LIB$_WRONUMARG Incorrect number of arguments.
LIB$MULTS_DELTA_TIME (Alpha and I64 Only)

Multiply Delta Time by an S-Floating Scalar

The Multiply Delta Time by an IEEE S-Floating Scalar routine multiplies a delta time by an IEEE S-floating scalar.

Format

LIB$MULTS_DELTA_TIME multiplier, delta-time

Returns

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

Arguments

multiplier
OpenVMS usage: floating_point
type: IEEE S_floating
access: read only
mechanism: by reference

The value by which LIB$MULTS_DELTA_TIME multiplies the delta time. The multiplier argument is the address of an IEEE S-floating value containing the scalar. If multiplier is negative, the absolute value of multiplier is used.

delta-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: modify
mechanism: by reference

The delta time to be multiplied. The delta-time argument is the address of an unsigned quadword containing the number to be multiplied. The initial delta-time argument must be greater than 0. After LIB$MULTS_DELTA_TIME performs the multiplication, the result is returned to delta-time. (The original delta-time value is overwritten.)

Description

LIB$MULTS_DELTA_TIME multiplies a delta time by an IEEE S-floating scalar. The result of the multiplication is returned to the delta-time argument.

Condition Values Returned

LIB$_NORMAL Routine successfully completed.
LIB$_IVTIME Invalid time.
LIB$_WRONUMARG Incorrect number of arguments.
The Parse Access Encoded Name String routine parses and translates a string of access names into a mask for a particular ownership category.

Format

```
LIB$PARSE_ACCESS_CODE access-string, [access-names,] ownership-category, access-mask,
[end-position]
```

Returns

- **OpenVMS usage**: cond_value
- **Type**: longword (unsigned)
- **Access**: write only
- **Mechanism**: by value

Arguments

- **access-string**
  - **OpenVMS usage**: char_string
  - **Type**: character-coded text string
  - **Access**: read only
  - **Mechanism**: by descriptor
  
  The address of a character-string descriptor pointing to a string of access names. Each access name is abbreviated to one letter. An example of a valid access string is RWE. Access names are specific to each of the different object classes. See the HP OpenVMS Guide to System Security for a complete list of all valid access names.

- **access-names**
  - **OpenVMS usage**: access_names
  - **Type**: array [0..31] of quadword string descriptor
  - **Access**: read only
  - **Mechanism**: by reference
  
  The address of the access name table for the associated object class. For example, it is the value returned by the LIB$GET_ACCNAM routine in the **accnam** longword. This parameter is optional and defaults to the access name table for the FILE object class.
**ownership-category**
OpenVMS usage: mask_word  
type: word (unsigned)  
access: read only  
mechanism: by reference

The address of a word that indicates the ownership category the access names refer to:

<table>
<thead>
<tr>
<th>Ownership Category</th>
<th>Mask Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>00000000000001111</td>
</tr>
<tr>
<td>Owner</td>
<td>00000000111100000</td>
</tr>
<tr>
<td>Group</td>
<td>00001111000000000</td>
</tr>
<tr>
<td>World</td>
<td>11110000000000000</td>
</tr>
</tbody>
</table>

**access-mask**
OpenVMS usage: mask_word  
type: word (unsigned)  
access: write only  
mechanism: by reference

The address of a word into which this routine writes the access mask. In this mask, a set bit means the access was requested for the specified ownership. Note that this is the opposite of the standard protection format where a set bit means no access.

**end-position**
OpenVMS usage: word_signed  
type: word (signed)  
access: write only  
mechanism: by reference

The number of characters from **access-string** processed by **LIB$PARSE_ACCESS_CODE**. In the case of an error in parsing the access string, the offset to the offending location is returned.

**Description**

**LIB$PARSE_ACCESS_CODE** parses a string of access names and translates the string into a mask for the requested ownership category. The string is a concatenated list of 1-letter abbreviations of access names.

This routine works for any protected object class by specifying the correct access name table. The address of the access name table can be obtained from the **LIB$GET_ACCNAM** routine.

This routine is useful for building a protection mask where the ownership names have already been parsed. Use **LIB$PARSE_SOGW_PROT** for parsing a string containing both ownership and access names.

The mask returned has bits set for the access requested for the specified ownership category. This is opposite the standard protection format where a set bit in the protection mask means no access.
The number of characters processed is optionally returned. This is useful for error processing. The end position will be the offset to the character that made the access category name string invalid.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_IVARG</td>
<td>Required parameter missing or a character in <strong>access-string</strong> did not represent a valid access type.</td>
</tr>
<tr>
<td>LIB$_WRONGNUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>
LIB$PARSE_SOGW_PROT

Parse Protection String

The Parse Protection String routine parses and translates a protection string into a protection mask.

Format

LIB$PARSE_SOGW_PROT protection-string, [access-names], protection-mask, ownership-mask, [end-position]

Returns

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

Arguments

**protection-string**
OpenVMS usage: char_string
type: character-coded text string
access: read only
mechanism: by descriptor

The address of a character-string descriptor pointing to the protection string. The string components are:

- **Ownership name** — System,Owner,Group,World. Ownership names can be specified in full or truncated to any number of characters. Matching is case blind, and spacing is ignored.
- **Access name** — Access names are always abbreviated to one letter. For example, access names for files are R (for read), W (for write), E (for execute), and D (for delete). Any combination can be passed. For example, RWE is a valid combination. A null access name specification means no access.
- **Separators** — Access names are separated from ownership names by either a colon (:) or an equal sign (=). The comma (,) is the list separator. A null access name specification means no access.

An example of a valid protection string is:

SYSTEM=RWED,OWNER:RWED,GROUP,WORLD:R

**access-names**
OpenVMS usage: access_names
type: array [0..31] of quadword string descriptor
access: read only
mechanism: by reference

The address of the access name table for the associated object class. For example, it is the value returned by the LIB$GET_ACCNAM routine in the **accnam** longword. This parameter is optional and defaults to the access name table for the FILE object class.
LIB$ Routines
LIB$PARSE_SOGW_PROT

**protection-mask**
OpenVMS usage: protection
type: word (unsigned)
access: write only
mechanism: by reference

The address of a word into which this routine writes a 16-bit protection mask translation of the protection string. Each bit set in the mask indicates no access for the access type it represents.

**ownership-mask**
OpenVMS usage: mask_word
type: word (unsigned)
access: write only
mechanism: by reference

The address of a word that indicates which ownership names were present in the protection string.

<table>
<thead>
<tr>
<th>Ownership Category</th>
<th>Mask Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>0000000000001111</td>
</tr>
<tr>
<td>Owner</td>
<td>0000000001111000</td>
</tr>
<tr>
<td>Group</td>
<td>0000111100000000</td>
</tr>
<tr>
<td>World</td>
<td>1111000000000000</td>
</tr>
</tbody>
</table>

**end-position**
OpenVMS usage: word_signed
type: word (signed)
access: write only
mechanism: by reference

The number of characters from **protection-string** processed by LIB$PARSE_SOGW_PROT. In the case of an error in parsing the protection string, the offset to the offending location is returned.

**Description**

LIB$PARSE_SOGW_PROT parses a protection string and translates the string into a 16-bit protection mask. LIB$PARSE_SOGW_PROT works for any protected object class by specifying the correct access name table.

The address of the access name table can be obtained from the LIB$GET_ACCNAM routine. Note that file access names are valid for any protected object class.

The number of characters processed is optionally returned. This is useful in error processing. The end position will be the offset to the character that made the protection string invalid. Note that the entire protection string must be valid, or an error is returned.

Several scenarios can cause the protection string to be invalid. The format of the protection string may be invalid, or the access category abbreviations may not be valid with respect to the access name tables.
### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _IVARG</td>
<td>Required parameter missing or invalid protection string.</td>
</tr>
<tr>
<td>LIB$ _WRONGNUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>
LIB$PAUSE

Pause Program Execution

The Pause Program Execution routine suspends program execution and returns control to the calling command level.

Format

LIB$PAUSE

Returns

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

Arguments

None.

Description

LIB$PAUSE suspends program execution and returns control to the calling command level. The suspended image may be continued with the CONTINUE command, or it may be terminated with the EXIT or STOP command. In the latter case, the image will not return to this routine.

Note that this routine functions only for interactive jobs. If this routine is invoked in batch mode, it has no effect.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_NOCLI</td>
<td>No CLI present. The calling process does not have a CLI or the CLI does not support the request. Note that DCL supports this function in INTERACTIVE mode only.</td>
</tr>
</tbody>
</table>
LIB$POLYD
Evaluate Polynomials

The Evaluate Polynomials routine (D-floating values) allows higher-level language users to evaluate D-floating value polynomials.

D-floating values are not supported in full precision in native OpenVMS Alpha and I64 programs. They are precise to 56 bits on VAX systems, 53 or 56 bits in translated VAX images, and 53 bits in native OpenVMS Alpha and I64 programs.

Format

LIB$POLYD polynomial-argument,degree,coefficient,floating-point-result

Returns

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

Arguments

polynomial-argument
OpenVMS usage: floating_point
type: D_floating
access: read only
mechanism: by reference

The address of a D-floating number that is the argument for the polynomial.

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

The address of a signed word integer that is the highest-numbered nonzero coefficient to participate in the evaluation.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

coefficient
OpenVMS usage: floating_point
type: D_floating
access: read only
mechanism: by reference, array reference

The address of an array of D-floating coefficients. The coefficient of the highest-order term of the polynomial is the lowest-addressed element in the array.

floating-point-result
OpenVMS usage: floating_point
type: D_floating
access: write only
mechanism: by reference
The address of a floating-point number that is the result of the calculation. LIB$POLYD writes the address of floating-point-result into a D-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (63 bits for POLYD).

**Description**

LIB$POLYD provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner’s Method. The result is computed as follows:

\[
\text{result} = C[0] + X \cdot (C[1] + X \cdot (C[2] + \ldots X \cdot (C[D]) \ldots))
\]

In the above result, \( D \) is the degree of the polynomial and \( X \) is the argument.

See the VAX Architecture Reference Manual for the detailed description of POLY.

**Condition Values Returned**

- **SS$_NORMAL** Routine successfully completed.
- **SS$_FLTOVF** Floating overflow.
- **SS$_ROPRAND** Reserved operand.

**Example**

The Fortran and Pascal examples provided in the description of LIB$POLYF also demonstrate how to use LIB$POLYD. Please refer to those examples for assistance in using this routine.
LIB$POLYF
Evaluate Polynomials

The Evaluate Polynomials routine (F-floating values) allows higher-level language users to evaluate F-floating polynomials.

Format

LIB$POLYF polynomial-argument,degree,coefficient,floating-point-result

Returns

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

Arguments

polynomial-argument
OpenVMS usage: floating_point
 type: F_floating
 access: read only
 mechanism: by reference

Argument for the polynomial. The polynomial-argument argument is the address of a floating-point number that contains this argument. The polynomial-argument argument is an F-floating number.

degree
OpenVMS usage: word_signed
 type: word (signed)
 access: read only
 mechanism: by reference

Highest-numbered nonzero coefficient to participate in the evaluation. The degree argument is the address of a signed word integer that contains this highest-numbered coefficient.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

coefficient
OpenVMS usage: floating_point
 type: F_floating
 access: read only
 mechanism: by reference, array reference

The address of an array of floating-point coefficients. The coefficient of the highest-order term of the polynomial is the lowest addressed element in the array. The coefficient argument is an array of F-floating numbers.
**floating-point-result**

OpenVMS usage: floating_point

- **type:** F_floating
- **access:** write only
- **mechanism:** by reference

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB$POLYF writes the address of **floating-point-result** into an F-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (31 bits for POLYF).

**Description**

LIB$POLYF provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner's Method. The result is computed as follows:

\[ \text{result} = C[0] + X \cdot (C[1] + X \cdot (C[2] + \ldots + (C[D]) \ldots)) \]

In the above result \( D \) is the degree of the polynomial and \( X \) is the argument.

**Condition Values Returned**

- **SS$_NORMAL** Routine successfully completed.
- **SS$_FLTOVF** Floating overflow.
- **SS$_ROPRAND** Reserved operand.

**Examples**

1. C+
   C This Fortran example demonstrates how to use
   C LIB$POLYF.
   C-
   
   ```fortran
   REAL*4 X,COEFF(5),RESULT
   INTEGER*2 DEG
   
   DATA COEFF/1.0,2.0,-1.0,1.0,-3.0/
   X = 2.0
   DEG = 4
   ! DEG has word length.
   
   RETURN = LIB$POLYF(X,DEG,COEFF,RESULT)
   TYPE *,’(2)^4 + 2*(2^3) -2^2 + 2 - 3 = ’,RESULT
   END
   ```

   This Fortran example demonstrates how to call LIB$POLYF. The output generated by this program is as follows:
2. PROGRAM POLYF(INPUT,OUTPUT);
{
This Pascal program demonstrates how to use
{ LIB$POLYF to evaluate a polynomial.
{-

    TYPE
        WORD = [WORD] 0..65535;
    VAR
        COEFF : ARRAY [0..2] OF REAL := (1.0,2.0,2.0);
        RESULT : REAL;
        RETURNED_STATUS : INTEGER;

    [EXTERNAL] FUNCTION LIB$POLYF(
        ARG   : REAL;
        DEGREE  : WORD;
        VAR RESULT : REAL
    ) : INTEGER; EXTERNAL;

    [EXTERNAL] FUNCTION LIB$STOP(
        Condition_Status : [IMMEDIATE,UNSAFE] UNSIGNED;
        FAO_Args         : [IMMEDIATE,UNSAFE,LIST] UNSIGNED
    ) : INTEGER; EXTERNAL;

BEGIN
{
    Call LIB$POLYF to evaluate 2(X**2) + 2*X + 1.
{-

    RETURNED_STATUS := LIB$POLYF(1.0,2,COEFF,RESULT);
    IF NOT ODD(RETURNED_STATUS)
        THEN
            LIB$STOP(RETURNED_STATUS);
    WRITELN('F(1.0) = ',RESULT:5:2);

END.

This example program demonstrates how to call LIB$POLYF from Pascal.
The output generated by this Pascal program is as follows:

$ RUN POLYF
F(1.0) = 5.00
LIB$ Routines
LIB$POLYG

LIB$POLYG
Evaluate Polynomials

The Evaluate Polynomials routine (G-floating values) allows higher-level language users to evaluate G-floating value polynomials.

Format

LIB$POLYG polynomial-argument, degree, coefficient, floating-point-result

Returns

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

Arguments

polynomial-argument
OpenVMS usage: floating_point
type: G_floating
access: read only
mechanism: by reference

Argument for the polynomial. The polynomial-argument argument is the address of a floating-point number that contains this argument. The polynomial-argument argument is a G-floating number.

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

Highest-numbered nonzero coefficient to participate in the evaluation. The degree argument is the address of a signed word integer that contains this highest-numbered coefficient.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

coefficient
OpenVMS usage: floating_point
type: G_floating
access: read only
mechanism: by reference, array reference

Floating-point coefficients. The coefficient argument is the address of an array of floating-point coefficients. The coefficient of the highest-order term of the polynomial is the lowest addressed element in the array. The coefficient argument is an array of G-floating numbers.
floating-point-result
OpenVMS usage: floating_point
type: G_floating
access: write only
mechanism: by reference

Result of the calculation. The floating-point-result argument is the address of a floating-point number that contains this result. LIB$POLYG writes the address of floating-point-result into a G-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (63 bits for POLYG).

Description
LIB$POLYG provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner’s Method. The result is computed as follows:

\[ \text{result} = C[0] + X \times (C[1] + X \times (C[2] + \ldots X \times (C[D]) \ldots )) \]

In the above result \( D \) is the degree of the polynomial and \( X \) is the argument.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_FLTQVF</td>
<td>Floating overflow.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand.</td>
</tr>
</tbody>
</table>

Example

The Fortran and Pascal examples provided in the description of LIB$POLYF also demonstrate how to use LIB$POLYG. Please refer to those examples for assistance in using this routine.
LIB$POLYH
Evaluate Polynomials (VAX Only)

On OpenVMS VAX systems, the Evaluate Polynomials routine (H-floating values) allows higher-level language users to evaluate H-floating value polynomials.

This routine is not available to native OpenVMS Alpha and I64 programs but is available to translated VAX images.

Format

LIB$POLYH polynomial-argument, degree, coefficient, floating-point-result

Returns

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

Arguments

polynomial-argument
OpenVMS usage: floating_point
type: H_floating
access: read only
mechanism: by reference

Argument for the polynomial. The polynomial-argument argument is the address of a floating-point number that contains this argument. The polynomial-argument argument is an H-floating number.

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

Highest-numbered nonzero coefficient to participate in the evaluation. The degree argument is the address of a signed word integer that contains this highest-numbered coefficient.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

coefficient
OpenVMS usage: floating_point
type: H_floating
access: read only
mechanism: by reference, array reference

Floating-point coefficients. The coefficient argument is the address of an array of floating-point coefficients. The coefficient of the highest-order term of the polynomial is the lowest addressed element in the array. The coefficient argument is an array of H-floating numbers.
**floating-point-result**
OpenVMS usage: floating_point
Type: H_floating
Access: write only
Mechanism: by reference

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB$POLYH writes the address of **floating-point-result** into an H-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (127 bits for POLYH).

**Description**

LIB$POLYH provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner’s Method. The result is computed as follows:

\[
\text{result} = C[0] + X \times (C[1] + X \times (C[2] + \ldots X \times (C[D]) \ldots ))
\]

In the above result \( D \) is the degree of the polynomial and \( X \) is the argument.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$ _FLTOVF</td>
<td>Floating overflow.</td>
</tr>
<tr>
<td>SS$ _ROPRAND</td>
<td>Reserved operand.</td>
</tr>
</tbody>
</table>

**Example**

The Fortran and Pascal examples provided in the description of LIB$POLYF also demonstrate how to use LIB$POLYH. Please refer to those examples for assistance in using this routine.
LIB$ Routines
LIB$POLYS (Alpha and I64 Only)

LIB$POLYS (Alpha and I64 Only)
Evaluate Polynomials

The Evaluate Polynomials routine (IEEE S-floating values) allows higher-level language users to evaluate IEEE S-floating polynomials.

Format

LIB$POLYS polynomial-argument,degree,coefficient,floating-point-result

Returns

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

Arguments

polynomial-argument
OpenVMS usage: floating_point
type: IEEE S_floating
access: read only
mechanism: by reference

Argument for the polynomial. The polynomial-argument argument is the address of a floating-point number that contains this argument. The polynomial-argument argument is an IEEE S-floating number.

degree
OpenVMS usage: word_signed
type: word (signed)
access: read only
mechanism: by reference

Highest-numbered nonzero coefficient to participate in the evaluation. The degree argument is the address of a signed word integer that contains this highest-numbered coefficient.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

coefficient
OpenVMS usage: floating_point
type: IEEE S_floating
access: read only
mechanism: by reference, array reference

The address of an array of floating-point coefficients. The coefficient of the highest-order term of the polynomial is the lowest addressed element in the array. The coefficient argument is an array of IEEE S-floating numbers.
floa**ting-point-result**

OpenVMS usage: floating_point

type: IEEE S_floating

access: write only

mechanism: by reference

Result of the calculation. The floating-point-result argument is the address of a floating-point number that contains this result. LIB$POLYS writes the address of floating-point-result into an IEEE S-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (31 bits for POLYS).

**Description**

LIB$POLYS provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner's Method. The result is computed as follows:

\[
\text{result} = C[0] + x(C[1] + x(C[2] + \ldots x(C[D] + \ldots)))
\]

In the above result, \(D\) is the degree of the polynomial and \(x\) is the argument.

**Condition Values Returned**

- **SS$_NORMAL** Routine successfully completed.
- **SS$_FLTOVF** Floating overflow.
- **SS$_ROPRAND** Reserved operand.

**Example**

```c
/*
** This C example demonstrates how to use LIB$POLYS.
*/

#if !(__IEEE_FLOAT)
#error "Compile module with /FLOAT=IEEE_FLOAT"
#endif

#include <stdio.h>
#include <lib$routines.h>

main ()
{
  float x = 2.0;
  float result = 0;
  float coeff[5] = {1.0, 2.0, -1.0, 1.0, -3.0};
  short deg = 4;
  int status;

  status = lib$polys(&x, &deg, &coeff, &result);
  if ((status & 1) != 1) lib$stop(status);

  printf("(2)^4 + 2*(2^3) -2^2 + 2 - 3 = %f (27.000000)\n", result);
}
```

This C example demonstrates how to call LIB$POLYS. The output generated by this program is as follows:

\[(2)^4 + 2*(2^3) -2^2 + 2 - 3 = 27.000000 (27.000000)\]
LIB$ POLYT (Alpha and I64 Only)

Evaluate Polynomials

The Evaluate Polynomials routine (IEEE T-floating values) allows higher-level language users to evaluate IEEE T-floating polynomials.

Format

LIB$POLYT polynomial-argument, degree, coefficient, floating-point-result

Returns

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

Arguments

polynomial-argument
OpenVMS usage: floating_point
type: IEEE T_floating
access: read only
mechanism: by reference

Argument for the polynomial. The polynomial-argument argument is the address of a floating-point number that contains this argument. The polynomial-argument argument is an IEEE T-floating number.

degree
OpenVMS usage: word_signed
type: word (signed)
access: read only
mechanism: by reference

Highest-numbered nonzero coefficient to participate in the evaluation. The degree argument is the address of a signed word integer that contains this highest-numbered coefficient.

If the degree is 0, the result equals C[0]. The range of the degree is 0 to 31.

coefficient
OpenVMS usage: floating_point
type: IEEE T_floating
access: read only
mechanism: by reference, array reference

The address of an array of floating-point coefficients. The coefficient of the highest-order term of the polynomial is the lowest addressed element in the array. The coefficient argument is an array of IEEE T-floating numbers.
**floating-point-result**
OpenVMS usage: floating_point
type: IEEE T_floating
access: write only
mechanism: by reference

Result of the calculation. The **floating-point-result** argument is the address of a floating-point number that contains this result. LIB$POLYT writes the address of **floating-point-result** into an IEEE T-floating number.

Intermediate multiplications are carried out using extended floating-point fractions (31 bits for POLYT).

**Description**

LIB$POLYT provides higher-level language users with the capability of evaluating polynomials.

The evaluation is carried out by Horner’s Method. The result is computed as follows:

\[ \text{result} = C[0] + X \cdot (C[1] + X \cdot (C[2] + \ldots X \cdot (C[D]) \ldots )) \]

In the above result, \(D\) is the degree of the polynomial and \(X\) is the argument.

**Condition Values Returned**

- **SS$_NORMAL**
  Routine successfully completed.
- **SS$_FLTOVF**
  Floating overflow.
- **SS$_ROPRAND**
  Reserved operand.
LIB$ Routines
LIB$PUT_COMMON

LIB$PUT_COMMON
Put String to Common

The Put String to Common routine copies the contents of a string into the common area. The common area is an area of storage that remains defined across multiple image activations in a process. Optionally, LIB$PUT_COMMON returns the actual number of characters copied. The maximum number of characters that can be copied is 252.

Format

LIB$PUT_COMMON source-string [,resultant-length]

Returns

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

Arguments

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

Source string to be copied to the common area by LIB$PUT_COMMON. The source-string argument is the address of a descriptor pointing to this source string.

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

Number of characters copied by LIB$PUT_COMMON to the common area. The resultant-length argument is the address of an unsigned word integer that contains this number of characters. LIB$PUT_COMMON writes this number into the resultant-length argument.

Description

LIB$PUT_COMMON and LIB$GET_COMMON allow programs to copy strings to and from the common area. The programs reading and writing the data in the common area must agree upon its amount and format. The maximum length of the destination string is defined as follows:

\[ \min(256, \text{the length of the data in the common storage area}) - 4 \]

Thus, the maximum length is 252.
In BASIC and Fortran, you can use these routines to allow a USEROPEN routine to pass information back to the routine that called it. A USEROPEN routine cannot write arguments. However, it can call LIB$PUT_COMMON to put information into the common area. The calling program can then use LIB$GET_COMMON to retrieve it.

You can also use these routines to pass information between images run successively, such as chained images run by LIB$RUN_PROGRAM. Since the common area is unique to each process, do not use LIB$GET_COMMON and LIB$PUT_COMMON to share information across processes.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Successfully completed, but the source string was truncated.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$PUT_INVO_REGISTERS (Alpha and I64 Only)

LIB$PUT_INVO_REGISTERS (Alpha and I64 Only)
Put Invocation Registers

The Put Invocation Registers routine modifies specified values in a procedure's invocation context. A procedure's invocation context consists of the values stored in the integer and floating-point registers as well as the program counter and the processor status registers.

LIB$PUT_INVO_REGISTERS updates internal register save areas with the new values. These values are written to the active register set by the time control returns to the procedure associated with the specified invocation handle.

Format

LIB$PUT_INVO_REGISTERS invo_handle, invo_context, invo_mask

Returns

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

Status value. A value of 1 indicates success. When the initial context represents the bottom of the call chain, a value of 0 is returned.

Arguments

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

Handle for the invocation to be updated.

invo_context
OpenVMS usage: invo_context_blk
type: structure
access: read only
mechanism: by reference

Address of an invocation context block that contains the values to be written to the registers.

Each register that is set in the invo_mask parameter is updated using the value found in the corresponding IREG or FREG field of the invocation context block. The program counter and processor status of the given invocation can also be updated in this way. No other fields of the invocation context block are used.
invo_mask
OpenVMS usage: mask_quadword
type: quadword (unsigned)
access: read only
mechanism: by reference

Address of a 64-bit vector, where each bit corresponds to a register field in
the passed invo_context. Bits 0 through 29 correspond to IREG[0] through
IREG[29], bit 30 corresponds to STACK_POINTER and cannot be changed, bit 31
corresponds to PROGRAM_COUNTER, bits 32 through 62 correspond to FREG[0]
through FREG[30], and bit 63 corresponds to PROCESSOR_STATUS.

Description
LIB$PUT_INVO_REGISTERS updates a given procedure invocation context’s
fields with new register contents.

Note
Only the conventional saved registers (R2 through R15) can be modified
reliably in this way. Any modification to scratch registers may be
overwritten by code in intervening procedure invocations. Any attempt
to modify the control register R29 may result in unpredictable program
behavior. The control register R30 cannot be modified. A value of 0 will
be returned if bit 30 is set.

Therefore, an action such as reading the context of a given procedure
invocation and then updating that context in its entirety may not produce
the desired results, whether or not you have made any modifications.

When using this routine, the caller should plan carefully and should
explicitly modify only those register values that need to be modified.

See the HP OpenVMS Calling Standard manual for additional information.

Condition Values Returned

None.
LIB$ Routines
LIB$PUT_OUTPUT

LIB$PUT_OUTPUT
Put Line to SYS$OUTPUT

The Put Line to SYS$OUTPUT routine writes a record to the current controlling output device, specified by SYS$OUTPUT using the OpenVMS RMS $PUT service.

Format

LIB$PUT_OUTPUT message-string

Returns

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

Argument

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

Message string written to the current controlling output device by LIB$PUT_OUTPUT. The message-string argument is the address of a descriptor pointing to this message string. RMS handles all formatting, so the message does not need to include such ASCII formatting instructions as carriage return (CR).

Description

When you log in, OpenVMS operating systems create three files as default I/O control streams for your process:
- SYS$INPUT, your default input device
- SYS$OUTPUT, your default output device
- SYS$COMMAND, the device that supplies the commands to your process

These files remain open until you log out. They are the interface between your interactive input and output or batch commands and the OpenVMS software. Initially, all three are equated with the terminal. However, with the DCL command ASSIGN, you can change these assignments to obtain information from a file or put information into a file. SYS$INPUT and SYS$COMMAND are usually identical, but the input and command streams can be different. For example, during the execution of an indirect command file from an interactive terminal, SYS$COMMAND refers to the terminal and SYS$INPUT refers to the command file.

On the first call to LIB$PUT_OUTPUT, if the output file is not a process-permanent file, LIB$PUT_OUTPUT opens the output file and positions it at the end-of-file mark. If no output file exists on the first call, LIB$PUT_OUTPUT creates a file. The RMS internal stream identifier (ISI) is stored in the routine’s static storage for subsequent calls.
LIB$PUT_OUTPUT uses RMS to format records on output, and RMS records have implied carriage control. That is, a record normally corresponds to a line of text. Therefore, if you want explicit carriage control, instead of implied carriage control, you must supply it yourself within the source string.

LIB$PUT_OUTPUT is the most convenient way for a MACRO or BLISS program to write information to SYS$OUTPUT.

If you have several shareable images that call LIB$PUT_OUTPUT, and if each shareable image includes its own copy of LIB$PUT_OUTPUT, your program could produce multiple output streams and multiple versions of your output file. A single application should reference one copy of LIB$PUT_OUTPUT.

**Condition Values Returned**

**SS$NORMAL** Routine successfully completed.

Any condition values returned by RMS.

**Example**

```
10 !+
! This BASIC program demonstrates how to use
! LIB$PUT_OUTPUT to output a simple message.
!-
MSGSTR$ = 'This is a sample message'
CALL LIB$PUT_OUTPUT(MSGSTR$)
!+
! In this example, the default value of
! SYS$OUTPUT is used. Therefore, the
! output is 'put' to the terminal screen.
!-
90 END
```

This BASIC program shows the use of LIB$PUT_OUTPUT. The output generated by this BASIC example is as follows:

This is a sample message
The Radix Point Symbol routine returns the system’s radix point symbol. This symbol is used inside a digit string to separate the integer part from the fraction part. This routine works by attempting to translate the logical name SYS$RADIX_POINT as a process, group, or system logical name.

**Format**

LIB$RADIX_POINT  radix-point-string [,resultant-length]

**Returns**

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

**Arguments**

**radix-point-string**  
OpenVMS usage: char_string  
type: character string  
access: write only  
mechanism: by descriptor

Radix point string. The **radix-point-string** argument is the address of a descriptor pointing to this radix point string.

**resultant-length**  
OpenVMS usage: word_unsigned  
type: word (unsigned)  
access: write only  
mechanism: by reference

The number of characters written into **radix-point-string**, not counting padding in the case of a fixed-length string. The **resultant-length** argument is the address of an unsigned word that contains this number.

If the **radix-point-string** argument is the address of a fixed-length string descriptor, there may not be enough characters in the fixed-length string to contain the whole radix point string, and the radix point string is truncated. If the radix point string is truncated to the size specified in a fixed-length string descriptor, **resultant-length** is set to this size. Therefore, **resultant-length** can always be used by the calling program to access a valid substring of **radix-point-string**.
Description

If unable to translate the logical name SYS$RADIX_POINT, LIB$RADIX_POINT returns the United States radix point symbol (.). If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS$RADIX_POINT as a systemwide logical name to provide a default for all users, and an individual user with a special need can define SYS$RADIX_POINT as a process logical name to override the default.

LIB$RADIX_POINT is used implicitly by BASIC.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Successfully completed, but the radix point string was truncated.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
</tbody>
</table>
LIB$ Routines  
LIB$REMQHI  

LIB$REMQHI  
Remove Entry from Head of Queue

The Remove Entry from Head of Queue routine removes an entry from the head of the specified self-relative longword interlocked queue. † LIB$REMQHI makes the REMQHI instruction available as a callable routine.

Format

LIB$REMQHI  header, remque-address [,retry-count]

Returns

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

Arguments

header  
OpenVMS usage:  quadword_signed  
type:  quadword integer (signed)  
access:  modify  
mechanism:  by reference

Queue header specifying the queue from which entry will be removed. The header argument contains the address of this signed aligned quadword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

On Alpha and I64 systems, the header argument must contain a 32-bit address. A 64-bit address results in an illegal operand exception.

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

Address of the removed entry. The remque-address argument is the address of an unsigned longword that contains this address. If the queue was empty, remque-address is set to the address of the header.

On Alpha and I64 systems, the remque-address argument must contain a 32-bit address. A 64-bit address results in an illegal operand exception.

retry-count  
OpenVMS usage:  longword_unsinged  
type:  longword (unsigned)  
access:  read only  
mechanism:  by reference

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
The number of times the operation is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of a longword that contains the retry count value. A value of 1 causes no retries. The default value is 10.

**Description**

The queue from which LIB$REMQHI removes an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

**Self-Relative Queues**

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry’s predecessor and successor. If these links are longwords, the queue is referred to as a self-relative longword queue.

You can use the LIB$INSQHI, LIB$INSQTI, LIB$REMQHI, and LIB$REMQTI routines to manage your self-relative longword queue on a VAX, Alpha, or I64 system. These routines implement the INSQHI, INSQTI, REMQHI, and REMQTI instructions that allow you to insert and remove an entry at the head or tail of a self-relative longword queue.

**Synchronization**

When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

**Alignment**

Use of the self-relative longword queue routines requires that the queue header and each of the queue entries be quadword aligned. You can use the Run-Time Library routine LIB$GET_VM on a VAX, Alpha, or I64 system to allocate quadword-aligned virtual memory for a queue.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was removed from the head of the queue, and the resulting queue contains one or more entries.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not quadword aligned, or the header address equals the entry address.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$REMQHI

LIB$ ONEENTQUE
Routine successfully completed. The entry was removed from the head of the queue, and the resulting queue is empty.

LIB$ QUEWASEMP
The queue was empty. The queue is not modified.

LIB$ SECINTFAI
A secondary interlock failure occurred; the insertion was attempted the number of times specified by *retry-count*. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.
LIB$REMQHIQ (Alpha and I64 Only)
Remove Entry from Head of Queue

The Remove Entry from Head of Queue routine removes an entry from the head of the specified self-relative quadword interlocked queue. LIB$REMQHIQ makes the REMQHIQ instruction available as a callable routine.

Format

LIB$REMQHIQ header,remque-address [,retry-count]

Returns

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

Arguments

header
OpenVMS usage: octaword_signed
type: octaword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue from which entry will be removed. The header argument contains the address of this signed aligned octaword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

remque-address
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference

Address of the removed entry. The remque-address argument is the address of an unsigned quadword that contains this address. If the queue was empty, remque-address is set to the address of the header.

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

The number of times the operation is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of a longword that contains the retry count value. A value of 1 causes no retries. The default value is 10.
LIB$ Routines
LIB$REMQHIQ (Alpha and I64 Only)

Description
The queue from which LIB$REMQHIQ removes an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

Self-Relative Queues
A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry's predecessor and successor. If these links are quadwords, the queue is referred to as a self-relative quadword queue.

You can use the LIB$INSQHIQ, LIB$INSQTIQ, LIB$REMQHIQ, and LIB$REMQTIQ routines to manage your self-relative quadword queue on an Alpha or I64 system. These routines implement the INSQHIQ, INSQTIQ, REMQHIQ, and REMQTIQ instructions that allow you to insert and remove an entry at the head or tail of a self-relative quadword queue.

Synchronization
When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

Alignment
Use of the self-relative quadword queue routines requires that the queue header and each of the queue entries be octaword aligned. You can use the Run-Time Library routine LIB$GET_VM_64 to allocate octaword-aligned virtual memory for a queue.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was removed from the head of the queue, and the resulting queue contains one or more entries.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not octaword aligned, or the header address equals the entry address.</td>
</tr>
<tr>
<td>LIB$_ONEENTQUE</td>
<td>Routine successfully completed. The entry was removed from the head of the queue, and the resulting queue is empty.</td>
</tr>
<tr>
<td>LIB$_QUEWASEMP</td>
<td>The queue was empty. The queue is not modified.</td>
</tr>
</tbody>
</table>
LIB$SECINTFAI

A secondary interlock failure occurred; the insertion was attempted the number of times specified by retry-count. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.
LIB$REMQTI
Remove Entry from Tail of Queue

The Remove Entry from Tail of Queue routine removes an entry from the tail of the specified self-relative longword interlocked queue. † LIB$REMQTI makes the REMQTI instruction available as a callable routine.

Format

LIB$REMQTI header, remque-address [,retry-count]

Returns

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

Arguments

header
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue from which the entry is to be deleted. The header argument contains the address of this signed aligned quadword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

On Alpha and I64 systems, the header argument must contain a 32-bit sign-extended address. An illegal operand exception occurs for any other form of address.

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

Address of the removed entry. The remque-address argument is the address of a longword that contains this address. If the queue was empty, remque-address is set to the address of the header.

On Alpha and I64 systems, the remque-address argument must contain a 32-bit sign-extended address. An illegal operand exception occurs for any other form of address.

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

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
The number of times the operation is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of a longword that is this retry count value. A value of 1 causes no retries. The default value is 10.

Description

The queue from which LIB$REMQTI removes an in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

Self-Relative Queues

A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry’s predecessor and successor. If these links are longwords, the queue is referred to as a self-relative longword queue.

You can use the LIB$INSQHI, LIB$INSQTI, LIB$REMQHI, and LIB$REMQTI routines to manage your self-relative longword queue on a VAX, Alpha, or I64 system. These routines implement the INSQHI, INSQTI, REMQHI, and REMQTI instructions that allow you to insert and remove an entry at the head or tail of a self-relative longword queue.

Synchronization

When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

Alignment

Use of the self-relative longword queue routines requires that the queue header and each of the queue entries be quadword aligned. You can use the Run-Time Library routine LIB$GET_VM on a VAX, Alpha, or I64 system to allocate quadword-aligned virtual memory for a queue.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was removed from the queue tail, and the resulting queue contains one or more entries.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not quadword aligned, or the header address equals the entry address.</td>
</tr>
<tr>
<td>Routines</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LIB$ ONEENTQUE</td>
<td>Routine successfully completed. The entry was removed from the queue tail, and the resulting queue is empty.</td>
</tr>
<tr>
<td>LIB$ QUEWASEMP</td>
<td>Queue was empty. The queue is not modified.</td>
</tr>
<tr>
<td>LIB$ SECINTFAI</td>
<td>A secondary interlock failure occurred; the insertion was attempted the number of times specified by <code>retry-count</code>. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.</td>
</tr>
</tbody>
</table>
LIB$REMQTIQ (Alpha and I64 Only)
Remove Entry from Tail of Queue

The Remove Entry from Tail of Queue routine removes an entry from the tail of the specified self-relative quadword interlocked queue. LIB$REMQTIQ makes the REMQTIQ instruction available as a callable routine.

Format

LIB$REMQTIQ header, remque-address [, retry-count]

Returns

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

Arguments

header
OpenVMS usage: octaword_signed
type: octaword integer (signed)
access: modify
mechanism: by reference

Queue header specifying the queue from which the entry is to be deleted. The header argument contains the address of this signed aligned octaword integer. The header argument must be initialized to zero before first use of the queue; zero means an empty queue.

remque-address
OpenVMS usage: address
type: quadword (unsigned)
access: write only
mechanism: by reference

Address of the removed entry. The remque-address argument is the address of a quadword that contains this address. If the queue was empty, remque-address is set to the address of the header.

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

The number of times the operation is to be retried in case of secondary-interlock failure of the queue instruction in a processor-shared memory application. The retry-count argument is the address of a longword that is this retry count value. A value of 1 causes no retries. The default value is 10.
LIB$ Routines
LIB$REMQTIQ (Alpha and I64 Only)

Description
The queue from which LIB$REMQTIQ removes an entry can be in process-private, processor-private, or processor-shareable memory to implement per-process, per-processor, or across-processor queues.

Self-Relative Queues
A queue is a doubly linked list. A Run-Time Library routine specifies a queue entry by its address.

A self-relative queue is a queue in which the links between entries are the displacements of the current entry’s predecessor and successor. If these links are quadwords, the queue is referred to as a self-relative quadword queue.

You can use the LIB$INSQHIQ, LIB$INSQTIQ, LIB$REMQHIQ, and LIB$REMQTIQ routines to manage your self-relative quadword queue on an Alpha or I64 system. These routines implement the INSQHIQ, INSQTIQ, REMQHIQ, and REMQTIQ instructions that allow you to insert and remove an entry at the head or tail of a self-relative quadword queue.

Synchronization
When you insert or remove a queue entry using the self-relative queue routines, the queue pointers are changed as an atomic operation. This ensures that no other process can interrupt the operation to insert or remove a queue entry of its own.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an AST.

If you do not use the self-relative queue routines to insert or remove a queue entry, you must ensure that the operation cannot be interrupted.

Alignment
Use of the self-relative quadword queue routines requires that the queue header and each of the queue entries be octaword aligned. You can use the Run-Time Library routine LIB$GET_VM_64 to allocate octaword-aligned virtual memory for a queue.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. The entry was removed from the queue tail, and the resulting queue contains one or more entries.</td>
</tr>
<tr>
<td>SS$_ROPRAND</td>
<td>Reserved operand fault. Either the entry or the header is at an address that is not octaword aligned, or the header address equals the entry address.</td>
</tr>
<tr>
<td>LIB$_ONEENTQUE</td>
<td>Routine successfully completed. The entry was removed from the queue tail, and the resulting queue is empty.</td>
</tr>
<tr>
<td>LIB$_QUEWASEMP</td>
<td>Queue was empty. The queue is not modified.</td>
</tr>
</tbody>
</table>
A secondary interlock failure occurred; the insertion was attempted the number of times specified by `retry-count`. This is a severe error. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.
LIB$ Routines
LIB$RENAME_FILE

LIB$RENAME_FILE
Rename One or More Files

The Rename One or More Files routine changes the names of one or more files. The specification of the files to be renamed can include wildcards.

LIB$RENAME_FILE is similar in function to the DCL command RENAME.

Format

[,user-success-procedure] [,user-error-procedure] [,user-confirm-procedure]
[,user-specified-argument] [,old-resultant-name] [,new-resultant-name]
[,file-scan-context]

Returns

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

Arguments

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

File specification of the files to be renamed. The old-filespec argument is the address of a descriptor pointing to the old file specification. The specification may include wildcards, in which case each file that matches the specification will be renamed. If running on Alpha or I64 and flag LIB$M_FIL_LONG_NAMES is set, the string must not contain more characters than specified by NAML$C_MAXRSS, otherwise the string must not contain more than 255 characters. Any string class is supported.

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

File specification for the new file names. The new-filespec argument is the address of a descriptor pointing to the new file specification.

This specification need not be complete; fields omitted or specified by using the wildcard character (*) will be filled in from the existing file's name using the same rules as for the DCL command RENAME. If running on Alpha or I64 and flag LIB$M_FIL_LONG_NAMES is set, the string must not contain more characters than specified by NAML$C_MAXRSS, otherwise the string must not contain more than 255 characters. Any string class is supported.
defualt-filespec
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Default file specification of the files to be renamed. The \texttt{default-filespec} argument is the address of a descriptor pointing to the default file specification.

This is an optional argument; if omitted, the default is the null string. See the \textit{OpenVMS Record Management Services Reference Manual} for information on default file specifications. If running on Alpha or I64 and flag LIB$M_FIL_LONG_NAMES is set, the string must not contain more characters than specified by NAML$C_MAXRSS, otherwise the string must not contain more than 255 characters. Any string class is supported.

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

Related file specification of the files to be renamed. The \texttt{related-filespec} argument is the address of a descriptor pointing to the related file specification.

This is an optional argument; if omitted, the default is the null string. Any string class is supported.

Input file parsing is used. (See the \textit{OpenVMS Record Management Services Reference Manual} for information on related file specifications and input file parsing.)

The related file specification is useful when you are processing lists of file specifications. Unspecified portions of the file specification are inherited from the last file processed. Any string class is supported. This is an optional argument.

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

Longword of flag bits designating optional behavior. The \texttt{flags} argument is the address of an unsigned longword containing the flag bits. This is an optional argument; if omitted, the default is that all flags are clear.

The bit number and its meaning are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIB$M_FIL_CUR_VER</td>
<td>If \texttt{new-filespec} does not specify a version number, this flag controls whether a new version number for the output file is to be assigned. If this bit is set, the current version number of the file is used. If this bit is clear, the file is given a version number 1 higher than any previously existing file of the same file name and file type. This is the default action.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$RENAME_FILE

Bit   Symbol          Description

If a file already exists with the same file name, type and version number, the error RMS$FEX is given. This flag is equivalent to the /NONEW_VERSION qualifier of the DCL command RENAME.

1  LIB$M_FIL_INH_SECUR Controls whether the renamed file takes on security attributes of the new location or keeps its existing security attributes. If this bit is clear, the attributes of the renamed file are inherited from the next lower version of the new file name, if any, the new parent directory, or both.

If this bit is clear, the file's security attributes are not changed; this is the default action.

For more information on file security, see the HP OpenVMS Guide to System Security. This flag is equivalent to the /INHERIT_SECURITY qualifier of the DCL command RENAME.

2  LIB$M_FIL_LONG_NAMES (Alpha and I64 only) Controls whether to accept file specifications greater than 255 characters in length. If this bit is set, LIB$RENAME_FILE can process files specifications with a maximum length of NAML$C_MAXRSS characters.

If this bit is clear, LIB$RENAME_FILE can process files names with a maximum length of 255 characters.

user-success-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied success routine that LIB$RENAME_FILE calls after each successful rename.

For further information on the success routine, see Call Format for a Success Routine in the Description section.

user-error-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied error routine that LIB$RENAME_FILE calls when it detects an error. The value returned by the error routine determines whether LIB$RENAME_FILE processes more files. For further information on the error routine, see Call Format for an Error Routine in the Description section.

user-confirm-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value
User-supplied confirm routine that LIB$RENAME_FILE calls before it renames a file. The value returned by the confirm routine determines whether or not LIB$RENAME_FILE renames the file.

The confirm routine can be used to select specific files for renaming based on criteria such as expiration date, size, and so on.

For further information on the confirm routine, see Call Format for a Confirm Routine in the Description section.

**user-specified-argument**

OpenVMS usage: user_arg

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

Value that LIB$RENAME_FILE passes to the success, error, and confirm routines each time they are called. Whatever mechanism is used to pass `user-specified-argument` to LIB$RENAME_FILE is also used to pass it to the user-supplied routines. This is an optional argument; if omitted, zero is passed by value.

**old-resultant-name**

OpenVMS usage: char_string

- **type**: character string
- **access**: write only
- **mechanism**: by descriptor

String into which LIB$RENAME_FILE copies the old resultant file specification of the last file processed. This is an optional argument. If present, it is used to store the file specification passed to the user-supplied routines instead of a default class S, type T string. Any string class is supported.

If you are specifying one or more of the action routine arguments, be sure that the descriptor class used to pass `resultant-name` is the same as the descriptor class required by the action routine. For example, VAX Ada requires a class SB descriptor for string arguments to Ada routines, but will use a class A descriptor by default when calling external routines. Refer to your language manual to determine the proper descriptor class to use.

**new-resultant-name**

OpenVMS usage: char_string

- **type**: character string
- **access**: write only
- **mechanism**: by descriptor

String into which LIB$RENAME_FILE writes the new OpenVMS RMS resultant file specification of the last file processed. The `new-resultant-name` argument is the address of a descriptor pointing to the new name. This is an optional argument. If present, it is used to store the file specification passed to the user-supplied routines instead of a class S, type T string. Any string class is supported.

If you are specifying one or more of the action routine arguments, be sure that the descriptor class used to pass `resultant-name` is the same as the descriptor class required by the action routine. For example, VAX Ada requires a class SB descriptor for string arguments to Ada routines, but will use a class A descriptor by default when calling external routines. Refer to your language manual to determine the proper descriptor class to use.
LIB$ Routines
LIB$RENAME_FILE

file-scan-context
OpenVMS usage: context
type: longword (unsigned)
access: modify
mechanism: by reference

Context for renaming a list of file specifications. The file-scan-context is the address of a longword that contains this context. You must initialize this longword to zero before the first of a series of calls to LIB$RENAME_FILE. LIB$RENAME_FILE uses the file scan context to retain the file context for multiple input files.

LIB$FILE_SCAN uses this context to retain multiple input file related file context. This is an optional argument; it need only be specified if you are using multiple input files, as the DCL command RENAME does. You may deallocate the context allocated by LIB$FILE_SCAN while processing the LIB$RENAME_FILE requests by calling LIB$FILE_SCAN_END after all calls to LIB$RENAME_FILE have been completed. See the description of LIB$FILE_SCAN for a more detailed description of this argument.

Description

This description is divided into three parts:

• Call Format for a Success Routine
• Call Format for an Error Routine
• Call Format for a Confirm Routine

Call Format for a Success Routine
The success routine is optional; it is called only if the user-success-procedure argument is specified in the call to LIB$RENAME_FILE.

The calling format of a success routine is as follows:

user-success-procedure old-filespec,new-filespec [,user-specified-argument]

old-filespec
OpenVMS usage: char_string
type: character string
access: read only
mechanism: descriptor

RMS resultant file specification of the file before it was renamed. If old-resultant-name was specified, it is used to pass the string to the success routine. Otherwise, a class S, type T string is passed. Any string class is supported.

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

RMS resultant file specification of the newly renamed file. If new-resultant-name was specified, it is used to pass the string to the success routine. Otherwise, a class S, type T string is passed. Any string class is supported.
user-specified-argument
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: unspecified

Value of user-specified-argument passed by LIB$RENAME_FILE to the
success routine using the same passing mechanism that was used to pass it
to LIB$RENAME_FILE.

Call Format for an Error Routine
The error routine returns a success/fail value that LIB$RENAME_FILE uses to
determine whether or not more files will be processed if an error is encountered.
The error routine is called only if the user-error-procedure argument was
specified in the call to LIB$RENAME_FILE. If the user-error-procedure
argument was not specified, the default is to continue processing.

The calling format of the error routine is as follows:

user-error-procedure old-filespec ,new-filespec ,rms-sts ,rms-stv ,error-source ,user-specified-argument

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

RMS resultant file specification of the file being renamed when the error occurred.
If old-resultant-name was specified, it is used to pass the string to the error
routine. Otherwise, a class S, type T string is passed. Any string class is
supported.

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

RMS resultant file specification of the new file name being used when the error
occurred. If new-resultant-name was specified, it is used to pass the string to the error
routine. Otherwise, a class S, type T string is passed. Any string class is
supported.

rms-sts
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by reference

Primary condition code (FAB$L_STS) which describes the error that occurred.
The rms-sts argument is the address of an unsigned longword that contains this
primary condition code.

rms-stv
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by reference
Secondary condition code (FAB$L_STV) which describes the error that occurred. The **rms-stv** argument is the address of an unsigned longword that contains this secondary condition code.

**error-source**
- **OpenVMS usage:** longword_signed
- **type:** longword integer (signed)
- **access:** read only
- **mechanism:** by reference

Integer code indicating where the error was found. The **error-source** argument is the address of a longword containing the error source.

The values of **error-source** and their meanings are as follows:

0️⃣ Error searching for **old-filespec**
1️⃣ Error parsing **new-filespec**
2️⃣ Error renaming file

**user-specified-argument**
- **OpenVMS usage:** user_arg
- **type:** longword (unsigned)
- **access:** read only
- **mechanism:** unspecified

Value of **user-specified-argument** that LIB$RENAME_FILE passes to the error routine using the same passing mechanism that was used to pass it to LIB$RENAME_FILE.

If the error routine returns a success status (bit 0 set), then LIB$RENAME_FILE will continue processing files. If the error routine returns a failure status (bit 0 clear), processing ceases immediately and LIB$RENAME_FILE returns with an error status.

If the **user-error-procedure** argument is not specified, LIB$RENAME_FILE will return to its caller the most severe error status encountered while renaming the files. If the error routine is called for an error, the success status LIB$_ERRORROUCAL is returned.

The error routine is not called for errors related to string copying.

**Call Format for a Confirm Routine**
The calling format of a confirm routine is as follows:

```
user-confirm-procedure old-filespec ,new-filespec ,old-fab [,user-specified-argument]
```

**old-filespec**
- **OpenVMS usage:** char_string
- **type:** character string
- **access:** read only
- **mechanism:** by descriptor

RMS resultant file specification of the file about to be renamed. If **old-resultant-name** was specified, it is used to pass the string to the confirm routine. Otherwise, a class S, type T string is passed. Any string class is supported.
**new-filespec**  
OpenVMS usage: char_string  
type: character string  
access: read only  
mechanism: by descriptor  

RMS resultant file specification which the file will be given. If **new-resultant-name** was specified, it is used to pass the string to the confirm routine. Otherwise, a class S, type T string is passed. Any string class is supported.

**old-fab**  
OpenVMS usage: fab  
type: unspecified  
access: read only  
mechanism: by reference  

Address of the RMS FAB that describes the file being renamed. Your program may perform an RMS $OPEN on the FAB to obtain file attributes it needs to determine whether the file should be renamed, but must close the file with $CLOSE before returning to LIB$RENAME_FILE.

(Alpha and I64 only) If the LIB$M_FIL_LONG_NAMES FLAGS is set, the FAB references a NAML block rather than a NAM block. The NAML block supports the use of long file specifications with a maximum length of NAML$C_MAXRSS. See the *OpenVMS Record Management Services Reference Manual* for information on NAML blocks.

**user-specified-argument**  
OpenVMS usage: user_arg  
type: longword (unsigned)  
access: read only  
mechanism: unspecified  

Value of **user-specified-argument** passed by LIB$RENAME_FILE to the confirm routine using the same passing mechanism that was used to pass it to LIB$RENAME_FILE. This is an optional argument.

If the confirm routine returns a success value (bit 0 set), the file is renamed; otherwise, the file is not renamed.

---

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_ERRROUCAL</td>
<td>Success—error routine called. A file error was encountered but the error routine was called to handle the condition.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument. The <strong>flags</strong> argument has one or more undefined bits set.</td>
</tr>
<tr>
<td>LIB$_INVFILSPE</td>
<td>Invalid file specification. On VAX, <strong>Old-filespec</strong>, <strong>new-filespec</strong>, or <strong>default-filespec</strong> contains more than 255 characters. On Alpha and I64, <strong>Old-filespec</strong>, <strong>new-filespec</strong>, or <strong>default-filespec</strong> contains more than NAML$C_MAXRSS characters.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$RENAME_FILE

LIB$_INVSTRDES  Invalid string descriptor. One of the string argument descriptors was not a valid string descriptor.

LIB$_WRONUMARG  Wrong number of arguments. An incorrect number of arguments was passed to LIB$RENAME_FILE.

Any condition value returned by LIB$SCOPY_xxx; truncation errors are ignored.

Any condition value returned by RMS. If the user-error-procedure argument was not specified, this is the most severe of the RMS errors which occurred while renaming the files.
LIB$RESERVE_EF
Reserve Event Flag

The Reserve Event Flag routine allocates a local event flag number specified by `event-flag-number`.

Format

```
LIB$RESERVE_EF  event-flag-number
```

Returns

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

Argument

```
event-flag-number
```

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

Event flag number to be allocated by LIB$RESERVE_EF. The `event-flag-number` argument contains the address of a signed longword integer that is this event flag number.

Description

LIB$RESERVE_EF allocates a specific local event flag. It differs from LIB$GET_EF, which allocates an arbitrary local event flag, which is the recommended procedure. Reserving a specific local event flag is not recommended because another routine may attempt to use the same flag, and the flag will no longer function as expected.

The following table lists the availability of local event flags.

<table>
<thead>
<tr>
<th>Number</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never used by this routine and always available</td>
</tr>
<tr>
<td>1 through 23</td>
<td>Initially reserved; available after being freed by LIB$FREE_EF</td>
</tr>
<tr>
<td>24 through 31</td>
<td>Reserved to OpenVMS</td>
</tr>
<tr>
<td>32 through 63</td>
<td>Initially free</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$RESERVE_EF

**Note**

Beware of running multiple images linked with /NOSYSSHR in the same process and having more than one image make calls to LIB$RESERVE_EF. Each image contains its own copy of the event flag bit array that is designed to be process-wide and synchronize ownership of event flags. Multiple calls to LIB$GET_EF could cause the same event flag to be allocated more than once.

See the *HP OpenVMS Programming Concepts Manual* for more information.

**Condition Values Returned**

- **SS$_NORMAL**  
  Routine successfully completed.

- **LIB$_EF_ALRRES**  
  Event flag already reserved.

- **LIB$_EF_RESSYS**  
  Event flag reserved to system. This occurs if the event flag number is outside the ranges of 1 through 23 and 32 through 63.

**Example**

```pascal
PROGRAM RESERVE_EF(INPUT, OUTPUT);
routine LIB$RESERVE_EF(%REF EVENT_FLAG_NUM : INTEGER); EXTERN;
routine LIB$FREE_EF(%REF EVENT_FLAG_NUM : INTEGER); EXTERN;
VAR
  FLAG_NUM : INTEGER;
BEGIN
  FLAG_NUM := 37;
  LIB$RESERVE_EF(FLAG_NUM);
  WRITELN(FLAG_NUM);
  LIB$FREE_EF(FLAG_NUM);
END.
```

This Pascal program generates the following output:

```
37
```
LIB$RESET_VM_ZONE
Reset Virtual Memory Zone

The Reset Virtual Memory Zone routine frees all blocks of memory that were previously allocated from a zone in the 32-bit virtual address space.†

Format

LIB$RESET_VM_ZONE zone-id

Returns

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

Argument

zone-id
OpenVMS usage: identifier
type: longword (unsigned)
access: read only
mechanism: by reference

Zone identifier. The zone-id is the address of a longword that contains the identifier of a zone created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE.

Description

LIB$RESET_VM_ZONE frees all the blocks of memory that were previously allocated from the zone. The memory becomes available to satisfy further allocation requests for the zone; the memory is not returned to the processwide 32-bit page pool managed by LIB$GET_VM_PAGE. Your program can continue to use the zone after you call LIB$RESET_VM_ZONE.

Resetting a zone is a much more efficient way to reuse storage than individually freeing each allocated object in the zone.

It is the caller’s responsibility to ensure that he or she has “exclusive” access to the zone while the reset operation is being performed. Results are unpredictable if another thread of control attempts to perform any operation on the zone while LIB$RESET_VM_ZONE is in progress.

If you specified deallocation filling when you created the zone, LIB$RESET_VM_ZONE will fill all of the allocated blocks that are freed.

If the zone you are resetting was created using the LIB$CREATE_USER_VM_ZONE routine, then you must have an appropriate action routine for the reset operation. That is, in your call to LIB$CREATE_USER_VM_ZONE, you must have specified a user-reset-procedure.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$RESET_VM_ZONE

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADDLOADR</td>
<td>An invalid zone-id argument.</td>
</tr>
</tbody>
</table>
LIB$RESET_VM_ZONE_64 (Alpha and I64 Only)

Reset Virtual Memory Zone

The Reset Virtual Memory Zone routine frees all blocks of memory that were previously allocated from a zone in the 64-bit virtual address space.

Format

LIB$RESET_VM_ZONE_64 zone-id

Returns

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

Argument

zone-id
OpenVMS usage: identifier
type: quadword (unsigned)
access: read only
mechanism: by reference

Zone identifier. The zone-id is the address of a quadword that contains the identifier of a zone created by a previous call to LIB$CREATE_VM_ZONE_64 or LIB$CREATE_USER_VM_ZONE_64.

Description

LIB$RESET_VM_ZONE_64 frees all the blocks of memory that were previously allocated from the zone. The memory becomes available to satisfy further allocation requests for the zone; the memory is not returned to the processwide 64-bit page pool managed by LIB$GET_VM_PAGE_64. Your program can continue to use the zone after you call LIB$RESET_VM_ZONE_64.

Resetting a zone is a much more efficient way to reuse storage than individually freeing each allocated object in the zone.

It is the caller’s responsibility to ensure that he or she has “exclusive” access to the zone while the reset operation is being performed. Results are unpredictable if another thread of control attempts to perform any operation on the zone while LIB$RESET_VM_ZONE_64 is in progress.

If you specified deallocation filling when you created the zone, LIB$RESET_VM_ZONE_64 will fill all of the allocated blocks that are freed.

If the zone you are resetting was created using the LIB$CREATE_USER_VM_ZONE_64 routine, then you must have an appropriate action routine for the reset operation. That is, in your call to LIB$CREATE_USER_VM_ZONE_64, you must have specified a user-reset-procedure.
LIB$ Routines
LIB$RESET_VM_ZONE_64 (Alpha and I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADBLOADR</td>
<td>An invalid <code>zone-id</code> argument.</td>
</tr>
</tbody>
</table>
LIB$REVERT
Revert to the Handler of the Routine Activator

The Revert to the Handler of the Routine Activator routine deletes the condition handler established by LIB$ESTABLISH by clearing the address pointing to the condition handler from the activated routine's stack frame. †

This routine is not available to native OpenVMS Alpha and I64 programs but is recognized and handled appropriately by most HP high-level language compilers.

Format

LIB$REVERT

Returns

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

Previous contents of SF$A_HANDLER (longword 0) of the caller’s stack frame. This is the address of the condition handler previously in effect. If no condition handler was in effect, zero is returned.

Arguments

None.

Description

LIB$REVERT returns the address that it clears from the calling routine's stack frame. LIB$REVERT is used only if your routine is to establish and then cancel a condition handler for a portion of its execution.

LIB$REVERT is provided primarily for use with languages without built-in error-handling facilities, such as Fortran. Do not use LIB$REVERT from BASIC, COBOL, Pascal, or PL/I. See the documentation for the language you are using for information about how that language handles errors.

In VAX MACRO, you merely use the following instruction rather than calling LIB$REVERT:

```
CLRL (FP) ; set handler address to 0
; in current stack frame
```

Condition Values Returned

None.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$RUN_PROGRAM

LIB$RUN_PROGRAM
Run New Program

The Run New Program routine causes the current program to stop running and begins execution of another program.

Format

LIB$RUN_PROGRAM program-name

Returns

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

Argument

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

File name of the program to be run in place of the current program. The program-name argument contains the address of a descriptor pointing to this file name string.

The maximum length of the file name is 255 characters. The default file type is .EXE.

Description

LIB$RUN_PROGRAM stops execution of the current program and begins execution of another program.

• If successful, control does not return to the calling program. Instead, the $EXIT system service is called, the new program image replaces the old image in the user process, and the command language interpreter (CLI) gives control to the new image.

• If unsuccessful, control returns to the command interpreter.

This routine is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In those cases, the error status LIB$_NOCLI is returned.

LIB$RUN_PROGRAM causes the current image to exit at the point of the call and directs the CLI, if one is present, to start running another program. If LIB$RUN_PROGRAM executes successfully, control passes to the second program; if not, control passes to the CLI. The calling program cannot regain control. This technique is called chaining.

This routine is provided primarily for compatibility with PDP-11 systems, where chaining is used to extend the address space of a system.
This routine may also be useful in an OpenVMS environment where address space is severely limited and large images are not possible. For example, you might use chaining to perform system generation on a small virtual address space, for a large page file.

With LIB$RUN_PROGRAM, the calling program can pass arguments to the next program in the chain only by using the common storage area. One way to do this is for the calling program to call LIB$PUT_COMMON to pass the information into the common storage area. Then the called program calls LIB$GET_COMMON to retrieve the data.

In general, this practice is not recommended. There is no convenient way to specify the order and type of arguments passed into the common storage area; so programs that pass arguments in this way must know about the format of the data before it is passed. When you use common storage, it is very difficult to keep your program modular and AST-reentrant; a method of arbitration must be designated to define which program can modify common storage and when.

Further, LIB$RUN_PROGRAM cannot be used if no command language interpreter is present, as in the case of image subprocesses and detached subprocesses.

If you want control to return to the caller, use LIB$SPAWN instead.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>LIB$_NOCLI</td>
<td>No CLI present to perform function. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.</td>
</tr>
<tr>
<td>LIB$_UNECLIERR</td>
<td>Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL or MCR CLIs, please report the problem to your HP support representative.</td>
</tr>
</tbody>
</table>
LIB$SCANC
Scan for Characters and Return Relative Position

The Scan for Characters and Return Relative Position routine is used to find a specified set of characters in the source string. LIB$SCANC makes the VAX SCANC instruction available as a callable routine. ¹

Format

LIB$SCANC source-string ,table-array ,byte-integer-mask

Returns

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

Relative position in the source string of the character that terminated the operation, or zero if the terminator character is not found. If the source string has a zero length, then a zero is returned.

Arguments

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

Source string used by LIB$SCANC to index into a table. The source-string argument contains the address of a descriptor pointing to this source string.

table-array
OpenVMS usage: vector_mask_byte
type: byte (unsigned)
access: read only
mechanism: by reference, array reference

Table that LIB$SCANC indexes into and performs a logical AND operation with the byte-integer-mask byte. The table-array argument contains the address of an unsigned byte array that is this table.

byte-integer-mask
OpenVMS usage: mask_byte
type: byte (unsigned)
access: read only
mechanism: by reference

Mask on which a logical AND operation is performed with bytes in table-array. The byte-integer-mask argument contains the address of an unsigned byte that is this mask.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Description
LIB$SCANC uses successive bytes of the string specified by source-string to index into a table. The byte selected from the table is the byte on which a logical AND operation is performed with the mask byte. The operation is terminated when the result of the AND operation is equal to 1.

Condition Values Returned
None.
LIB$ SCOPY_DXDX
Copy Source String Passed by Descriptor to Destination

The Copy Source String Passed by Descriptor to Destination routine copies a source string passed by descriptor to a destination string.

Format

LIB$SCOPY_DXDX source-string ,destination-string

Corresponding JSB Entry Point
LIB$SCOPY_DXDX6

Returns

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

Arguments

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

Source string to be copied to the destination string by LIB$SCOPY_DXDX. The source-string argument contains the address of a descriptor pointing to this source string. The descriptor class can be unspecified, fixed-length, decimal string, array, noncontiguous array, varying, or dynamic.

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

Destination string to which the source string is copied. The destination-string argument contains the address of a descriptor pointing to this destination string.
The following actions occur depending on the class of the destination string’s descriptor:

<table>
<thead>
<tr>
<th>Descriptor Class</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S, Z, SD, A, NCA</td>
<td>Copy the source string. If needed, space-fill or truncate on the right.</td>
</tr>
<tr>
<td>D</td>
<td>If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation (if any) and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.</td>
</tr>
<tr>
<td>VS</td>
<td>Copy source string to destination string up to the limit of the descriptor MAXSTRLEN field with no padding. Readjust the current length (CURLEN) field to the actual number of bytes copied.</td>
</tr>
</tbody>
</table>

**Description**

LIB$SCOPY_DXDX returns all condition values as a status; truncation is a qualified success condition value (bit 0 set to 1).

In addition, an equivalent JSB entry point is available, with R0 containing the first argument and R1 containing the second.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. All characters in the input string were copied to the destination string.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. String truncated. The destination string could not contain all of the characters copied from the source string.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
</tbody>
</table>
LIB$SCOPY_R_DX
Copy Source String Passed by Reference to Destination String

The Copy Source String Passed by Reference to Destination String routine copies a source string passed by reference to a destination string, passed by descriptor.

**Format**

```
LIB$SCOPY_R_DX  word-integer-source-length ,source-string ,destination-string
```

**Corresponding JSB Entry Point**

LIB$SCOPY_R_DX6

**Returns**

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

**Arguments**

**word-integer-source-length**

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

Length of the source string in bytes. The `word-integer-source-length` argument is the address of an unsigned word that contains the length of the source string.

**source-string**

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

Source string to be copied to the destination string by LIB$SCOPY_R_DX. The `source-string` argument is the address of this source string.

**destination-string**

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

Destination string to which the source string is copied. The `destination-string` argument contains the address of a descriptor pointing to this destination string.
Description

LIB$SCOPY_R_DX copies a source string, passed by reference, to a destination string, passed by descriptor. It returns the status as a condition value. Truncation is a qualified success; LIB$SCOPY_R_DX sets bit 0 of the condition value to 1.

The actions taken by LIB$SCOPY_R_DX depend on the descriptor class of the destination string. The following table describes these actions for each descriptor class:

<table>
<thead>
<tr>
<th>Descriptor Class</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S, Z, SD, A, NCA</td>
<td>Copy the source string. If needed, space fill or truncate on the right.</td>
</tr>
</tbody>
</table>
| D                | If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor.  
|                  | If the area specified is not large enough, return the previous space allocation, if any, and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor. |
| VS               | Copy source string to destination string up to the limit of the descriptor's MAXSTRLEN field with no padding. Readjust the string's current length (CURLEN) field to the actual number of bytes copied. |

An equivalent JSB entry is available, with R0 being the first argument, R1 the second, and R2 the third. The length argument is passed in bits 15:0 of R0.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed. All characters in the input string were copied to the destination string.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. String truncated. The destination string could not contain all of the characters copied from the source string.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$SCOPY_R_DX_64 (Alpha and I64 Only)

LIB$SCOPY_R_DX_64 (Alpha and I64 Only)
Copy Source String Passed by Reference to Destination String

The Copy Source String Passed by Reference to Destination String routine copies a source string passed by reference to a destination string, passed by descriptor.

Format

LIB$SCOPY_R_DX_64 quad-integer-source-length,source-string,destination-string

Returns

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

Arguments

quad-integer-source-length
OpenVMS usage: quadword_unsigned
type: quadword (unsigned)
access: read only
mechanism: by reference

Length of the source string in bytes. The quad-integer-source-length argument is the address of an unsigned quadword that contains the length of the source string.

source-string
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by reference

Source string to be copied to the destination string by LIB$SCOPY_R_DX_64. The source-string argument is the address of this source string.

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

Destination string to which the source string is copied. The destination-string argument contains the address of a descriptor pointing to this destination string.

Description

LIB$SCOPY_R_DX_64 copies a source string, passed by reference, to a destination string, passed by descriptor. It returns the status as a condition value. Truncation is a qualified success; LIB$SCOPY_R_DX_64 sets bit 0 of the condition value to 1.
The actions taken by LIB$SCOPY_R_DX_64 depend on the descriptor class of the destination string. The following table describes these actions for each descriptor class:

<table>
<thead>
<tr>
<th>Descriptor Class</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S, Z, SD, A, NCA</td>
<td>Copy the source string. If needed, space fill or truncate on the right.</td>
</tr>
<tr>
<td>D</td>
<td>If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation, if any, and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.</td>
</tr>
<tr>
<td>VS</td>
<td>Copy source string to destination string up to the limit of the descriptor’s MAXSTRLEN field with no padding. Readjust the string’s current length (CURLEN) field to the actual number of bytes copied.</td>
</tr>
</tbody>
</table>

**Condition Values Returned**

<table>
<thead>
<tr>
<th>SS$_NORMAL</th>
<th>Routine successfully completed. All characters in the input string were copied to the destination string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. String truncated. The destination string could not contain all of the characters copied from the source string.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
</tbody>
</table>
LIB$ Set Logical Name

The Set Logical Name routine requests the calling process’s command language interpreter (CLI) to define or redefine a supervisor-mode process logical name. It provides the same function as the DCL command DEFINE.

**Format**

```
LIB$SET_LOGICAL logical-name [,value-string] [,table] [,attributes] [,item-list]
```

Either the **item-list** or **value-string** argument must be specified. If both **item-list** and **value-string** are specified, the **value-string** argument is ignored.

**Returns**

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

**Arguments**

**logical-name**
OpenVMS usage: logical_name
type: character string
access: read only
mechanism: by descriptor

Logical name to be defined or redefined. The **logical-name** argument contains the address of a descriptor pointing to this logical name string. The maximum length of a logical name is 255 characters. Note that logical names are case sensitive.

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

Value to be given to the logical name. The **value-string** argument contains the address of a descriptor pointing to this value string. The maximum length of a logical name value is 255 characters.

If omitted, an item list must be present to specify the values of the logical name.

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

Name of the table in which to create the logical name. The **table** argument contains the address of a descriptor pointing to the logical name table. If no table is specified, LNM$PROCESS is used as the default.
attributes
OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by reference

Logical name or translation attributes. The attributes argument is the address of a longword bit mask that contains the logical name or translation attributes.

LNMSM_CONFINE and LNMSM_NO_ALIAS are currently available logical name attributes. See the description of the $CRELNM system service in the HP OpenVMS System Services Reference Manual: A–GETUAI for definitions of LNMSM_CONFINE and LNMSM_NO_ALIAS. If omitted, no special logical name attribute is established.

If no item-list is specified, the translation attributes LNMSM_CONCEALED and LNMSM_TERMINAL may be specified. See the description of the ASSIGN command in the HP OpenVMS DCL Dictionary for definitions of these attributes. If an item-list is specified, it will contain the translation attributes for each equivalence string in the attribute.

description
If the optional table argument is defined, the logical name will be placed in the table specified by the table argument; otherwise, the logical name is placed in the LNMSPROCESS table.

Unlike the system services $CRELOG and $CRELNM, LIB$SET_LOGICAL does not require the caller to be executing in supervisor mode to define a supervisor-mode logical name. Supervisor-mode logical names are not deleted when an image exits. A program can obtain the current value of any logical name by calling the system service $TRNLNM. For more information on logical names, see the HP OpenVMS System Services Reference Manual.

This routine is supported for use with the DCL and MCR CLIs. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In that case, the error status LIB$_NOCLI is returned.
This routine does not support the DCL DEFINE and DEASSIGN commands’ special side-effect of opening and closing a process-permanent file if the logical name SYS$OUTPUT is specified.

See the *HP OpenVMS DCL Dictionary* for a description of the DEFINE command.

### Condition Values Returned

- **SS$_NORMAL**: Routine successfully completed.
- **SS$_ACCVIO**: Access violation. The logical name or its value could not be read.
- **SS$_BADPARAM**: Bad argument.
- **SS$_BUFFEROVF**: Routine successfully completed; however, a buffer overflow occurred.
- **SS$_INSFMEM**: Insufficient dynamic memory.
- **SS$_IVLOGNAM**: Invalid logical name. The logical name or its value contained more than 255 characters.
- **SS$_IVLOGTAB**: Invalid logical name table.
- **SS$_NOPRIV**: No privileges for attempted operation.
- **SS$_SUPERSEDE**: Routine successfully completed; the previous definition of the logical name was replaced.
- **SS$_TOOMANYLNAM**: Logical name translation exceeded allowed depth.
- **LIB$$_INVARG**: Neither the *value-string* nor the *item-list* argument was specified.
- **LIB$$_INVSTRDES**: Invalid string descriptor.
- **LIB$$_NOCLI**: No CLI present to perform function. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.
- **LIB$$_UNECLIERR**: Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL command language interpreter, please report the problem to your HP support representative.

### Example

```
!+ ! Initialize value for logical name MY_LOG
!- SYMBOL$ = 'MY_LOG'
SETVAL$ = 'OFF'
CALL LIB$SET_LOGICAL (SYMBOL$, SETVAL$)
END
```

The BASIC program above sets the logical MY_LOG to OFF. This value can be displayed after the program is run with SHOW LOGICAL as follows:

```
$ SHOW LOGICAL MY_LOG
"MY_LOG" = "OFF" (LN$PROCESS_TABLE)
```
LIB$SET_SYMBOL
Set Value of CLI Symbol

The Set Value of CLI Symbol routine requests the calling process’s command language interpreter (CLI) to define or redefine a CLI symbol.

Format

LIB$SET_SYMBOL symbol ,value-string [,table-type-indicator]

Returns

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

Arguments

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

Name of the symbol to be defined or modified by LIB$SET_SYMBOL. The symbol argument is the address of a descriptor pointing to this symbol string. If you redefine a previously defined CLI symbol, the symbol value is modified to the new value that you provide.

The symbol name is converted to uppercase and trailing blanks are removed before use. The symbol argument must begin with a letter, a digit, a dollar sign ($), a hyphen (-), or an underscore (_). The maximum length of symbol is 255 characters.

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

Value to be given to the symbol. The value-string argument is the address of a descriptor pointing to this value string.

Trailing blanks are not removed from the value string before use. The maximum length of value-string is 1024 characters. Integer values are not allowed; LIB$SET_SYMBOL is intended to set string CLI symbols, not integer CLI symbols.
**LIB$ Routines**

**LIB$SET_SYMBOL**

**table-type-indicator**

OpenVMS usage: longwordSigned

Type: longword integer (signed)

Access: read only

Mechanism: by reference

Indicator of the table that will contain the defined symbol. The **table-type-indicator** argument is the address of a signed longword integer that is this table indicator.

If omitted, the local symbol table is used. The following are possible values for **table-type-indicator**:

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Value</th>
<th>Table Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_CLI_LOCAL_SYM</td>
<td>1</td>
<td>Local symbol table</td>
</tr>
<tr>
<td>LIB$K_CLI_GLOBAL_SYM</td>
<td>2</td>
<td>Global symbol table</td>
</tr>
</tbody>
</table>

**Description**

LIB$SET_SYMBOL requests the calling process’s CLI to define or redefine a CLI symbol.

CLI symbols created using LIB$SET_SYMBOL may be inaccessible by other CLI commands. To avoid this situation, make sure that your symbol names are alphanumeric and that the first character is alphabetic. LIB$SET_SYMBOL is intended to set string CLI symbols, not integer CLI symbols.

LIB$K_CLI_LOCAL_SYM and LIB$K_CLI_GLOBAL_SYM are defined as global symbols and in symbol libraries supplied by HP (macro or module name $LIBCLIDEF).

This routine is supported for use with the DCL CLI. If used with the MCR CLI, the error status LIB$NOCLI will be returned. If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In this case, the error status LIB$NOCLI is returned.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_AMBSYMDEF</td>
<td>Ambiguous symbol definition. The symbol name you want to define is ambiguous when compared with existing symbol names. This condition might arise if abbreviated symbols have been defined previously. See the <em>HP OpenVMS DCL Dictionary</em> for more information on abbreviated symbols.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
</tbody>
</table>
LIB$_INSCLIMEM

Insufficient CLI memory. The CLI could not get enough virtual memory to assign another symbol. This condition may be caused by having too many symbols defined; deleting some symbol definitions may make enough room for the new symbol.

LIB$_INSVIRMEM

Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.

LIB$_INVARG

Invalid argument. The value of table-type-indicator was invalid or the length of value-string was greater than 1024 characters.

LIB$_INVSTRDES

Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.

LIB$_INVSYMNM

Invalid symbol name. The length of symbol was greater than 255 characters or symbol did not begin with a letter.

LIB$_NOCLI

No CLI present to perform function. The calling process did not have a CLI to perform the function or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.

LIB$_UNECLIERR

Unexpected CLI error. The CLI returned an error status which was not recognized. This error may be caused by use of a nonstandard CLI. If this error occurs while using the DCL command language interpreter, please report the problem to your HP support representative.

Example

```basic
!+  
! Initialize value and symbol name  
!-  
SYMBOL$ = 'MY_SYM'
SETVAL$ = 'ON'
CALL LIB$SET_SYMBOL (SYMBOL$, SETVAL$)
END
```

The BASIC program above sets the symbol MY_SYM to ON. This value can be displayed after the program is run with SHOW SYMBOL as follows:

```
$ SHOW SYMBOL MY_SYM
 "MY_SYM" = "ON" (LNM$PROCESS_TABLE)
```
LIB$SFREE1_DD
Free One Dynamic String

The Free One Dynamic String routine returns the dynamically allocated storage for a dynamic string.

Format

LIB$SFREE1_DD descriptor-address

Corresponding JSB Entry Point

LIB$SFREE1_DD6

Returns

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

Argument

descriptor-address
OpenVMS usage: descriptor
type: quadword (unsigned)
access: modify
mechanism: by reference

Dynamic descriptor specifying the area to be deallocated. The descriptor-address argument is the address of an unsigned quadword that is this descriptor. The descriptor is assumed to be dynamic and its class field is not checked.

Description

Before a routine deallocates a dynamic descriptor, it must use LIB$SFREE1_DD or LIB$SFREEN_DD to deallocate the string storage space specified by the dynamic descriptor. Otherwise, string storage is not deallocated and your program can run out of memory.

This routine deallocates the described string space and flags the descriptor as describing no string at all. The descriptor's POINTER and LENGTH fields contain zero (0).

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_FATERRLIB Fatal internal error.
LIB$SFREEN_DD
Free One or More Dynamic Strings

The Free One or More Dynamic Strings routine returns one or more dynamic strings to free storage.

Format

LIB$SFREEN_DD  number-of-descriptors ,first-descriptor-array

Corresponding JSB Entry Point

LIB$SFREEN_DD6

Returns

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

Arguments

number-of-descriptors
OpenVMS usage:  longword_unsigned
type:  longword (unsigned)
access:  read only
mechanism:  by reference

Number of adjacent descriptors freed by LIB$SFREEN_DD. The number-of-descriptors argument contains the address of an unsigned longword that is this number. The deallocated area is returned to free storage.

first-descriptor-array
OpenVMS usage:  descriptor_array
type:  quadword (unsigned)
access:  modify
mechanism:  by reference, array reference

First descriptor of an array of descriptors. The first-descriptor-array argument contains the address of this first descriptor. The descriptors are assumed to be dynamic, and their class fields are not checked.

The descriptor array must contain all 32-bit descriptors or all 64-bit descriptors. They cannot be mixed.

Description

Before a routine that allocates space returns to its caller, it must use LIB$SFREE1_DD or LIB$SFREEN_DD to deallocate the string storage space specified by any descriptors located in the stack. Otherwise, space is not deallocated and your program could run out of virtual memory.

LIB$SFREEN_DD deallocates the described string space and flags each descriptor as describing no string at all by setting the descriptor’s POINTER and LENGTH fields to 0 (zero).
## Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _FATERRLIB</td>
<td>Fatal internal error.</td>
</tr>
</tbody>
</table>
LIB$SGET1_DD
Get One Dynamic String

The Get One Dynamic String routine allocates dynamic virtual memory to the string descriptor you specify.

Format

LIB$SGET1_DD   word-integer-length ,descriptor-part

Corresponding JSB Entry Point

LIB$SGET1_DD_R6

Returns

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

Arguments

word-integer-length
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

Number of bytes of dynamic virtual memory to be allocated by LIB$SGET1_DD. The word-integer-length argument is the address of an unsigned word that contains this number. The amount of storage allocated may be rounded up automatically.

descr-iptor-part
OpenVMS usage: quadword_unsigned
type: quadword (unsigned)
access: write only
mechanism: by reference

Descriptor of the dynamic string to which LIB$SGET1_DD allocates the dynamic virtual memory. The descriptor-part argument contains the address of this descriptor.

The descriptor-part argument must contain the address of a dynamic string descriptor; LIB$SGET1_DD returns an unpredictable result if any other type of descriptor is specified by this argument.

The descriptor CLASS field is not checked but is set to dynamic (2). The LENGTH field is set to word-integer-length, and the POINTER field points to the string area allocated.
LIB$ Routines
LIB$SGET1_DD

Description
LIB$SGET1_DD is similar to LIB$SCOPY_DXDX except that no source string is copied. You can write anything you want in the allocated area.

If descriptor-part already has dynamic memory allocated to it, but the amount allocated is less than word-integer-length, that space is deallocated before LIB$SGET1_DD allocates new space.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
</tbody>
</table>
LIB$SGET1_DD_64 (Alpha and I64 Only)
Get One Dynamic String

The Get One Dynamic String routine allocates dynamic virtual memory to the string descriptor you specify.

Format

LIB$SGET1_DD_64 quad-integer-length ,descriptor-part

Returns

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

Arguments

quad-integer-length
OpenVMS usage: quadword_unsigned
type: quadword (unsigned)
access: read only
mechanism: by reference

Number of bytes of dynamic virtual memory to be allocated by LIB$SGET1_DD_64. The quad-integer-length argument is the address of an unsigned quadword that contains this number. The amount of storage allocated can be rounded up automatically.

descriptor-part
OpenVMS usage: quadword_unsigned
type: quadword (unsigned)
access: write only
mechanism: by reference

Descriptor of the dynamic string to which LIB$SGET1_DD_64 allocates the dynamic virtual memory. The descriptor-part argument contains the address of this descriptor.

The descriptor-part argument must contain the address of a dynamic string descriptor; LIB$SGET1_DD_64 returns an unpredictable result if any other type of descriptor is specified by this argument.

The descriptor CLASS field is not checked but is set to dynamic (2). The LENGTH field is set to quad-integer-length, and the POINTER field points to the string area allocated.

Description

LIB$SGET1_DD_64 is similar to LIB$SCOPY_DXDX except that no source string is copied. You can write anything you want in the allocated area.

If descriptor-part already has dynamic memory allocated to it, but the amount allocated is less than quad-integer-length, that space is deallocated before LIB$SGET1_DD_64 allocates new space.
LIB$ Routines
LIB$SGET1_DD_64 (Alpha and I64 Only)

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
</tbody>
</table>
LIB$SHOW_TIMER
Show Accumulated Times and Counts

The Show Accumulated Times and Counts routine returns times and counts accumulated since the last call to LIB$INIT_TIMER and displays them on SYS$OUTPUT. (LIB$INIT_TIMER must be called prior to invoking this routine.) A user-supplied action routine may change this default behavior.

Format

LIB$SHOW_TIMER [handle-address] [,code] [,user-action-procedure] [,user-argument-value]

Returns

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

Arguments

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

Block of storage containing the value returned by a previous call to LIB$INIT_TIMER. The handle-address argument is the address of an unsigned longword integer containing that value.

• If specified, the pointer must be the same value returned by a previous call to LIB$INIT_TIMER.
• If omitted, LIB$SHOW_TIMER will use a block of memory allocated by LIB$INIT_TIMER.
• If handle-address is omitted and LIB$INIT_TIMER has not been called previously, the error LIB$_INVARG is returned. LIB$INIT_TIMER must be called prior to a call to LIB$SHOW_TIMER.

LIB$SHOW_TIMER assumes that LIB$INIT_TIMER has been previously called, and that the results of that call are stored either in a block pointed to by handle-address, or in the memory allocated by LIB$INIT_TIMER.

Note that the handle-address argument is the same as the context argument used in the LIB$INIT_TIMER call.

code
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

Integer specifying the statistic you want; if it is omitted or zero, all five statistics are returned on one line. The code argument is the address of a signed longword integer containing the statistic code.
The following values are allowed for the **code** argument:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Elapsed time</td>
</tr>
<tr>
<td>2</td>
<td>CPU time</td>
</tr>
<tr>
<td>3</td>
<td>Buffered I/O</td>
</tr>
<tr>
<td>4</td>
<td>Direct I/O</td>
</tr>
<tr>
<td>5</td>
<td>Page faults</td>
</tr>
</tbody>
</table>

**user-action-procedure**

OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied action routine called by LIB$SHOW_TIMER. The default action of LIB$SHOW_TIMER is to write the results to SYS$OUTPUT. An action routine is useful if you want to write the results to a file or, in general, anywhere other than SYS$OUTPUT.

The action routine returns either a success or failure condition value; this status is returned to the calling program as the value of LIB$SHOW_TIMER.

**user-argument-value**

OpenVMS usage: user-arg
type: longword (unsigned) (on VAX systems)
        quadword (unsigned) (on Alpha and I64 systems)
access: read only
mechanism: by value

A value to be passed to the action routine without interpretation. If omitted, LIB$SHOW_TIMER passes a zero by value to the user routine.

**Description**

LIB$SHOW_TIMER returns the times and counts accumulated since the last call to LIB$INIT_TIMER. By default, when neither **code** nor **user-action-procedure** is specified in the call, LIB$SHOW_TIMER writes to SYS$OUTPUT a line giving the following information:

<table>
<thead>
<tr>
<th>Shown on Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELAPSED = dddd hh:mm:ss.cc</td>
<td>Elapsed real time</td>
</tr>
<tr>
<td>CPU = hhhh:mm:ss.cc</td>
<td>Elapsed CPU time</td>
</tr>
<tr>
<td>BUFIO = nnnn</td>
<td>Count of buffered I/O operations</td>
</tr>
<tr>
<td>DIRIO = nnnn</td>
<td>Count of direct I/O operations</td>
</tr>
<tr>
<td>PAGEFAULTS = nnnn</td>
<td>Count of page faults</td>
</tr>
</tbody>
</table>

Any one or all five statistics can be written to SYS$OUTPUT or passed to your user-supplied action routine for other processing.
Call Format for an Action Routine

Action routine is a user-supplied routine called by LIB$SHOW_TIMER. The action routine is used when you want to write results to anywhere other than SYS$OUTPUT. The action routine is called only when you specify the `user-action-procedure` argument in the call to LIB$SHOW_TIMER.

LIB$SHOW_TIMER calls the action routine using this format:

```
user-action-procedure out-str [user-argument-value]
```

**out-str**

OpenVMS usage: char_string

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

Fixed-length string containing the statistics requested. The string is formatted exactly as it would be if written to SYS$OUTPUT. The leading character is blank.

**user-argument-value**

OpenVMS usage: user-arg

- **type:** longword (unsigned) (on VAX systems) quadword (unsigned) (on Alpha and I64 systems)
- **access:** read only
- **mechanism:** by value

A value passed to LIB$SHOW_TIMER. The user argument is passed without interpretation to the action routine.

Condition Values Returned

- SS$NORMAL: Routine successfully completed.
- LIB$_INVARG: Invalid argument. Either code or handle-address was invalid.

Any condition values returned by LIB$PUT_OUTPUT or your action routine.

Example

```pascal
PROGRAM SHOW_TIMER(INPUT,OUTPUT);
{+}
{ This Pascal example demonstrates how to use LIB$SHOW_TIMER. }
{-}

VAR
  RETURNED_STATUS : INTEGER;

[EXTERNAL] FUNCTION LIB$INIT_TIMER(
  HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0
) : INTEGER; EXTERNAL;

[EXTERNAL] FUNCTION LIB$SHOW_TIMER(
  HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0;
  CODE : INTEGER;
    USER_ARG : INTEGER) := %IMMED 0;
  USER_ARG : INTEGER := %IMMED 0
) : INTEGER; EXTERNAL;
```
LIB$ Routines
LIB$SHOW_TIMER

[EXTERNAL] FUNCTION LIB$STOP(
    CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;
    FAO_ARGS : [IMMEDIATE,UNSAFE,LIST] UNSIGNED
) : INTEGER; EXTERNAL;

ROUTINE USER_ACTION_RTN(
    USER_ARG : INTEGER);

BEGIN
    WRITELN('User argument is ',USER_ARG:1);
    WRITELN(OUT_STR);
END;

BEGIN
{+}
{ Call LIB$INIT_TIMER to initialize RTL internal counters.
}{-}
RETURNED_STATUS := LIB$INIT_TIMER;
IF NOT ODD(RETURNED_STATUS)
THEN
    LIB$STOP(RETURNED_STATUS);
{+}
{ Print a line of text to waste time.
}{-}
WRITELN('Spend time to acquire elapsed real time and page faults');
{+}
{ Call LIB$SHOW_TIMER to display counters.
}{-}
RETURNED_STATUS := LIB$SHOW_TIMER(,0,USER_ACTION_RTN,5);
END.

This Pascal program demonstrates how to call LIB$SHOW_TIMER. The output generated by this Pascal example is as follows:

$ RUN SHOW_TIMER
Spend time to acquire elapsed real time and page faults
User argument is 5
    ELAPSED: 0 00:00:00.44  CPU: 0:00:00.04
    BUFIO: 1  DIRIO: 0  FAULTS: 18
LIB$SHOW_VM

Show Virtual Memory Statistics

The Show Virtual Memory Statistics routine returns the statistics accumulated from calls to LIB$GET_VM/LIB$FREE_VM and LIB$GET_VM_PAGE/LIB$FREE_VM_PAGE. †

Format

LIB$SHOW_VM [code] [,user-action-procedure] [,user-specified-argument]

Returns

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

Arguments

code
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Code specifying any one of the statistics to be written to SYS$OUTPUT or passed to an action routine for processing. The code argument is the address of a signed longword integer containing the statistic code. This is an optional argument.

If the statistic code is omitted or is zero, statistics for values 1, 2, and 3 are returned on one line.

The following values are allowed for the code argument:

<table>
<thead>
<tr>
<th>Value</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Statistics for values 1, 2, and 3 are returned.</td>
</tr>
<tr>
<td>1</td>
<td>Number of successful calls to LIB$GET_VM.</td>
</tr>
<tr>
<td>2</td>
<td>Number of successful calls to LIB$FREE_VM.</td>
</tr>
<tr>
<td>3</td>
<td>Number of bytes allocated by LIB$GET_VM but not yet deallocated by LIB$FREE_VM.</td>
</tr>
<tr>
<td>4</td>
<td>Statistics for values 5, 6, and 7 are returned.</td>
</tr>
<tr>
<td>5</td>
<td>Number of calls to LIB$GET_VM_PAGE.</td>
</tr>
<tr>
<td>6</td>
<td>Number of calls to LIB$FREE_VM_PAGE.</td>
</tr>
<tr>
<td>7</td>
<td>Number of VAX pages or Alpha pagelets allocated by LIB$GET_VM_PAGE but not yet deallocated by LIB$FREE_VM_PAGE.</td>
</tr>
</tbody>
</table>

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
**LIB$ Routines**

**LIB$SHOW_VM**

**user-action-procedure**

OpenVMS usage: procedure  
type: procedure value  
access: function call (before return)  
mechanism: by value

User-supplied action routine called by LIB$SHOW_VM. By default, LIB$SHOW_VM returns statistics to SYS$OUTPUT. An action routine is useful when you want to return statistics to a file or, in general, to any place other than SYS$OUTPUT. The routine returns either a success or failure condition value, which will be returned as the value of LIB$SHOW_VM.

For more information on the action routine, see Call Format for an Action Routine in the Description section.

**user-specified-argument**

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

A 32-bit value to be passed directly to the action routine without interpretation. That is, the contents of the argument list entry `user-specified-argument` are copied to the argument list entry for `user-action-procedure`.

**Description**

LIB$SHOW_VM returns the statistics accumulated from calls to LIB$GET_VM/LIB$FREE_VM and LIB$GET_VM_PAGE/LIB$FREE_VM_PAGE. By default, with neither `code` nor `user-action-procedure` specified in the call, LIB$SHOW_VM writes a line giving the following information to SYS$OUTPUT:

```
mmm calls to LIB$GET_VM, nnn calls to LIB$FREE_VM, ppp bytes still allocated
```

Optionally, any one of six statistics can be output to SYS$OUTPUT and/or the line of information can be passed to a user-specified “action routine” for processing different from the default.

**Call Format for an Action Routine**

The action routine is a user-supplied routine that LIB$SHOW_VM calls if you specify the `user-action-procedure` argument in the call to LIB$SHOW_VM.

The call format for an action routine is:

```
user-action-procedure resultant-string ,user-specified-argument
```

**resultant-string**

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

Statistics supplied by LIB$SHOW_VM. The `resultant-string` argument is the address of a descriptor pointing to an address into which LIB$SHOW_VM writes the statistics. The string is formatted exactly as it would be if written to SYS$OUTPUT. The first character is a blank; carriage-return/line-feed combinations are not included.
**user-specified-argument**

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

The 32-bit value passed to LIB$SHOW_VM is passed to the action routine without interpretation. If the **user-specified-argument** argument is omitted in the call to LIB$SHOW_VM, a zero is passed by value to the user routine.

### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$ _INVARG</td>
<td>Invalid arguments. This can be caused by an invalid value for <strong>code</strong>.</td>
</tr>
</tbody>
</table>

Any condition values returned by LIB$PUT_OUTPUT or your action routine.
LIB$ Routines
LIB$SHOW_VM_64 (Alpha and I64 Only)

LIB$SHOW_VM_64 (Alpha and I64 Only)
Show Virtual Memory Statistics

The Show Virtual Memory Statistics routine returns the statistics accumulated from calls to LIB$GET_VM_64/LIB$FREE_VM_64 and LIB$GET_VM_PAGE_64/LIB$FREE_VM_PAGE_64.

Format

LIB$SHOW_VM_64 [code] [,user-action-procedure] [,user-specified-argument]

Returns

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

Arguments

code
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Code specifying any one of the statistics to be written to SYS$OUTPUT or passed to an action routine for processing. The code argument is the address of a signed quadword integer containing the statistic code. This is an optional argument. If the statistic code is omitted or is zero, statistics for values 1, 2, and 3 are returned on one line.

The following values are allowed for the code argument:

<table>
<thead>
<tr>
<th>Value</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Statistics for values 1, 2, and 3 are returned.</td>
</tr>
<tr>
<td>1</td>
<td>Number of successful calls to LIB$GET_VM_64.</td>
</tr>
<tr>
<td>2</td>
<td>Number of successful calls to LIB$FREE_VM_64.</td>
</tr>
<tr>
<td>3</td>
<td>Number of bytes allocated by LIB$GET_VM_64 but not yet deallocated by LIB$FREE_VM_64.</td>
</tr>
<tr>
<td>4</td>
<td>Statistics for values 5, 6, and 7 are returned.</td>
</tr>
<tr>
<td>5</td>
<td>Number of calls to LIB$GET_VM_PAGE_64.</td>
</tr>
<tr>
<td>6</td>
<td>Number of calls to LIB$FREE_VM_PAGE_64.</td>
</tr>
<tr>
<td>7</td>
<td>Number of Alpha or I64 pagelets allocated by LIB$GET_VM_PAGE_64 but not yet deallocated by LIB$FREE_VM_PAGE_64.</td>
</tr>
</tbody>
</table>
user-action-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied action routine called by LIB$SHOW_VM_64. By default, LIB$SHOW_VM_64 returns statistics to SYS$OUTPUT. An action routine is useful when you want to return statistics to a file or, in general, to any place other than SYS$OUTPUT. The routine returns either a success or failure condition value, which will be returned as the value of LIB$SHOW_VM_64.

For more information on the action routine, see Call Format for an Action Routine in the Description section.

user-specified-argument
OpenVMS usage: user_arg
type: quadword (unsigned)
access: read only
mechanism: by value

A 64-bit value to be passed directly to the action routine without interpretation. That is, the contents of the argument list entry user-specified-argument are copied to the argument list entry for user-action-procedure.

Description
LIB$SHOW_VM_64 returns the statistics accumulated from calls to LIB$GET_VM_64/LIB$FREE_VM_64 and LIB$GET_VM_PAGE_64/LIB$FREE_VM_PAGE_64. By default, with neither code nor user-action-procedure specified in the call, LIB$SHOW_VM_64 writes a line giving the following information to SYS$OUTPUT:

mmm calls to LIB$GET_VM_64, nnn calls to LIB$FREE_VM_64, ppp bytes still allocated

Optionally, any one of six statistics can be output to SYS$OUTPUT and/or the line of information can be passed to a user-specified “action routine” for processing different from the default.

Call Format for an Action Routine
The action routine is a user-supplied routine that LIB$SHOW_VM_64 calls if you specify the user-action-procedure argument in the call to LIB$SHOW_VM_64.

The call format for an action routine is:

user-action-procedure resultant-string ,user-specified-argument

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

Statistics supplied by LIB$SHOW_VM_64. The resultant-string argument is the address of a descriptor pointing to an address into which LIB$SHOW_VM_64 writes the statistics. The string is formatted exactly as it would be if written to SYS$OUTPUT. The first character is a blank; carriage-return/line-feed combinations are not included.
LIB$ Routines
LIB$SHOW_VM_64 (Alpha and I64 Only)

user-specified-argument
OpenVMS usage: user_arg
type: quadword (unsigned)
access: read only
mechanism: by value

The 64-bit value passed to LIB$SHOW_VM_64 is passed to the action routine without interpretation. If the user-specified-argument argument is omitted in the call to LIB$SHOW_VM_64, a zero is passed by value to the user routine.

Condition Values Returned

SS$_NORMAL Route successfully completed.
LIB$_INVARG Invalid arguments. This can be caused by an invalid value for code.

Any condition values returned by LIB$PUT_OUTPUT or your action routine.
LIB$SHOW_VM_ZONE
Return Information About a Zone

The Return Information About a Zone routine returns formatted information about a zone in the 32-bit virtual address space, detailing such information as the zone's name, characteristics, and areas, and then passes the information to the specified or default action routine. †

Format

LIB$SHOW_VM_ZONE zone-id [,detail-level] [,user-action-procedure] [,user-arg]

Returns

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

Arguments

zone-id
OpenVMS usage: identifier
type: longword (unsigned)
access: read only
mechanism: by reference

Zone identifier. The zone-id argument is the address of an unsigned longword containing this identifier. Use zero to indicate the 32-bit default zone.

detail-level
OpenVMS usage: longword_signed
type: longword (signed)
access: read only
mechanism: by reference

An identifier code specifying the level of detail required by the user. The detail-level argument is the address of a signed longword containing this code. The default is minimal information. The following are valid values for detail-level:

0  zone-id and name
1  zone-id, name, algorithm, flags, and size information
2  zone-id, name, algorithm, flags, size information, cache information, and area summary
3  zone-id, name, algorithm, flags, size information, cache information, area summary, and queue validation

user-action-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$SHOW_VM_ZONE

Optional user-supplied action routine called by LIB$SHOW_VM_ZONE. By default, LIB$SHOW_VM_ZONE prints statistics to SYS$OUTPUT by means of LIB$PUT_OUTPUT. An action routine is useful when you want to return statistics to a file or, in general, to any location other than SYS$OUTPUT. If user-action-procedure fails, LIB$SHOW_VM_ZONE terminates and returns a failure code. Success codes are ignored.

For more information on the action routine, see the Description section.

**user-arg**
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: by value

Optional 32-bit value to be passed directly to the action routine without interpretation. That is, the contents of the argument list entry user-arg are copied to the argument list entry for user-action-procedure.

**Description**

LIB$SHOW_VM_ZONE returns formatted information about the specified zone and passes it to the action routine. The detail-level argument determines the degree of detail of the zone information returned, and this information is formatted into a readable display and passed to either a user action routine or to LIB$PUT_OUTPUT.

The action routine is a user-supplied routine that LIB$SHOW_VM_ZONE calls if you specify the action-routine argument in the call to LIB$SHOW_VM_ZONE. If you do not specify action-routine, the information is passed to LIB$PUT_OUTPUT for output to SYS$OUTPUT. The call format for an action routine is as follows:

action-routine string, user-arg

**Arguments**

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

Information supplied by LIB$SHOW_VM_ZONE. The string argument is the address of a descriptor pointing to an address into which LIB$SHOW_VM_ZONE writes the requested information. The string is formatted exactly as it would be if written to SYS$OUTPUT.

**user-arg**
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: by value

The 32-bit value passed to LIB$SHOW_VM_ZONE is passed to the action routine without interpretation. If the user-arg argument is omitted in the call to LIB$SHOW_VM_ZONE, a zero is passed by value to the user routine.

If no zone-id is specified (0 is passed), the 32-bit default zone is used.
You must ensure that you have exclusive access to the zone while information is being displayed. Results are unpredictable and may be inconsistent if another thread of control modifies the zone while this routine is displaying data or scanning control blocks.

While scanning the queues and free lists, this routine may detect errors.

If the lookaside list summary discovers a block improperly linked into the list so that the list appears disjointed, the count of the number of blocks of that particular size will be displayed as asterisks.

Table lib–7 lists error and warning messages that can be displayed during the lookaside list and area free list scans. The format is:

**** ERROR -- error description ****
**** WARNING -- warning description ****

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid block size</td>
<td>The size of the block is either not large enough to contain the necessary queue links or is unreasonably large. The size field has been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Block not owned by zone</td>
<td>The current block is not within a section of the virtual address space controlled by this zone. It is possibly attempting to free a block not originally allocated from this zone.</td>
</tr>
<tr>
<td>Block extends past the end of area; truncated</td>
<td>The end of the block is not in the area from which the block has been allocated. The size field may have been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Block extends into “unallocated” block, truncated</td>
<td>The end of the block extends past the allocated section of the area. The size field may have been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Current block not completely accessible</td>
<td>The current block extends into a nonexistent part of the virtual address space. The size field may have been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Back link does not return to previous block</td>
<td>The back link in a doubly linked list does not point to the previous block.</td>
</tr>
<tr>
<td>Forward link does not point to valid address</td>
<td>The forward link of current block points to a location that is not in the virtual address space.</td>
</tr>
<tr>
<td>Free-fill mismatch</td>
<td>One of the locations filled when the block was freed has been modified.</td>
</tr>
</tbody>
</table>

(continued on next page)
Table lib–7 (Cont.) LIB$SHOW_VM_ZONE Error and Warning Messages

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary tag mismatch</td>
<td>One of the boundary tags of the block is not valid.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Warning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward link of current block may not be valid</td>
<td>The back link of the block pointed to by the forward link of the current block does not point to the current block.</td>
</tr>
<tr>
<td>Block at nnnnnnnn is not accessible</td>
<td>The block at location nnnnnnnn could not be accessed and cannot be dumped.</td>
</tr>
<tr>
<td>Block truncated to nnnnnnnn bytes to prevent ACCVIO</td>
<td>The block to be dumped extends into the inaccessible part of the address space. The size of the block is reduced so that the block to be dumped fits within the accessible addresses.</td>
</tr>
</tbody>
</table>

When a block forward link is suspected of pointing to an invalid next block, the information from the next block is replaced by asterisks. The following is a sample error display:

```
**** ERROR -- forward-link does not point to valid address ****
Link Analysis for Current Block:

<table>
<thead>
<tr>
<th>Previous</th>
<th>Current</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0014B270</td>
<td>0014C200</td>
<td>6B6E754A</td>
</tr>
</tbody>
</table>

Block size = 32
Block contents:

00000000 00000000 6B6E754A 00000020 ...Junk........ 00000 0014C200
0014B270 00000008 00000000 00000000 ............p².. 00010 0014C210
```

Condition Values Returned

- SS$_NORMAL: Routine successfully completed.
- LIB$_BADZONE: Invalid zone. Routine was called with a zone-id that does not represent a valid VM zone.
- LIB$_INSVIRMEM: Insufficient virtual memory.
- LIB$_INVARG: Invalid argument.
- LIB$_INVOPEZON: Invalid operation for zone; invalid use of unspecified user zone action routine.
- LIB$_NOTFOU: Could not find another VM zone (alternate success status).
- LIB$_WRONUMARG: Wrong number of arguments.

Any condition value returned by the user-formatted output action routine or LIB$PUT_OUTPUT.
Examples

1. #include <lib$routines.h>

   main()
   {
       long zone_id = 0;
       long detail_level = 1;
       LIB$SHOW_VM_ZONE(&zone_id, &detail_level);
   }

   An example of the output generated by this C program using detail-level 1
   is as follows:

   Zone Id = 7FB96160, Zone name = "DEFAULT_ZONE"
   Algorithm = LIB$K_VM_FIRST_FIT
   Flags = 00000020
      LIB$VM_EXTEND_AREA
   Initial size = 124 pages Current size = 0 pages in 0 areas
   Extend size = 128 pages Page limit = None
   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries
   0 bytes have been freed and not yet reallocated
   72 bytes are used for zone and area control blocks, or 100.0% overhead

2. #include <descrip.h>
   #include <libvmdef.h>
   #include <lib$routines.h>
   #include <stdlib.h>

   main()
   {
       long zone_id;
       long algorithm = LIB$K_VM_QUICK_FIT;
       long algorithm_arg = 16;
       long flags = LIB$M_VM_FREE_FILL0 | LIB$M_VM_EXTEND_AREA;
       long detail_level = 3;
       $DESCRIPTOR(zone_name, "Mix of lookaside list and area blocks");
       int i;
       #define NUM_BLOCKS 250
       char *blocks[NUM_BLOCKS];
       long sizes[NUM_BLOCKS];
       LIB$CREATE_VM_ZONE(&zone_id, &algorithm, &algorithm_arg, &flags,
          0, 0, 0, 0, 0, 0, 7* Omitted arguments */
          &zone_name, 0, 0);
       for (i = 0; i < NUM_BLOCKS; i++)
       {
           sizes[i] = rand() % 300 + 9;
           LIB$GET_VM(&sizes[i], &blocks[i], &zone_id);
       }
       for (i = 0; i < NUM_BLOCKS; i++)
           LIB$FREE_VM(&sizes[i], &blocks[i], &zone_id);
       LIB$SHOW_VM_ZONE(&zone_id, &detail_level);
   }
An example of the output generated by this C program using `detail-level` 3 is as follows:

Zone Id = 00045000, Zone name = "Mix of lookaside list and area blocks"

Algorithm = LIB$K VM QUICK FIT with 16 Lookaside Lists ranging from a minimum blocksize of 8, to a maximum blocksize of 128

Flags = 00000028

LIB$M_VM_FREE_FILL0
LIB$M_VM_EXTEND_AREA

Initial size = 16 pages Current size = 96 pages in 1 area
Extend size = 16 pages Page limit = None

Requests are rounded up to a multiple of 8 bytes, naturally aligned on 8 byte boundaries

41512 bytes have been freed and not yet reallocated
312 bytes are used for zone and area control blocks, or 0.6% overhead

Quick Fit Lookaside List Summary:

<table>
<thead>
<tr>
<th>List number</th>
<th>Block size</th>
<th>Number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>48</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>56</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>72</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>88</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>96</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>104</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>112</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>120</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>10</td>
</tr>
</tbody>
</table>

Area Summary:

<table>
<thead>
<tr>
<th>First address</th>
<th>Last address</th>
<th>Pages assigned</th>
<th>Bytes not yet allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>00045800</td>
<td>000517FF</td>
<td>96</td>
<td>7640</td>
</tr>
</tbody>
</table>

Scanning Lookaside Lists in Zone Control Block
Scanning Free List for Area at 00045800
Number of blocks = 62, Min blocksize = 136, Max blocksize = 3160
LIB$SHOW_VM_ZONE_64 (Alpha and I64 Only)

Return Information About a Zone

The Return Information About a Zone routine returns formatted information about a zone in the 64-bit virtual address space, detailing such information as the zone's name, characteristics, and areas, and then passes the information to the specified or default action routine.

Format

LIB$SHOW_VM_ZONE_64 zone-id [,detail-level] [,user-action-procedure] [,user-arg]

Returns

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

Arguments

zone-id
OpenVMS usage: identifier
type: quadword (unsigned)
access: read only
mechanism: by reference

Zone identifier. The zone-id argument is the address of an unsigned quadword containing this identifier. Use zero to indicate the 64-bit default zone.

detail-level
OpenVMS usage: quadword_signed
type: quadword (signed)
access: read only
mechanism: by reference

An identifier code specifying the level of detail required by the user. The detail-level argument is the address of a signed quadword containing this code. The default is minimal information. The following are valid values for detail-level:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zone-id and name</td>
</tr>
<tr>
<td>1</td>
<td>zone-id, name, algorithm, flags, and size information</td>
</tr>
<tr>
<td>2</td>
<td>zone-id, name, algorithm, flags, size information, cache information, and area summary</td>
</tr>
<tr>
<td>3</td>
<td>zone-id, name, algorithm, flags, size information, cache information, area summary, and queue validation</td>
</tr>
</tbody>
</table>

user-action-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

Optional user-supplied action routine called by LIB$SHOW_VM_ZONE_64. By default, LIB$SHOW_VM_ZONE_64 prints statistics to SYS$OUTPUT by
LIB$ Routines
LIB$SHOW_VM_ZONE_64 (Alpha and I64 Only)

means of LIB$PUT_OUTPUT. An action routine is useful when you want to return statistics to a file or, in general, to any location other than SYS$OUTPUT. If user-action-procedure fails, LIB$SHOW_VM_ZONE_64 terminates and returns a failure code. Success codes are ignored.

For more information on the action routine, see the Description section.

user-arg
OpenVMS usage: user_arg
type: quadword (unsigned)
access: read only
mechanism: by value

Optional 64-bit value to be passed directly to the action routine without interpretation. That is, the contents of the argument list entry user-arg are copied to the argument list entry for user-action-procedure.

Description
LIB$SHOW_VM_ZONE_64 returns formatted information about the specified zone and passes it to the action routine. The detail-level argument determines the degree of detail of the zone information returned, and this information is formatted into a readable display and passed to either a user action routine or to LIB$PUT_OUTPUT.

The action routine is a user-supplied routine that LIB$SHOW_VM_ZONE_64 calls if you specify the action-routine argument in the call to LIB$SHOW_VM_ZONE_64. If you do not specify action-routine, the information is passed to LIB$PUT_OUTPUT for output to SYS$OUTPUT. The call format for an action routine is as follows:

action-routine string, user-arg

Arguments

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

Information supplied by LIB$SHOW_VM_ZONE_64. The string argument is the address of a descriptor pointing to an address into which LIB$SHOW_VM_ZONE_64 writes the requested information. The string is formatted exactly as it would be if written to SYS$OUTPUT.

user-arg
OpenVMS usage: user_arg
type: quadword (unsigned)
access: read only
mechanism: by value

The 64-bit value passed to LIB$SHOW_VM_ZONE_64 is passed to the action routine without interpretation. If the user-arg argument is omitted in the call to LIB$SHOW_VM_ZONE_64, a zero is passed by value to the user routine.

If no zone-id is specified (0 is passed), the 64-bit default zone is used.
LIB$ Routines

LIB$SHOW_VM_ZONE_64 (Alpha and I64 Only)

You must ensure that you have exclusive access to the zone while information is being displayed. Results are unpredictable and may be inconsistent if another thread of control modifies the zone while this routine is displaying data or scanning control blocks.

While scanning the queues and free lists, this routine may detect errors.

If the lookaside list summary discovers a block improperly linked into the list so that the list appears disjointed, the count of the number of blocks of that particular size will be displayed as asterisks.

Table lib–8 lists error and warning messages that may be displayed during the lookaside list and area free list scans. The format is as follows:

**** ERROR -- error description ****
**** WARNING -- warning description ****

Table lib–8 LIB$SHOW_VM_ZONE_64 Error and Warning Messages

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid block size</td>
<td>The size of the block is either not large enough to contain the necessary queue links or is unreasonably large. The size field has been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Block not owned by zone</td>
<td>The current block is not within a section of the virtual address space controlled by this zone. It may be attempting to free a block not originally allocated from this zone.</td>
</tr>
<tr>
<td>Block extends past the end of area; truncated</td>
<td>The end of the block is not in the area from which the block has been allocated. The size field may have been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Block extends into “unallocated” block, truncated</td>
<td>The end of the block extends past the allocated section of the area. The size field may have been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Current block not completely accessible</td>
<td>The current block extends into a nonexistent part of the virtual address space. The size field may have been corrupted. Therefore, the size of the block is reduced so the block to be dumped fits within the area.</td>
</tr>
<tr>
<td>Back link does not return to previous block</td>
<td>The back link in a doubly linked list does not point to the previous block.</td>
</tr>
<tr>
<td>Forward link does not point to valid address</td>
<td>The forward link of current block points to a location that is not in the virtual address space.</td>
</tr>
<tr>
<td>Free-fill mismatch</td>
<td>One of the locations filled when the block was freed has been modified.</td>
</tr>
</tbody>
</table>

(continued on next page)
LIB$ Routines
LIB$SHOW_VM_ZONE_64 (Alpha and I64 Only)

Table lib–8 (Cont.) LIB$SHOW_VM_ZONE_64 Error and Warning Messages

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary tag mismatch</td>
<td>One of the boundary tags of the block is not valid.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Warning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward link of current block may not be valid</td>
<td>The back link of the block pointed to by the forward link of the current block does not point to the current block.</td>
</tr>
<tr>
<td>Block at nnnnnnnn is not accessible</td>
<td>The block at location nnnnnnnn could not be accessed and cannot be dumped.</td>
</tr>
<tr>
<td>Block truncated to nnnnnnnn bytes to prevent ACCVIO</td>
<td>The block to be dumped extends into the inaccessible part of the address space. The size of the block is reduced so that the block to be dumped fits within the accessible addresses.</td>
</tr>
</tbody>
</table>

When a block forward link is suspected of pointing to an invalid next block, the information from the next block is replaced by asterisks. The following is a sample error display:

**** ERROR -- forward-link does not point to valid address ****

<table>
<thead>
<tr>
<th>Link Analysis for Current Block:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Block adr : 00000001C00000050 00000001C0002040 4B4E556A6B6E754A</td>
</tr>
<tr>
<td>Forw link (abs): 00000001C0002040 4B4E556A6B6E754A ********************</td>
</tr>
<tr>
<td>Block size = 64</td>
</tr>
<tr>
<td>Block contents: 4B4E556A 6B6E754A 00000000 00000040 @.......JunkjUNK 0000 00000001C0002040 00000000 00000000 00000000 00010 00000001C0002050</td>
</tr>
</tbody>
</table>

**Condition Values Returned**

<table>
<thead>
<tr>
<th>SS$_NORMAL</th>
<th>Routine successfully completed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_BADZONE</td>
<td>Invalid zone. Routine was called with a <strong>zone-id</strong> that does not represent a valid VM zone.</td>
</tr>
<tr>
<td>LIB$_INVAR</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>LIB$_INVOPEZON</td>
<td>Invalid operation for zone; invalid use of unspecified user zone action routine.</td>
</tr>
<tr>
<td>LIB$_NOTFOU</td>
<td>Could not find another VM zone (alternate success status).</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>

Any condition value returned by the user-formatted output action routine or LIB$PUT_OUTPUT.
Examples

1. #include <lib$routines.h>

main()
{
    __int64 zone_id = 0;
    __int64 detail_level = 1;
    LIB$SHOW_VM_ZONE_64(&zone_id, &detail_level);
}

An example of the output generated by this C program using `detail-level 1` is as follows:

Zone Id = 0000000000020040, Zone name = "DEFAULT_ZONE"
Algorithm = LIB$K_VM_FIRST_FIT
Flags = 00000020
    LIB$M_VM_EXTEND_AREA
Initial size = 124 pages Current size = 0 pages in 0 areas
Extend size = 128 pages Page limit = None
Requests are rounded up to a multiple of 16 bytes,
naturally aligned on 16 byte boundaries
0 bytes have been freed and not yet reallocated
128 bytes are used for zone and area control blocks, or 100.0% overhead

2. #include <descrip.h>
#include <libvmdef.h>
#include <lib$routines.h>
#include <stdlib.h>

#pragma pointer_size(long)

main()
{
    __int64 zone_id;
    __int64 algorithm = LIB$K_VM_QUICK_FIT;
    __int64 algorithm_arg = 16;
    __int64 flags = LIB$M_VM_FREE_FILL0 | LIB$M_VM_EXTEND_AREA;
    __int64 detail_level = 3;
    #DESCRIPTOR(zone_name, "Lookaside list and area blocks");
    int i;
    #define NUM_BLOCKS 250
    char *blocks[NUM_BLOCKS];
    __int64 sizes[NUM_BLOCKS];
    LIB$CREATE_VM_ZONE_64(&zone_id, &algorithm, &algorithm_arg, &flags,
        0, 0, 0, 0, 0, 0, /* Omitted arguments */
        &zone_name, 0, 0);
    for (i = 0; i < NUM_BLOCKS; i++)
    {
        sizes[i] = rand() % 400 + 17;
        LIB$GET_VM_64(&sizes[i], &blocks[i], &zone_id);
    }
    for (i = 0; i < NUM_BLOCKS; i++)
    { LIB$FREE_VM_64(&sizes[i], &blocks[i], &zone_id); }
    LIB$SHOW_VM_ZONE_64(&zone_id, &detail_level); }
LIB$ Routines  
LIB$SHOW_VM_ZONE_64 (Alpha and I64 Only)

An example of the output generated by this C program using detail-level 3 is as follows:

Zone Id = 00000001C0002000, Zone name = "Lookaside list and area blocks"
Algorithm = LIB$K_VM_QUICK_FIT with 16 Lookaside Lists ranging from a minimum blocksize of 16, to a maximum blocksize of 256
Flags = 00000028
LIB$M_VM_FREE_FILL0
LIB$M_VM_EXTEND_AREA
Initial size = 16 pages Current size = 112 pages in 1 area
Extend size = 16 pages Page limit = None
Requests are rounded up to a multiple of 16 bytes, naturally aligned on 16 byte boundaries
56992 bytes have been freed and not yet reallocated
576 bytes are used for zone and area control blocks, or 0.9% overhead

Quick Fit Lookaside List Summary:

<table>
<thead>
<tr>
<th>List number</th>
<th>Block size</th>
<th>Number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>80</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>96</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>112</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>144</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>176</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>192</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>208</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>224</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>240</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>10</td>
</tr>
</tbody>
</table>

Area Summary:

<table>
<thead>
<tr>
<th>First address</th>
<th>Last address</th>
<th>Pages assigned</th>
<th>Bytes not yet allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001C0004000</td>
<td>00000001C0011FFF</td>
<td>112</td>
<td>352</td>
</tr>
</tbody>
</table>

Scanning Lookaside Lists in Zone Control Block
Scanning Free List for Area at 00000001C0004000
Number of blocks = 63, Min blocksize = 272, Max blocksize = 1360
LIB$SIGNAL

Signal Exception Condition

The Signal Exception Condition routine generates a signal that indicates that an exception condition has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program will exit.

Format

LIB$SIGNAL condition-value [,condition-argument...] [,condition-value-n [,condition-argument-n...]]

Returns

None.

Arguments

condition-value
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by value

OpenVMS 32-bit condition value. The condition-value argument is an unsigned longword that contains this condition value.

The HP OpenVMS Programming Concepts Manual explains the format of an OpenVMS condition value.

ccondition-argument
OpenVMS usage: varying_arg
type: unspecified
access: read only
mechanism: by value

As many arguments as are required to process the exception specified by condition-value. Note that these arguments are also used as FAO (formatted ASCII output) arguments to format a message.

The HP OpenVMS Programming Concepts Manual explains the message format.

ccondition-value-n
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by value

OpenVMS 32-bit condition value. The optional condition-value-n argument is an unsigned longword that contains this condition value. The calling routine can specify additional conditions to be processed by specifying condition-value-2 through condition-value-n, with each condition value followed by any arguments required to process the condition specified. However, the total number of arguments in the call to LIB$SIGNAL must not exceed 253.

The HP OpenVMS Programming Concepts Manual explains the format of an OpenVMS condition value.
**LIB$ Routines**

**LIB$SIGNAL**

**condition-argument-n**

OpenVMS usage: varying_arg  
Type: unspecified  
Access: read only  
Mechanism: by value

As many arguments as are required to create the message reporting the exception specified by **condition-value-n**.

The *HP OpenVMS Programming Concepts Manual* explains the message format.

**Description**

A routine calls LIB$SIGNAL to indicate an exception condition or output a message rather than return a status code to its caller.

LIB$SIGNAL creates a signal argument vector that contains all the arguments passed to it, with the PC and PSL (VAX) or PS (Alpha or I64) appended to it. LIB$SIGNAL also creates a mechanism argument vector that contains the state of the process at the time of the exception. LIB$SIGNAL then searches for a condition handler to process the exception condition.

LIB$SIGNAL first examines the primary and secondary exception vectors, then scans the stack, beginning with the most recent frame, searching for declared condition handlers. LIB$SIGNAL calls, in succession, each condition handler it finds, until a condition handler

- Returns a continue code
- Calls system service $UNWIND
- Calls LIB$STOP

LIB$SIGNAL uses each frame’s saved frame pointer (FP) to chain back through the stack frames. The *HP OpenVMS Programming Concepts Manual* provides additional information on this process.

The condition handler can do one of the following:

- Successfully process the condition and return a continue code (that is, any success completion code with bit 0 set to 1). In this case, LIB$SIGNAL returns to its caller, which should be prepared to continue execution.
- Fail to process the condition. The handler then returns a resignal code (that is, any completion code with bit 0 set to 0) and LIB$SIGNAL scans the stack for the next specified handler.
- Dismiss the signal and system service $UNWIND to cause the Condition Handling Facility (CHF) to perform some call stack cleanup and resume program execution (at a level specified by the condition handler) up on the call stack.

LIB$SIGNAL can, as necessary, scan up to 65,536 previous stack frames and then finally examine the last-chance exception vector. If called, the last-chance exception handler formats a message based on the condition codes and arguments contained within the signal argument vector.
Condition Values Returned

None.

Examples

1. C
   This Fortran example program demonstrates the use of
   LIB$SIGNAL.
   This program defines SS$... signals and then calls LIB$SIGNAL
   passing the access violation code as the argument.
   INCLUDE '(SSDEF)'
   CALL LIB$SIGNAL ( %VAL(SS$_ACCVIO) )
   END

   In Fortran, this code fragment signals the standard system message ACCESS VIOLATION.
   The output generated by this Fortran program on an OpenVMS Alpha system is as follows:

   %SYSTEM-F-ACCVIO, access violation, reason mask=10, virtual address=03C00020,
   PC=00000000, PS=08000000
   %TRACE-F-TRACEBACK, symbolic stack dump follows
   module name routine name line rel PC abs PC
   D2$MAIN D2$MAIN 683 00000010 00000410

2. ;
   This VAX MACRO example program demonstrates the use of LIB$SIGNAL
   by forcing an access violation to be signaled.
   .EXTRN SS$_ACCVIO ; Declare external symbol
   .ENTRY START,0
   PUSHL #SS$_ACCVIO ; Condition value symbol
   ; for access violation
   CALLS #1, LIB$SIGNAL ; Signal the condition
   RET
   .END START

   This example shows the equivalent VAX MACRO code. The output generated
   by this program on an OpenVMS VAX system is as follows:

   %SYSTEM-F-ACCVIO, access violation, reason mask=0F, virtual address=03C00000,
   PC=00000000, PSL=00000000
   %TRACE-F-TRACEBACK, symbolic stack dump follows
   module name routine name line rel PC abs PC
   .MAIN. START 0000000F 0000020F
3. ```c
#include <ssdef.h>
#include <lib$routines.h>

main()
{
    /*
     * lib$signal will append the PC/PS to argument list,
     * so pass only first two FAO arguments to lib$signal
     */
    lib$signal(SS$ ACCVIO, 4, -559038737); /* Shouldn't return */
    return (SS$ _NORMAL); /* Exit if it does */
}
```

This example shows the equivalent C code. The output generated by this program on an OpenVMS Alpha system is as follows:

```
%SYSTEM-F-ACCVIO, access violation, reason mask=04, virtual address=DEADBEEF,
PC=00020034, PS=0000001B
%TRACE-F-TRACEBACK, symbolic stack dump follows
Image Name Module Name Routine Name Line Number rel PC abs PC
LIB$SIGNAL 0 00010034 00020034
LIB$SIGNAL 0 000100A0 000200A0
         082F01158 82F011580 7FF190D0 7FF190D0
```

4. ```c
#include <stdio>
#include <ssdef>
#include <tlib$routines>

/* Condition handler: */
/* */
/* This condition handler will print out the signal array, based on */
/* the argument count in the first element of the array. The error */
/* is resigalled and should be picked up by the last chance condition */
/* handler which will format and print error messages and terminate the */
/* program. */
/* */
int handler (int* sig, int*mech)
{
    int i;
    printf ("*** Caught signal:

    ");
    for (i = 0; i <= sig[0]; i++)
        {printf (" %08X
", sig[i]);
    }
    printf ("\n");
    return SS$ _RESIGNAL;
}
```

/* Main program: */
/* */
/* Signal errors: */
/* */
/* SS$_BADPARAM has no arguments */
/* SS$ ACCVIO has 4 arguments, the last two (PC and PS) are */
/* automatically provided by LIB$SIGNAL. */
/* */
main ()
{
    lib$establish (handler);
    lib$signal (SS$ _BADPARAM, SS$ _ACCVIO, 2, 0xFACE);
}
```

This C example demonstrates the use of a condition handler to capture the signal generated by LIB$SIGNAL. The output is as follows:
$ CC SIGNAL.C
$ LINK SIGNAL
$ RUN SIGNAL
*** Caught signal:

00000006
00000014
0000000C
00000002
0000FACE
000201A0
0000001B

%SYSTEM-F-BADPARAM, bad parameter value
-SYSTEM-F-ACCVIO, access violation, reason mask=02,
virtual address=000000000000FACE, PC=00000000000201A0, PS=0000001B
%TRACE-F-TRACEBACK, symbolic stack dump follows

<table>
<thead>
<tr>
<th>image</th>
<th>module</th>
<th>routine</th>
<th>line</th>
<th>rel PC</th>
<th>abs PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>main</td>
<td>5961</td>
<td>0000000000001A0</td>
<td>00000000000201A0</td>
</tr>
<tr>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>__main</td>
<td>0</td>
<td>000000000000050</td>
<td>0000000000020050</td>
</tr>
</tbody>
</table>

0 FFFFFFFF82204914 FFFFFFFF82204914

lib–527
LIB$ SIG_TO_RET
Signal Converted to a Return Status

The Signal Converted to a Return Status routine converts any signaled condition value to a value returned as a function. The signaled condition is returned to the caller of the user routine that established the handler that is calling LIB$SIG_TO_RET. This routine may be established as or called from a condition handler.

Format

LIB$SIG_TO_RET signal-arguments,mechanism-arguments

Returns

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

Arguments

signal-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: read only
mechanism: by reference, array reference

Signal argument vector. The signal-arguments argument contains the address of an array that is this signal argument vector stack.

See the HP OpenVMS Programming Concepts Manual for a description of the signal argument vector.

mechanism-arguments
OpenVMS usage: structure
type: unspecified
access: read only
mechanism: by reference

Mechanism arguments vector. The mechanism-arguments argument contains the address of a structure that is this mechanism argument vector stack.

See the HP OpenVMS Programming Concepts Manual for a description of the mechanism argument vector.

Description

LIB$SIG_TO_RET is called with the argument list that was passed to a condition handler by the OpenVMS Condition Handling Facility. The signaled condition is converted to a value returned to the routine that called the routine that established the handler. That action is performed by unwinding the stack to the caller of the establisher of the condition handler. The condition code is returned as the value in R0. See the HP OpenVMS Programming Concepts Manual for more information on condition handling.

LIB$SIG_TO_RET causes the stack to be unwound to the caller of the routine that established the handler which was called by the signal.
Condition Values Returned

`SS$\_NORMAL`  
Routine successfully completed; `SS$\_UNWIND` completed. Otherwise, the error code from `SS$\_UNWIND` is returned.

Example

Fortran example demonstrates how to use `LIB$SIG_TO_RET`. This function subroutine inverts each entry in an array. That is, \( a(i,j) \) becomes \( \frac{1}{a(i,j)} \). The subroutine has been declared as an integer function so that the status of the inversion may be returned. The status should be success, unless one of the \( a(i,j) \) entries is zero. If one of the \( a(i,j) = 0 \), then \( \frac{1}{a(i,j)} \) is division by zero. This division by zero does not cause a division by zero error, rather, the routine will return a failure.

```fortran
INTEGER\*4 FUNCTION FLIP(A,N)
DIMENSION A(N,N)
EXTERNAL LIB$SIG_TO_RET
CALL LIB$ESTABLISH (LIB$SIG_TO_RET)
FLIP = .TRUE.

C Flip each entry.

DO I = 1, N
  DO J = 1, N
    A(I,J) = 1.0/A(I,J)
  END DO
END DO

C This is the main code.

INTEGER STATUS, FLIP
REAL ARRAY_1(2,2), ARRAY_2(3,3)
DATA ARRAY_1/1,2,3,4/, ARRAY_2/1,2,3,5,0,5,6,7,2/
CHARACTER\*32 TEXT(2), STRING
DATA TEXT(1)/' This array could be flipped. '/,
  TEXT(2)/' This array could not be flipped. '/
STRING = TEXT(1)
STATUS = FLIP(ARRAY_1,2)
IF ( .NOT. STATUS) STRING = TEXT(2)
  TYPE '(a)', STRING
STRING = TEXT(1)
STATUS = FLIP(ARRAY_2,3)
IF ( .NOT. STATUS) STRING = TEXT(2)
  TYPE '(a)', STRING
END
```

This Fortran example program inverts each entry in an array. The output generated by this program is as follows:

This array could be flipped.
This array could not be flipped.
LIB$ SIG_TO_STOP
Convert a Signaled Condition to a Signaled Stop

The Convert a Signaled Condition to a Signaled Stop routine converts a signaled condition to a signaled condition that cannot be continued.

Format

LIB$SIG_TO_STOP signal-arguments,mechanism-arguments

Returns

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

Arguments

signal-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: modify
mechanism: by reference, array reference

Signal argument vector. The signal-arguments argument contains the address of an array that is this signal argument vector stack.

See the HP OpenVMS Programming Concepts Manual for a description of the signal argument vector.

mechanism-arguments
OpenVMS usage: structure
type: unspecified
access: read only
mechanism: by reference

Mechanism argument vector. The mechanism-arguments argument contains the address of a structure that is this mechanism argument vector stack.

See the HP OpenVMS Programming Concepts Manual for a description of the mechanism argument vector.

Description

LIB$SIG_TO_STOP causes a signal to appear as though it had been signaled by a call to LIB$STOP. When a signal is generated by LIB$STOP, the severity code is forced to SEVERE and control cannot return to the routine that signaled the condition. LIB$SIG_TO_STOP may be enabled as a condition handler for a routine or it may be called from a condition handler.

If the condition value in signal-arguments is SS$_UNWIND, then LIB$SIG_TO_STOP returns the error condition LIB$_INVARG.
### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed; SS$ _UNWIND completed. Otherwise, the error code from SS$ _UNWIND is returned.</td>
</tr>
<tr>
<td>LIB$ _INVARG</td>
<td>Invalid argument. The condition code in <code>signal-arguments</code> is SS$ _UNWIND.</td>
</tr>
</tbody>
</table>
LIB$ SIM_TRAP

Simulate Floating Trap

The Simulate Floating Trap routine converts floating faults to floating traps. It can be enabled as a condition handler or can be called by one. †

This routine is not available to native OpenVMS Alpha or I64 programs but is available to translated VAX images.

Format

LIB$SIM_TRAP signal-arguments, mechanism-arguments

Returns

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

Arguments

signal-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: modify
mechanism: by reference, array reference

Signal argument vector. The signal-arguments argument contains the address of an array that is this signal argument vector stack.

See the HP OpenVMS Programming Concepts Manual for a description of the signal argument vector.

mechanism-arguments
OpenVMS usage: vector_longword_unsigned
type: unspecified
access: read only
mechanism: by reference, array reference

Mechanism argument vector. The mechanism-arguments argument contains the address of an array that is this mechanism argument vector stack.

See the HP OpenVMS Programming Concepts Manual for a description of the mechanism argument vector.

Description

LIB$SIM_TRAP converts floating faults to floating traps. It can be enabled as a condition handler or can be called by one.

LIB$SIM_TRAP intercepts floating overflow, underflow, and divide-by-zero faults. It simulates the instruction causing the condition up to the point where a fault should be signaled, then signals the corresponding floating trap.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
Since LIB$SIM_TRAP nullifies the condition handling for the original fault condition, the final condition signaled by the routine will be from the context of the instruction itself, rather than from the condition handler. The signaling path is identical to that of a hardware-generated trap. The signal argument vector is placed so the last entry in the vector will be the user’s stack pointer at the completion of the instruction (for a trap), or at the beginning of the instruction (for a fault).

See the VAX Architecture Reference Manual for more information on faults and traps.

**Condition Values Returned**

- **SS$_RESIGNAL**: Resignal condition to next handler. The exception was one that LIB$SIM_TRAP could not handle.
LIB$SKPC
Skip Equal Characters

The Skip Equal Characters routine compares each character of a given string with a given character and returns the relative position of the first nonequal character as an index. LIB$SKPC makes the VAX SKPC instruction available as a callable routine.¹

Format

LIB$SKPC  character-string ,source-string

Returns

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

The relative position in the source string of the first unequal character. LIB$SKPC returns a zero if the source string was of zero length or if every character in source-string was equal to character-string.

Arguments

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

String whose initial character is to be used by LIB$SKPC in the comparison. The character-string argument contains the address of a descriptor pointing to this string. Only the first character of character-string is used, and the length of character-string is not checked.

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

String to be searched by LIB$SKPC. The source-string argument contains the address of a descriptor pointing to this string.

Description

LIB$SKPC compares the initial character of character-string with successive characters of source-string until it finds an inequality or reaches the end of the source-string. It returns the relative position of this unequal character as an index, which is the relative position of the first occurrence of a substring in the source string.

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Condition Values Returned

None.

Example

C+  
C This Fortran example program shows the use of LIB$SKPC.  
C LIB$SKPC compares each character of a given string with a given character. 
C It returns the relative position of the first nonequal character as an index.  
C-  
     I = LIB$SKPC (’ ’, ’ ABC’) 
     TYPE 1, I  
     1 FORMAT(’The blank character matches the’,I2,’nd character in’) 
     TYPE *,’the string ” ABC”’  
     J = LIB$SKPC (’A’, ’ AAA’) 
     TYPE 2, J  
     2 FORMAT(’The character ”A” matches the’,I2,’th character in’) 
     TYPE *,’the string ” AAA”’  
END

This Fortran example generates the following output:

The blank character matches the 2nd character in 
the string ” ABC”
The character ”A” matches the 0th character in 
the string ” AAA”
LIB$SPANC
Skip Selected Characters

The Skip Selected Characters routine is used to skip a specified set of characters in the source string. LIB$SPANC makes the VAX SPANC instruction available as a callable routine.\(^1\)

Format

```
LIB$SPANC  source-string ,table-array ,byte-integer-mask
```

Returns

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

The relative position in the source string of the character that terminated the operation is returned if such a character is found. Otherwise, zero is returned. If the source string has a zero length, then a zero is returned.

Arguments

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

Source string used by LIB$SPANC to index into table-array. The source-string argument contains the address of a descriptor pointing to this source string.

**table-array**
OpenVMS usage: vector_mask_byte
type: byte (unsigned)
access: read only
mechanism: by reference, array reference

Table that LIB$SPANC indexes into and performs an AND operation with the byte-integer-mask byte. The table-array argument contains the address of an unsigned byte array that is this table.

**byte-integer-mask**
OpenVMS usage: mask_byte
type: byte (unsigned)
access: read only
mechanism: by reference

Mask that an AND operation is performed with bytes in table-array. The byte-integer-mask argument contains the address of an unsigned byte that is this mask.

---

\(^1\) On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
Description

LIB$SPANC uses successive bytes of the string specified by source-string to index into a table. An AND operation is performed on the byte selected from the table and the mask byte.

The operation is terminated when the result of the AND operation is zero.

Condition Values Returned

None.

Example

!+
! This Fortran program demonstrates how to use
! LIB$SCANC and STR$UPCASE.
!+
! Declare the Run-Time Library routines to be used.
!-
  INTEGER*4 STR$UPCASE   ! Translate to upper case
  INTEGER*4 LIB$SCANC    ! Look for characters
  INTEGER*4 LIB$SPANC    ! Skip over characters
!+
! Declare the alphabet from which "words" are constructed.
!-
  CHARACTER*(38) ALPHABET
  DATA ALPHABET /'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789$_'/
!+
! Local variable declarations
!-
  INTEGER*4 WORD_COUNT /0/     ! Count of words found
  INTEGER*4 WORD_LENGTH /0/    ! Length of a word
  INTEGER*4 TOTAL_LENGTH /0/   ! Sum of word lengths
  INTEGER*4 START_POS /0/      ! Position of start of word
  INTEGER*4 END_POS /0/        ! Position of end of word
  REAL*4 AVERAGE_LENGTH /0.0/  ! Average length of words
  CHARACTER*80 LINE           ! Line to examine for words
  BYTE MATCH_TABLE(0:255) /256*0/ ! Match table for scanning
!+
! The routines LIB$SCANC and LIB$SPANC require a table with an entry
! for each possible character. Create a match table from ALPHABET
! with an entry of 1 if the character is in ALPHABET, 0 otherwise.
! MATCH_TABLE has already been initialized to zeros.
!-
  DO I = 1, LEN(ALPHABET)
    MATCH_TABLE(ICHAR(ALPHABET(I:I))) = 1
  END DO
!+
! Loop forever finding words in LINE. When LINE is exhausted,
! indicated by a START_POS of zero, read another one. Upon
! end-of-file, leave the loop and print the statistics.
!-
OPEN( UNIT = 1, FILE = 'TEST.DAT', TYPE = 'OLD' )
DO WHILE (.TRUE.)
  DO WHILE (START_POS .EQ. 0) ! Get a new line
    READ (1,'(A)',END=900) LINE ! If EOF, skip to 900
    CALL STR$UPCASE (LINE,LINE) ! Convert to upper case for matching
    START_POS = LIB$SCANC (LINE,MATCH_TABLE,1) ! Find beginning
  ! of first word
  END DO
  ! START_POS now points to the beginning of a word. Call LIB$SPANC to
  ! find the first character that is not part of the word. Set
  ! START_POS to beginning of next word. If LIB$SPANC does not
  ! find a non-word character, it returns zero.
  !
  END_POS = 1 START_POS + LIB$SPANC (LINE(START_POS:), MATCH_TABLE,1) - 1
  IF (END_POS .LT. START_POS) THEN ! Word goes to end of line
    WORD_LENGTH = (LEN(LINE) + 1) - START_POS
    START_POS = 0 ! Indicate line exhausted
  ELSE
    WORD_LENGTH = END_POS - START_POS
    START_POS = 1 END_POS + LIB$SCANC (LINE(END_POS:),MATCH_TABLE,1) - 1
    IF (START_POS .LT. END_POS) START_POS = 0 ! No more words on line
  END IF
  ! Update count and length statistics.
  WORD_COUNT = WORD_COUNT + 1
  TOTAL_LENGTH = TOTAL_LENGTH + WORD_LENGTH
END DO
900 CONTINUE

! Compute average word length and display statistics.
!
IF (WORD_COUNT .NE. 0)
  AVERAGE_LENGTH = FLOAT(TOTAL_LENGTH) / FLOAT(WORD_COUNT)
  TYPE 901,WORD_COUNT,AVERAGE_LENGTH
  FORMAT (1X,I10,' words found, average length was ',
      1 F4.1,' letters.')
901   CLOSE (1)
END

This Fortran program reads text from the default input unit and looks for words. A word is defined as a string containing only the characters A through Z (uppercase or lowercase), 0 through 9, and the dollar sign ($) and underscore (_) symbols. The program reports the total number of words found and their average length.
The program uses three Run-Time Library routines: STR$UPCASE, LIB$SCANC, and LIB$SPANC.

1. The string is converted to uppercase using STR$UPCASE so that the search for words will ignore the case of letters.

2. LIB$SCANC searches through the string for one of a set of characters, the set being specified as nonzero elements in a 256-byte table.

3. Similarly, LIB$SPANC uses the VAX SPANC instruction to search through a string for a character whose table entry is not zero. 1

The value returned by each routine is the index into the string where the first matching (or nonmatching) character was found, or zero if no match was found.

The output generated by this Fortran program is as follows:

   12 words found, average length was 4.2 letters.

---

1 On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$SPAWN
Spawn Subprocess

The Spawn Subprocess routine requests the command language interpreter (CLI) of the calling process to spawn a subprocess for executing CLI commands. LIB$SPAWN provides the same function as the DCL command SPAWN.

Format


Returns

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

Arguments

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

CLI command to be executed by the spawned subprocess. The command-string argument is the address of a descriptor pointing to this CLI command string. If command-string is omitted, commands are taken from the file specified by input-file.

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

Equivalence name to be associated with the logical name SYS$INPUT in the logical name table for the subprocess. The input-file argument is the address of a descriptor pointing to this equivalence string. If input-file is omitted, the default is the caller’s SYS$INPUT.

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

Equivalence name to be associated with the logical names SYS$OUTPUT and SYS$ERROR in the logical name table for the subprocess. The output-file argument is the address of a descriptor pointing to this equivalence string. If output-file is omitted, the default is the caller’s SYS$OUTPUT.
flags
OpenVMS usage: mask_longword

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOWAIT</td>
<td>If this bit is set, the calling process continues executing in parallel with the subprocess. If this bit is clear, the calling process hibernates until the subprocess completes.</td>
</tr>
<tr>
<td>1</td>
<td>NOCLISYM</td>
<td>If this bit is set, the spawned subprocess does not inherit CLI symbols from its caller. If this bit is clear, the subprocess inherits all currently defined CLI symbols. You may want to specify NOCLISYM to help prevent commands redefined by symbol assignments from affecting the spawned commands.</td>
</tr>
<tr>
<td>2</td>
<td>NOLOGNAM</td>
<td>If this bit is set, the spawned subprocess does not inherit process logical names from its caller. If this bit is clear, the subprocess inherits all currently defined process logical names. You may want to specify NOLOGNAM to help prevent commands redefined by logical name assignments from affecting the spawned commands.</td>
</tr>
<tr>
<td>3</td>
<td>NOKEYPAD</td>
<td>If this bit is set, the keypad symbols and state are not passed to the subprocess. If this bit is not set, the keypad settings are passed to the subprocess.</td>
</tr>
<tr>
<td>4</td>
<td>NOTIFY</td>
<td>If this bit is set, a message is broadcast to SYS$OUTPUT when the subprocess completes or aborts. If this bit is not set, no message is broadcast. This bit should not be set unless the NOWAIT bit is also set.</td>
</tr>
<tr>
<td>5</td>
<td>NOCONTROL</td>
<td>If this bit is set, no carriage-return/line-feed is prefixed to any prompt string. If this bit is not set, a carriage-return/line-feed is prefixed to any prompt string specified.</td>
</tr>
<tr>
<td>6</td>
<td>TRUSTED</td>
<td>If this bit is set, it indicates a SPAWN command on behalf of the application. If this bit is not set, it indicates that the SPAWN command originates from user. SPAWN commands originating from users are disallowed in captive accounts (DCL).</td>
</tr>
<tr>
<td>7</td>
<td>AUTHPRIV</td>
<td>If this bit is set, the subprocess inherits the caller’s authorized privileges. If this bit is clear, the spawned processes’ authorized mask is set equal to the caller’s current (active) privilege mask.</td>
</tr>
<tr>
<td>8</td>
<td>SUBSYSTEM</td>
<td>If this bit is set, a spawned process inherits protected subsystem IDs for the duration of LOGINOUT.EXE (used to map the CLI). The IDs will be removed in the process of transferring control to the CLI (as a user mode $RUNDWN is performed). If this bit is clear, LOGINOUT does not execute under the subsystem IDs.</td>
</tr>
</tbody>
</table>
Bits 9 through 31 are reserved for future expansion and must be zero. Symbolic flag names are defined in libraries supplied by HP in module $CLIDEF. They are CLI$M_NOWAIT, CLI$M_NOCLISYM, CLI$M_NOLOGNAM, CLI$M_NOKEYPAD, CLI$M_NOTIFY, CLI$M_NOCONTROL, CLI$M_TRUSTED, CLI$M_AUTHPRIV, and CLI$M_SUBSYSTEM.

**process-name**

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

Name defined for the subprocess. The *process-name* argument is the address of a descriptor pointing to this name string. If *process-name* is omitted, a unique process name will be generated. If you supply a name and it is not unique, LIB$SPAWN will return the condition value SS$_DUPLNAM.

The DCL_CTLFLAGS is a bitmask used to alter default behavior for certain commands on a systemwide basis. Currently, only the low bit of the bitmask is defined. The low bit controls the default *process-name* assignment for a subprocess created using the LIB$SPAWN routine.

Prior to OpenVMS Version 7.3-1, if no process name was supplied, the system constructed a name by appending _n to the username, where n was the next available non-duplicate integer for any process currently in the system. For example, the first spawned process from user SYSTEM would be called SYSTEM_1, the second, SYSTEM_2, and so on. The next available number was chosen, as soon as a gap was found.

Beginning in OpenVMS Version 7.3-1, the default constructed process name for subprocesses has changed. Instead of incrementally searching for the next unique number, a random number is chosen to append to the username. Therefore, the first processes that are spawned from user SYSTEM might be SYSTEM_154, SYSTEM_42, SYSTEM_87, and so on. This procedure results in a very high probability of finding a unique number on the first try since it is unlikely the same number is already in use.

However, some applications might rely on the previous method of assigning subprocess names. The DCL_CTLFLAGS parameter is available to allow you to configure the system as necessary.

Bit 0 of DCL_CTLFLAGS selects the behavior for assigning default subprocess names, as explained in the following:

- If clear, the new behavior is used. If the process name is not specified, it will be the username with a random number suffix. This is the default setting.
- If set, the previous behavior is used. If the process name is not specified, it will be the username with the next available number suffix.

**process-id**

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

Process identification of the spawned subprocess. The *process-id* argument is the address of an unsigned longword that contains this process identification value.
This process identification value is meaningful only if the NOWAIT flags bit is set.

**completion-status-address**
OpenVMS usage: address
type: address
access: read only
mechanism: by value

The final completion status of the subprocess. The completion-status-address argument contains the address of the status. The system writes the value of the final completion status of the subprocess into completion-status-address when the subprocess completes. If the subprocess returns a status code of 0, the system writes SS$NORMAL into this address.

If the NOWAIT flags bit is set, the completion-status-address is updated asynchronously when the subprocess completes. Use the byte-integer-event-flag-num or AST-address arguments to determine when the subprocess has completed. Your program must ensure that the address is still valid when the value is written.

**byte-integer-event-flag-num**
OpenVMS usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

The number of a local event flag to be set when the spawned subprocess completes. The byte-integer-event-flag-num argument is the address of an unsigned byte that contains this event flag number. If byte-integer-event-flag-num is omitted, no event flag is set.

Specifying byte-integer-event-flag-num is meaningful only if the NOWAIT flags bit is set.

**AST-address**
OpenVMS usage: procedure
type: procedure value
access: call without stack unwinding
mechanism: by value

Routine to be called by means of an AST when the subprocess completes.

Specifying AST-address is meaningful only if the NOWAIT flags bit is set.

**varying-AST-argument**
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: by value

A value to be passed to the AST routine. Typically, the varying-AST-argument argument is the address of a block of storage the AST routine will use.

Specifying varying-AST-argument is meaningful only if the NOWAIT flags bit is set and if AST-address has been specified.
**prompt-string**
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

Prompt string to use in the subprocess. The **prompt-string** argument is the address of a descriptor pointing to this prompt string. If **prompt-string** is omitted, the subprocess uses the same prompt string that the parent process uses.

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

File specification for the command language interpreter (CLI) to be run in the subprocess. The **cli** argument is the address of this file specification string’s descriptor. The CLI specified must reside in SYS$SYSTEM with a file type of .EXE, and it must be installed. No directory or file type may be specified. The **cli** argument must be specified in uppercase characters.

If **cli** is omitted, the subprocess uses the same CLI as the parent process. If **cli** is specified, no context is copied to the subprocess.

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

File specification for the command tables to be used by the spawned process. The **table** argument is the address of this file specification string’s descriptor. The table specified must reside in SYS$SHARE with a file type of .EXE, and it must be installed.

If **table** is omitted, the subprocess uses the same table as the parent process.

**Description**
The subprocess created by LIB$SPAWN inherits the following attributes from the caller’s environment:

- Process logical names
- Global and local CLI symbols
- Default device and directory
- Process privileges
- Process nondeductible quotas
- Current command verification setting

The subprocess does not inherit process-permanent files nor routine or image context.
Though the subprocess inherits the caller’s process privileges as its own process privileges, the set of authorized privileges in the subprocess is inherited from the caller’s current privileges. If the calling image is installed with elevated privileges, these privileges are not available to the the subprocess until a SET PROCESS/PRIVILEGE command or equivalent $SETPRV call is performed in the subprocess to enable these privileges.

If the calling image is installed with elevated privileges, it should disable those privileges around the call to LIB$SPAWN unless the environment of the subprocess is strictly controlled. Otherwise, there is a possibility of a security breach due to elevated privileges accidentally being made available to the user.

If neither command-string nor input-file is present, command input is taken from the parent terminal. If both command-string and input-file are present, the subprocess first executes command-string and then reads from input-file. If only command-string is specified, the command is executed, and the subprocess is terminated by either a LOGOUT command or an end-of-file.

The subprocess does not inherit process-permanent files nor routine or image context. No LOGIN.COM file is executed.

Unless the NOWAIT flags bit is set, the caller’s process is put into hibernation until the subprocess finishes. Because the caller’s process hibernates in supervisor mode, any user-mode ASTs queued for delivery to the caller are not delivered until the caller reawakes. Control can also be restored to the caller by means of an ATTACH command or by a suitable call to LIB$ATTACH from the subprocess.

This routine is supported for use only with the DCL command language interpreter. If used when the current CLI is MCR, the error status LIB$_NOCLI is returned.

If an image is run directly as a subprocess or as a detached process, there is no CLI present to perform this function. In such cases, the error status LIB$_NOCLI is returned.

Programs depending on embedded DCL commands may not function properly when run under other command language interpreters that may be supported by future versions of OpenVMS operating systems.

See the HP OpenVMS DCL Dictionary for a complete description of the SPAWN command.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_ACCVIO</td>
<td>Access violation. One of the string arguments to LIB$SPAWN could not be read, or <strong>completion-status-address</strong> could not be written.</td>
</tr>
<tr>
<td>SS$_DUPLNAM</td>
<td>Duplicate process name. If the argument <strong>process-name</strong> was specified, it duplicated an existing process name. If <strong>process-name</strong> was omitted, LIB$SPAWN was unable to create a unique name for the subprocess.</td>
</tr>
</tbody>
</table>
fac$_xxx  Other error trying to create subprocess.
LIB$_INVARG Invalid argument. The optional argument \textit{flags} was specified, and a bit other than bits 0 through 8 was set.
LIB$_INVSTRDES Invalid string descriptor. One of the string arguments had an invalid descriptor.
LIB$_NOCLI No CLI present to perform function. The calling process did not have a CLI to perform the function, or the CLI did not support the request type. Note that an image run as a subprocess or detached process does not have a CLI.

If an error is encountered during subprocess creation, the status value for that error is returned by LIB$SPAWN.

\textbf{Example}

\begin{verbatim}
ISTAT=LIB$SPAWN(,,,CLI$M_NOKEYPAD,,,,,,,'> ')
IF (.NOT. ISTAT) CALL LIB$STOP(%VAL(ISTAT))
\end{verbatim}

This Fortran fragment shows a call to LIB$SPAWN from within a Fortran program. A subprocess is spawned taking input from SYS$INPUT and giving output to SYS$OUTPUT. The keypad state is not passed to the subprocess. A prompt string of "> " is specified for the subprocess.
LIB$STAT_TIMER
Statistics, Return Accumulated Times and Counts

The Statistics, Return Accumulated Times and Counts routine returns to its caller one of five available statistics accumulated since the last call to LIB$INIT_TIMER. Unlike LIB$SHOW_TIMER, which formats the values for output, LIB$STAT_TIMER returns the value as an unsigned longword or quadword.

Format

LIB$STAT_TIMER code,value-argument [,handle-address]

Returns

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

Arguments

code
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

The address of a signed longword integer that contains a code to specify the statistic to be returned. The code specification must be an integer from 1 to 5.

The following values are allowed for code:

<table>
<thead>
<tr>
<th>Value</th>
<th>Statistic Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Elapsed real time (quadword, in system time format)</td>
</tr>
<tr>
<td>2</td>
<td>Elapsed CPU time (longword, in 10 millisecond increments)</td>
</tr>
<tr>
<td>3</td>
<td>Count of buffered I/O operations (longword)</td>
</tr>
<tr>
<td>4</td>
<td>Count of direct I/O operations (longword)</td>
</tr>
<tr>
<td>5</td>
<td>Count of page faults (longword)</td>
</tr>
</tbody>
</table>

value-argument

OpenVMS usage: user_arg
type: unspecified
access: write only
mechanism: by reference

The statistic returned by LIB$STAT_TIMER. The value-argument argument contains the address of a longword or quadword that is this statistic. All statistics are longword integers except elapsed real time, which is a quadword.

See the HP OpenVMS System Services Reference Manual for more details on the system time format.
LIB$ Routines
LIB$STAT_TIMER

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

Pointer to a block of storage. The optional handle-address argument contains the address of an unsigned longword that is this pointer.

If handle-address is specified, LIB$STAT_TIMER assumes that LIB$INIT_TIMER has been called with the same value of handle-address. Handle-address is an optional argument. If it is not specified, LIB$STAT_TIMER uses internal storage.

Description

Only one of the five statistics is returned by each call to LIB$STAT_TIMER. The elapsed time is returned in the system quadword format. Therefore the receiving area should be eight bytes long. All other returned values are longwords.

LIB$SHOW_TIMER and LIB$STAT_TIMER are relatively simple tools for testing the performance of a new application. Note that LIB$INIT_TIMER must be called prior to any calls to LIB$SHOW_TIMER or LIB$STAT_TIMER.

To obtain more detailed information, use LIB$GETJPI (Get Job/Process Information) or the system service $GETTIM.

The following summary shows the differences between LIB$SHOW_TIMER and LIB$STAT_TIMER:

<table>
<thead>
<tr>
<th>Code</th>
<th>Statistic</th>
<th>Format for LIB$SHOW_TIMER</th>
<th>Format for LIB$STAT_TIMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Elapsed real time</td>
<td>hhh:mm:ss.cc</td>
<td>Quadword in system time format</td>
</tr>
<tr>
<td>2</td>
<td>Elapsed CPU time</td>
<td>hhh:mm:ss.cc</td>
<td>Longword in 10-millisecond increments</td>
</tr>
<tr>
<td>3</td>
<td>Count of buffered I/O operations</td>
<td>nnnn</td>
<td>Longword</td>
</tr>
<tr>
<td>4</td>
<td>Count of direct I/O operations</td>
<td>nnnn</td>
<td>Longword</td>
</tr>
<tr>
<td>5</td>
<td>Count of page faults</td>
<td>nnnn</td>
<td>Longword</td>
</tr>
</tbody>
</table>

When you call LIB$INIT_TIMER, you must use the optional handle-address argument only if you want to keep several sets of statistics simultaneously. This argument points to a block in heap storage where the statistics are to be stored.

You need to call LIB$FREE_TIMER only if you have specified handle-address in LIB$INIT_TIMER and you want to deallocate all heap storage resources. In most cases, the implicit deallocation at program exit time will be sufficient.
**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument. Either <strong>code</strong> or <strong>handle-address</strong> is invalid.</td>
</tr>
</tbody>
</table>

**Example**

```pascal
PROGRAM STAT_TIMER(INPUT,OUTPUT);
{+}
{ This Pascal example program demonstrates the use of
  { LIB$STAT_TIMER.
  {-}

  TYPE
  BYTE = [BYTE] 0..255;
  WORD = [WORD] 0..65535;
  QUADWORD_SYSTEM_TIME = [QUAD] RECORD
    FIRST_LONGWORD : UNSIGNED;
    SECOND_LONGWORD : UNSIGNED;
  END;

  VAR
  ELAPSED_REAL_TIME : QUADWORD_SYSTEM_TIME;
  ELAPSED_STRING  : VARYING [32] OF CHAR;
  PAGE_FAULT_COUNT : UNSIGNED;
  RETURNED_STATUS : UNSIGNED;

  [EXTERNAL] FUNCTION LIB$INIT_TIMER(
    HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0 ) : INTEGER; EXTERNAL;

  [EXTERNAL] FUNCTION LIB$STAT_TIMER(  
    CODE : INTEGER;
    HANDLE_ADR : [REFERENCE] UNSIGNED := %IMMED 0
  ) : INTEGER; EXTERNAL;

  [EXTERNAL] FUNCTION LIB$STOP(  
    CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;
    FAO_ARGS : [IMMEDIATE,UNSAFE,LIST] UNSIGNED
  ) : INTEGER; EXTERNAL;

  [EXTERNAL] FUNCTION LIB$SYS_ASCTIM(  
    OUT_LEN : [REFERENCE] WORD := %IMMED 0;
    USER_TIME : QUADWORD_SYSTEM_TIME := %IMMED 0;
    CNV_FLG : UNSIGNED := %IMMED 0
  ) : INTEGER; EXTERNAL;

BEGIN
{+}
{ Call LIB$INIT_TIMER to initialize RTL internal counters.
{-}

  RETURNED_STATUS := LIB$INIT_TIMER;
  IF NOT ODD(RETURNED_STATUS) THEN
    LIB$STOP(RETURNED_STATUS);
{+}
{ Print a line of text to waste time.
{-}

  WRITELN('Spend time to acquire elapsed real time and page faults');
```

```
{+}
{ Call LIB$STAT_TIMER to retrieve statistics values. }
{-}
RETURNED_STATUS := LIB$STAT_TIMER(1,ELAPSED_REAL_TIME);
IF NOT ODD(RETURNED_STATUS)
 THEN
   LIB$STOP(RETURNED_STATUS);
RETURNED_STATUS := LIB$STAT_TIMER(5,PAGE_FAULT_COUNT);
IF NOT ODD(RETURNED_STATUS)
 THEN
   LIB$STOP(RETURNED_STATUS);
{+}
{ Print the statistics retrieved from LIB$STAT_TIMER. }
{-}
WRITELN('Page fault count is ',PAGE_FAULT_COUNT:1);
RETURNED_STATUS := LIB$SYS_ASCTIM(
   ELAPSED_STRING.LENGTH,ELAPSED_STRING.BODY,ELAPSED_REAL_TIME,
   1);
IF NOT ODD(RETURNED_STATUS)
 THEN
   LIB$STOP(RETURNED_STATUS);
WRITELN('Elapsed real time is ',ELAPSED_STRING);
END.

This Pascal program demonstrates the use of LIB$STAT_TIMER. The output generated by this program is as follows:

   Spend time to acquire elapsed real time and page faults
   Page fault count is 22
   Elapsed real time is 00:00:00.61
LIB$STAT_VM
Return Virtual Memory Statistics

The Return Virtual Memory Statistics routine returns to its caller one of six statistics available from calls to LIB$GET_VM/LIB$FREE_VM and LIB$GET_VM_PAGE/LIB$FREE_VM_PAGE. † Unlike LIB$SHOW_VM, which formats the values for output and displays them on SYS$OUTPUT, LIB$STAT_VM returns the statistic in the **value-argument** argument. Only one of the statistics is returned by each call to LIB$STAT_VM.

**Format**

LIB$STAT_VM code ,value-argument

**Returns**

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

**Arguments**

**code**

OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Code specifying which statistic is to be returned. The **code** argument contains the address of a signed longword integer that is this code.

<table>
<thead>
<tr>
<th>Code</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of successful calls to LIB$GET_VM</td>
</tr>
<tr>
<td>2</td>
<td>Number of successful calls to LIB$FREE_VM</td>
</tr>
<tr>
<td>3</td>
<td>Number of bytes allocated by LIB$GET_VM but not yet deallocated by LIB$FREE_VM</td>
</tr>
<tr>
<td>5</td>
<td>Number of calls to LIB$GET_VM_PAGE</td>
</tr>
<tr>
<td>6</td>
<td>Number of calls to LIB$FREE_VM_PAGE</td>
</tr>
<tr>
<td>7</td>
<td>Number of VAX pages or Alpha pagelets allocated by LIB$GET_VM_PAGE but not yet deallocated by LIB$FREE_VM_PAGE</td>
</tr>
</tbody>
</table>

Note that it is invalid to omit **code** or to give a **code** of 0 or 4.

**value-argument**

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

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$ Routines
LIB$STAT_VM

Value of the statistic returned by LIB$STAT_VM. The value-argument argument contains the address of an unsigned longword integer that is this value.

Description
LIB$STAT_VM returns to its caller one of six available statistics. Unlike LIB$SHOW_VM, which formats the values for output, LIB$STAT_VM returns the value to a location specified as an argument.

Only one of the six statistics can be returned by one call to LIB$STAT_VM. The argument code must be one of six values described for LIB$SHOW_VM. A code value of 0 or 4 is invalid.

Unlike LIB$SHOW_VM, which produces ASCII values for output, LIB$STAT_VM returns the value in binary form to a location specified as an argument.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument. The value of code was not one of the values allowed by LIB$STAT_VM.</td>
</tr>
</tbody>
</table>
LIB$STAT_VM_64 (Alpha and I64 Only)

Return Virtual Memory Statistics

The Return Virtual Memory Statistics routine returns to its caller one of six statistics available from calls to LIB$GET_VM_64 and LIB$FREE_VM_64, as well as LIB$GET_VM_PAGE_64 and LIB$FREE_VM_PAGE_64. Unlike LIB$SHOW_VM_64, which formats the values for output and displays them on SYS$OUTPUT, LIB$STAT_VM_64 returns the statistic in the value-argument argument. Only one of the statistics is returned by each call to LIB$STAT_VM_64.

Format

LIB$STAT_VM_64 code,value-argument

Returns

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

Arguments

code
OpenVMS usage: quadword_signed
type: quadword integer (signed)
access: read only
mechanism: by reference

Code specifying which statistic is to be returned. The code argument contains the address of a signed quadword integer that is this code.

<table>
<thead>
<tr>
<th>Code</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of successful calls to LIB$GET_VM_64</td>
</tr>
<tr>
<td>2</td>
<td>Number of successful calls to LIB$FREE_VM_64</td>
</tr>
<tr>
<td>3</td>
<td>Number of bytes allocated by LIB$GET_VM_64 but not yet deallocated by LIB$FREE_VM_64</td>
</tr>
<tr>
<td>5</td>
<td>Number of calls to LIB$GET_VM_PAGE_64</td>
</tr>
<tr>
<td>6</td>
<td>Number of calls to LIB$FREE_VM_PAGE_64</td>
</tr>
<tr>
<td>7</td>
<td>Number of Alpha or I64 pagelets allocated by LIB$GET_VM_PAGE_64 but not yet deallocated by LIB$FREE_VM_PAGE_64</td>
</tr>
</tbody>
</table>

Note that it is invalid to omit code or to give a code of 0 or 4.

value-argument
OpenVMS usage: user_arg
type: quadword (unsigned)
access: write only
mechanism: by reference
LIB$ Routines
LIB$STAT_VM_64 (Alpha and I64 Only)

Value of the statistic returned by LIB$STAT_VM_64. The value-argument argument contains the address of an unsigned quadword integer that is this value.

Description
LIB$STAT_VM_64 returns to its caller one of six available statistics. Unlike LIB$SHOW_VM_64, which formats the values for output, LIB$STAT_VM_64 returns the value to a location specified as an argument.

Only one of the six statistics can be returned by one call to LIB$STAT_VM_64. The code argument must be one of six values described for LIB$SHOW_VM_64. A value of 0 or 4 is invalid.

Unlike LIB$SHOW_VM_64, which produces ASCII values for output, LIB$STAT_VM_64 returns the value in binary form to a location specified as an argument.

Condition Values Returned

- SS$_NORMAL Routine successfully completed.
- LIB$_INVARG Invalid argument. The value of code was not one of the values allowed by LIB$STAT_VM_64.
LIB$STOP
Stop Execution and Signal the Condition

The Stop Execution and Signal the Condition routine generates a signal that indicates that an exception condition has occurred in your program. Exception conditions signaled by LIB$STOP cannot be continued from the point of the signal.

Format

LIB$STOP condition-value [,number-of-arguments] [,FAO-argument...] 

Returns

LIB$STOP generates a signal and stops execution of the calling program. No condition values are returned.

Arguments

c-condition-value
OpenVMS usage: cond_value
type: longword (unsigned)
access: read only
mechanism: by value

OpenVMS 32-bit condition value. The condition-value argument is an unsigned longword that contains this condition value.

The HP OpenVMS Programming Concepts Manual explains the format of a condition value.

number-of-arguments
OpenVMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by value

Number of FAO arguments associated with condition-value. The optional number-of-arguments argument is a signed longword integer that contains this number. If omitted or specified as zero, no FAO arguments follow.

FAO-argument
OpenVMS usage: varying_arg
type: unspecified
access: read only
mechanism: by value

Optional FAO (formatted ASCII output) argument that is associated with the specified condition value.

The HP OpenVMS Programming Concepts Manual explains the message format.
LIB$ Routines
LIB$STOP

Description
LIB$STOP is called whenever your program must indicate an exception condition because it is impossible to continue execution or return a status code to the calling program.

LIB$STOP scans the stack frame by frame, starting with the most recent frame, calling each established handler (see the HP OpenVMS Programming Concepts Manual). LIB$STOP guarantees that control will not return to the caller.

The LIB$STOP argument list, the Program Counter (PC) and Processor Status Longword (PSL on OpenVMS VAX systems, PS on OpenVMS Alpha and I64 systems) of the caller are appended to build the signal argument vector.

The severity of condition-value is forced to SEVERE before each call to a handler.

If any handler attempts to continue by returning a success completion code, the error message ATTEMPT TO CONTINUE FROM STOP is printed and your program exits.

If the handler called by LIB$STOP in turn calls system service $UNWIND, control will not return to LIB$STOP’s caller, thus changing the program flow. A handler can also modify the saved copy of R0/R1 in the mechanism vector, changing registers R0 and R1 after the stack has been unwound. If a handler does neither of these things, then all registers including R0/R1 and the hardware condition codes are preserved.¹

The only way a handler can prevent the image from exiting after a call to LIB$STOP is to unwind the stack using the $UNWIND system service.

Condition Values Returned

None.

Example

10 EXTERNAL LONG FUNCTION LIB$RESERVE_EF
DECLARE LONG RET_STATUS
RET_STATUS = LIB$RESERVE_EF( 2% )
IF (RET_STATUS AND 1%) = 0% THEN
    CALL LIB$STOP( RET_STATUS BY VALUE )
END IF
PRINT "Event flag 2 reserved successfully"
END

This BASIC example program uses LIB$STOP to stop executing if an error is signaled. This BASIC program tries to reserve an event flag that is not accessible to user programs, thus ensuring that an error will be signaled.

The output generated by this BASIC program is as follows:

%LIB-F-EF_ALRRES, event flag already reserved
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name routine name line rel PC abs PC
2822XBLST$MAIN 2822XBLST$MAIN 6 00000044 00000644

¹ On Alpha systems, OpenVMS Alpha instructions perform the equivalent operation.
LIB$SUBX
Multiple-Precision Binary Subtraction

The Multiple-Precision Binary Subtraction routine performs subtraction on signed two's complement integers of arbitrary length.

Format

LIB$SUBX  minuend-array ,subtrahend-array ,difference-array [,array-length]

Returns

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

Arguments

minuend-array
OpenVMS usage:  vector_longword_signed
type:  unspecified
access:  read only
mechanism:  by reference, array reference
Minuend; a multiple-precision, signed two's complement integer. The minuend-array argument is the address of an array of signed longword integers that contains the minuend.

subtrahend-array
OpenVMS usage:  vector_longword_signed
type:  unspecified
access:  read only
mechanism:  by reference, array reference
Subtrahend; a multiple-precision, signed two's complement integer. The subtrahend-array argument is the address of an array of signed longword integers that contains the subtrahend.

difference-array
OpenVMS usage:  vector_longword_signed
type:  unspecified
access:  write only
mechanism:  by reference, array reference
Difference; a multiple-precision, signed two's complement integer result. The difference-array argument is the address of an array of signed longword integers that contains the difference.

array-length
OpenVMS usage:  longword_signed
type:  longword integer (signed)
access:  read only
mechanism:  by reference
LIB$ Routines
LIB$SUBX

Length in longwords of the arrays to be operated on by LIB$SUBX. The array-length argument contains the address of a signed longword integer that is this length. The array-length argument must not be negative. The default length is 2 units.

Description
LIB$SUBX performs subtraction on signed two's complement integers of arbitrary length. The integers are located in arrays of longwords. The higher addresses contain the higher-precision parts of the values. The highest-addressed longword contains the sign and 31 bits of precision. The remaining longwords contain 32 bits of precision in each. The number of longwords to be operated on is given by the optional argument, array-length. The default length is 2, which corresponds to the OpenVMS quadword data type.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
SS$_INTOVF Integer overflow. The result is correct, except that the sign bit is lost.
LIB$_INVARG Invalid argument. Length is negative. The output array is unchanged.

Example

C+ C This Fortran example program demonstrates the use of LIB$SUBX.
C- INTEGER A(2),B(2),C(2),RETURN
C+ C Let "A" have the value 72057594037927937 = ’1000000000000001’x.
C Let "B" have the value 4294967295 = ’00000000FFFFFFFF’x.
C-
A(1) = ’00000001’x
A(2) = ’10000000’x
B(1) = ’FFFFFFFF’x
B(2) = ’00000000’x
C+
C Then "A" - "B" is 72057589742960642.
C-
RETURN = LIB$SUBX(A,B,C)
TYPE *,’
TYPE *,’Let A = 72057594037927937 and B = 4294967295.‘
TYPE *,’Then C = A - B = 72057589742960642.’
TYPE 2,C(2),C(1)
2 FORMAT(’72057589742960642 is represented as ’,1H’,28,28,3H’x.)
TYPE *, 51HThat is, C(2) = ’0FFFFFFF’x and C(1) = ’00000002’x.
END

This Fortran example demonstrates how to call LIB$SUBX. The output generated by this program is as follows:

Let A = 72057594037927937 and B = 4294967295.
Then C = A - B = 72057589742960642.
72057589742960642 is represented as ’ FFFFFFF 2’x.
That is, C(2) = ’0FFFFFFF’x and C(1) = ’00000002’x.

lib-558
LIB$SUB_TIMES
Subtract Two Quadword Times

The Subtract Two Quadword Times routine subtracts two OpenVMS internal-time-format times.

Format

LIB$SUB_TIMES time1 ,time2 ,resultant-time

Returns

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

Arguments

time1
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

First time, from which LIB$SUB_TIMES subtracts the second time. The time1 argument is the address of an unsigned quadword containing this time. The time1 argument must represent a later or equal time or a longer or equal time interval than time2. The time1 argument may be either absolute time or delta time as long as time2 is of the same type. If time1 and time2 are of different types, time1 must be the absolute time.

Second time, which LIB$SUB_TIMES subtracts from the first time. The time2 argument is the address of an unsigned quadword containing this time. The time2 argument must represent an earlier or equal time or a shorter or equal time interval than time1. The time2 argument may be either absolute time or delta time as long as time1 is of the same type. If time2 and time1 are of different types, time2 must be the delta time.

resultant-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: write only
mechanism: by reference

The result of subtracting time2 from time1. The resultant-time argument is the address of an unsigned quadword containing the result. If both time1 and time2 are delta times, then resultant-time is a delta time. If both time1 and time2 are absolute times, then resultant-time is a delta time. If time1 is an
absolute time and time2 is a delta time, then resultant-time is an absolute time.

Description

LIB$SUB_TIMES subtracts two OpenVMS internal times. The second time, specified by time2, is subtracted from time1. The following table shows the only combinations of times you can subtract:

<table>
<thead>
<tr>
<th>Time1</th>
<th>Time2</th>
<th>Subtraction</th>
<th>Resultant-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>delta</td>
<td>delta</td>
<td>time1 − time2</td>
<td>delta</td>
</tr>
<tr>
<td>absolute</td>
<td>absolute</td>
<td>time1 − time2</td>
<td>delta</td>
</tr>
<tr>
<td>absolute</td>
<td>delta</td>
<td>time1 − time2</td>
<td>absolute</td>
</tr>
</tbody>
</table>

Delta time values cannot be a zero and always reflect time in the future. Binary format number will always be negative. Therefore, if time1 and time2 are equal, resultant-time cannot be 0. Instead, resultant-time is represented by .1 of one microsecond (the smallest interval of time recognized by the OpenVMS operating system). This interval is shown as “0 00:00:00.00” when formatted by the standard techniques.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_INVARGORD</td>
<td>Invalid ordering of arguments.</td>
</tr>
<tr>
<td>LIB$_IVTIME</td>
<td>Invalid time.</td>
</tr>
<tr>
<td>LIB$_NEGTIM</td>
<td>Negative time computed.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Incorrect number of arguments.</td>
</tr>
</tbody>
</table>
LIB$SYS_ASCTIM
Invoke $ASCTIM to Convert Binary Time to ASCII String

The Invoke $ASCTIM to Convert Binary Time to ASCII String routine calls the system service $ASCTIM to convert a binary date and time value, returning the ASCII string using the semantics of the caller's string.

Format

LIB$SYS_ASCTIM [resultant-length],time-string [,user-time] [,flags]

Returns

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

Arguments

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

Number of bytes written into time-string, not counting padding in the case of a fixed-length string. The resultant-length argument contains the address of an unsigned word integer that is this number.

If the input string is truncated to the size specified in the time-string descriptor, resultant-length is set to this size. Therefore, resultant-length can always be used by the calling program to access a valid substring of time-string.

time-string
OpenVMS usage: time_name
type: character string
access: write only
mechanism: by descriptor

Destination string into which LIB$SYS_ASCTIM writes the ASCII time string. The time-string argument contains the address of a descriptor pointing to the destination string.

user-time
OpenVMS usage: date_time
type: quadword (unsigned)
access: read only
mechanism: by reference

Value that LIB$SYS_ASCTIM converts to ASCII string form. The user-time argument contains the address of a signed quadword integer that is this value.

If 0 or no address is specified, the current system date and time are returned. A positive value represents an absolute time. A negative value represents a delta time. Delta times must be less than 10,000 days.
LIB$ Routines
LIB$SYS_ASCTIM

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

Conversion indicator specifying which date and time fields LIB$SYS_ASCTIM
should return. The **flags** argument is the address of an unsigned bit mask that
contains this conversion indicator.

A value of 1 causes only the hour, minute, second, and hundredths of a second
to be returned, depending on the length of the buffer. A value of 0 (the default)
causes the full date and time to be returned, depending on the length of the
buffer.

The results of specifying some possible combinations for the values of the **flags**
and **time-string** arguments are shown below:

<table>
<thead>
<tr>
<th>Time Value</th>
<th>Time-String Length</th>
<th>Flags Value</th>
<th>Information Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>23</td>
<td>0</td>
<td>Date and time</td>
</tr>
<tr>
<td>Absolute</td>
<td>12</td>
<td>0</td>
<td>Date</td>
</tr>
<tr>
<td>Absolute</td>
<td>11</td>
<td>1</td>
<td>Time</td>
</tr>
<tr>
<td>Delta</td>
<td>16</td>
<td>0</td>
<td>Days and time</td>
</tr>
<tr>
<td>Delta</td>
<td>11</td>
<td>1</td>
<td>Time</td>
</tr>
</tbody>
</table>

The **flags** argument is passed to LIB$SYS_ASCTIM by reference and is changed
to value for use by $ASCTIM.

Description

See the *HP OpenVMS System Services Reference Manual: A–GETUAI* for a
complete description of $ASCTIM.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_IVTIME</td>
<td>The specified delta time is greater than or equal to 10,000 days.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to your HP support representative.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory. Your program has exceeded the image quota for virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed, but the source string was truncated on copy.</td>
</tr>
</tbody>
</table>
LIB$SYS_FAO
Invoke $FAO System Service to Format Output

The Invoke $FAO System Service to Format Output routine calls the $FAO system service, returning a string in the semantics you provide. If called with other than a fixed-length string for output, the length of the resultant string is limited to 256 bytes and truncation occurs.

Format

LIB$SYS_FAO  character-string, [resultant-length],resultant-string [,directive-argument ,...]

Returns

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

Arguments

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

ASCII control string, consisting of the fixed text of the output string and FAO directives. The character-string argument contains the address of a descriptor pointing to this control string.

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

Length of the output string. The resultant-length argument contains the address of an unsigned word integer that is this length.

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

Fully formatted output string returned by LIB$SYS_FAO. The resultant-string argument contains the address of a descriptor pointing to this output string.
**directive-argument**

OpenVMS usage: varying_arg  
type: unspecified  
access: read only  
mechanism: unspecified

Directive argument contained in longwords. Depending on the directive, a **directive-argument** argument can be a value to be converted, the address of the string to be inserted, or a length or argument count. The passing mechanism for each of these arguments should be the one expected by the $FAO system service.

**Description**

See the *HP OpenVMS System Services Reference Manual: A–GETUAI* for a complete description of $FAO.

**Condition Values Returned**

- **SS$_NORMAL**: Routine successfully completed.
- **SS$_BADPARAM**: An invalid directive was specified in the FAO control string.
- **SS$_BUFFEROVF**: Successfully completed, but the formatted output string overflowed the output buffer and was truncated.
- **LIB$_STRTRU**: Success, but the source string was truncated on copy.
- **LIB$_INSVIRMEM**: Insufficient virtual memory to allocate dynamic string.
- **LIB$_INVSTRDES**: Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.
LIB$SYS_FAOL
Invoke $FAOL System Service to Format Output

The Invoke $FAOL System Service to Format Output routine calls the $FAOL system service, returning the string in the semantics you provide. If called with other than a fixed-length string for output, the length of the resultant string is limited to 256 bytes and truncation occurs.

Format

LIB$SYS_FAOL character-string [,resultant-length] ,resultant-string ,directive-argument-address

Returns

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

Arguments

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

ASCII control string, consisting of the fixed text of the output string and FAO directives. The character-string argument contains the address of a descriptor pointing to this control string.

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

Length of the output string. The resultant-length argument contains the address of an unsigned word integer that is this length.

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

Fully formatted output string returned by LIB$SYS_FAOL. The resultant-string argument contains the address of a descriptor pointing to this output string.
**directive-argument-address**

OpenVMS usage: address

type: longword (unsigned)

access: read only

mechanism: unspecified

Directive arguments. The **directive-argument-address** arguments are contained in an array of unsigned longword directive arguments. Depending on the directive, a **directive-argument-address** argument can be a value to be converted, the address of the string to be inserted, or a length or argument count. The passing mechanism for each of these arguments should be the one expected by the $FAOL system service.

**Description**

See the *HP OpenVMS System Services Reference Manual: A–GETUAI* for a complete description of $FAOL.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ _NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$ _BADPARAM</td>
<td>An invalid directive was specified in the FAO control string.</td>
</tr>
<tr>
<td>SS$ _BUFFEROVF</td>
<td>Successfully completed, but the formatted output string overflowed the output buffer and was truncated.</td>
</tr>
<tr>
<td>LIB$ _INSVIRMEM</td>
<td>Insufficient virtual memory to allocate dynamic string.</td>
</tr>
<tr>
<td>LIB$ _INVSTRDES</td>
<td>Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.</td>
</tr>
<tr>
<td>LIB$ _STRTRU</td>
<td>Success, but the source string was truncated on copy.</td>
</tr>
</tbody>
</table>
LIB$SYS_FAOL_64 (Alpha and I64 Only)
Invoke $FAOL_64 System Service to Format Output

The Invoke $FAOL_64 System Service to Format Output routine calls the $FAOL_64 system service, returning the string in the semantics you provide. If called with other than a fixed-length string for output, the length of the resultant string is limited to 256 bytes and truncation occurs.

Format

LIB$SYS_FAOL_64  character-string [,resultant-length] ,resultant-string ,directive-argument-address

Returns

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

Arguments

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

ASCII control string, consisting of the fixed text of the output string and FAO directives. The character-string argument contains the address of a descriptor pointing to this control string.

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

Length of the output string. The resultant-length argument contains the address of an unsigned word integer that is this length.

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

Fully formatted output string returned by LIB$SYS_FAOL_64. The resultant-string argument contains the address of a descriptor pointing to this output string.
LIB$ Routines
LIBSYS_FAOL_64 (Alpha and I64 Only)

directive-argument-address
OpenVMS usage: address
   type: quadword (unsigned)
   access: read only
   mechanism: unspecified

Directive arguments. The directive-argument-address arguments are contained in an array of unsigned quadword directive arguments. Depending on the directive, a directive-argument-address argument can be a value to be converted, the address of the string to be inserted, or a length or argument count. The passing mechanism for each of these arguments should be the one expected by the $FAOL_64 system service.

Description
See the HP OpenVMS System Services Reference Manual: A–GETUA1 for a complete description of $FAOL_64.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
SS$_BADPARAM An invalid directive was specified in the FAO control string.
SS$_BUFFEROVF Successfully completed, but the formatted output string overflowed the output buffer and was truncated.
LIB$_INSVIRMEM Insufficient virtual memory to allocate dynamic string.
LIB$_INVSTRDES Invalid string descriptor. A string descriptor has an invalid value in its CLASS field.
LIB$_STRTRU Success, but the source string was truncated on copy.
LIB$SYS_GETMSG
Invoke $GETMSG System Service to Get Message Text

The Invoke $GETMSG System Service to Get Message Text routine calls the system service $GETMSG and returns a message string into destination-string using the semantics of the caller’s string.

Format

LIB$SYS_GETMSG  message-id [,message-length] ,destination-string [,flags] [,unsigned-resultant-array]

Returns

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

Arguments

message-id
OpenVMS usage:  identifier
type:  longword (unsigned)
access:  read only
mechanism:  by reference

Message identification to be retrieved by LIB$SYS_GETMSG. The message-id argument contains the address of an unsigned longword integer that is this message identification.

message-length
OpenVMS usage:  word_unsigned
type:  word integer (unsigned)
access:  write only
mechanism:  by reference

Number of characters written into destination-string, not counting padding in the case of a fixed-length string. The message-length argument contains the address of an unsigned word integer that is this number.

If the input string is truncated to the size specified in the destination-string descriptor, message-length is set to this size. Therefore, message-length can always be used by the calling program to access a valid substring of destination-string.

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

Destination string. The destination-string argument contains the address of a descriptor pointing to this destination string. LIB$SYS_GETMSG writes the message that has been returned by $GETMSG into destination-string.
LIB$ Routines
LIB$SYS_GETMSG

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

Four flag bits for message content. The flags argument is the address of an
unsigned longword that contains these flag bits. The default value is a longword
with bits 0 through 3 set to 1. The flags argument is passed to LIB$SYS_
GETMSG by reference and changed to value for use by $GETMSG.

The following table lists the bit numbers, their values, and corresponding
descriptions:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Include text of message.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Do not include text of message.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Include message identifier.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Do not include message identifier.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Include severity indicator.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Do not include severity indicator.</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Include facility name.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Do not include facility name.</td>
</tr>
</tbody>
</table>

unsigned-resultant-array
OpenVMS usage: unspecified
type: unspecified
access: write only
mechanism: by reference, array reference

A 4-byte array to receive message-specific information. The unsigned-resultant-
array argument contains the address of this array.

The contents of this 4-byte array are as follows:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Count of FAO arguments</td>
</tr>
<tr>
<td>2</td>
<td>User value</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Description

LIB$SYS_GETMSG calls the $GETMSG system service and returns a message
string using the semantics of the caller’s string. Note that, in order to retrieve a
message string for a LIB$ facility message, you must include the file $LIBDEF in
your program.

See the HP OpenVMS System Services Reference Manual: A–GETUAI for a more
complete description of $GETMSG.
### Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>SS$_BUFFEROVF</td>
<td>Successfully completed, but the resultant string overflowed the buffer provided and was truncated.</td>
</tr>
<tr>
<td>SS$_MSGNOTFND</td>
<td>Successfully completed, but the message code does not have an associated message on file.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Successfully completed, but the source string was truncated on copy.</td>
</tr>
<tr>
<td>LIB$_FATERRLIB</td>
<td>Fatal internal error.</td>
</tr>
<tr>
<td>LIB$_INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
</tbody>
</table>
Table-Driven Finite-State Parser

The Table-Driven Finite-State Parser routine is a general-purpose, table-driven parser implemented as a finite-state automaton, with extensions that make it suitable for a wide range of applications. It parses a string and returns a message indicating whether or not the input string is valid.

LIB$TABLE_PARSE is called with the address of an argument block, the address of a state table, and the address of a keyword table. The input string is specified as part of the argument block.

The LIB$ facility supports the following two versions of the Table-Driven Finite-State Parser:

LIB$TPARSE Available on VAX systems.

LIB$TPARSE is available on Alpha and I64 systems in translated form. In this form, it is applicable to translated VAX images only.

LIB$TABLE_PARSE Available on VAX, Alpha, and I64 systems.

LIB$TPARSE and LIB$TABLE_PARSE differ mainly in the way they pass arguments to action routines.

The term LIB$TABLE_PARSE is used here to describe concepts that apply to both LIB$TPARSE and LIB$TABLE_PARSE.

Format

LIB$TPARSE/LIB$TABLE_PARSE argument-block, state-table, key-table

Returns

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

Arguments

argument-block
   OpenVMS usage: unspecified
   type: unspecified
   access: modify
   mechanism: by reference

LIB$TABLE_PARSE argument block. The argument-block argument contains the address of this argument block.

The LIB$TABLE_PARSE argument block contains information about the state of the parse operation. It is a means of communication between LIB$TABLE_PARSE and the user’s program. It is passed as an argument to all action routines.

‡ No support for arguments passed by 64-bit address reference or the use of 64-bit descriptors is planned for LIB$TPARSE. On Alpha and I64 systems, LIB$TABLE_PARSE supports arguments passed by 64-bit address reference and the use of 64-bit descriptors.
You must declare and initialize the argument block. Section 1.4 describes the argument block in detail. Section 2.2 illustrates the coding for an argument block declaration and discusses its initialization.

LIB$T[ABLE_]PARSE supports the following argument blocks:

- A 32-bit argument block that accommodates longword addresses, values, and input tokens on VAX, Alpha, and I64 systems.
  On Alpha and I64 systems, this argument block also accommodates a numeric token whose binary representation is less than or equal to 2**64.
- A 64-bit argument block that accommodates quadword addresses, values, and input tokens on Alpha and I64 systems.

**state-table**
OpenVMS usage: unspecified
type: unspecified
access: read only
mechanism: by reference

Starting state in the state table. The **state-table** argument is the address of this starting state. Usually, the name appearing as the first argument of the $INIT_STATE macro is used.

You must define the state table for your parser. LIB$T[ABLE_]PARSE provides macros in the MACRO and BLISS languages for this purpose. Section 1.3 describes these macros.

**key-table**
OpenVMS usage: unspecified
type: unspecified
access: read only
mechanism: by reference

Keyword table. The **key-table** argument is the address of this keyword table. This name must be the same as that which appears as the second argument of the $INIT_STATE macro.

You must only assign a name to the keyword table. The LIB$T[ABLE_]PARSE macros allocate and define the table. See Section 4 for more information about the keyword table.

**Description**

The following sections explain in detail how LIB$T[ABLE_]PARSE works and how to call it from both the MACRO assembly language and high-level languages:

1. How LIB$T[ABLE_]PARSE Works — Describes the data structures used by LIB$T[ABLE_]PARSE and how LIB$T[ABLE_]PARSE operates on them.
2. Coding and Using a Simple State Table — Explains how to construct and use a simple state table.
3. Using Advanced LIB$T[ABLE_]PARSE Features — Explains how to use subexpressions, abbreviations, action routines, and other advanced features.
4. Data Representation — Includes information for the low-level-language programmer, such as the binary representation of state table data.
1 How LIB$T[ABLE_]PARSE Works

LIB$T[ABLE_]PARSE analyzes an input string according to a set of states and transitions presented in a state table you define. It determines whether the input string is valid according to the rules you define for the input language.

There are three parts to any parsing operation:

- The set of symbol types, or alphabet, from which you can choose the vocabulary of your language.
  
  You specify a symbol type for each transition you define. The symbol type specifies what constitutes a matching substring from the input string.
  
  LIB$T[ABLE_]PARSE recognizes the ASCII character set and provides symbolic names for the most common combinations of ASCII characters, such as alphabetic and alphanumeric strings, OpenVMS symbols, and numbers. See Section 1.2 for a list of the symbol types that comprise the LIB$T[ABLE_]PARSE alphabet.

- The rules that govern how the alphabet is used—in other words, the language's grammar.
  
  You specify the rules for a language in a state table. A LIB$T[ABLE_]PARSE state table lists the possible states for your language. Each state consists of a list of the transitions to other states and the operations to be performed when a transition is executed (see Section 1.3).

- The string to be parsed.
  
  The argument block specifies the input string. It also contains additional information about the state of the parse—how much of the string has not been interpreted, what the current token is, and so forth (see Section 1.4).

1.1 Overview

Before discussing the alphabet, the state table, and the argument block in detail, this section provides an overview of how these three parts work together.

1.1.1 Evaluating the Input String

LIB$T[ABLE_]PARSE evaluates the input string from left to right as it transitions from state to state. For a particular transition in a particular state, it evaluates the beginning of the unprocessed part of the input string against the symbol type you specify for the transition to determine whether there is a match.

LIB$T[ABLE_]PARSE compares each character of the remaining input string, from left to right, against the transition's symbol type until it encounters a character in the input string that does not match. It takes the substring that matches the symbol type and stores a pointer to it in the argument block as the current token. In this way, any character in the input string that does not belong to the symbol type's constituent character set effectively becomes a separator.

If LIB$T[ABLE_]PARSE finds a match, it executes the transition.

If the input string does not match, LIB$T[ABLE_]PARSE attempts to match the next transition. It performs the comparison using the transitions in the order in which you define them for the state.
1.1.2 Executing a Transition
When LIB$T[ABLE_]PARSE finds a match with a transition, it performs the following steps:

1. Stores a pointer to the current token in the argument block. If the token matches one of the numeric symbol types, it also stores the token’s binary representation in the argument block.

2. Calls the action routine, if any, specified by the transition and passes it the argument block and any additional user-specified arguments.
   You can use an action routine to reject a transition. In this case, LIB$T[ABLE_]PARSE performs none of the following steps. See Section 3.1 for more information.

3. Performs one of the following operations:
   • Stores the mask, if any, specified by the transition in the location specified by the transition.
   • Stores the value of token in the program location specified by the transition.

4. Transfers control to the specified state, if any, or to the next state in the state table.

1.1.3 Exiting LIB$T[ABLE_]PARSE
LIB$T[ABLE_]PARSE continues to match and execute transitions from state to state until one of the following occurs:

• For a valid match, it executes a user-specified transition to TPA$_EXIT$ at main level. It returns the value SS$_NORMAL$.

• A transition requests that LIB$T[ABLE_]PARSE consider the string invalid by specifying a transition to TPA$_FAIL$ at main level (rather than at the level of a subexpression). LIB$T[ABLE_]PARSE returns with the value LIB$_SYNTAXERR$.
   You can also request a transition to TPA$_FAIL$ from an action routine. The action routine can provide an alternate failure status.

• An error occurs at the main level. The error can be:
   – A syntax error. All transitions in the current state fail to match the remaining input string. LIB$T[ABLE_]PARSE returns LIB$_SYNTAXERR$ or an alternate failure status returned by an action routine.
   – A state table format error. One of your state table entries is invalid. LIB$T[ABLE_]PARSE returns LIB$_INVTYPE$.

   Note
LIB$T[ABLE_]PARSE generates no signals and establishes no condition handler; action routines can signal through LIB$T[ABLE_]PARSE back to the calling program.

When LIB$T[ABLE_]PARSE cannot successfully parse the entire string, it defines the current token, as follows, and stores it in the argument block before returning:

• If LIB$T[ABLE_]PARSE fails to match a transition in the current state, it attempts to define the current token as the beginning of the remaining
input string. You can incorporate this token in an error message or use it to
determine the logical flow of your program.

LIB$T[ABLE_]PARSE attempts to match the characters from the beginning
of the remaining input string, one at a time, against the TPA$_SYMBOL
alphabet symbol type until it encounters a character that does not match.
The TPA$_SYMBOL symbol type consists of all the characters of the standard
OpenVMS symbol constituent set.

- If LIB$T[ABLE_]PARSE successfully matches one or more consecutive
  characters from the input string against TPA$_SYMBOL, then the
  substring that matched TPA$_SYMBOL becomes the current token.
- If the first character of the remaining input string does not match TPA$_
  SYMBOL, the first character becomes the current token.

• If LIB$T[ABLE_]PARSE matches the symbol type for a transition that
  specifies TPA$_FAIL as the next state, it leaves the token that matched the
  transition as the current token.

1.2 Alphabet of LIB$T[ABLE_]PARSE
The LIB$T[ABLE_]PARSE alphabet consists of a set of symbol types defined in
Table lib–9. This alphabet includes strings made up of elements of the ASCII
character set. It provides all the basic building blocks needed for constructing
a grammar using the ASCII character set. The alphabet also includes symbol
types that represent the more complex constructions found in programming and
command language grammar.

Use the symbols types that comprise the LIB$T[ABLE_]PARSE alphabet to define
a vocabulary and grammar for your language. For each transition you define, you
specify one of the alphabet symbol types. LIB$T[ABLE_]PARSE compares the
characters at the beginning of the remaining input string with this symbol type of
each of the possible transitions. If LIB$T[ABLE_]PARSE finds a match, it enters
the state specified by that transition.

<table>
<thead>
<tr>
<th>Symbol Type</th>
<th>Characters Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>'x'</td>
<td>The particular ASCII character. In a state table, it is expressed by enclosing the character in single quotation marks. The character can be any member of the 8-bit ASCII code set. LIB$T[ABLE_]PARSE does not consider uppercase and lowercase alphabetic characters and codes with different values in bit 7 to be equivalent.</td>
</tr>
<tr>
<td>TPA$_ANY</td>
<td>Any single character.</td>
</tr>
<tr>
<td>TPA$_ALPHA</td>
<td>Any alphabetic character, which includes the DEC multinational character set.</td>
</tr>
<tr>
<td>TPA$_DIGIT</td>
<td>Any numeric character, that is, 0 through 9.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table lib–9 (Cont.) The Alphabet of LIB$TABLE\_PARSE

<table>
<thead>
<tr>
<th>Symbol Type</th>
<th>Characters Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$_STRING</td>
<td>Any string of one or more alphanumeric characters, that is, uppercase or lowercase A through Z, and the numeric characters 0 through 9. The string can be any length. It is bounded on the right by the first nonalphanumeric character or by the end of the string.</td>
</tr>
<tr>
<td>TPA$_SYMBOL</td>
<td>Any string of one or more through characters of the standard OpenVMS symbol constituent set, that is, uppercase and lowercase A through Z and all DEC multinational characters, in addition to the dollar sign ($), and the underscore (_). The string is bounded on the right by some character not in the symbol constituent set (usually a blank) or by the end of the string.</td>
</tr>
<tr>
<td>'keyword'</td>
<td>The string of characters enclosed in single quotation marks. A keyword can consist of one or more characters of the OpenVMS symbol constituent set, that is, uppercase and lowercase A through Z, the numeric characters 0 through 9, the dollar sign ($), and the underscore (_). Uppercase and lowercase alphabetics are treated as different characters. A state table can contain up to 220 keywords. The keyword is bounded on the right by a character not in the symbol constituent set or by the end of the string. Keywords that are one character in length are expressed in the form ‘x*’ to distinguish them from the single-character symbol (‘x’). They must be differentiated because they are not the same in operation. For example, in the input string AB+C, the single character ‘A’ would match the first character of this string, whereas the keyword ‘A*’ would not, because B in the string is in the symbol constituent set.</td>
</tr>
<tr>
<td>TPA$_BLANK</td>
<td>Any string of one or more blanks and/or tabs.</td>
</tr>
<tr>
<td>TPA$_OCTAL</td>
<td>Any octal number (that is, any string of one or more numeric characters 0 through 7) whose magnitude is less than $2^{32}$ for a 32-bit argument block or less than $2^{64}$ for a 64-bit argument block.</td>
</tr>
<tr>
<td>TPA$_DECIMAL</td>
<td>Any decimal number (that is, any string of one or more numeric characters 0 through 9) whose magnitude is less than $2^{32}$ for a 32-bit argument block or less than $2^{64}$ for a 64-bit argument block.</td>
</tr>
<tr>
<td>TPA$_HEX</td>
<td>Any hexadecimal number (that is, any string of one or more numeric characters 0 through 9, A through F) whose magnitude is less than $2^{32}$ for a 32-bit argument block or less than $2^{64}$ for a 64-bit argument block.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table lib–9 (Cont.)  The Alphabet of LIB$TABLEPARSE

<table>
<thead>
<tr>
<th>Symbol Type</th>
<th>Characters Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>‡TPA$_OCTAL_64</td>
<td>Any octal number (that is, any string of one or more numeric characters 0 through 7) whose magnitude is less than $2^{64}$.</td>
</tr>
<tr>
<td>‡TPA$_DECIMAL_64</td>
<td>Any decimal number (that is, any string of one or more numeric characters 0 through 9) whose magnitude is less than $2^{64}$.</td>
</tr>
<tr>
<td>‡TPA$_HEX_64</td>
<td>Any hexadecimal number (that is, any string of one or more numeric characters 0 through 9, A through F) whose magnitude is less than $2^{64}$.</td>
</tr>
<tr>
<td>TPA$_FILESPEC</td>
<td>Any string that constitutes a valid OpenVMS file specification. The string is bounded on the right by the first character that either is not a file specification constituent character or would cause the string to violate the syntax rules of a file specification.</td>
</tr>
<tr>
<td>TPA$_NODE</td>
<td>Matches a full node specification including the double colon (::).</td>
</tr>
<tr>
<td>TPA$_NODE_ACS</td>
<td>Matches a primary node specification including the access control string, if any, but not the double colon (::).</td>
</tr>
<tr>
<td>TPA$_NODE_PRIMARY</td>
<td>Matches a primary node specification excluding both the access control string, if any, and the double colon (::).</td>
</tr>
<tr>
<td>TPA$_UIC</td>
<td>Any string that constitutes a valid OpenVMS numerical UIC specification, bounded by square brackets or angle brackets. The binary value of the UIC, converted in octal radix, is placed in the argument block. The wildcard character (* ) is permitted in the group and/or member fields; its presence results in that field being set to its largest possible value in the binary representation.</td>
</tr>
</tbody>
</table>
| TPA$_IDENT                       | Any string that constitutes a valid OpenVMS identifier. Identifiers may be given as numerical UICs according to the rules for TPA$_UIC, or as alphabetic identifier names that appear in the system's rights database. The binary value of the identifier, converted in either octal or hexadecimal radix or by lookup in the system rights database, is placed in the argument block. Identifiers can be entered in any of the following forms:  

\[
\begin{align*}
[n,m] & \quad <n,m> \\
[name1,name2] & \quad <name1,name2> \\
[name] & \quad <name> \\
\text{name} & \quad %X\text{hex-value}
\end{align*}
\]

You can use a wildcard (*) in place of any occurrence of *number* or *name* in an identifier form.

‡Alpha and I64 specific

(continued on next page)
Table lib–9 (Cont.)  The Alphabet of LIB$TABLEPARSE

<table>
<thead>
<tr>
<th>Symbol Type</th>
<th>Characters Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$_LAMBDA</td>
<td>The empty string (always matches). As it executes the transition, LIB$TABLEPARSE does not remove any characters from the input string. LAMBDA transitions are useful in getting action routines called under otherwise awkward circumstances, providing unconditional GOTOs to link portions of a state table together, and providing default actions in certain cases.</td>
</tr>
<tr>
<td>TPA$_EOS</td>
<td>The end of the input string.</td>
</tr>
<tr>
<td>state label</td>
<td>The label of a state that functions as a subexpression. A subexpression is analogous to a subroutine within the state table. The subexpression facility permits complex syntactic constructs that appear in many places in grammar to appear only once in the state table. It also permits a degree of nondeterministic or pushdown parsing with a parser that is otherwise deterministic and finite-state. See Section 3.5 for detailed information about subexpressions and examples of their use.</td>
</tr>
</tbody>
</table>

Note

By default, LIB$TABLEPARSE treats blanks (defined to be either spaces or tabs), as though they belong to no symbol type constituent set. Effectively, this makes the blank a separator. LIB$TABLEPARSE begins its next comparison with the first nonblank character following the blanks. To have LIB$TABLEPARSE evaluate a blank as it would any other character in the input string, set the TPA$V_BLANKS flag in the argument block. Section 3.2 provides an example of the use of this flag.

1.3 State Tables

This section describes state table generation and the macros used to construct state tables. Section 2 explains how to use these macros.

The state table must be set up using either MACRO or BLISS. Everything else, including any action routines, can be coded in the language of your choice. Simply compile the state table separately, then link it with your program.

The body of the state table consists of one or more states, each of which defines one or more transitions to the same or other states. The order of the states and the order of the transitions for each state are important:

- If a transition does not specify a target state, LIB$TABLEPARSE transitions to the next state after the current state in the state table.
- For a given state, LIB$TABLEPARSE evaluates the input string against the transitions in the order in which they are defined and executes the first transition it matches.
  - If a state defines more than one transition with symbol types that match overlapping sets of tokens, the order of transition definitions within the state is significant. For example, the characters 123 followed by a comma...
(, ) could match TPA$_{DECIMAL}$, TPA$_{OCTAL}$, TPA$_{STRING}$, or one of several other symbol types.

– It is best to order transitions in order of increasing generality of their symbol types. For example, the TPA$_{SYMBOL}$ symbol type matches all keyword strings. In general, LIB$T[ABLE_\_PARSE$ never executes a keyword transition that follows a TPA$_{SYMBOL}$ transition. The symbol types, in order of increasing generality, are as follows:

```latex
\texttt{\{keyword\}}
\texttt{\{x\}}
TPA$_{EOS}$
TPA$_{ALPHA}$
TPA$_{DIGIT}$
TPA$_{BLANK}$
TPA$_{OCTAL}$
TPA$_{OCTAL\_64}$ (Alpha and I64 only)
TPA$_{DECIMAL}$
TPA$_{DECIMAL\_64}$ (Alpha and I64 only)
TPA$_{HEX}$
TPA$_{HEX\_64}$ (Alpha and I64 only)
TPA$_{STRING}$
TPA$_{SYMBOL}$
TPA$_{UIC}$
TPA$_{IDENT}$
TPA$_{NODE\_PRIMARY}$
TPA$_{NODE\_ACS}$
TPA$_{NODE}$
TPA$_{FILESPEC}$
TPA$_{ANY}$
TPA$_{LAMBDA}$
```

The list of symbol types does not include subexpression calls, because the generality of these calls depends on the symbol types recognized within the subexpression. If you use action routines to reject certain transitions, you can change the order in which that symbol type is placed in this order. In any case, LIB$T[ABLE_\_PARSE$ executes the first transition listed in a state that you permit to match the leftmost portion of the remaining input string.

### Note

1.3.1 MACRO State Table Generation Macro Calls

The OpenVMS system MACRO library contains a set of assembler macros that allow convenient and readable coding of a LIB$T[ABLE_\_PARSE$ state table. These macros generate symbol definitions and tables. They do not produce any executable code or routine calls.

There are four MACRO state table generation macros:

- **$\text{INIT\_STATE}$**—Initializes the LIB$T[ABLE_\_PARSE$ macros and declares the beginning of a state table (see Section 1.3.1.1)
- **$\text{STATE}$**—Defines a state (see Section 1.3.1.2)
- **$\text{TRAN}$**—Defines a state transition (see Section 1.3.1.3)
• $END_STATE—Ends the state table (see Section 1.3.1.4)

A state table begins with a call to $INIT_STATE and ends with a call to $END_STATE. Within the state table, define each state by a call to $STATE immediately followed by as many calls to $TRAN as you need to define the transitions from that state.

1.3.1.1 $INIT_STATE—Initializes the LIB$TABLE_PARSE Macros

The $INIT_STATE macro declares the beginning of a state table. It initializes the internals of the table generator macros and declares the locations of the state table and the keyword table:

• The state table is the structure containing the definitions of the states and the transitions between them. LIB$TABLE_PARSE builds the state table as it processes the $STATE and $TRAN macros you use to define the table.

• The keyword table contains the text of the keywords used in the state table. LIB$TABLE_PARSE builds the keyword table as it processes the calls to $TRAN for each state.

Section 4 provides specific information on the allocation and binary representations of the state table and the keyword table. This information may be useful in debugging your program.

$INIT_STATE state-table ,key-table

state-table
The name assigned to the state table. LIB$TABLE_PARSE equates this label to the start of the first state in the state table.

key-table
The name assigned to the keyword table. LIB$TABLE_PARSE equates this label to the start of the keyword table.

You must supply both the address of the state table and the address of the keyword table in the call to LIB$TABLE_PARSE to perform a parse. The $INIT_STATE macro can appear more than once in a program. Each occurrence defines a separate state table. No part of any state table can refer to part of any other state table.

1.3.1.2 $STATE—Defines a State

The $STATE macro declares the beginning of a state.

$STATE [label]

label
An optional label for the state. LIB$TABLE_PARSE equates the label, if present, to the starting address of the state.

1.3.1.3 $TRAN—Defines a State Transition

The $TRAN macro defines a transition from the state in which it is defined to some other (or to the same) state. The arguments of the macro define, among other things, the symbol type that causes the transition to be executed, the state to which to transfer, and the action routine to call, if any. The transition defined by a $TRAN macro belongs to the state defined by the last preceding $STATE macro.

**LIB$ Routines**

**LIB$TPARSE/LIB$TABLE_PARSE**

**type**
The symbol type, taken from the LIB$T[ABLE_]PARSE alphabet, that is recognized by this transition. The transition is taken if the characters from the beginning of the remaining input string match the specified symbol type.

If the transition calls a subexpression to determine a match, the symbol type syntax includes the state label of the subexpression to be called. It is indicated with the MACRO expression `!label`. See Section 3.5 for information about subexpressions.

**label**
The optional target state of this transition. If present, it must be the label assigned to some state in the state table. If no label argument is present, LIB$T[ABLE_]PARSE transfers control to the state immediately following the current state in the state table.

LIB$T[ABLE_]PARSE defines two expressions you can also specify as the target state in the label argument:

- **TPA$_EXIT** — The parsing operation in progress terminates with a success status.
- **TPA$_FAIL** — The parsing operation stops with a failure status, as if a syntax error had occurred.

**action**
The optional address of a user-supplied action routine. If this argument is present, LIB$T[ABLE_]PARSE calls the named action routine before it executes the transition. Section 3.1 describes the calling sequence of action routines and the information available to them.

Because the action routine address is self-relative, it cannot be in a shared image separate from the state table.

**mask**
An optional 32-bit mask value used with the msk-adr argument.

When LIB$T[ABLE_]PARSE executes the transition, it performs an inclusive OR operation using the mask value and the contents of msk-adr and stores the result in msk-adr.

You can associate one or more bits in mask with a particular transition and set those bits. When LIB$T[ABLE_]PARSE returns, you can check the bits in msk-adr to determine which transitions were executed. You can also use an action routine to check the bit and ensure that a transition is executed only once.

If the mask argument is present, the msk-adr argument must also be present.

**msk-adr**
The msk-adr argument provides two mutually exclusive capabilities depending on whether the mask argument is present:

- If mask is present, msk-adr is the address of a longword associated with the preceding mask argument. LIB$T[ABLE_]PARSE performs the inclusive OR operation on the contents of this address and the mask argument and stores the result in msk-adr.

  Initialize the contents of msk-adr to zero before calling LIB$T[ABLE_]PARSE.
If *mask* is not present, you can use *msk-adr* to specify the address of a location where LIB$[ABLE_]PARSE stores information about the matching token. No OR operation is performed. This capability lets a program extract the most commonly needed information from the input string without using action routines.

The kind of information that LIB$[ABLE_]PARSE stores in the location you specify as the *msk-adr* argument depends on the symbol type specified for the *type* argument and on the argument block, as follows:

- If the symbol type is TPA$_DECIMAL$, TPA$_OCTAL$, or TPA$_HEX$, LIB$[ABLE_]PARSE stores the binary representation of the matching number as an unsigned longword for a 32-bit argument block and as an unsigned quadword for a 64-bit argument block.

- If the symbol type is TPA$_DECIMAL_64$, TPA$_OCTAL_64$, or TPA$_HEX_64$, LIB$[ABLE_]PARSE stores the binary representation of the matching number as an unsigned quadword for both 32-bit and 64-bit argument blocks.

- If the symbol type is ‘*x*’, TPA$_ANY$, TPA$_ALPHA$, or TPA$_DIGIT$, LIB$[ABLE_]PARSE stores the 8-bit matching character as an unsigned byte.

- If the symbol is of any other type, you must specify *msk-adr* as the address of a 32-bit or 64-bit string descriptor, as appropriate, that you allocate in your program. LIB$[ABLE_]PARSE assumes a 32-bit or 64-bit descriptor if the argument block with which you called it is 32-bit or 64-bit, respectively.

For a 32-bit descriptor, LIB$[ABLE_]PARSE stores the length of the token in the first 32 bits (longword) of the descriptor. It stores a pointer to the token in the second longword. This pointer is the address of the token in the input string.

For a 64-bit descriptor, LIB$[ABLE_]PARSE stores the length of the token in the second quadword of the descriptor and stores the address of the token in the input string in the third quadword. On entry, LIB$[ABLE_]PARSE writes the fields of the first quadword as follows:

```plaintext
DSC64$B_CLASS = DSC64$K_CLASS_S
DSC64$B_DTYPE = DSC64$K_DTYPE_T
DSC64$L_MBMO = -1
DSC64$W_MBO = +1
```

Using *msk-adr* makes your parsing program nonmodular. The resulting program, which contains this state table, includes code that is not position independent.

Because the address specified by *msk-adr* is self-relative, it cannot be in a shared image separate from the state table.

### Argument

An optional 32-bit value that LIB$[ABLE_]PARSE passes to the action routine without interpretation. This argument can be an identifier number, an address, or any other information your action routine needs. It allows a single action routine to serve many transitions for which similar, but slightly varying, actions must be performed.
Because LIB$T[ABLE_]PARSE does not know the form or meaning of argument the value is stored in its absolute form. If you use argument to pass an address, you must store the address in its absolute form rather than as a self-relative pointer. In this case the resulting program, which contains this state table, is nonmodular. That is, it includes code that is not position independent.

1.3.1.4 $END_STATE—Ends the State Table
The $END_STATE macro declares the end of the state table. It is mandatory, in order to permit the orderly cleanup of the LIB$T[ABLE_]PARSE macro system. The $END_STATE macro has no arguments. You code it as follows:

$END_STATE

1.3.2 BLISS State Table Generation Macro Calls
The SYS$LIBRARY:TPAMAC.L32 and SYS$LIBRARY:TPAMAC.L64 files each contain a set of BLISS macros that allow convenient and readable coding of LIB$T[ABLE_]PARSE state tables in BLISS.

Use one of the following BLISS state table generation macros:

• $INIT_STATE—Initializes the macros (see Section 1.3.2.1)
• $STATE—Defines a state and its transitions (see Section 1.3.2.2)

To make the macros available to the program, include the following declaration in the module containing the state tables:

LIBRARY 'SYS$LIBRARY:TPAMAC';

The BLISS compiler you use, BLISS-32 or BLISS-64, chooses the corresponding SYS$LIBRARY:TPAMAC file.

The BLISS table generation macros contain no BEGIN or END statements. This allows $STATE macros to refer to each other. They generate all storage with OWN declarations. This means that the macros modify PSECT declarations for OWN and GLOBAL storage. Thus if other data declarations follow the state table declarations, they may not have the correct attributes. You cannot simply surround the state table with BEGIN/END, because this constitutes an expression. No declarations of any kind, including ROUTINE declarations, can follow an expression.

Use one of the following techniques to include LIB$T[ABLE_]PARSE a state table in a BLISS module:

• Follow the state table with explicit redeclarations of the OWN and GLOBAL PSECTs. Example 3 illustrates this technique.
• Place the state table in a separate module. The high-level language examples in the next section use this technique.
• Place the state table between BEGIN and END statements after the declarations within a routine body.
• Place the state table between BEGIN and END statements at the end of a module.

In all cases you must define all action routines, masks, addresses, and arguments with suitable declarations (which can be FORWARD or EXTERNAL). The LIB$T[ABLE_]PARSE macros handle the necessary FORWARD declarations for forward references to labels within the state table.
1.3.2.1 $\text{INIT\_STATE}—\text{Initializes the LIB\$T[ABLE\_]PARSE Macros}$

The $\text{INIT\_STATE}$ macro initializes the LIB$\$T[ABLE\_]PARSE macro system in the same manner it does for MACRO.

$$\text{INIT\_STATE} \ (\text{state-table, key-table});$$

**state-table**

The name assigned to the state table. LIB$\$T[ABLE\_]PARSE equates this label to the start of the first state in the state table.

**key-table**

The name assigned to the keyword table. LIB$\$T[ABLE\_]PARSE equates this label to the start of the keyword table.

Both names are declared as global vectors of length zero. As with the MACRO state table generation macros, you can invoke $\text{INIT\_STATE}$ more than once to declare several state tables within a single module.

1.3.2.2 $\text{STATE}—\text{Declares a State and Its Transitions}$

In BLISS, you use the $\text{STATE}$ macro to declare a state in its entirety, including its transitions.

$$\text{STATE} \ ((\text{label}), \ (\text{transition}), \ (\text{transition}), \ (\text{transition}) \ . \ . \ . \ . \ );$$

**label**

Optional address of the start of the state. The compiler declares `label` as a local vector of length zero. Note that the comma following the optional label is mandatory.

**transition**

Each transition appears within parentheses in the same form as the transition argument list for the MACRO $\text{TRAN}$ macro.

`type[,label][,action][,mask][,msk-adr][,argument]`

The arguments of each transition are expressed in exactly the same format as in the MACRO macros, with the exception of the subexpression symbol type. In BLISS, this symbol type has the form (label).

Note that the transitions are not specified as keyword macros. Therefore, you must use commas to indicate arguments you have skipped.

1.4 LIB$\$T[ABLE\_]PARSE Argument Block

LIB$\$T[ABLE\_]PARSE finds the input string through the argument block. This argument block is the impure database upon which LIB$\$T[ABLE\_]PARSE operates. That is, it is a set of variable data that can be written as well as read. It contains information about the string to be parsed, option flags for LIB$\$T[ABLE\_]PARSE, and data about the current token. If LIB$\$T[ABLE\_]PARSE calls an action routine, it passes the argument block to the action routine. This permits the action routine efficient reference to relevant data.
1.4.1 Choosing an Argument Block
LIB$TABLEPARSE provides an argument block for 32-bit operations on VAX, Alpha, and I64 systems. It also provides an argument block for 64-bit operations on Alpha and I64 systems.

1.4.1.1 32-Bit Argument Block
The 32-bit LIB$TABLEPARSE argument block accommodates longword addresses and values as well as input tokens whose binary representations require no more than 32 bits.

On Alpha and I64 systems, the LIB$TABLEPARSE 32-bit argument block can also accommodate a numeric input token whose binary representation requires up to 64 bits.

LIB$TABLEPARSE defines the first 9 longwords of the 32-bit argument block as shown in Figure lib–20. You must pass an argument block of at least this length as the first argument to LIB$TABLEPARSE. You can add fields to the end of the argument block as a means of passing user-defined data to action routines.

The TPA$K_LENGTH0 symbol represents the number of bytes (36) in the basic 32-bit argument block. You can use this symbol to determine the start of any user-defined fields you add to the argument block.

Table lib–10 describes the argument block fields.

Figure lib–20 LIB$TABLEPARSE 32-Bit Argument Block

1.4.1.2 64-Bit Argument Block (Alpha Only)
The 64-bit LIB$TABLEPARSE argument block accommodates quadword addresses and values as well as input tokens whose binary representations require no more than 64 bits.

LIB$TABLEPARSE defines the first 10 words of the 64-bit argument block as shown in Figure lib–21. You can add fields to the end of the argument block as a means of passing data to action routines.
The TPA64$K_LENGTH0 symbol represents the number of bytes (80) in the basic 64-bit argument block. You can use this symbol to determine the start of any user-defined fields you add to the argument block.

Table lib–10 describes the argument block fields.

**Figure lib–21 LIB$TABLE_PARSE 64-Bit Argument Block (Alpha and i64 Only)**

<table>
<thead>
<tr>
<th>TPAS$L_OPTIONS</th>
<th>TPA64$S_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA64$Q_STRINGDESC</td>
<td></td>
</tr>
<tr>
<td>TPA64$Q_STRINGCNT</td>
<td></td>
</tr>
<tr>
<td>TPA64$Q_STRINGPTR</td>
<td></td>
</tr>
<tr>
<td>TPA64$Q_TOKENDESC</td>
<td></td>
</tr>
<tr>
<td>TPA64$Q_TOKENCNT</td>
<td></td>
</tr>
<tr>
<td>TPA64$Q_TOKENPTR</td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td>TPA64$S_CHAR</td>
</tr>
<tr>
<td>TPA64$Q_NUMBER</td>
<td></td>
</tr>
<tr>
<td>TPA64$Q_PARAM</td>
<td></td>
</tr>
</tbody>
</table>

User defined fields

1.4.2 **Symbolic Names for Argument Block Fields**

The fields in each type of argument block have symbolic names. Figure lib–20 and Figure lib–21 show some of these symbolic names. This section tells you how to access these names in some of the most commonly used languages:

- **MACRO assembly language** — MACRO language programs can define both the 32-bit and 64-bit argument block names by invoking the macro $TPADEF (automatically loaded from the system macro library). The field names define the byte offset of the field from the start of the argument block. This includes the bit fields ($V_names). In addition, bit mask values ($M_names) are available for the bit fields.

- **BLISS** — The field names are also available to BLISS programs from the system macro SYS$LIBRARY:STARLET.L32 and SYS$LIBRARY:STARLET.L64 libraries. Each name (except for the $M_names) is defined as a fixed-reference macro that operates on a byte-based block. The $M_names are defined as literals.

- **C** — The same field names are available to C programs from the `tpadef.h` file. For the 32-bit and 64-bit argument blocks, the names are defined as elements of the `tpadef` and `tpa64def` structures, respectively.

See Section 2.2 for an example of an argument block declaration.
1.4.3 32-Bit and 64-Bit Argument Block Fields

Table lib–10 describes the fields of the 32-bit and 64-bit argument blocks.

Note that most fields have two symbols and one description. The symbol that begins with the prefix TPA$ is used with a 32-bit argument block, while the symbol that begins with the prefix TPA64$ is used with a 64-bit argument block. To prevent cumbersome explanations, Table lib–10 uses only the main part of a field name, without the prefix used in the actual code, when referring to a field for both the 32-bit and 64-bit argument blocks. For example, the options field is referred to as OPTIONS rather than specifying both TPA$L_OPTIONS and TPA64$L_OPTIONS. The complete field name is used only when referring to a field for one particular form of argument block.

Table lib–10 LIB$TABLE_PARSE Argument Block Fields

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$L_COUNT</td>
<td>A longword containing the value of TPA$K_COUNT0 for 32-bit argument blocks or TPA64$K_COUNT0 for 64-bit argument blocks. TPA$K_COUNT0 is defined to be 8. TPA64$K_COUNT0 is defined to be –1. If the value contained in this longword is greater than or equal to 8, LIB$TABLE_PARSE treats the argument block as a 32-bit argument block. If the value is –1, LIB$TABLE_PARSE treats the argument block as a 64-bit argument block. For LIB$TPARSE (VAX only), a longword containing the number of longwords that make up the rest of the argument block. This longword functions as the argument count when the argument block becomes the argument list to an action routine. This field must contain a value that is greater than or equal to the value of TPA$K_COUNT0, whose numeric value is 8.</td>
</tr>
<tr>
<td>TPA64$L_COUNT</td>
<td></td>
</tr>
</tbody>
</table>

(continued on next page)
Table lib–10 (Cont.)  LIB$TABLEPARSE Argument Block Fields

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$L_OPTIONS</td>
<td>Contains various flag bits and other options. The defined flags are as follows:</td>
</tr>
<tr>
<td>TPA64$L_OPTIONS</td>
<td>Contains various flag bits and other options. The defined flags are as follows:</td>
</tr>
</tbody>
</table>

- **TPA$V_BLANKS, TPA64$V_BLANKS** — Setting this bit causes LIB$TABLEPARSE to process blanks and tabs explicitly, rather than treating them as separators. See Section 3.2 for information about processing blanks.
- **TPA$V_ABBRFM, TPA64$V_ABBRFM** — Setting this bit allows keywords to be abbreviated to any length. If an abbreviated keyword string is ambiguous, the first eligible transition listed in the state matches it.
- **TPA$V_ABBREV, TPA64$V_ABBREV** — Setting this bit allows keywords to be abbreviated to the shortest length that is unambiguous in that state. See the Abbreviating Keywords section.
- **TPA$V_AMBIG, TPA64$V_AMBIG** — LIB$TABLEPARSE sets this bit when it has detected an ambiguous keyword string in the current state.

The OPTIONS field also contains the following option:

- **TPA$B_MCOUNT, TPA64$B_MCOUNT** — This byte contains the minimum number of characters allowed for the abbreviation of a keyword. If its value is zero, abbreviations are not allowed. Preventing ambiguity is the responsibility of the state table designer. If the ABBRFM or ABBREV flag is set, LIB$TABLEPARSE ignores MCOUNT. MCOUNT is the high byte of the OPTIONS field.

TPA64$Q_STRINGDESC | For a 64-bit argument block, the three quadwords starting with TPA64$Q_STRINGDESC form an embedded 64-bit descriptor for the input string. On entry, LIB$TABLEPARSE writes the fields of TPA64$Q_STRINGDESC as follows:

- **DSC64$B_CLASS = DSC64$K_CLASS_S**
- **DSC64$B_DTYPE = DSC64$K_DTYPE_T**
- **DSC64$L_MBMO = –1**
- **DSC64$W_MBO = +1**

---

1 LIB$TABLEPARSE defines bit masks TPA$M_BLANKS, TPA$M_ABBRFM, TPA$M_ABBREV, and TPA$M_AMBIG for use by languages such as MACRO. These bit masks correspond to the location of the $V_ fields in the OPTIONS field.

2 See the HP OpenVMS Calling Standard manual for information about string descriptor fields.
Table lib–10 (Cont.) LIB$T[ABLE_]PARSE Argument Block Fields

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$L_STRINGCNT</td>
<td>Contains the number of characters remaining in the input string.</td>
</tr>
<tr>
<td>TPA64$Q_STRINGCNT</td>
<td>For a 32-bit argument block, TPA$L_STRINGCNT and TPA$L_STRINGPTR form an embedded 32-bit descriptor for the input string.²</td>
</tr>
<tr>
<td></td>
<td>For both 32-bit and 64-bit argument blocks:</td>
</tr>
<tr>
<td></td>
<td>• You must initialize the STRINGCNT and STRINGPTR fields to describe the input string. Use LIB$ANALYZE_SDESC or LIB$ANALYZE_SDESC_64 to read the string length and address from the string's descriptor and write them in STRINGCNT and STRINGPTR, respectively.</td>
</tr>
<tr>
<td></td>
<td>• Before LIB$T[ABLE_]PARSE calls an action routine, it modifies STRINGCNT and STRINGPTR to describe the remainder of the input string.</td>
</tr>
<tr>
<td></td>
<td>• When LIB$T[ABLE_]PARSE returns, STRINGCNT and STRINGPTR describe the portion of the input string that LIB$T[ABLE_]PARSE did not process. This occurs whether LIB$T[ABLE_]PARSE returns success or failure.</td>
</tr>
<tr>
<td>TPA$L_STRINGPTR</td>
<td>Contains the address of the remainder of the string being parsed.</td>
</tr>
<tr>
<td>TPA64$Q_STRINGPTR</td>
<td>For a 64-bit argument block, the three quadwords starting with TPA64$Q_TOKENDESC form an embedded 64-bit descriptor for the current token.²</td>
</tr>
<tr>
<td>TPA64$Q_TOKENDESC</td>
<td>On entry, LIB$T[ABLE_]PARSE writes the fields of TPA64$Q_TOKENDESC as follows:</td>
</tr>
<tr>
<td></td>
<td>DSC64$B_CLASS = DSC64$K_CLASS_S</td>
</tr>
<tr>
<td></td>
<td>DSC64$B_DTYPE = DSC64$K_DTYPE_T</td>
</tr>
<tr>
<td></td>
<td>DSC64$L_MBMO = –1</td>
</tr>
<tr>
<td></td>
<td>DSC64$W_MBO = +1</td>
</tr>
<tr>
<td>TPA$L_TOKENCNT</td>
<td>Contains the number of characters in the current token.</td>
</tr>
<tr>
<td>TPA64$Q_TOKENCNT</td>
<td>For a 32-bit argument block, TPA$L_TOKENCNT and TPA$L_TOKENPTR form an embedded 32-bit descriptor for the input token.²</td>
</tr>
</tbody>
</table>

²See the HP OpenVMS Calling Standard manual for information about string descriptor fields.

(continued on next page)
Table lib–10 (Cont.) LIB$[ABLE_]PARSE Argument Block Fields

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$L_TOKENPTR</td>
<td>Contains the address of the current token.</td>
</tr>
<tr>
<td>TPA64$Q_TOKENPTR</td>
<td>Contains the address of the current token.</td>
</tr>
<tr>
<td>TPA$B_CHAR3</td>
<td>Contains the character matched by one of the single-character symbol types: 'x', TPA$_ANY, TPA$_ALPHA, or TPA$_DIGIT.</td>
</tr>
<tr>
<td>TPA64$B_CHAR3</td>
<td>Contains the character matched by one of the single-character symbol types: 'x', TPA$_ANY, TPA$_ALPHA, or TPA$_DIGIT.</td>
</tr>
<tr>
<td>TPA$L_NUMBER3</td>
<td>Contains the binary representation of a numeric token that matches TPA$_OCTAL, TPA$_DECIMAL, TPA$_HEX, TPA$_UIC, or TPA$_IDENT. For a 64-bit argument block, it can also contain the binary representation of a numeric token that matches TPA$_DECIMAL_64, TPA$_OCTAL_64, or TPA$_HEX_64.</td>
</tr>
<tr>
<td>TPA64$Q_NUMBER3</td>
<td>For a 32-bit argument block on an Alpha system, contains the binary representation of a numeric token that matches TPA$_DECIMAL_64, TPA$_OCTAL_64, or TPA$<em>HEX_64. LIB$[ABLE</em>]PARSE converts the numeric token in the appropriate radix before storing it in the TPA$Q_NUMBER field. In the 32-bit argument block, TPA$Q_NUMBER overlays TPA$L_NUMBER and the longword in which TPA$B_CHAR resides.</td>
</tr>
<tr>
<td>(Alpha and I64 specific) TPA$Q_NUMBER3</td>
<td>For a 32-bit argument block on an Alpha system, contains the binary representation of a numeric token that matches TPA$_DECIMAL_64, TPA$_OCTAL_64, or TPA$<em>HEX_64. LIB$[ABLE</em>]PARSE converts the numeric token in the appropriate radix before storing it in the TPA$Q_NUMBER field. In the 32-bit argument block, TPA$Q_NUMBER overlays TPA$L_NUMBER and the longword in which TPA$B_CHAR resides.</td>
</tr>
<tr>
<td>TPA$L_PARAM</td>
<td>Contains the optional 32-bit argument supplied by the state transition in its argument.</td>
</tr>
<tr>
<td>TPA64$Q_PARAM</td>
<td>For a 64-bit argument block, LIB$[ABLE_]PARSE sign-extends the argument value before storing it in TPA64$Q_PARAM.</td>
</tr>
</tbody>
</table>

LIB$[ABLE_]PARSE modifies TPA$Q_NUMBER prior to calling an action routine from a transition whose symbol type is listed in the TPA$Q_NUMBER Description column. It does not modify this field while executing a transition that specifies any other symbol type.

2 Coding and Using a Simple State Table
LIB$[ABLE_]PARSE can parse programming languages, command languages, or any other grammar for which a deterministic parser is the best choice.

To code a program to use LIB$[ABLE_]PARSE, perform the following steps:
1. Set up state tables to implement the language's grammar (See Section 2.1)
2. Define the argument block and other common variables (See Section 2.2)
3. Include the call to LIB$[ABLE_]PARSE in the main program (See Section 2.3)

This section provides examples that demonstrate the use of LIB$[ABLE_]PARSE to perform these three steps. The examples parse the command language of a simple report management utility. This hypothetical utility allows a user to perform the following activities:

- Obtain a list of available reports (SHOW command).
- Read reports on the terminal (READ command).
- Print reports (PRINT command).
Store new reports (FILE command).

The examples use the BASIC programming language for everything except the state and keyword tables, which are coded in BLISS.

This simple state table program does not use any action routines or other arguments. See Section 3 for information about how to use these features of LIB$TABLE_PARSE.

2.1 Setting Up a State Table
A state table associates the parser's alphabet with a set of possible transitions.

It is often helpful to create a graphical representation of a state table before attempting to code it. The following section illustrates two possible approaches.

2.1.1 Diagramming the Transitions
One way to set up these tables is to start from a transition diagram of the language you want to parse. (If you do not know how to construct a transition diagram, you might find it helpful to read an introductory text about compiler design and construction before you start.) Each circle represents a state in the state table. Each arrow, labeled with an input option, represents a transition out of one state to another state or within the same state.

Figure lib–22 shows a transition diagram for the hypothetical utility described in this section.

Figure lib–22 Transition Diagram for a Hypothetical Utility

Another technique for developing a state table starts with a tabular diagram in which the first column is the starting state, the second column identifies the input token, or keyword, and the third gives the resultant state.

Figure lib–23 is a tabular diagram of the utility that appears in Figure lib–22.
## Figure lib–23  Tabular Diagram of a Hypothetical Utility

<table>
<thead>
<tr>
<th>Starting State</th>
<th>Input</th>
<th>Resulting State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>PRINT, READ, FILE, SHOW, Other</td>
<td>State1, State1, State1, State1, Error</td>
</tr>
<tr>
<td></td>
<td>Report Name, End of String, Other</td>
<td>State1, Exit, Error</td>
</tr>
</tbody>
</table>

In this case, each unique entry in the Starting State or Resulting State column represents a state in the state table. Each entry in the Input column represents a possible transition out of the state in the Starting State column to a state in the Resulting State column.

### 2.1.2 Coding a State Table

For both MACRO and BLISS, you begin the state table with an `$INIT_STATE` macro. If you use MACRO to define your state table, then:

- Use the `$STATE` macro to define each state.
- Follow each `$STATE` macro with one instance of the `$TRAN` macro for each transition from this state to another state or within the same state.

If you use BLISS to define the state table, then:

- Use the `$STATE` macro to define each state and its associated transitions.

**Note**

The order in which you define the states is important. If you do not specify a target state for a transition, LIB$TABLE_PARSE transfers control to the next state in the state table.

The following MACRO and BLISS examples code the state table for the hypothetical utility diagrammed in Figure lib–22 and Figure lib–23. Note that neither of these state tables includes the error state, because LIB$TABLE_PARSE automatically generates an error if the input token does not match a transition in the current state. To provide a transition to your own error state, code the last transition in the state with the TPA$_LAMBDA symbol type and specify a transition to your error state. The TPA$_LAMBDA symbol type matches any input token.

The state table, coded using MACRO, for this simple language looks like this:

```plaintext
.TITLE simplelang
.ident 'v1'

$TPADEF

$INIT_STATE SIMPLE_LANGUAGE_TABLE, SIMPLE_KEYWORD_TABLE
```
Using the BLISS macros yields the following state table definition:

```
MODULE simple_statetable =
BEGIN
  !+ These libraries contain the macros and other definitions
  ! needed to generate the state tables.
  !-
  LIBRARY 'SYS$LIBRARY:STARLET';
  LIBRARY 'SYS$LIBRARY:TPAMAC';
  !+ UFD_STATE is the name you are giving the state table.
  ! UFD_KEY names the keyword table.
  ! Be sure to use the same name in the call to LIB$TABLE_PARSE.
  !-
  $INIT_STATE (UFD_STATE, UFD_KEY);
  !+ ! Read the command name (to the first blank in the command).
  ! Each string is a keyword; you are limited to 220 keywords
  ! per state table.
  !-
  $STATE (START, 
   ('CREATE',STATE1),
   ('FILE',STATE1),
   ('PRINT',STATE1),
   ('READ',STATE1)
   );

  $STATE (STATE1,
   (TPA$_STRING, STATE1),
   (TPA$_EOS, TPA$_EXIT)
   );

END
ELUDOM ! End of module CREATE_TABLE
```

Assemble or compile this module as you would any other program module.

### 2.2 Defining the Argument Block

After you have set up the state tables, you need to declare the LIB$TABLE_PARSE argument block in such a way that both your program and LIB$TABLE_PARSE can use it. This means the data must be defined in an area common to the calling program and the program module containing the state table definitions.

In most programming languages you will use a combination of EXTERNAL statements and common data definitions to create and access a separate data PSECT. If you do not know what mechanisms the language you are using provides, consult the documentation for that language.
The following example shows the LIB$TABLEPARSE argument block defined for use in a BASIC program.

```
!LIB$TABLEPARSE requires that TPA$K_COUNT0 be eight.
DECLARE INTEGER CONSTANT TPA$K_COUNT0 = 8,
  TPA$L_COUNT = 0,
  TPA$L_OPTIONS=1,
  TPA$L_STRINGCNT=2,
  TPA$L_STRINGPTR=3,
  TPA$L_TOKENCNT=4,
  TPA$L_TOKENPTR=5,
  TPA$B_CHAR=6,
  TPA$L_NUMBER=7,
  TPA$L_PARAM=8

! Redefining the map allows you to use the standard
! LIB$TABLEPARSE symbolic names. TPA$L_STRINGCNT,
! for example, references the same storage location
! as TPARSE_ARRAY(2) and TPARSE_ARRAY(BTPA$L_STRINGCNT).

MAP (TPARSE_BLOCK) LONG TPARSE_ARRAY (TPA$K_COUNT0)
```

Before your program can call LIB$TABLEPARSE, it must place the necessary information in the argument block.

The example utility does not need to set any flags because it uses the LIB$TABLEPARSE defaults for options such as blanks processing and abbreviations. However, it must put the address and length of the string to be parsed into the TPA$L_STRINGCNT and TPA$L_STRINGPTR fields.

The address and the length of the string to be parsed are available in the descriptor of the input string (called COMMAND_LINE in the following program). However, BASIC, like most high-level languages, does not allow you to look at the descriptors of your strings. Instead, you can use LIB$ANALYZE_SDESC or LIB$ANALYZE_SDESC_64 to read the length and address from the string descriptor and place them in the argument block.
2.3 Coding the Call to LIB$TABLE_PARSE

The following example demonstrates calling LIB$TABLE_PARSE from a high-level language (BLISS). This program uses the BLISS state table described in Section 2.1.2.

```plaintext
5 %TITLE "BLISS Program to Call LIB$TABLE_PARSE"

OPTION TYPE=EXPLICIT

!+ ! COMMAND_LINE is the string to receive the input command from the terminal.
! ERROR_MSG_TEXT is the system error message returned from LIB$SYS_GETMSG
! (used in the error handling routine)
!-
DECLARE STRING COMMAND_LINE, ERROR_MSG_TEXT

!+ ! RET_STATUS receives the status from the system calls.
! SAVE_STATUS is used when an error occurs and the error handling routine calls LIB$SYS_GETMSG to obtain the error text.
!-
DECLARE LONG RET_STATUS, SAVE_STATUS

!+ ! UFD_STATE is the address of the state table.
! UFD_KEY is the address of the key table.
! Both addresses are set up by the macros in module SIMPLE_STATETABLE32.
!-
EXTERNAL LONG UFD_STATE, UFD_KEY

!+ ! To allow us to compare returned statuses more easily.
!-
EXTERNAL INTEGER CONSTANT SS_NORMAL, &
  LIB$_SYNTAXERR, &
  LIB$_INVTYPE

!+ ! This program calls the following Run-Time Library routines:
! ! LIB$TABLE_PARSE to parse the input string
! ! LIB$ANALYZE_SDESC to get the length and starting address of the command string and place them in the LIB$TABLE_PARSE argument block.
! ! LIB$SYS_GETMSG to find the facility, severity, and text of any system errors that occur during program execution.
!-
EXTERNAL LONG FUNCTION LIB$TABLE_PARSE, &
  LIB$ANALYZE_SDESC, &
  LIB$SYS_GETMSG
```
This file defines the argument block that is passed to LIB$TABLE_PARSE. It also defines subscripts that make it easier to access the array.

Keeping the argument block definitions in a separate file makes them easier to modify and lets other programs use the same definitions.

%INCLUDE "SIMPLE_TPARSE_BLOCK"

ON ERROR GOTO ERROR_HANDLER

LIB$TABLE_PARSE requires that TPA$L_COUNT, the first field in the argument block, have a value of TPA$K_COUNT0, whose value is 8.

TPA$L_COUNT = TPA$K_COUNT0

Prompt at the terminal for the user’s action. A real utility should provide a friendlier, clearer interface.

GET_INPUT: PRINT "Your options are: " , " READ report "
            PRINT , " FILE report "
            PRINT , " PRINT report "
            PRINT , " CREATE report "
            PRINT
            INPUT "What would you like to do"; COMMAND_LINE

Get the length and starting address of the command line and place them in the LIB$TABLE_PARSE argument block. Note that LIB$ANALYZE_SDESC stores the length as a word.

RET_STATUS = LIB$ANALYZE_SDESC (COMMAND_LINE BY DESC, &
                                         TPARSE_ARRAY (BTPA$L_STRINGCNT) BY REF, &
                                         TPARSE_ARRAY (BTPA$L_STRINGPTR) BY REF)

IF RET_STATUS <> SS$NORMAL THEN
    GOTO ERROR_HANDLER
END IF

Call LIB$TABLE_PARSE to process the input string.

Note that LIB$TABLE_PARSE expects to receive its arguments by reference, while BASIC’s default for arrays and strings is by descriptor. Therefore the BY REF clauses are required. Without them, LIB$TABLE_PARSE cannot find the input string and the parse will always fail.

RET_STATUS = LIB$TABLE_PARSE (TPARSE_ARRAY () BY REF, &
                                         UFO_STATE BY REF,
                                         UFO_KEY BY REF)

This simple program provides no information except that a valid command was entered. The next section discusses techniques for gathering more information.
LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

IF RET_STATUS = SS$_NORMAL
  !+ For now, exit on success.
  !-
  THEN PRINT "Parse successful"
  GOTO 9999
  !+ If the parse failed, give the user a chance to try again.
  !-
  ELSE IF RET_STATUS = LIB$_SYNTAXERR THEN
    PRINT "You did not enter a valid command."
    PRINT "Please try again."
    GOTO GET_INPUT
  !+ ! If a more serious error occurred, inform the user
  ! and exit.
  !-
  ELSE
    GOTO ERROR_HANDLER
  END IF
END IF

500 ERROR_HANDLER: SAVE_STATUS = RET_STATUS
RET_STATUS = LIB$SYS_GETMSG (SAVE_STATUS,,ERROR_MSG_TEXT)
PRINT "Something went wrong."
PRINT ERL, ERROR_MSG_TEXT
RESUME 9999

9999 END

Compile this program as you would any other BASIC program.

When both the state tables and the main program have been compiled, link them
together to form a single executable image, as follows:

$ LINK SIMPLANG,SIMPLANG_STATETABLE

3 Using Advanced LIB$TABLE_PARSE Features
The LIB$TABLE_PARSE call in the previous program tells you that the
command the user entered was valid, but nothing else—not even which command
was entered. A program usually needs more information than this.

The following sections describe some of the more complicated ways to process
input strings or to gather extra information for your program, including:

• Action routines (see 3.1)
• Blanks in the input string (see 3.2)
• Special characters in the input string (see 3.3)
• Abbreviated keywords (see 3.4)
• Subexpressions (see 3.5)
• Modular use of LIB$TABLE_PARSE (see 3.6)
3.1 Using Action Routines

After LIB$TABLE_PARSE finds a match between a transition and the leading portion of the input string, it determines if the transition that made the match specified an action routine. If it did, LIB$TABLE_PARSE stores the value of the transition’s argument longword, if any, in the argument block PARAM field and calls the action routine.

- If the action routine returns success, LIB$TABLE_PARSE processes the mask or msk-adr arguments, if any, and continues to execute the transition as it would if there was no action routine.
- If the action routine returns failure, LIB$TABLE_PARSE does not execute the transition and continues attempting to match successive transitions.

3.1.1 Passing Data to an Action Routine

An action routine has only one argument, the argument block. You can pass additional data to the action routine using:

- The transition’s optional argument argument
- Fields you add to the end of the argument block

LIB$TABLE_PARSE and LIB$TPARSE use different linkages for passing the argument block to the action routine:

- LIB$TABLE_PARSE uses the standard calling mechanism and passes the argument block, by reference, as the only argument to the action routine.

  Therefore, for OpenVMS systems, action routines are written as:

  ROUTINE TEST( TPARSE_ARGUMENT_BLOCK : REF BLOCK[ , BYTE ] ) =
  BEGIN
    TPARSE_ARGUMENT_BLOCK[ TPA$V_ABBREV ] = 1
  END;

- On VAX systems, LIB$TPARSE uses a nonstandard linkage that establishes the address of the argument block as the routine’s actual argument pointer. Therefore an action routine can reference fields in the argument block by their symbolic offsets relative to the AP (argument pointer) register.

  For example:

  ROUTINE TEST =
  BEGIN
    BUILTIN
    AP;
    BIND
    TPARSE_ARGUMENT_BLOCK = AP : REF BLOCK[ , BYTE ];
    TPARSE_ARGUMENT_BLOCK[ TPA$V_ABBREV ] = 1
  END;

3.1.2 Action Routine Return Values

The action routine returns a value to LIB$TABLE_PARSE in R0 that controls execution of the current state transition. If the action routine returns success (low bit set in R0) then LIB$TABLE_PARSE proceeds with the execution of the state transition. If the action routine returns failure (low bit clear in R0), LIB$TABLE_PARSE rejects the transition that was being processed and acts as if the symbol type of that transition had not matched. It proceeds to evaluate other transitions in that state for eligibility.
Prior to calling an action routine, LIB$TABLE_PARSE sets the low bit of R0 to make it easier for the action routine to return success.

If an action routine returns a nonzero failure status to LIB$TABLE_PARSE and no subsequent transitions in that state match, LIB$TABLE_PARSE will return the status of the action routine, rather than the status LIB$_SYNTAXERR. In longword-valued functions in high-level languages, this value is returned in R0.

### 3.1.3 Using an Action Routine to Reject a Transition

An action routine can intentionally return a failure status to force LIB$TABLE_PARSE to reject a transition. This allows you to implement symbol types specific to particular applications. To recognize a specialized symbol type, code a state transition using a LIB$TABLE_PARSE symbol type that describes a superset of the desired set of possible tokens. The associated action routine then performs the additional discrimination necessary and returns success or failure to LIB$TABLE_PARSE, which then accordingly executes or fails to execute the transition.

A pure finite-state machine, for instance, has difficulty recognizing strings that are shorter than some maximum length or accepting numeric values confined to some particular range.

### 3.2 Blanks in the Input String

The default mode of operation in LIB$TABLE_PARSE is to treat blanks as separators. That is, they can appear between any two tokens in the string being parsed without being called for by transitions in the state table. Because blanks are significant in some situations, LIB$TABLE_PARSE processes blanks if you have set the bit TPA$V_BLANKS in the options longword of the argument block. The following input string shows the difference in operation:

```
ABC DEF
```

LIB$TABLE_PARSE recognizes the string by the following sequences of state transitions, depending on the state of the blanks control flag. The following examples illustrate processing with and without TPA$V_BLANKS set:
• TPA$V_BLANKS set:
  \$STATE
  \$TRAN TPA$_STRING
  \$STATE
  \$TRAN TPA$_BLANK
  \$STATE
  \$TRAN TPA$_STRING

• TPA$V_BLANKS clear:
  \$STATE
  \$TRAN TPA$_STRING
  \$STATE
  \$TRAN TPA$_STRING

Your action routines can set or clear TPA$V_BLANKS as LIB$TABLE_PARSE enters or leaves sections of the state table in which blanks are significant. LIB$TABLE_PARSE always checks the blanks control flag as it enters a state. If the flag is clear, it removes any space or tab characters present at the front of the input string before it proceeds to evaluate transitions. Note that when the TPA$V_BLANKS flag is clear, the TPA$BLANK symbol type will never match. If TPA$V_BLANKS is set, you must explicitly process blanks.

3.3 Special Characters in the Input String
Not all members of the ASCII character set can be entered directly in the state table definitions. Examples include the single quotation mark and all control characters.

In MACRO state tables, such characters can be specified as the symbol type with any assembler expression that is equivalent to the ASCII code of the desired character, not including the single quotes. For example, you could code a transition to match a backspace character as follows:

BACKSPACE = 8
  ...
  ...
  \$TRAN BACKSPACE, ...

MACRO places extra restrictions on the use of a comma in arguments to macros; often they must be surrounded by one or more angle brackets. Using a symbolic name for the comma will avoid such difficulties.

To build a transition matching such a single character in a BLISS state table, you can use the %CHAR lexical function as follows:

LITERAL BACKSPACE = 8;
  ...
  ...
  \$STATE (label,
            (%CHAR (BACKSPACE), ... )
      ) ;
3.4 Abbreviating Keywords
The default mode of LIB$TPARSE/LIB$TABLE_PARSE is exact match. All keywords in the input string must exactly match their spelling, length and case in the state table. However, many languages (command languages in particular) allow you to abbreviate keywords. For this reason, LIB$TPARSE/LIB$TABLE_PARSE has three abbreviation facilities to permit the recognition of abbreviated keywords when the state table lists only the full spellings. All three are controlled by flags and options defined in the argument block OPTIONS field. Table lib–11 describes these flags.

Table lib–11 Keyword Abbreviation Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$B_MCOUNT</td>
<td>By setting a value in the MCOUNT argument block field, the calling program or action routine specifies a minimum number of characters from the abbreviated keyword that must be present for a match to occur. For example, setting the byte to the value 4 would allow the keyword DEASSIGN to appear in an input string as DEAS, DEASS, DEASSI, DEASSIG, or DEASSIGN. LIB$TPARSE checks all the characters of the keyword string. Incorrect spellings beyond the minimum abbreviation are not permitted.</td>
</tr>
<tr>
<td>TPA64$B_MCOUNT</td>
<td></td>
</tr>
<tr>
<td>TPA$V_ABBRFM</td>
<td>If you set the ABBRFM flag in the argument block OPTIONS field, LIB$TPARSE recognizes any leftmost substring of a keyword as a match for that keyword. LIB$TPARSE does not check for ambiguity; it matches the first keyword listed in the state table of which the input token is a subset. For proper recognition of ambiguous keywords, the keywords in each state must be arranged in alphabetical order by the ASCII collating sequence as follows: Dollar sign ($) Numerics Uppercase alphabets Underscore (_) Lowercase alphabets</td>
</tr>
<tr>
<td>TPA64$V_ABBRFM</td>
<td></td>
</tr>
</tbody>
</table>

(continued on next page)
Table lib–11 (Cont.) Keyword Abbreviation Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPA$V_ABBREV</td>
<td>If you set the ABBREV flag in the argument block OPTIONS field, LIB$TABLE_PARSE recognizes any abbreviation of a keyword as long as it is unambiguous among the keywords in that state.</td>
</tr>
<tr>
<td>TPA64$V_ABBREV</td>
<td>If LIB$TABLE_PARSE finds that the front of the input string contains an ambiguous keyword string, it sets the AMBIG flag in the OPTIONS field and refuses to recognize any keyword transitions in that state. (It still accepts other symbol types.) The AMBIG flag can be checked by an action routine that is called when coming out of that state, or by the calling program if LIB$TABLE_PARSE returns with a syntax error status. LIB$TABLE_PARSE clears the flag when it enters the next state.</td>
</tr>
</tbody>
</table>

If both the ABBRFM and ABBREV flags are set, ABBRFM takes precedence.

---

**Note**

Using a keyword abbreviation option can permit short abbreviations or ambiguity, which restricts the extensibility of a language. Adding a new keyword can make a formerly valid abbreviation ambiguous.

---

3.5 Using Subexpressions

LIB$TABLE_PARSE subexpressions are analogous to subroutines within the state table. You can use subexpressions as you would use subroutines in any program:

- To avoid replication of complex expressions.
- For a limited form of pushdown parsing, in which the state table contains recursively nested subexpressions.
- For nondeterministic parsing, that is, parsing in which you need some number of states of look-ahead. To do this, place each path of look-ahead in a separate subexpression and call the subexpressions in the transitions of the state that needs the look-ahead. When a look-ahead path fails, the subexpression failure mechanism causes LIB$TABLE_PARSE to back out and try another path.

A subexpression call is indicated with the MACRO expression `!label` or the BLISS expression `(label)` as the transition type argument. Transfer of control to a subexpression causes LIB$TABLE_PARSE to call itself recursively, using the same argument block and keyword table as the original call, and using the specified state label as a starting state.

The following statement is an example of a $TRAN macro that calls a subexpression:

```
$TRAN !Q_STRING,,,,Q_DESCRIPTOR
```

In this example, Q_STRING is the label of another state, a subexpression, in the same state table.
When LIB$T[ABLE_]PARSE evaluates a transition that transfers control to a subexpression, it evaluates the subexpression’s transitions and processes the remaining input string.

- If the subexpression succeeds, it returns success to LIB$T[ABLE_]PARSE by executing a transition to TPA$_EXIT$. LIB$T[ABLE_]PARSE thus considers the calling transition to have made a match. It calls that transition’s action routine, if any, and executes the transition.
- If the subexpression fails, LIB$T[ABLE_]PARSE considers the calling transition to have no match. It backs up the input string, leaving it as it was at the start of the subexpression, and continues processing by evaluating the remaining transitions in the calling state.

3.5.1 Using Action Routines and Storing Data in a Subexpression

Be careful when designing subexpressions whose transitions provide action routines or use the mask and msk-adr arguments. As LIB$T[ABLE_]PARSE processes the state transitions of a subexpression, it calls the specified action routines and stores the mask and msk-adr. If the subexpression fails, LIB$T[ABLE_]PARSE backs up the input string and resumes processing in the calling state. However, any effect that an action routine has had on the caller’s database cannot be undone.

If subexpressions are used only as state table subroutines, there is usually no harm done, because when a subexpression fails in this mode, the parse generally fails. This is not true of pushdown or nondeterministic parsing. In applications where you expect subexpressions to fail, design action routines to store results in temporary storage. You can then make these results permanent at the main level, where the flow of control is deterministic.

3.5.2 An Example: Parsing a Quoted String

The following example is an excerpt of a state table that parses a string quoted by an arbitrary character. The table interprets the first character that appears as a quote character. Many text editors and some programming languages contain this sort of construction.

LIB$T[ABLE_]PARSE processes a transition that invokes a subexpression as it would any other transition:

- If the subexpression returns success by executing a transition to TPA$_EXIT$, LIB$T[ABLE_]PARSE considers the calling transition to have a match. It updates Q_DESCRIPTOR to describe the substring parsed by the subexpression and executes the transition to the next state in the state table.
- If the subexpression returns failure by executing a transition to TPA$_FAIL$, LIB$T[ABLE_]PARSE considers the calling transition to have no match. It restores the input string as it was when the subexpression was called and continues by evaluating the next transition in the state.
Main level state table. The first transition accepts and stores the quoting character.

```
$STATE STRING
  $TRAN TPA\$\_ANY,,,Q\_CHAR
```

Call the subexpression to accept the quoted string and store the string descriptor. Note that the descriptor spans all the characters accepted by the subexpression.

```
$STATE
  $TRAN !Q\_STRING,,,Q\__DESCRIPTOR
  $TRAN TPA\$\_LAMBDA,TPA\$\_FAIL
```

Accept the trailing quote character, left behind by the subexpression.

```
$STATE
  $TRAN TPA\$\_ANY,NEXT
```

Subexpression to scan the quoted string. The second transition matches until it is rejected by the action routine. The subexpression should never encounter the end of string before the final quoting character.

```
$STATE Q\_STRING
  $TRAN TPA\$\_EOS,TPA\$\_FAIL
  $TRAN TPA\$\_ANY,Q\_STRING,TEST\_Q
  $TRAN TPA\$\_LAMBDA,TPA\$\_EXIT
```

The following MACRO subroutine compares the current character with the quoting character and returns failure if it matches.

```
TEST\_Q: .WORD 0 ; null entry mask
  CMPB TPA\$\_B\_CHAR(AP),Q\_CHAR ; check the character
  BNEQ 10$ ; note R0 is already 1
  CLR R0 ; match - reject transition
10$: RET
```

### 3.5.3 An Example: Parsing a Complex Grammar

The following example is an excerpt from a state table that shows how to use subexpressions to parse a complex grammar. The state table accepts a number followed by a keyword qualifier. Depending on the keyword, the table interprets the number as decimal, octal, or hexadecimal. The state table accepts strings such as the following:

- 10/OCTAL
- 32768/DECIMAL
- 77AF/HEX

This sort of grammar is difficult to parse with a deterministic finite-state machine. Using a subexpression look-ahead of two states permits a simpler expression of the state tables.

```
$STATE
  $TRAN !OCT\_NUM,NEXT,,,NUMBER
  $TRAN !DEC\_NUM,NEXT,,,NUMBER
  $TRAN !HEX\_NUM,NEXT,,,NUMBER
```
LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

++; Subexpressions to accept an octal number followed by the OCTAL
; qualifier.
--; $STATE OCT_NUM
$TRAN TPA$$_OCTAL
$STATE
$TRAN '/'
$STATE
$TRAN 'OCTAL',TPA$_EXIT

++; Subexpression to accept a decimal number followed by the DECIMAL
; qualifier.
--; $STATE DEC_NUM
$TRAN TPA$$_DECIMAL
$STATE
$TRAN '/'
$STATE
$TRAN 'DECIMAL',TPA$_EXIT

++; Subexpression to accept a hex number followed by the HEX
; qualifier.
--; $STATE HEX_NUM
$TRAN TPA$$_HEX
$STATE
$TRAN '/'
$STATE
$TRAN 'HEX',TPA$_EXIT

Note that the transitions that follow a match with a numeric token do not
disturb the NUMBER field in the argument block. This allows the main level
subexpression call to retrieve it when the subexpression returns.

3.6 LIB$TABLE_PARSE and Modularity
To use LIB$TABLE_PARSE in a modular and shareable fashion:

- Avoid using OWN storage. Instead, allocate the argument block on the stack
  or the heap.
- Do not use the msk-adr argument.
- Do not use the argument argument as an address. If additional context is
  needed, allocate it at the end of the argument block.
- Use action routines to control flags such as TPA$V_BLANKS. The MACRO
  example at the end of the LIB$TPARSE/LIB$TABLE_PARSE section shows
  such an action routine, though the program itself is not modular.

4 Data Representation
This section describes the binary representation and allocation of a
LIB$TABLE_PARSE state table and a keyword table. While this information
is not required to use LIB$TABLE_PARSE, it may be useful in debugging your
program.

4.1 State Table Representation
Each state consists of its transitions concatenated in memory.
LIB$TABLE_PARSE equates the state label to the address of the first byte
of the first transition. A marker in the last transition identifies the end of the
state. The LIB$TABLE_PARSE table macros build the state table in the PSECT
_LIB$STATE$.
Each transition in a state consists of 2 to 23 bytes containing the arguments of the transition. The state table generation macros do not allocate storage for arguments not specified in the transition macro. This allows simple transitions to be represented efficiently. For example, the following transition, which simply accepts the character ‘?’ and falls through to the next state, is represented in two bytes:

$TRAN ‘?’

In this section, pointers described as self-relative are signed displacements from the address following the end of the pointer (this is identical to branch displacements in the OpenVMS VAX instruction set).

Table lib–12 describes the elements of a state transition in the order in which they appear, if present, in the transition. If a transition does not include a specific option, no bytes are assigned to the option within the transition.
Table lib-12  Binary Representation of a LIB$TABLE_PARSE State Transition

<table>
<thead>
<tr>
<th>Transition Element</th>
<th>No. of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol type</td>
<td>1</td>
<td>The first byte of a transition always contains the binary coding of the symbol type accepted by this transition. Flag bit 0 in the flags byte controls the interpretation of the type byte. If the flag is clear, the type byte represents a single character (the ‘x’ construct). If the flag bit is set, the type byte is one of the other type codes (keyword, number, and so on). The following table lists the symbol types accepted by LIB$TABLE_PARSE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol Type</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘x’</td>
<td>ASCII code of the character (8 bits)</td>
</tr>
<tr>
<td>‘keyword’</td>
<td>The keyword index (0 to 219)</td>
</tr>
<tr>
<td>TPA$_DECIMAL_64</td>
<td>228</td>
</tr>
<tr>
<td>(Alpha and I64 only)</td>
<td></td>
</tr>
<tr>
<td>TPA$_OCTAL_64</td>
<td>229</td>
</tr>
<tr>
<td>(Alpha and I64 only)</td>
<td></td>
</tr>
<tr>
<td>TPA$_HEX_64</td>
<td>230</td>
</tr>
<tr>
<td>(Alpha and I64 only)</td>
<td></td>
</tr>
<tr>
<td>TPA$_NODE_ACS</td>
<td>231</td>
</tr>
<tr>
<td>TPA$_NODE_PRIMARY</td>
<td>232</td>
</tr>
<tr>
<td>TPA$_NODE</td>
<td>233</td>
</tr>
<tr>
<td>TPA$_FILESPEC</td>
<td>234</td>
</tr>
<tr>
<td>TPA$_UIC</td>
<td>235</td>
</tr>
<tr>
<td>TPA$_IDENT</td>
<td>236</td>
</tr>
<tr>
<td>TPA$_ANY</td>
<td>237</td>
</tr>
<tr>
<td>TPA$_ALPHA</td>
<td>238</td>
</tr>
<tr>
<td>TPA$_DIGIT</td>
<td>239</td>
</tr>
<tr>
<td>TPA$_STRING</td>
<td>240</td>
</tr>
<tr>
<td>TPA$_SYMBOL</td>
<td>241</td>
</tr>
<tr>
<td>TPA$_BLANK</td>
<td>242</td>
</tr>
<tr>
<td>TPA$_DECIMAL</td>
<td>243</td>
</tr>
<tr>
<td>TPA$_OCTAL</td>
<td>244</td>
</tr>
<tr>
<td>TPA$_HEX</td>
<td>245</td>
</tr>
<tr>
<td>TPA$_LAMBDA</td>
<td>246</td>
</tr>
<tr>
<td>TPA$_EOS</td>
<td>247</td>
</tr>
<tr>
<td>TPA$_SUBEXPR</td>
<td>248</td>
</tr>
<tr>
<td>(subexpression call)</td>
<td></td>
</tr>
<tr>
<td>(Other codes are reserved for expansion)</td>
<td></td>
</tr>
</tbody>
</table>

(continued on next page)
Table lib–12 (Cont.) Binary Representation of a LIB$TABLEPARSE State Transition

<table>
<thead>
<tr>
<th>Transition Element</th>
<th>No. of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of the TPA$FILESPEC, TPA$NODE, TPA$NODE_PRIMARY, or TPA$NODE_ACS symbol type results in calls to the $FILESCAN system service. Use of the symbol type TPA$IDENT results in calls to the $ASCTOID system service. If your application of LIB$TABLEPARSE runs in an environment other than OpenVMS user mode, you must carefully evaluate whether use of these services is consistent with your environment.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>First flags byte</td>
<td>1</td>
<td>This byte contains the following bits, which specify the options of the transition. It is always present.</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Set if the type byte is not a single character.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Set if the second flags byte is present.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Set if this is the last transition in the state.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Set if a subexpression pointer is present.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Set if an explicit target state is present.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Set if the mask longword is present.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Set if the msk-adr longword is present.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Set if an action routine address is present.</td>
<td></td>
</tr>
<tr>
<td>Second flags byte</td>
<td>1</td>
<td>This byte is present if any of its flag bits is set. It contains an additional flag describing the transition. It is used as follows:</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Set if the action routine argument is present.</td>
<td></td>
</tr>
<tr>
<td>Subexpression pointer</td>
<td>2</td>
<td>This word is present in transitions that are subexpression calls. It is a 16-bit signed self-relative pointer to the starting state of the subexpression.</td>
</tr>
<tr>
<td>Argument longword</td>
<td>4</td>
<td>This longword field contains the 32-bit action routine argument, when specified.</td>
</tr>
<tr>
<td>Action routine address</td>
<td>4</td>
<td>This longword contains a self-relative pointer to the action routine, when specified.</td>
</tr>
<tr>
<td>Bit mask</td>
<td>4</td>
<td>This longword contains the mask argument, when specified.</td>
</tr>
</tbody>
</table>

(continued on next page)
Table lib–12 (Cont.)  Binary Representation of a LIB$TABLE_PARSE State Transition

<table>
<thead>
<tr>
<th>Transition Element</th>
<th>No. of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask address</td>
<td>4</td>
<td>This longword, when specified, contains a self-relative pointer through which the mask, or data that depends on the symbol type, is to be stored. Because the pointer is self-relative, when it points to an absolute location, the state table is not PIC (position-independent code).</td>
</tr>
<tr>
<td>Transition target</td>
<td>2</td>
<td>This word, when specified, contains the address of the target state of the transition. The address is stored as a 16-bit signed self-relative pointer. The final state TPA$_EXIT$ is coded as a word whose value is –1; the failure state TPA$_FAIL$ is coded as a word whose value is –2.</td>
</tr>
</tbody>
</table>

4.2  Keyword Table Representation

The keyword table is a vector of 16-bit signed pointers that address locations in the keyword string area, relative to the start of the keyword vector. It is the structure to which the $INIT_STATE$ macro equates its second argument.

The LIB$TABLE_PARSE$ macros assign an index number to each keyword. The index number is stored in the symbol type byte in the transition; it locates the associated keyword vector entry. The keyword strings are stored in the order encountered in the state table. Each keyword string is terminated by a byte containing the value –1. Between the keywords of adjacent states is an additional –1 byte to stop the ambiguous keyword scan.

To ensure that the keyword vector is adjacent to the keyword string area, the keyword vector is located in PSECT _LIB$KEY0$ and the keyword strings and stored in PSECT _LIB$KEY1$.

Your program should not use any of the three PSECTs used by LIB$TABLE_PARSE$ (_LIB$STATE$, _LIB$KEY0$, and _LIB$KEY1$). The PSECTs _LIB$KEY0$ and _LIB$KEY1$ refer to each other using 16-bit displacements, so user PSECTs inserted between them can cause truncation errors from the linker.

Condition Values Returned

- **SS$_\_NORMAL**  Routine successfully completed.
  - LIB$TABLE_PARSE$ has executed a transition to TPA$_EXIT$ at main level, not within a subexpression.

- **LIB$_\_SYNTAXERR**  Parse completed with syntax error.
  - LIB$TABLE_PARSE$ has encountered a state at main level in which none of the transitions match the input string, or in which a transition to TPA$_FAIL$ was executed.

- **LIB$_\_INVTYPE**  State table error. LIB$TABLE_PARSE$ has encountered an invalid entry in the state table.
Other If an action routine returns a failure status other than zero, and the parse consequently fails, LIB$TABLE_PARSE returns the status returned by the action routine.

Examples

Example 1a
The following DEC C program accepts and parses the command line of a CREATE/DIRECTORY command using LIB$TABLE_PARSE. It uses the state table defined in Example 1b.

```c
/*
** This DEC C program accepts and parses the command line of a CREATE/DIRECTORY
** command. This program uses the LIB$GET_FOREIGN call to acquire the command
** line from the CLI and parse it with LIB$TABLE_PARSE, leaving the necessary
** information in its global data base. The command line is of
** the following format:
**
** CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
**     /OWNER_UIC=[2437,25]
**     /ENTRIES=100
**     /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
**
** The three qualifiers are optional. Alternatively, the command
** may take the form:
**
** CREATE/DIR DEVICE:[202,31]
**
** using any of the optional qualifiers.
**
** The source for this program can be found in:
**
** SYS$EXAMPLES:LIB$TABLE_PARSE_DEMO.COM
**
*/

/*
** Specify the required header files
*/
#include <tpadef.h>
#include <descrip.h>
#include <starlet.h>
#include <lib$routines.h>

/*
** Specify macro definitions
*/
#define max_name_count 8
#define max_token_size 9
#define uic_string_size 6
#define command_buffer_size 256

/*
** Specify persistent data that’s local to this module
*/
LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

static
union
    uic_union {
        __int32 bits;
        struct {
            char first;
            char second;
        } bytes;
        struct {
            __int16 first;
            __int16 second;
        } words;
    }
    file_owner; /* Actual file owner UIC */

static
int
    name_count; /* Number of directory names */

static
char
    uic_string[ uic_string_size + 1 ]; /* Buffer for string */

static
struct
dsc$descriptor_s
    name_vector[ max_name_count ]; /* Vector of descriptors */

/*
 ** Specify persistent data that’s global to this module.
 ** This data is referenced externally by the state table definitions.
 */

union
    uic_union
        uIC_group, /* Tempt for UIC group */
        uIC_member; /* Tempt for UIC member */

int
    parser_flags, /* Keyword flags */
    entry_count, /* Space to preallocate */
    file_protect; /* Directory file protection */

struct
dsc$descriptor_s
    device_string = /* Device string descriptor */
        { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, (char *) 0 };

/*
 ** Specify the user action routines.
 **
 ** Please note that if it were LIB$TPARSE being called, the user action
 ** routines would have to be coded as follows:
 **
 ** int user_action_routine( __int32 psuedo_ap )
 ** {
 **    struct tpadef
 **        *tparse_block = (tpadef *) (&psuedo_ap - 1);
 **    printf( "Parameter value: %d\n",
 **        tparse_block->tpa$l_param
 **    );
 ** }
 **
 */

/*
 ** Shut off explicit blank processing after passing the command name.
 */
int blanks_off( struct tpadef *tparse_block ) {
    tparse_block->tpa$v_blanks = 0;
    return( 1 );
}

/*
 ** Check the UIC for legal value range.
 */
int check_uic( struct tpadef *tparse_block ) {
    if ( (uic_group.words.second != 0) ||
        (uic_member.words.second != 0) )
        return( 0 );

    file_owner.words.first = uic_member.words.first;
    file_owner.words.second = uic_group.words.first;
    return( 1 );
}

/*
 ** Store a directory name component.
 */
int store_name( struct tpadef *tparse_block ) {
    if ( (name_count >= max_name_count) ||
        (tparse_block->tpa$l_tokencnt > max_token_size) )
        return( 0 );

    name_vector[ name_count ].dsc$w_length = tparse_block->tpa$l_tokencnt;
    name_vector[ name_count ].dsc$b_dtype = DSC$K_DTYPE_T;
    name_vector[ name_count ].dsc$b_class = DSC$K_CLASS_S;
    name_vector[ name_count++ ].dsc$a_pointer = tparse_block->tpa$l_tokenptr;
    return( 1 );
}

/*
 ** Convert a UIC into its equivalent directory file name.
 */
int make_uic( struct tpadef *tparse_block ) {
    $DESCRIPTOR( control_string, "!OB!OB" );
    $DESCRIPTOR( dirname, uic_string );
    if ( (uic_group.bytes.second != '\0') ||
        (uic_member.bytes.second != '\0') )
        return( 0 );

    sys$fao( &control_string,
        &dirname.dsc$w_length,
        &dirname,
        uic_group.bytes.first,
        uic_member.bytes.first
    );

    return( 1 );
}

/*
 ** The main program section starts here.
 */
main( ) {
LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

/*
 ** This program creates a directory. It gets the command
 ** line from the CLI and parses it with LIB$TABLE_PARSE.
 */

extern
  char
    ufd_state,
    ufd_key;

char
  command_buffer[ command_buffer_size + 1 ];

int
  status;

$DESCRIPTOR( prompt, "Command> " );
$DESCRIPTOR( command_descriptor, command_buffer );

struct
tpadef
    tparse_block = { TPA$K_COUNT0,     /* Longword count */
                   TPA$M_ABBREV      /* Allow abbreviation */
                   TPA$M_BLANKS      /* Process spaces explicitly */
                   };

status = lib$get_foreign( &command_descriptor,
                         &prompt,
                         command_descriptor.dsc$w_length
                     );

if ( (status & 1) == 0 )
    return( status );

/*
 ** Copy the input string descriptor into the control block
 ** and then call LIB$TABLE_PARSE. Note that impure storage is assumed
 ** to be zero.
 */

  tparse_block.tpa$l_stringcnt = command_descriptor.dsc$w_length;
  tparse_block.tpa$l_stringptr = command_descriptor.dsc$a_pointer;

return( status = lib$table_parse( &tparse_block, &ufd_state, &ufd_key ) );
}
Example 1b
The following MACRO assembly language program module defines the state tables for the preceding sample program.

```
.TITLE CREATE_DIR_TABLES - Create Directory File (tables)
.IDENT "X-1"

;+ 
; This module defines the state tables for the preceding
; sample program, which accepts and parses the command line of the
; CREATE/DIRECTORY command. The command line has the following format:
; CREATE/DIRECTORY DEVICE:[MARANTZ.ACCOUNT.OLD]
; /OWNER_UIC=[2437,25]
; /ENTRIES=100
; /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
;
; The three qualifiers are optional. Alternatively, the command
; may take the form
; CREATE/DIRECTORY DEVICE:[202,31]
;
; using any of the optional qualifiers.
;
;+ 
; Global data, control blocks, etc.
;
;+ 
; Define control block offsets
;
; $CLIDEF
; $TPADEF

;+ 
; Define parser flag bits for flags longword
;
;+ 
; UIC_FLAG = 1 ; /UIC seen
ENTRYES_FLAG = 2 ; /ENTRIES seen
PROT_FLAG = 4 ; /PROTECTION seen

;SBTTL Parser State Table
;+ 
; Assign values for protection flags to be used when parsing protection
; string.
;"
LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

SYSTEM_READ_FLAG = 'X0001
SYSTEM_WRITE_FLAG = 'X0002
SYSTEM_EXECUTE_FLAG = 'X0004
SYSTEM_DELETE_FLAG = 'X0008
OWNER_READ_FLAG = 'X0010
OWNER_WRITE_FLAG = 'X0020
OWNER_EXECUTE_FLAG = 'X0040
OWNER_DELETE_FLAG = 'X0080
GROUP_READ_FLAG = 'X0100
GROUP_WRITE_FLAG = 'X0200
GROUP_EXECUTE_FLAG = 'X0400
GROUP_DELETE_FLAG = 'X0800
WORLD_READ_FLAG = 'X1000
WORLD_WRITE_FLAG = 'X2000
WORLD_EXECUTE_FLAG = 'X4000
WORLD_DELETE_FLAG = 'X8000

$INIT_STATE UFD_STATE,UFD_KEY

;+ Read over the command name (to the first blank in the command).
;-

; STATE START
$TRAN TPA$ BLANK,,BLANKS_OFF
$TRAN TPA$ ANY,START

;+ Read device name string and trailing colon.
;-

; STATE
$TRAN TPA$ SYMBOL,,,,DEVICE_STRING
$STATE
$TRAN ';

; Read directory string, which is either a UIC string or a general
; directory string.
;-

$STATE
$TRAN !UIC,,MAKE_UIC
$TRAN !NAME

;+ Scan for options until end of line is reached
;-

$STATE OPTIONS
$TRAN '/'
$TRAN TPA$ EOS,TPA$ EXIT
$STATE
$TRAN 'OWNER_UIC',PARSE_UIC,,UIC_FLAG,PARSER_FLAGS
$TRAN 'ENTRIES',PARSE_ENTRIES,,ENTRIES_FLAG,PARSER_FLAGS
$TRAN 'PROTECTION',PARSE PROT,,PROT_FLAG,PARSER_FLAGS

;+ Get file owner UIC.
;-

$STATE PARSE_UIC
$TRAN ':
$TRAN '='
$STATE
$TRAN !UIC,OPTIONS
;+ Get number of directory entries.
;-

SiSTATE PARSE_ENTRIES
$TRAN ':
$TRAN '='
$STATE
$TRAN TPA$-_DECIMAL,OPTIONS,,,ENTRY_COUNT

;+ Get directory file protection. Note that the bit masks generate the
; protection in complements form. It will be uncomplemented by the main
; program.
;-

SiSTATE PARSE_PROT
$TRAN ':
$TRAN '='
$STATE
$TRAN '('
$STATE NEXT_PRO
$TRAN 'SYSTEM', SYPR
$TRAN 'OWNER', OWPR
$TRAN 'GROUP', GRPR
$TRAN 'WORLD', WOPR
$STATE SYPR
$TRAN ':
$TRAN '='
$STATE SYPRO
$TRAN 'R',SYPRO,,SYSTEM_READ_FLAG,FILE_PROTECT
$TRAN 'W',SYPRO,,SYSTEM_WRITE_FLAG,FILE_PROTECT
$TRAN 'E',SYPRO,,SYSTEM_EXECUTE_FLAG,FILE_PROTECT
$TRAN 'D',SYPRO,,SYSTEM_DELETE_FLAG,FILE_PROTECT
$TRAN TPA$-_LAMBDA,ENDPRO
$STATE OWPR
$TRAN ':
$TRAN '='
$STATE OWPRO
$TRAN 'R',OWPRO,,OWNER_READ_FLAG,FILE_PROTECT
$TRAN 'W',OWPRO,,OWNER_WRITE_FLAG,FILE_PROTECT
$TRAN 'E',OWPRO,,OWNER_EXECUTE_FLAG,FILE_PROTECT
$TRAN 'D',OWPRO,,OWNER_DELETE_FLAG,FILE_PROTECT
$TRAN TPA$-_LAMBDA,ENDPRO
$STATE GRPR
$TRAN ':
$TRAN '='
$STATE GRPRO
$TRAN 'R',GRPRO,,GROUP_READ_FLAG,FILE_PROTECT
$TRAN 'W',GRPRO,,GROUP_WRITE_FLAG,FILE_PROTECT
$TRAN 'E',GRPRO,,GROUP_EXECUTE_FLAG,FILE_PROTECT
$TRAN 'D',GRPRO,,GROUP_DELETE_FLAG,FILE_PROTECT
$TRAN TPA$-_LAMBDA,ENDPRO
$STATE WOPR
$TRAN ':
$TRAN '='

LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

$STATE WOPRO
$TRAN 'R',WOPRO,,WORLD_READ_FLAG,FILE_PROTECT
$TRAN 'W',WOPRO,,WORLD_WRITE_FLAG,FILE_PROTECT
$TRAN 'E',WOPRO,,WORLD_EXECUTE_FLAG,FILE_PROTECT
$TRAN 'D',WOPRO,,WORLD_DELETE_FLAG,FILE_PROTECT
$TRAN TPAS$_LAMBDA,ENDPRO

$STATE ENDPRO
$TRAN <$,>,NEXT_PRO
$TRAN ')',OPTIONS

++; Subexpression to parse a UIC string.
;-

$STATE UIC
$TRAN '['

$TRAN TPAS$_OCTAL,,,,UIC_GROUP

$STATE $TRAN <','>; The comma character must be
; surrounded by angle brackets
; because MACRO restricts the use
; of commas in arguments to macros.

$STATE $TRAN TPAS$_OCTAL,,,,UIC_MEMBER

$STATE $TRAN ']',TPAS$_EXIT,CHECK_UIC

++; Subexpression to parse a general directory string
;-

$STATE NAME
$TRAN '['

$STATE NAMEO $TRAN TPAS$_STRING,,STORE_NAME

$STATE $TRAN '.,NAMEO
$TRAN ']',TPAS$_EXIT
$END_STATE

.END
Example 2

The following OpenVMS BLISS program accepts and parses the command line of a CREATE/DIRECTORY command using LIB$TPARSE.

```
MODULE CREATE_DIR ( ! Create directory file
                      IDENT = 'X0000',
                      MAIN = CREATE_DIR) =

BEGIN

! This OpenVMS BLISS program accepts and parses the command line
! of a CREATE/DIRECTORY command. This program uses the
! LIB$GET_FOREIGN call to acquire the command line from
! the CLI and parse it with LIB$TPARSE, leaving the necessary
! information in its global data base. The command line is of
! the following format:
!
! CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
! /UIC=[2437,25]
! /ENTRIES=100
! /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
!
! The three qualifiers are optional. Alternatively, the command
! may take the form
!
! CREATE/DIR DEVICE:[202,31]
!
! using any of the optional qualifiers.

! Global data, control blocks, etc.

LIBRARY 'SYS$LIBRARY:STARLET';
LIBRARY 'SYS$LIBRARY:TPAMAC.L32';

! Macro to make the LIB$TPARSE control block addressable as a block
! through the argument pointer.

MACRO
TPARSE_ARGS =
   BUILTIN AP;
   MAP AP : REF BLOCK [,BYTE];
%;

! Declare routines in this module.

FORWARD ROUTINE
CREATE_DIR, ! Mail program
BLANKS_OFF, ! No explicit blank processing
CHECK_UIC, ! Validate and assemble UIC
STORE_NAME, ! Store next directory name
MAKE_UIC; ! Make UIC into directory name

! Define parser flag bits for flags longword.

```
LITERAL
UIC_FLAG = 0, ! /UIC seen
ENTRIES_FLAG = 1, ! /ENTRIES seen
PROT_FLAG = 2; ! /PROTECTION seen

OWN
!+! This is the LIB$GET_FOREIGN descriptor block to get the command line.
!-
COMMAND_DESC : BLOCK [DSC$K_S_BLN, BYTE],
COMMAND_BUFF : VECTOR [256, BYTE],

!+! This is the LIB$TPARSE argument block.
!-
TPARSE_BLOCK : BLOCK [TPA$K_LENGTH0, BYTE]
  INITIAL (TPA$K_COUNT0, ! Longword count
  TPA$M_ABBREV ! Allow abbreviation
  OR TPA$M_BLANKS), ! Process spaces explicitly

!+! Parser global data:
!-
PARSER_FLAGS : BITVECTOR [32], ! Keyword flags
DEVICE_STRING : VECTOR [2], ! Device string descriptor
ENTRY_COUNT, ! Space to preallocate
FILE_PROTECT, ! Directory file protection
UIC_GROUP, ! Temp for UIC group
UIC_MEMBER, ! Temp for UIC member
FILE_OWNER, ! Actual file owner UIC
NAME_COUNT, ! Number of directory names
UIC_STRING : VECTOR [6, BYTE], ! Buffer for string
NAME_VECTOR : BLOCKVECTOR [0, 2], ! Vector of descriptors

DIRNAME1 : VECTOR [2], ! Name descriptor 1
DIRNAME2 : VECTOR [2], ! Name descriptor 2
DIRNAME3 : VECTOR [2], ! Name descriptor 3
DIRNAME4 : VECTOR [2], ! Name descriptor 4
DIRNAME5 : VECTOR [2], ! Name descriptor 5
DIRNAME6 : VECTOR [2], ! Name descriptor 6
DIRNAME7 : VECTOR [2], ! Name descriptor 7
DIRNAME8 : VECTOR [2]; ! Name descriptor 8

!+! Structure macro to reference the descriptor fields in the vector of
! descriptors.
!-
MACRO
STRING_COUNT = 0, 0, 32, 0%, ! Count field
STRING_ADDR = 1, 0, 32, 0%; ! Address field

!+! LIB$TPARSE state table to parse the command line
!-
$INIT_STATE (UFD_STATE, UFD_KEY);

!+! Read over the command name (to the first blank in the command).
!-
$STATE (START,  
  (TPA$_BLANK, , BLANKS_OFF),  
  (TPA$_ANY, START)  
);  
!+  
! Read device name string and trailing colon.  
!-  
$STATE (,  
  (TPA$_SYMBOL,,,, DEVICE_STRING)  
);  
$STATE (,  
  (':')  
);  
!+  
! Read directory string, which is either a UIC string or a general  
! directory string.  
!-  
$STATE (,  
  ((UIC),, MAKE_UIC),  
  ((NAME))  
);  
!+  
! Scan for options until end of line is reached.  
!-  
$STATE (OPTIONS,  
  ('/'),(TPA$_EOS, TPA$_EXIT)  
);  
$STATE (,  
  ('UIC', PARSE_UIC,, 1^UIC_FLAG, PARSER_FLAGS),  
  ('ENTRIES', PARSE_ENTRIES,, 1^ENTRIES_FLAG, PARSER_FLAGS),  
  ('PROTECTION', PARSE_PROT,, 1^PROT_FLAG, PARSER_FLAGS)  
);  
!+  
! Get file owner UIC.  
!-  
$STATE (PARSE_UIC,  
  (':'),  
  ('='))  
);  
$STATE (,  
  ((UIC), OPTIONS)  
);  
!+  
! Get number of directory entries.  
!-  
$STATE (PARSE_ENTRIES,  
  (':'),  
  ('='))  
);  
$STATE (,  
  (TPA$_DECIMAL, OPTIONS,,,, ENTRY_COUNT)  
);
Get directory file protection. Note that the bit masks generate the
protection in complement form. It will be uncomplemented by the main
program.

$STATE (PARSE_PROT,
     ('=',)
     ('='))
$STATE (,
     ('(')
     )
$STATE (NEXT_PRO,
     ('SYSTEM', SYPR),
     ('OWNER', OWPR),
     ('GROUP', GRPR),
     ('WORLD', WOPR)
     )
$STATE (SYPR,
     ('=',)
     ('='))
$STATE (SYPR0,
     ('R', SYPR0,, %X'0001', FILE_PROTECT),
     ('W', SYPR0,, %X'0002', FILE_PROTECT),
     ('E', SYPR0,, %X'0004', FILE_PROTECT),
     ('D', SYPR0,, %X'0008', FILE_PROTECT),
     (TPA$_LAMBDA, ENDPRO)
     )
$STATE (OWPR,
     ('=',)
     ('='))
$STATE (OWPR0,
     ('R', OWPR0,, %X'0020', FILE_PROTECT),
     ('W', OWPR0,, %X'0040', FILE_PROTECT),
     ('E', OWPR0,, %X'0000', FILE_PROTECT),
     ('D', OWPR0,, %X'0080', FILE_PROTECT),
     (TPA$_LAMBDA, ENDPRO)
     )
$STATE (GRPR,
     ('=',)
     ('='))
$STATE (GRPR0,
     ('R', GRPR0,, %X'0000', FILE_PROTECT),
     ('W', GRPR0,, %X'0000', FILE_PROTECT),
     ('E', GRPR0,, %X'0000', FILE_PROTECT),
     ('D', GRPR0,, %X'0000', FILE_PROTECT),
     (TPA$_LAMBDA, ENDPRO)
     )
$STATE (WOPR,
     ('=',)
     ('='))
GLOBAL ROUTINE CREATE_DIR (START_ADDR, CLI_CALLBACK) =
BEGIN
  !+  ! This program creates a directory. It gets the command
  ! line from the CLI and parses it with LIB$TPARSE.
  !-  
  LOCAL
    STATUS, ! Status from LIB$TPARSE
    OUT_LEN : WORD; ! length of returned command line
  EXTERNAL
    SS$_NORMAL;
LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

EXTERNAL ROUTINE
LIB$GET_FOREIGN : ADDRESSING_MODE (GENERAL),
LIB$TPARSE : ADDRESSING_MODE (GENERAL);

COMMAND_DESC [DSC$W_LENGTH] = 256;
COMMAND_DESC [DSC$B_DTYPE] = DSC$K_DTYPE_T;
COMMAND_DESC [DSC$B_CLASS] = DSC$K_CLASS_S;
COMMAND_DESC [DSC$A_POINTER] = COMMAND_BUFF;

STATUS = LIB$GET_FOREIGN (COMMAND_DESC,
%ASCID 'COMMAND: ',
OUT_LEN
);

IF NOT .STATUS
THEN
  SIGNAL (STATUS);

! Copy the input string descriptor into the LIB$TPARSE control block
! and call LIB$TPARSE. Note that impure storage is assumed to be zero.
!
TPARSE_BLOCK [TPA$L_STRINGCNT] = .OUT_LEN;
TPARSE_BLOCK [TPA$L_STRINGPTR] = .COMMAND_DESC [DSC$A_POINTER];

STATUS = LIB$TPARSE (TPARSE_BLOCK, UFD_STATE, UFD_KEY);

IF NOT .STATUS
THEN
  RETURN 0;

RETURN SS$NORMAL; ! End of routine CREATE_DIR

! Parser action routines
!

ROUTINE BLANKS_OFF =
BEGIN
  TPARSE_ARGS;

  AP [TPA$V_BLANKS] = 0;
  1
END;

! Shut off explicit blank processing after passing the command name.
!
ROUTINE CHECK_UIC =
BEGIN
  TPARSE_ARGS;

  IF .UIC_GROUP<16,16> NEQ 0
  OR .UIC_MEMBER<16,16> NEQ 0
  THEN RETURN 0;

  FILE_OWNER<0,16> = .UIC_MEMBER;
  FILE_OWNER<16,16> = .UIC_GROUP;
  1
END;

! Check the UIC for legal value range.
!
ROUTINE STORE_DIRNAME =
BEGIN
  TPARSE_ARGS;

  IF .UIC_GROUP<16,16> NEQ 0
  OR .UIC_MEMBER<16,16> NEQ 0
  THEN RETURN 0;

  FILE_OWNER<0,16> = .UIC_MEMBER;
  FILE_OWNER<16,16> = .UIC_GROUP;
  1
END;

! Store a directory name component.
!
ROUTINE STORE_NAME =
    BEGIN
    TPARSE_ARGS;
    IF .NAME_COUNT GEQU 8
    OR .AP[TPA$L_TOKENCNT] GTRU 9
    THEN RETURN 0;
    NAME_COUNT = .NAME_COUNT + 1;
    NAME_VECTOR [.NAME_COUNT, STRING_COUNT] = .AP[TPA$L_TOKENCNT];
    NAME_VECTOR [.NAME_COUNT, STRING_ADDR] = .AP[TPA$L_TOKENPTR];
    END;

!+ ! Convert a UIC into its equivalent directory file name. !-
ROUTINE MAKE_UIC =
    BEGIN
    TPARSE_ARGS;
    IF .UIC_GROUP<8,8> NEQ 0
    OR .UIC_MEMBER<8,8> NEQ 0
    THEN RETURN 0;
    DIRNAME1[0] = 0;
    DIRNAME1[1] = UIC_STRING;
    $FAOL (CTRSTR = UPLIT (6, UPLIT BYTE ('!OB!OB')),
           OUTBUF = DIRNAME1,
           PRMLST = UIC_GROUP);
    END;
END
ELUDOM ! End of module CREATE_DIR
Example 3
The following MACRO assembly language program accepts and parses the command line of a CREATE/DIRECTORY command using LIB$TPARSE. It also defines the state table for the parser.

```
.TITLE CREATE_DIR - Create Directory File
.IDENT "X0000"
;
; This is a sample OpenVMS MACRO program that accepts and parses the command
; line of the CREATE/DIRECTORY command. This program contains the OpenVMS
; call to acquire the command line from the command interpreter
; and parse it with LIB$TPARSE, leaving the necessary information in
; its global data base. The command line has the following format:
;
; CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
; /OWNER_UIC=[2437,25]
; /ENTRIES=100
; /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
;
; The three qualifiers are optional. Alternatively, the command
; may take the form
;
; CREATE/DIR DEVICE:[202,31]
;
; using any of the optional qualifiers.
;
; Global data, control blocks, etc.
;
; .PSECT IMPURE,WRT,NOEXE
;
; Define control block offsets
;
; $CLIDEF$TPADEF
;
; Define parser flag bits for flags longword
;
; UIC_FLAG = 1 ; /UIC seen
; ENTRIES_FLAG = 2 ; /ENTRIES seen
; PROT_FLAG = 4 ; /PROTECTION seen
;
; LIB$GET_FOREIGN string descriptors to get the line to be parsed
;
; STRING_LEN = 256
; STRING_DESC:
; .WORD STRING_LEN
; .BYTE DSC$K_DTYPE_T
; .BYTE DSC$K_CLASS_S
; .ADDRESS STRING_AREA
;
; STRING_AREA:
; .BLKB STRING_LEN
;
; PROMPT_DESC:
; .WORD PROMPT_LEN
; .BYTE DSC$K_DTYPE_T
; .BYTE DSC$K_CLASS_S
; .ADDRESS PROMPT
```
PROMPT: 
.ASCII /qualifiers: / 
PROMPT_LEN = .-PROMPT

;+ TPARSE argument block ;-

TPARSE_BLOCK:  
.LONG TPA$K_COUNT0 ; Longword count
.LONG TPA$M_ABBREV!- ; Allow abbreviation
.TPA$M_BLANKS ; Process spaces explicitly
.LONG TPA$LENGTH0-8 ; Remainder set at run time

;+ ; Parser global data ;-

RET_LEN: .BLKW 1 ; LENGTH OF RETURNED COMMAND LINE
PARSER_FLAGS: .BLKL 1 ; Keyword flags
DEVICE_STRING: .BLKL 2 ; Device string descriptor
ENTRY_COUNT: .BLKL 1 ; Space to preallocate
FILE_PROTECT: .BLKL 1 ; Directory file protection
UIC_GROUP: .BLKL 1 ; Temp for UIC group
UIC_MEMBER: .BLKL 1 ; Temp for UIC member
UIC_STRING: .BLKB 6 ; String to receive converted UIC
FILE_OWNER: .BLKL 1 ; Actual file owner UIC
NAME_COUNT: .BLKL 1 ; Number of directory names
DIRNAME1: .BLKL 2 ; Name descriptor 1
DIRNAME2: .BLKL 2 ; Name descriptor 2
DIRNAME3: .BLKL 2 ; Name descriptor 3
DIRNAME4: .BLKL 2 ; Name descriptor 4
DIRNAME5: .BLKL 2 ; Name descriptor 5
DIRNAME6: .BLKL 2 ; Name descriptor 6
DIRNAME7: .BLKL 2 ; Name descriptor 7
DIRNAME8: .BLKL 2 ; Name descriptor 8

.SBTTL Main Program
;+ ; This program gets the CREATE/DIRECTORY command line from ; the command interpreter and parses it. ;-

.CREAT E_DIR:; 
.WORD ^M<R2,R3,R4,R5> ; Save registers

;+ ; Call the command interpreter to obtain the command line. ;-

PUSHAW RET_LEN
PUSHAQ PROMPT_DESC
PUSHAQ STRING_DESC
CALLS #3,G^LIB$GET_FOREIGN ; Call to get command line
BLBC R0, SYNTAX_ERR

;+ ; Copy the input string descriptor into the TPARSE control block ; and call LIB$TPARSE. Note that impure storage is assumed to be zero. ;-

MOV2WL RET_LEN, TPARSE_BLOCK+TPA$L_STRINGCNT
MOVAL STRING_AREA, TPARSE_BLOCK+TPA$L_STRINGPTR
PUSHAL UPD_KEY
PUSHAL UPD_STATE
PUSHAL TPARSE_BLOCK
CALLS #3,G^LIB$TPARSE
BLBC R0, SYNTAX_ERR
;+ Parsing is complete.
;
; You can include here code to process the string just parsed, to call
; another program to process the command, or to return control to
; a calling program, if any.
;-
SYNTAX_ERR:
;+
; Code to handle parsing errors.
;-
    RET

.SBITL    Parser State Table

;+ ; Assign values for protection flags to be used when parsing protection
; string.
;-
SYSTEM_READ_FLAG = ^X0001
SYSTEM_WRITE_FLAG = ^X0002
SYSTEM_EXECUTE_FLAG = ^X0004
SYSTEM_DELETE_FLAG = ^X0008
OWNER_READ_FLAG = ^X0010
OWNER_WRITE_FLAG = ^X0020
OWNER_EXECUTE_FLAG = ^X0040
OWNER_DELETE_FLAG = ^X0080
GROUP_READ_FLAG = ^X0100
GROUP_WRITE_FLAG = ^X0200
GROUP_EXECUTE_FLAG = ^X0400
GROUP_DELETE_FLAG = ^X0800
WORLD_READ_FLAG = ^X1000
WORLD_WRITE_FLAG = ^X2000
WORLD_EXECUTE_FLAG = ^X4000
WORLD_DELETE_FLAG = ^X8000

$INIT_STATE    UFD_STATE,UFD_KEY
;+
; Read over the command name (to the first blank in the command).
;-
    $STATE    START
    $TRAN    TPA$_BLANK,,BLANKS_OFF
    $TRAN    TPA$_ANY,START
;+
; Read device name string and trailing colon.
;-
    $STATE
    $TRAN    TPA$_SYMBOL,,,,DEVICE_STRING
    $STATE
    $TRAN    ‘:’
;+
; Read directory string, which is either a UIC string or a general
; directory string.
;-
    $STATE
    $TRAN    !UIC,,,,MAKE_UIC
    $TRAN    !NAME
;+
; Scan for options until end of line is reached
;-

LIB$ Routines
LIB$TPARSE/LIB$TABLE_PARSE

$STATE OPTIONS
$TRAN '/'
$TRAN TPA$ EOS, TPA$ EXIT
$STATE
$TRAN 'OWNER UIC',PARSE_UIC,,UIC_FLAG,PARSER_FLAGS
$TRAN 'ENTRIES',PARSE_ENTRIES,,ENTRIES_FLAG,PARSER_FLAGS
$TRAN 'PROTECTION',PARSE_PROT,,PROT_FLAG,PARSER_FLAGS

/+ ; Get file owner UIC.
-/
$STATE PARSE_UIC
$TRAN ':'
$TRAN '='
$STATE
$TRAN !UIC,OPTIONS

/+ ; Get number of directory entries.
-/
$STATE PARSE_ENTRIES
$TRAN ':'
$TRAN '='
$STATE
$TRAN TPA$ DECIMAL,OPTIONS,,ENTRY_COUNT

/+ ; Get directory file protection. Note that the bit masks generate the
; protection in complement form. It will be uncomplemented by the main
; program.
-/
$STATE PARSE_PROT
$TRAN ':'
$TRAN '='
$STATE
$TRAN '('
$STATE NEXT_PRO
$TRAN 'SYSTEM', SYPR
$TRAN 'OWNER', OWPR
$TRAN 'GROUP', GRPR
$TRAN 'WORLD', WOPR
$STATE SYPR
$TRAN ':'
$TRAN '='
$STATE SYPRO
$TRAN 'R',SYPRO,,SYSTEM_READ_FLAG,FILE_PROTECT
$TRAN 'W',SYPRO,,SYSTEM_WRITE_FLAG,FILE_PROTECT
$TRAN 'E',SYPRO,,SYSTEM_EXECUTE_FLAG,FILE_PROTECT
$TRAN 'D',SYPRO,,SYSTEM_DELETE_FLAG,FILE_PROTECT
$TRAN TPA$ LAMBDA, ENDPRO
$STATE OWPR
$TRAN ':'
$TRAN '='
$STATE OWPRO
$TRAN 'R',OWPRO,,OWNER_READ_FLAG,FILE_PROTECT
$TRAN 'W',OWPRO,,OWNER_WRITE_FLAG,FILE_PROTECT
$TRAN 'E',OWPRO,,OWNER_EXECUTE_FLAG,FILE_PROTECT
$TRAN 'D',OWPRO,,OWNER_DELETE_FLAG,FILE_PROTECT
$TRAN TPA$ LAMBDA, ENDPRO
;+ Subexpression to parse a UIC string.
;-

$STATE UIC
$TRAN ‘[’

$STATE TPA$ _OCTAL,,,,,UIC_GROUP

$STATE $TRAN <’,’> ; The comma character must be
 ; surrounded by angle brackets
 ; because MACRO restricts the use
 ; of commas in arguments to macros.

$STATE $TRAN TPA$ _OCTAL,,,,,UIC_MEMBER

$STATE $TRAN ‘]’,TPA$ _EXIT, CHECK_UIC

;+ Subexpression to parse a general directory string
;-

$STATE NAME
$TRAN ‘[’

$STATE NAMEO
$TRAN TPA$ _STRING,,STORE_NAME

$STATE $TRAN ‘.’,NAMEO
$TRAN ‘]’,TPA$ _EXIT
$END STATE

.SBTTL Parser Action Routines
.PSECT CODE,EXE,NOWRT

;+ Shut off explicit blank processing after passing the command name.
;-


BLANKS_OFF:
  .WORD  0          ; No registers saved (or used)
  BBCC #TPA$V_BLANKS,TPA$L_OPTIONS(AP),10$
10$:
  RET

; Check the UIC for legal value range.
;+
CHECK_UIC:
  .WORD  0          ; No registers saved (or used)
  TSTW UIC_GROUP+2 ; UIC components are 16 bits
  BNEQ 10$ TSTW UIC_MEMBER+2
  BNEQ 10$ MOVW UIC_GROUP,FILE_OWNER+2 ; Store actual UIC
  MOVW UIC_MEMBER,FILE_OWNER ; after checking
  RET
10$:
  CLRL R0          ; Value out of range - fail
  RET

; Store a directory name component.
;+
STORE_NAME:
  .WORD  0          ; No registers saved (or used)
  MOVL NAME_COUNT,R1 ; Get count of names so far
  CMPL R1,#8          ; Maximum of 8 permitted
  BGEQU 10$ INCL NAME_COUNT ; Count this name
  MOVAL UIC_STRING,DIRNAME1+4 ; Point to string buffer
  $FAOL CTRSTR=FAO_STRING,- ; Convert UIC to octal string
  OUTBUF=DIRNAME1,-
  PRMLST=UIC_GROUP
  RET
10$:
  CLRL R0          ; Error in directory name
  RET

; Convert a UIC into its equivalent directory file name.
;+
MAKE_UIC:
  .WORD  0          ; No registers saved (or used)
  TSTB UIC_GROUP+1 ; Check UIC for byte values,
  BNEQ 10$          ; because UIC type directories
  TSTB UIC_MEMBER+1 ; are restricted to this form
  BNEQ 10$ MOVV #6,DIRNAME1 ; Directory name is 6 bytes
  MOVAL UIC_STRING,DIRNAME1+4 ; Point to string buffer
  $FAOL CTRSTR=FAO_STRING,- ; Convert UIC to octal string
  OUTBUF=DIRNAME1,-
  PRMLST=UIC_GROUP
  RET
10$:
  CLRL R0          ; Range error - fail it
  RET

FAO_STRING: .LONG STRING_END-STRING_START
STRING_START: .ASCII '1OB1OB'
STRING_END: .END CREATE_DIR
LIB$ Routines
LIB$TRAVERSE_TREE

LIB$TRAVERSE_TREE
Traverse a Balanced Binary Tree

The Traverse a Balanced Binary Tree routine calls an action routine for each node in a binary tree. †

Format

LIB$TRAVERSE_TREE treehead ,user-action-procedure [,user-data-address]

Returns

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

Arguments

treehead
OpenVMS usage: address
type: address
access: read only
mechanism: by reference

Tree head of the binary tree. The treehead argument is the address of an unsigned longword that is the tree head in the binary tree traversal.

user-action-procedure
OpenVMS usage: procedure
type: procedure value
access: function call (before return)
mechanism: by value

User-supplied action routine called by LIB$TRAVERSE_TREE for each node in the tree. The user-action-procedure argument must return a success status for LIB$TRAVERSE_TREE to continue traversal.

For more information, see Call Format for an Action Routine in the Description section.

user-data-address
OpenVMS usage: user_arg
type: longword (unsigned)
access: read only
mechanism: by reference

User data that LIB$TRAVERSE_TREE passes to your action routine. The user-data-address argument contains the address of this user data. This is an optional argument; the default value is 0.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
Description
LIB$TRAVERSE_TREE calls a user-supplied action routine for each node to traverse a balanced binary tree.

Call Format for an Action Routine
The format of the call is as follows:

user-action-procedure node, user-data-address

LIB$TRAVERSE_TREE passes the node and user-data-address arguments to your action routine by reference.

This action routine is defined by you to fit your own purposes. A common use of an action routine here is to print the contents of each node during the tree traversal.

The following is one example of a user-supplied action routine.

```c
struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    char Text[80];
};
static long Print_Node(struct Full_node* Node, void* dummy)
{
    /*
    ** Print the string contained in the current node
    */
    printf("%s\n", Node->Text);
    return LIB$_NORMAL;
}
```

Condition Values Returned

LIB$_NORMAL Routine successfully completed.

Any condition value returned by your action routine.

Example
The C example provided in the description of LIB$INSERT_TREE also demonstrates the use of LIB$TRAVERSE_TREE. Refer to that example for assistance in using this routine.
LIB$ Routines
LIB$TRAVERSE_TREE_64 (Alpha and I64 Only)

LIB$TRAVERSE_TREE_64 (Alpha and I64 Only)
Traverse a Balanced Binary Tree

The Traverse a Balanced Binary Tree routine calls an action routine for each node in a binary tree.

Format

LIB$TRAVERSE_TREE_64 treehead ,user-action-procedure [,user-data-address]

Returns

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

Arguments

treehead
  OpenVMS usage: address
  type:    address
  access:  read only
  mechanism: by reference

Tree head of the binary tree. The treehead argument is the address of an unsigned quadword that is the tree head in the binary tree traversal.

user-action-procedure
  OpenVMS usage: procedure
  type:    procedure value
  access:  function call (before return)
  mechanism: by value

User-supplied action routine called by LIB$TRAVERSE_TREE_64 for each node in the tree. The user-action-procedure argument must return a success status for LIB$TRAVERSE_TREE_64 to continue traversal.

For more information, see Call Format for an Action Routine in the Description section.

user-data-address
  OpenVMS usage: user_arg
  type:    quadword (unsigned)
  access:  read only
  mechanism: by reference

User data that LIB$TRAVERSE_TREE_64 passes to your action routine. The user-data-address argument contains the address of this user data. This is an optional argument; the default value is 0.
LIB$ Routines
LIB$TRAVERSE_TREE_64 (Alpha and I64 Only)

Description
LIB$TRAVERSE_TREE_64 calls a user-supplied action routine for each node to traverse a balanced binary tree.

Call Format for an Action Routine
The format of the call is as follows:

user-action-procedure node ,user-data-address

LIB$TRAVERSE_TREE_64 passes the node and user-data-address arguments to your action routine by reference.

This action routine is defined by you to fit your own purposes. A common use of an action routine here is to print the contents of each node during the tree traversal.

The following is one example of a user-supplied action routine.

```c
struct Full_node
{
    void* left_link;
    void* right_link;
    short reserved;
    char Text[80];
};

static long Print_Node(struct Full_node* Node, void* dummy)
{
    /*
    ** Print the string contained in the current node
    */
    printf("%s\n", Node->Text);
    return LIB$_NORMAL;
}
```

Condition Values Returned

LIB$_NORMAL Routine successfully completed.

Any condition value returned by your action routine.

Example

The C example provided in the description of LIB$INSERT_TREE_64 also demonstrates the use of LIB$TRAVERSE_TREE_64. Refer to that example for assistance in using this routine.
LIB$ Routines
LIB$TRA_ASC_EBC

LIB$TRA_ASC_EBC
Translate ASCII to EBCDIC

The Translate ASCII to EBCDIC routine translates an ASCII string to an EBCDIC string.

Format

LIB$TRA_ASC_EBC source-string ,byte-integer-dest-string

Returns

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

Arguments

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

Source string (ASCII) to be translated by LIB$TRA_ASC_EBC. The source-string argument contains the address of a descriptor pointing to this source string.

byte-integer-dest-string
OpenVMS usage: char_string
type: character string
access: write only
mechanism: by descriptor

Destination string (EBCDIC). The byte-integer-dest-string argument contains the address of a descriptor pointing to this destination string.

Description

LIB$TRA_ASC_EBC translates an ASCII string to an EBCDIC string. If the destination string is a fixed-length string, its length must match the length of the input string. The length of both the source and destination strings is limited to 65,535 characters. No filling is done.

A similar operation can be accomplished by specifying the ASCII to EBCDIC translation table, LIB$AB_ASC_EBC, in a routine using LIB$MOVTC, but no testing for untranslatable characters is done under those circumstances.

The LIB$TRA_ASC_EBC routine uses the ASCII to EBCDIC translation table.
ASCII to EBCDIC Translation Table

- The numbers on the left represent the low-order bits of the ASCII characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the ASCII characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent EBCDIC characters in hexadecimal notation.

Figure lib–24 is the ASCII to EBCDIC translation table.

**Figure lib–24 LIBSAB_ASC_EBC**

<table>
<thead>
<tr>
<th>Row Bits 0 – 3</th>
<th>Column Bits 4 – 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F</td>
</tr>
<tr>
<td>0</td>
<td>00    10    40    F0    7C    D7    79    97    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>1</td>
<td>01    11    4F    F1    C1    D8    81    98    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>2</td>
<td>02    12    7F    F2    C2    D9    82    99    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>3</td>
<td>03    13    7B    F3    C3    E2    83    A2    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>4</td>
<td>37    3C    5B    F4    C4    E3    84    A3    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>5</td>
<td>2D    3D    6C    F5    C5    E4    85    A4    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>6</td>
<td>2E    32    50    F6    C6    E5    86    A5    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>7</td>
<td>2F    26    7D    F7    C7    E6    87    A6    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>8</td>
<td>16    18    4D    F8    C8    E7    88    A7    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>9</td>
<td>05    19    5D    F9    C9    E8    89    A8    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>A</td>
<td>25    3F    5C    7A    D1    E9    91    A9    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>B</td>
<td>0B    27    4E    SE    D3    49    92    C0    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>C</td>
<td>0C    1C    6B    4C    D3    E0    93    6A    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>D</td>
<td>0D    1D    60    7E    D4    5A    94    D0    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>E</td>
<td>0E    1E    4B    6E    D5    5F    95    A1    3F    3F    3F    3F    3F    3F    3F</td>
</tr>
<tr>
<td>F</td>
<td>0F    1F    61    6F    D6    96    07    3F    3F    3F    3F    3F    3F    3F    FF</td>
</tr>
</tbody>
</table>

All ASCII graphics are translated to their equivalent EBCDIC graphics except for the graphics noted in Table lib–13.

**Table lib–13 ASCII Graphics Not Translated to EBCDIC Equivalent by LIBSTRA_ASC_EBC**

<table>
<thead>
<tr>
<th>ASCII Graphic</th>
<th>EBCDIC Graphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ (left square bracket)</td>
<td>ç (cents sign)</td>
</tr>
<tr>
<td>! (exclamation point)</td>
<td></td>
</tr>
<tr>
<td>^ (circumflex)</td>
<td>~ (logical not)</td>
</tr>
<tr>
<td>] (right square bracket)</td>
<td>! (exclamation point)</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$TRA_ASC_EBC
Condition Values Returned
SS$_NORMAL
LIB$_INVARG

Routine successfully completed.
If the destination string is a fixed-length string
and its length is not the same as the source
string length, or if the length of the input string
is greater than 65,535 characters, no translation
is attempted.
One or more occurrences of an untranslatable
character have been detected during the
translation.

LIB$_INVCHA

Example
This COBOL program uses LIB$TRA_ASC_EBC to translate an ASCII string to
EBCDIC. If successful, it then uses LIB$MOVTC to translate the EBCDIC string
back to ASCII.

IDENTIFICATION DIVISION.
PROGRAM-ID. TRANS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 INPUT-STRING PIC X(4).
01 EBCDIC-STRING PIC X(4).
01 OUT-STRING PIC X(4).
01 FILL-CHAR PIC X VALUE "@".
01 SS-STATUS PIC S9(9)
COMP.
88 SS-NORMAL VALUE 01.
01 EBCDIC-TABLE.
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)
05 FILLER PIC X(16)

VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE
VALUE

"@@@@@@@@@@@@@@@@".
"@@@@@@@@@@@@@@@@".
"@@@@@@@@@@@@@@@@".
"@@@@@@@@@@@@@@@@".
" @@@@@@@@@@.<(+|".
"&@@@@@@@@@!$*);@".
"-/@@@@@@@@@,%_>?".
"@@@@@@@@@@:#@’=""".
"@abcdefghi@@@@@@".
"@jklmnopqr@@@@@@".
"@@stuvwxyz@@@@@@".
"@@@@@@@@@@@@@@@@".
"@ABCDEFGHI@@@@@@".
"!JKLMNOPQR@@@@@@".
"@@STUVWXYZ@@@@@@".
"0123456789@@@@@@".

ROUTINE DIVISION.
001-MAIN.
DISPLAY " ".
DISPLAY "ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC: "
WITH NO ADVANCING.
ACCEPT INPUT-STRING
AT END STOP RUN.
IF INPUT-STRING = "EXIT" OR "exit" OR "
"
STOP RUN.

lib–638


CALL "LIB$TRA_ASC_EBC"
   USING BY DESCRIPTOR INPUT-STRING, EBCDIC-STRING
   GIVING SS-STATUS.
IF SS-NORMAL
   CALL "LIB$MOVTC"
   USING BY DESCRIPTOR EBCDIC-STRING,
      FILL-CHAR, 
      EBCDIC-TABLE, 
      OUT-STRING,
   GIVING SS-STATUS 
   IF SS-NORMAL
      DISPLAY "ASCII ENTERED WAS: " INPUT-STRING
      DISPLAY "EBCDIC TRANSLATED IS: " OUT-STRING 
   ELSE
      DISPLAY "*** LIB$MOVTC TRANSLATION UNSUCCESSFUL ***"
   ELSE
      DISPLAY "*** LIB$TRA_ASC_EBC TRANSLATION UNSUCCESSFUL ***".
GO TO 001-MAIN.

To exit from this program, you must press Ctrl/Z. The output generated by this
COBOL program is as follows:

$ RUN TRANS

ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC:  abdc
  ASCII ENTERED WAS:  abdc
  EBCDIC TRANSLATED IS:  abdc

ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC: ~==b&
  ASCII ENTERED WAS:  =~b&
  EBCDIC TRANSLATED IS:  @=b&

ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC: 8^%$
  ASCII ENTERED WAS:  8^%$
  EBCDIC TRANSLATED IS:  8@%$

ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC: /x\}
  ASCII ENTERED WAS:  /x\}
  EBCDIC TRANSLATED IS:  /x@!

ENTER 4 CHARACTERS TO BE TRANSLATED ASCII TO EBCDIC: Ctrl/Z
LIB$Routines
LIB$TRA_EBC_ASC

LIB$TRA_EBC_ASC
Translate EBCDIC to ASCII

The Translate EBCDIC to ASCII routine translates an EBCDIC string to an ASCII string.

Format

LIB$TRA_EBC_ASC byte-integer-source-string,destination-string

Returns

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

Arguments

byte-integer-source-string
OpenVMS usage: char_string
type: character string
access: read only
mechanism: by descriptor

String (EBCDIC) to be translated by LIB$TRA_EBC_ASC. The byte-integer-source-string argument contains the address of a descriptor pointing to this source string.

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

Destination string (ASCII). The destination-string argument contains the address of the descriptor of this destination string.

The LIB$TRA_EBC_ASC routine uses the EBCDIC to ASCII translation table, LIB$AB_EBC_ASC.

Description

LIB$TRA_EBC_ASC translates an EBCDIC string to an ASCII string. If the destination string is a fixed-length string, its length must match the length of the input string. The length of both the source and destination strings is limited to 65,535 characters. No filling is done.

A similar operation can be accomplished by specifying the EBCDIC to ASCII translation table, LIB$AB_EBC_ASC, in a routine using LIB$MOVTC, but no testing for untranslatable characters is done under these circumstances.
The LIB$TRA_EBC_ASC routine uses the EBCDIC to ASCII translation shown in Figure lib–25.

**Figure lib–25 LIB$AB_EBC_ASC**

<table>
<thead>
<tr>
<th>Row</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>10</td>
<td>5C</td>
<td>5C</td>
<td>20</td>
<td>26</td>
<td>2D</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>7B</td>
<td>7D</td>
<td>5C</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
<td>5C</td>
<td>5C</td>
<td>2F</td>
<td>5C</td>
<td>61</td>
<td>6A</td>
<td>7E</td>
<td>5C</td>
<td>41</td>
<td>4A</td>
<td>5C</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>12</td>
<td>5C</td>
<td>5C</td>
<td>62</td>
<td>6B</td>
<td>73</td>
<td>5C</td>
<td>42</td>
<td>4B</td>
<td>53</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>13</td>
<td>5C</td>
<td>5C</td>
<td>63</td>
<td>6C</td>
<td>74</td>
<td>5C</td>
<td>43</td>
<td>4C</td>
<td>54</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>14</td>
<td>5C</td>
<td>5C</td>
<td>64</td>
<td>6D</td>
<td>75</td>
<td>5C</td>
<td>44</td>
<td>4D</td>
<td>55</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>15</td>
<td>5C</td>
<td>5C</td>
<td>65</td>
<td>6E</td>
<td>76</td>
<td>5C</td>
<td>45</td>
<td>4E</td>
<td>56</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>16</td>
<td>5C</td>
<td>5C</td>
<td>66</td>
<td>6F</td>
<td>77</td>
<td>5C</td>
<td>46</td>
<td>4F</td>
<td>57</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>17</td>
<td>5C</td>
<td>5C</td>
<td>67</td>
<td>70</td>
<td>71</td>
<td>5C</td>
<td>47</td>
<td>50</td>
<td>58</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>18</td>
<td>5C</td>
<td>5C</td>
<td>68</td>
<td>71</td>
<td>72</td>
<td>5C</td>
<td>48</td>
<td>51</td>
<td>59</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>19</td>
<td>5C</td>
<td>5C</td>
<td>69</td>
<td>72</td>
<td>73</td>
<td>5C</td>
<td>49</td>
<td>52</td>
<td>5A</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0A</td>
<td>1A</td>
<td>5C</td>
<td>5C</td>
<td>5B</td>
<td>5D</td>
<td>7C</td>
<td>3A</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>1B</td>
<td>5C</td>
<td>5C</td>
<td>5E</td>
<td>24</td>
<td>2C</td>
<td>23</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0C</td>
<td>1C</td>
<td>5C</td>
<td>14</td>
<td>3C</td>
<td>2A</td>
<td>25</td>
<td>40</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0D</td>
<td>1D</td>
<td>05</td>
<td>15</td>
<td>28</td>
<td>29</td>
<td>5F</td>
<td>27</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0E</td>
<td>1E</td>
<td>06</td>
<td>5C</td>
<td>2B</td>
<td>3B</td>
<td>3E</td>
<td>3D</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0F</td>
<td>1F</td>
<td>07</td>
<td>1A</td>
<td>21</td>
<td>5E</td>
<td>3F</td>
<td>22</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td>5C</td>
<td></td>
</tr>
</tbody>
</table>

**EBCDIC to ASCII Translation Table**

- The numbers on the left represent the low-order bits of the EBCDIC characters in hexadecimal notation.
- The numbers across the top represent the high-order bits of the EBCDIC characters in hexadecimal notation.
- The numbers in the body of the table represent the equivalent ASCII characters in hexadecimal notation.

All EBCDIC graphics are translated to their equivalent ASCII graphic except for the graphics noted in Table lib–14.

**Table lib–14 EBCDIC Graphics Not Translated to ASCII Equivalent by LIB$TRA_EBC_ASC**

<table>
<thead>
<tr>
<th>EBCDIC Graphic</th>
<th>ASCII Graphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>€ (cents sign)</td>
<td>[ (left square bracket)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[ (short vertical bar)</td>
<td>! (exclamation point)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~ (logical not)</td>
<td>^ (circumflex)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>! (exclamation point)</td>
<td>] (right square bracket)</td>
</tr>
</tbody>
</table>
LIB$ Routines
LIB$TRA_EBC_ASC

Condition Values Returned

SS$_NORMAL  Routine successfully completed.
LIB$_INVARG  If the destination string is a fixed-length string
and its length is not the same as the source
string length, or if the length of the input string
is greater than 65,535 characters, no translation
is attempted.
LIB$_INVCHA  One or more occurrences of an untranslatable
character have been detected during the
translation.
LIB$TRIM_FILESPEC
Fit Long File Specification into Fixed Field

The Fit Long File Specification into Fixed Field routine takes a file specification, such as an OpenVMS RMS resultant name string, and shortens it (if necessary) so that it fits into a field of fixed width.

Format

LIB$TRIM_FILESPEC old-filespec ,new-filespec [,word-integer-width] [,resultant-length]

Returns

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

Arguments

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

File specification to be trimmed. The old-filespec argument contains the address of a descriptor pointing to this file specification string.

The file specification should be an RMS resultant name string.

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

Trimmed file specification. The new-filespec argument contains the address of a descriptor pointing to this trimmed file specification string. LIB$TRIM_FILESPEC writes the trimmed file specification into new-filespec.

word-integer-width
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

Maximum field width desired. The word-integer-width argument is the address of an unsigned word that contains this maximum field width.

If omitted, the current length of new-filespec is used. If new-filespec is not a fixed-length string, you should specify word-integer-width to ensure that the desired width is used.
**LIB$ Routines**
**LIB$TRIM_FILESPEC**

**resultant-length**
OpenVMS usage: word_unsigned  
type: word (unsigned)  
access: write only  
mechanism: by reference

Length of the trimmed file specification, not including any blank padding or truncated characters. The **resultant-length** argument is the address of an unsigned word that contains this length. This is an optional argument.

**Description**
This routine trims file specifications in a consistent, predictable manner to fit in a fixed-length field using the same algorithm that HP software uses.

LIB$TRIM_FILESPEC allows compilers and other utilities which need to display file specifications in fixed-length fields, such as listing headers, to display file specifications in a consistent fashion.

If necessary to make the file specification fit into the specified field width, LIB$TRIM_FILESPEC removes portions of the file specification in this order:

1. Node (including access control)  
2. Device  
3. Directory  
4. Version  
5. Type

If, after removing all these fields, the file name is still longer than the field width, the file name is truncated and the alternate success status LIB$_STRTRU is returned.

LIB$TRIM_FILESPEC supports any string class for the **old-filespec** and **new-filespec** string arguments.

**Condition Values Returned**

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Success, but the output string was truncated. Significant characters of the trimmed file specification were truncated.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>

Any condition values returned by LIB$SCOPY_R_DX, or the $FILESCAN system service.
Example

PROGRAM TRIM_FILESPEC(INPUT,OUTPUT);
{+}
{ This PASCAL example program demonstrates the
{ use of LIB$TRIM_FILESPEC.}
{ -}

TYPE
    WORD = [WORD] 0..65535;

VAR
    INPUT_FILESPEC : VARYING [255] OF CHAR;
    OUTPUT_FILESPEC : VARYING [32] OF CHAR;
    RETURNED_STATUS : INTEGER;

[EXTERNAL] FUNCTION LIB$TRIM_FILESPEC(
    IN_FILE : VARYING [LEN1] OF CHAR;
    VAR OUT_FILE : VARYING [LEN2] OF CHAR;
    WIDTH    : WORD := %IMMED 0;
    OUT_LEN  : [REFERENCE] WORD := %IMMED 0
) : INTEGER; EXTERNAL;

[EXTERNAL] FUNCTION LIB$STOP(
    CONDITION_STATUS : [IMMEDIATE,UNSAFE] UNSIGNED;
    FAO_ARGS         : [IMMEDIATE,UNSAFE,LIST] UNSIGNED
) : INTEGER; EXTERNAL;

BEGIN
{+}
{ Start with a large INPUT_FILESPEC.}
{-}

INPUT_FILESPEC := 'DISK$NAME:\[DIRECTORY1.DIRECTORY2\]FILENAME.EXTENSION;1';
{+}
{ Use LIB$TRIM_FILESPEC to shorten it to fit a smaller variable.}
{-}

RETURNED_STATUS := LIB$TRIM_FILESPEC(
    INPUT_FILESPEC,
    OUTPUT_FILESPEC,
    SIZE(OUTPUT_FILESPEC.BODY));

IF NOT ODD(RETURNED_STATUS)
THEN
    LIB$STOP(RETURNED_STATUS);

{+}
{ Print out the original file name along with the
{ shortened file name.}
{-}

WRITELN('Original file specification ',INPUT_FILESPEC);
WRITELN('Shortened file specification ',OUTPUT_FILESPEC);

END.

This Pascal example program demonstrates the use of LIB$TRIM_FILESPEC.
The output generated by this program is as follows:

Original file specification DISK$NAME:\[DIRECTORY1.DIRECTORY2\]FILENAME.EXTENSION;1
Shortened file specification FILENAME.EXTENSION;1
LIB$TRIM_FULLNAME
Trim a Full Name to Fit into a Desired Output Field

The Trim a Full Name to Fit into a Desired Output Field routine trims a full name to fit into a desired output field. The trimming preserves the most significant part of the full name. †

Format

LIB$TRIM_FULLNAME  fullname, trimmed-nodename [,output-width] [,resultant-length]

Returns

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

Arguments

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

Full name to be trimmed. The fullname argument contains the address of a descriptor pointing to this full name string.

The error LIB$_INVARG is returned if fullname contains an invalid full name, points to a null string, or contains more than 1024 characters. The error LIB$_INVSTRDES is returned if fullname is an invalid descriptor.

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

Trimmed node name. The trimmed-nodename argument contains the address of a descriptor pointing to the trimmed node-name string. LIB$TRIM_FULLNAME writes the trimmed node name into the buffer pointed to by trimmed-nodename.

The error LIB$_INVSTRDES is returned if trimmed-nodename is an invalid descriptor.

The length field of the trimmed-nodename descriptor is not updated unless trimmed-nodename is a dynamic descriptor with a length less than the resultant trimmed node name. Refer to the OpenVMS RTL String Manipulation (STR$) Manual for dynamic string descriptor usage.

The trimmed-nodename argument contains an unusable result when LIB$TRIM_FULLNAME returns in error.

† No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
**output-width**
OpenVMS usage: word_unsigned
type: word (unsigned)
access: read only
mechanism: by reference

Field width desired for the trimmed node name. The `output-width` argument is the address of an unsigned word that contains this field width in bytes.

If `output-width` is omitted, the current length of `trimmed-nodename` is used. If `trimmed-nodename` is not a fixed-length string, specify `output-width` to ensure that the desired width is used.

If the lengths of both `trimmed-nodename` and `output-width` are specified, the length in `output-width` is used. In this case, if the current length of `trimmed-nodename` is smaller than the length of `output-width`, the output trimmed node name is truncated at the end, and the alternate successful status LIB$_STRTRU is returned.

**resultant-length**
OpenVMS usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Length of the trimmed node name. The `resultant-length` argument is the address of an unsigned word that contains this length in bytes.

The `resultant-length` argument contains an unusable result when LIB$TRIM_FULLNAME returns in error.

**Description**

This routine trims a full name to the length that fits the desired output field. It allows applications to trim long full names for displaying in a fixed-length field, such as listing headers, in a consistent manner.

Full names are validated. Valid full names are defined as full names expanded from using LIB$EXPAND_NODENAME. A node name must be expanded to a full name using LIB$EXPAND_NODENAME before calling LIB$TRIM_FULLNAME. The error LIB$_INVARF is returned if the input full name is invalid.

If the length of `fullname` is less than or equal to the desired output width, no trimming is performed, and `fullname` is returned in `trimmed-nodename`. Trailing blanks are padded if necessary.

Trimming is performed when the length of `fullname` is larger than the desired output width. The alternate successful status LIB$_STRTRU is returned.

The trimmed node name contains the significant part of the full name. This allows the most important information of a full name to be retained for display purposes. The significant part of a full name is determined by the underlying network services.

In a DECnet environment, trimming a DECnet-Plus full name results in the error condition LIB$_INVARF.
LIB$ Routines
LIB$TRIM_FULLNAME

If a usable short form of a node name is desired for display purposes, call LIB$COMPRESS_NODENAME first. If LIB$COMPRESS_NODENAME returns LIB$_STRTRU, LIB$TRIM_FULLNAME can then be used to return the trimmed node name.

LIB$TRIM_FULLNAME adds padding spaces to the end of the output buffer if the trimmed node name is shorter than the size of the output buffer. The argument resultant-length, if supplied, is set to the length of the trimmed node name, excluding any padding spaces.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_STRTRU</td>
<td>Routine successfully completed. Characters are truncated in the output buffer pointed to by trimmed-nodename.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid argument:</td>
</tr>
<tr>
<td></td>
<td>• fullname is invalid.</td>
</tr>
<tr>
<td></td>
<td>• fullname points to a null string.</td>
</tr>
<tr>
<td></td>
<td>• The length of the input full name is more than 1024 characters.</td>
</tr>
<tr>
<td></td>
<td>• The trimmed DECnet-Plus for OpenVMS node name is invalid in a DECnet for OpenVMS environment.</td>
</tr>
<tr>
<td>LIB$_INVSTRDES</td>
<td>Invalid string descriptor.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>

Any condition value returned by LIB$SCOPY_R_DX, or the $IPC DECnet service.

Examples

The following table gives some examples of the results of using LIB$TRIM_FULLNAME:

<table>
<thead>
<tr>
<th>Full Name</th>
<th>Size of Output Field</th>
<th>Trimmed Node Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td>3</td>
<td>NOD</td>
</tr>
<tr>
<td>NODE</td>
<td>8</td>
<td>NODE</td>
</tr>
<tr>
<td>DEC:.FOO.NODE</td>
<td>5</td>
<td>.NODE</td>
</tr>
<tr>
<td>DEC:.FOO.NODE</td>
<td>8</td>
<td>FOO.NODE</td>
</tr>
<tr>
<td>DEC:.FOO.NODE</td>
<td>20</td>
<td>DEC:.FOO.NODE</td>
</tr>
</tbody>
</table>
LIB$UNLOCK_IMAGE (Alpha and I64 Only)
Unlock an Image from Process Working Set

Unlocks the specified image in the process's working set.

Format

LIB$UNLOCK_IMAGE address

Returns

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

Arguments

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

Address of a byte within the image to be unlocked in the working set. If the address argument is 0, the current image (which contains the call to LIB$UNLOCK_IMAGE) is unlocked in the working set.

Description

LIB$UNLOCK_IMAGE unlocks the specified image in the process's working set.

This routine is typically used by a privileged user after the program, executing in kernel mode, lowers IPL to 0 or 2. Above IPL 2, paging is not allowed by the system. The program must access only pages valid in the process's working set. LIB$LOCK_IMAGE is used to lock the image in the working set.

Condition Values Returned

SS$_WASSET The specified image is unlocked in the working set and had previously been locked in the working set.

SS$_WASCLR The specified image is unlocked in the working set and had previously not been locked in the working set.

Other status codes returned by sys$lkwset_64.
LIB$ Routines
LIB$VERIFY_VM_ZONE

LIB$VERIFY_VM_ZONE
Verify a Zone

The Verify a Zone routine performs verification of a 32-bit zone. ↑

Format

LIB$VERIFY_VM_ZONE zone-id

Returns

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

Argument

zone-id
OpenVMS usage: identifier
type: longword (unsigned)
access: read only
mechanism: by reference

Zone identifier of the zone to be verified. The zone-id argument is the address of an unsigned longword that contains this zone identifier. A value of 0 indicates the 32-bit default zone.

Description

LIB$VERIFY_VM_ZONE verifies a zone. LIB$VERIFY_VM_ZONE performs verification of the zone header and scans all of the queues and lists maintained in the zone header; this includes the lookaside lists and the free lists. If the zone was created with LIB$M_VM_FREE_FILL0 or LIB$M_VM_FREE_FILL1, LIB$VERIFY_VM_ZONE also checks the contents of the free blocks.

As soon as an error is encountered, processing stops. If LIB$_BADZONE is returned, use the routine LIB$SHOW_VM_ZONE to dump the zone information.

You must have exclusive access to the zone while the verification is proceeding. Results are unpredictable if another thread of control modifies the zone while this routine is analyzing control data or scanning control blocks.

Condition Values Returned

SS$_NORMAL Routine successfully completed.
LIB$_BADZONE Invalid zone.
LIB$_INSVIRMEM Insufficient virtual memory.
LIB$_INVARG Invalid or null argument.
LIB$_WRONUMARG Wrong number of arguments.

↑ No support for arguments passed by 64-bit address reference or for use of 64-bit descriptors, if applicable, is planned for this routine.
LIB$VERIFY_VM_ZONE_64 (Alpha and I64 Only)

Verify a Zone

The Verify a Zone routine performs verification of a 64-bit zone.

Format

LIB$VERIFY_VM_ZONE_64 zone-id

Returns

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

Argument

zone-id
OpenVMS usage: identifier
type: quadword (unsigned)
access: read only
mechanism: by reference

Zone identifier of the zone to be verified. The zone-id argument is the address of an unsigned quadword that contains this zone identifier. A value of 0 indicates the 64-bit default zone.

Description

LIB$VERIFY_VM_ZONE_64 verifies a zone. LIB$VERIFY_VM_ZONE_64 performs verification of the zone header and scans all of the queues and lists maintained in the zone header; this includes the lookaside lists and the free lists. If the zone was created with the LIB$M_VM_FREE_FILL0 or LIB$M_VM_FREE_FILL1 algorithm, LIB$VERIFY_VM_ZONE_64 also checks the contents of the free blocks.

As soon as an error is encountered, processing stops. If LIB$_BADZONE is returned, use the routine LIB$SHOW_VM_ZONE_64 to dump the zone information.

You must have exclusive access to the zone while the verification is proceeding. Results are unpredictable if another thread of control modifies the zone while this routine is analyzing control data or scanning control blocks.

Condition Values Returned

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$_NORMAL</td>
<td>Routine successfully completed.</td>
</tr>
<tr>
<td>LIB$_BADZONE</td>
<td>Invalid zone.</td>
</tr>
<tr>
<td>LIB$_INVARG</td>
<td>Invalid or null argument.</td>
</tr>
<tr>
<td>LIB$_WRONUMARG</td>
<td>Wrong number of arguments.</td>
</tr>
</tbody>
</table>
LIB$ WAIT
Wait a Specified Period of Time

The Wait a Specified Period of Time routine places the current process into hibernation for the number of seconds specified in its argument.

Format

LIB$WAIT  seconds [,flags] [,float-type]

Returns

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

Arguments

seconds
OpenVMS usage:  floating_point
type:  F_floating
access:  read only
mechanism:  by reference

The number of seconds to wait. The seconds argument contains the address of an F-floating number that is this number.

The value is rounded to the nearest hundredth-second before use. Seconds must be between 0.0 and 100,000.0.

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

Control flags. The flags argument is the address of a longword integer that contains the control flags. The following flag is defined:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIB$K_NOWAKE</td>
<td>LIB$WAIT will not wake in the case of an interrupt.</td>
</tr>
</tbody>
</table>

This is an optional argument. If omitted, the default is 0, and LIB$WAIT will wake in the case of an interrupt.

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

lib–652
Float type. The float-type argument is the address of a longword integer that determines the floating-point type of the seconds argument. Use one of the following symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Floating-Point Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB$K_VAX_F</td>
<td>0</td>
<td>F_floating</td>
</tr>
<tr>
<td>LIB$K_VAX_D</td>
<td>1</td>
<td>D_floating</td>
</tr>
<tr>
<td>LIB$K_VAX_G</td>
<td>2</td>
<td>G_floating</td>
</tr>
<tr>
<td>LIB$K_VAX_H</td>
<td>3</td>
<td>H_floating</td>
</tr>
<tr>
<td>LIB$K_IEEE_S</td>
<td>4</td>
<td>IEEE_S_floating</td>
</tr>
<tr>
<td>LIB$K_IEEE_T</td>
<td>5</td>
<td>IEEE_T_floating</td>
</tr>
</tbody>
</table>

This is an optional argument. If omitted, the default is F_floating. F_floating is the required float-type when LIB$WAIT is called from a module written in a language that prototypes functions.

**Description**

LIB$WAIT rounds the value specified by seconds to the nearest hundredth-second, uses the $SCHDWK system service to schedule a wakeup for that interval, and then issues the $HIBER system service to hibernate until the wakeup occurs.

Because of other system activity, the length of time that the process actually waits may be somewhat longer than what was specified by seconds.

The process hibernates in the caller’s access mode; therefore, asynchronous system traps (ASTs) may be delivered while the process is hibernating. However, if the process hibernates at AST level, further ASTs can not be delivered.

When the LIB$K_NOWAIT control flag is used, LIB$WAIT makes use of the $SETIMR system service to schedule the wakeup, and then issues a $SYNCH system service call to check for the completion status. In this case, LIB$WAIT will not be interrupted by $WAKE. Use LIB$K_NOWAKE when it is necessary for the wait to be completed without interruption.

**Note**

The NOWAKE option makes use of the $SETIMR and $SYNCH system services. Because use of these services requires that an AST be delivered, you should not use LIB$WAIT with the LIB$K_NOWAKE control flag at AST level.

See the *HP OpenVMS System Services Reference Manual* for more information.

**Condition Values Returned**

- **SS$_NORMAL**: Routine successfully completed.
- **LIB$_INVAR**: Invalid argument. The value of seconds was less than 0 or greater than 100,000.0
LIB$ Routines
LIB$WAIT

LIB$_WRONUMARG Wrong number of arguments. An incorrect number of arguments was passed to LIB$WAIT.

Any condition values returned by the $SCHDWK or SETIMR system services, or by the RTL routine LIB$CVT_FTOF.

Example

IDENTIFICATION DIVISION.
PROGRAM-ID. T3.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 WAIT-TIME COMP-1.
  01 FLOAT-TYPE PIC 9(5) COMP VALUE 0.
PROCEDURE DIVISION.
p0. MOVE 10 TO WAIT-TIME.
    CALL "LIB$WAIT"
    USING BY REFERENCE WAIT_TIME, OMITTED,
          BY REFERENCE FLOAT-TYPE.
    STOP RUN.

This COBOL program demonstrates the use of LIB$WAIT on both OpenVMS VAX and OpenVMS Alpha and I64 systems. When run, the process performs a 10 second wait.
This part provides a detailed discussion of the routines provided by the OpenVMS RTL (CVT$) facility.
The Convert Floating-Point Data Type routine provides a simplified options-interface for converting a floating-point data type to another supported floating-point data type.

**Format**

```
CVT$CONVERT_FLOAT  input_value, input_type_code, output_value, output_type_code, options
```

**Returns**

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

**Arguments**

- **input_value**
  - **OpenVMS usage:** varying_arg
  - **type:** unspecified
  - **access:** read only
  - **mechanism:** by reference

The address of a data area containing a floating-point number that is to be converted. The `input_value` argument may contain floating-point data in F_Floating, D_Floating, G_Floating, H_Floating, IEEE_S_Floating, IEEE_T_Floating, IEEE_X_Floating, IBM_Long_Floating, IBM_Shoot_Floating, or CRAY_Floating format. The value of the `input_type_code` argument determines the format and size of the `input_value` argument.

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

The value of a longword bit mask specifying the type of floating-point data being passed in the `input_value` argument. Valid type codes are:

<table>
<thead>
<tr>
<th>input_type_code</th>
<th>Format</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$K_VAX_F</td>
<td>F_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_VAX_D</td>
<td>D_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_VAX_G</td>
<td>G_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_VAX_H</td>
<td>H_Floating</td>
<td>16</td>
</tr>
<tr>
<td>CVT$K_IEEE_S</td>
<td>IEEE_S_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_IEEE_T</td>
<td>IEEE_T_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_IEEE_X</td>
<td>IEEE_X_Floating</td>
<td>16</td>
</tr>
</tbody>
</table>
CVT$ Routine

CVT$CONVERT_FLOAT

<table>
<thead>
<tr>
<th>input_type_code</th>
<th>Format</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$K_IBM_LONG</td>
<td>IBM_Long_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_IBM_SHORT</td>
<td>IBM_Short_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_CRAY</td>
<td>CRAY_Floating</td>
<td>8</td>
</tr>
</tbody>
</table>

Declarations for the input_type_code argument are in the $CVTDEF module found in the system symbol libraries.

**output_value**

OpenVMS usage: varying_arg

type: unspecified

access: write only

mechanism: by reference

The address of a data area that receives the converted floating-point number.

The output_value argument can contain floating-point data in F_Floating, D_Floating, G_Floating, H_Floating, IEEE_S_Floating, IEEE_T_Floating, IEEE_X_Floating, IBM_Long_Floating, IBM_Short_Floating, or CRAY_Floating format.

The value of the output_type_code argument determines the size and format of the data placed into the output_value argument.

**output_type_code**

OpenVMS usage: longword_unsigned

type: longword (unsigned)

access: read only

mechanism: by value

The value of a longword bit mask specifying the type of floating-point data that the input_value argument will be converted into and returned in the output_value argument. Valid type codes are:

<table>
<thead>
<tr>
<th>output_type_code</th>
<th>Format</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$K_VAX_F</td>
<td>F_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_VAX_D</td>
<td>D_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_VAX_G</td>
<td>G_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_VAX_H</td>
<td>H_Floating</td>
<td>16</td>
</tr>
<tr>
<td>CVT$K_IEEE_S</td>
<td>IEEE_S_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_IEEE_T</td>
<td>IEEE_T_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_IEEE_X</td>
<td>IEEE_X_Floating</td>
<td>16</td>
</tr>
<tr>
<td>CVT$K_IBM_LONG</td>
<td>IBM_Long_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_IBM_SHORT</td>
<td>IBM_Short_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_CRAY</td>
<td>CRAY_Floating</td>
<td>8</td>
</tr>
</tbody>
</table>

Declarations for the output_type_code argument are in the $CVTDEF module found in the system symbol libraries.
options
OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by value

Conversion option specifier. The options argument is the address of a longword
bit mask in which each option bit set causes the corresponding option to be used
during the conversion.

The following options can be specified using the options argument:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$M_ROUND_TO_NEAREST</td>
<td>The default rounding option for conversions to IEEE data types. This IEEE Std. 754 rounding mode results in the representable output value nearest to the infinitely precise result. If the two nearest representable values are equally near, the one whose least significant bit is 0 is the result.</td>
</tr>
<tr>
<td>CVT$M_VAX_ROUNDING</td>
<td>The default rounding option for conversions to non-IEEE data types. Performs &quot;traditional&quot; style rounding. This mode results in the representable output value nearest to the infinitely precise result. If the two nearest representable values are equally near, the output value is the closest to either positive infinity or negative infinity, depending on the sign of the input value.</td>
</tr>
<tr>
<td>CVT$M_TRUNCATE</td>
<td>Round the output value toward zero (truncate).</td>
</tr>
<tr>
<td>CVT$M_ROUND_TO_POS</td>
<td>Round the output value toward positive infinity.</td>
</tr>
<tr>
<td>CVT$M_ROUND_TO_NEG</td>
<td>Round the output value toward negative infinity.</td>
</tr>
<tr>
<td>CVT$M_BIG_ENDIAN</td>
<td>Interprets IEEE data types as BigEndian.</td>
</tr>
<tr>
<td>CVT$M_ERR_UNDERFLOW</td>
<td>Report underflow conditions as errors.</td>
</tr>
</tbody>
</table>

Declarations for the options argument are in the $CVTDEF module found in the
system symbol libraries.

Description

CVT$CONVERT_FLOAT is a general-purpose, floating-point conversion routine
that converts any input_type_code floating-point data type into any output_
type_code floating-point data type. The conversion is subject to the options
specified in the options argument.
CVT$ Routine
CVT$CONVERT_FLOAT

---

**Note**

OpenVMS compilers do not support arithmetic operations for all of the above floating-point data types. Additional floating-point data types are supported by this routine for data conversion purposes only.

---

**Condition Values Returned**

- CVT$_NORMAL  Normal successful completion.
- CVT$_INPCONERR Input conversion error.
- CVT$_INVINPTYP Invalid input type code.
- CVT$_INVOPT Invalid option argument.
- CVT$_INVOUTTYP Invalid output type code.
- CVT$_INVVAL Input value was an invalid number or NaN.
- CVT$_NEGINF Input value was negative infinity.
- CVT$_OUTCONERR Output conversion error.
- CVT$_OVERFLOW Overflow detected during conversion.
- CVT$_POSINF Input value was positive infinity.
- CVT$_UNDERFLOW Underflow detected during conversion.

Return status values are in the $CVTMSG module found in the system symbol libraries.

---

**Example**

```c
/*
 ** ===================================================================
 ** Example of CVT$CONVERT_FLOAT
 ** -------------------------------------------------------------------
 ** This example program reads IEEE T floating-point numbers from an
 ** input file, converts them to VAX D floating-point numbers and
 ** writes the result to an output file.
 ** The input and output file names can be specified as the first and
 ** second arguments on the command line as follows:
 ** $ EXAMPLE IEEE_T_INPUT_FILE.DAT VAX_D_OUTPUT_FILE.DAT
 ** If the input or output files are not included on the command
 ** line then the program prompts the user for them.
 ** ===================================================================*/
#include <stdio.h>

unsigned long CVT$CONVERT_FLOAT(void *input_value, unsigned long input_type, void *output_value, unsigned long output_type, unsigned long options);
```

---

---
globalvalue CVT$K_VAX_D;
globalvalue CVT$K_IEEE_T;
globalvalue CVT$M_ROUND_TO_NEAREST;
globalvalue CVT$NORMAL;

main(int argc, char *argv[])
{
    double D_Float_number;
    unsigned long IEEE_Double_number[2];
    unsigned long options;
    char     in_filename[80];
    char     out_filename[80];
    FILE     *in_file, *out_file;
    unsigned long ret_status;

    /*
    ** Find out where we are going to get the data from.
    ** First look at the first argument of the command line.
    ** If nothing is there, then attempt to use IEEE_T_IN.DAT.
    **=================================================================
    */
    if (argc == 1)
    {
        printf("Enter input data file: [IEEE_T_IN.DAT]: ");
        if (gets(in_filename) == NULL)
            exit(1);
        if (strlen(in_filename) == 0)
            strcpy(in_filename, "IEEE_T_IN.DAT");
    }
    else
        strcpy(in_filename, argv[1]);

    /*
    ** Find out where we are going to put the output data.
    ** First look at the second argument of the command line.
    ** If nothing is there, then put it in VAX_D_OUT.DAT
    **=================================================================
    */
    if (argc <= 2)
    {
        printf("Enter output data file: [VAX_D_OUT.DAT]: ");
        if (gets(out_filename) == NULL)
            exit(1);
        if (strlen(out_filename) == 0)
            strcpy(out_filename, "VAX_D_OUT.DAT");
    }
    else
        strcpy(out_filename, argv[2]);

    /*
    ** Open the input and output files.
    **=================================================================
    */
    if ((in_file = fopen(in_filename, "r")) == NULL)
    {
        fprintf(stderr, "%s couldn't open file %s\n", argv[0], in_filename);
        exit(1);
    }

    out_file = fopen(out_filename, "wb");
    options    = CVT$M_ROUND_TO_NEAREST;
    ret_status = CVT$NORMAL;
CVT$ Routine
CVT$CONVERT_FLOAT

/*
 ** Read in each number from the file, convert it, and write it to
 ** the output file.
 ** =================================================================
 */
while ((fread (&IEEE_Double_number[0],
    sizeof(IEEE_Double_number),
    1, in_file) == 1) &&
    (ret_status == CVT$_NORMAL))
{
    ret_status = CVT$CONVERT_FLOAT(&IEEE_Double_number[0], CVT$K_IEEE_T,
        &D_Float_number, CVT$K_VAX_D,
        options);
    if (ret_status == CVT$NORMAL)
    {
        fwrite(&D_Float_number, sizeof(D_Float_number), 1, out_file);
        printf("Converted data: %lf.\n", D_Float_number);
    }
}
fclose(in_file);
fclose(out_file);
if (ret_status == CVT$NORMAL)
    exit(1);
else
    exit(ret_status);
CVT$FTOF
Convert Floating-Point Data Type

The Convert Floating-Point Data Type routine converts floating-point data types to other supported floating-point data types and allows additional control over the converted results. CVT$FTOF functionality is also available on other platforms supported by HP.

Format

\[ \text{status} = \text{CVT$FTOF} \quad \text{input\_value, input\_type\_code, output\_value, output\_type\_code, options} \]

Returns

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

The \text{status} return value is an unsigned longword bit mask containing the condition codes raised by the function. CVT$FTOF returns CVT$K_NORMAL; otherwise, it sets one or more recoverable and unrecoverable conditions. Use the following condition names to determine which conditions are set:

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Condition (always reported by default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$K_NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>CVT$M_INVALID_INPUT_TYPE</td>
<td>Invalid input type code.</td>
</tr>
<tr>
<td>CVT$M_INVALID_OUTPUT_TYPE</td>
<td>Invalid output type code.</td>
</tr>
<tr>
<td>CVT$M_INVALID_OPTION</td>
<td>Invalid option argument.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Condition (reported only if the CVTSM_REPORT_ALL option is selected)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$M_RESULT_INFINITE</td>
<td>Conversion produced an infinite result.¹</td>
</tr>
<tr>
<td>CVT$M_RESULT_DENORMALIZED</td>
<td>Conversion produced a denormalized result.¹</td>
</tr>
<tr>
<td>CVT$M_RESULT_OVERFLOW_RANGE</td>
<td>Conversion yielded an exponent greater than 60000 (8).²</td>
</tr>
<tr>
<td>CVT$M_RESULT_UNDERFLOW_RANGE</td>
<td>Conversion yielded an exponent less than 20000 (8).²</td>
</tr>
<tr>
<td>CVT$M_RESULT_UNNORMALIZED</td>
<td>Conversion produced an unnormalized result.³</td>
</tr>
</tbody>
</table>

¹For IEEE data type conversions.
²For CRAY data type conversions.
³For IBM data type conversions.
CVT$ Routine

CVT$FTOF

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Condition (reported only if the CVTSM_REPORT_ALL option is selected)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$M_RESULT_INVALID</td>
<td>Conversion result is either ROP (reserved operand), NaN (not a number), or closest equivalent. CRAY and IBM data types return 0.4</td>
</tr>
<tr>
<td>CVT$M_RESULT_OVERFLOW</td>
<td>Conversion resulted in overflow.4</td>
</tr>
<tr>
<td>CVT$M_RESULT_UNDERFLOW</td>
<td>Conversion resulted in underflow.4</td>
</tr>
<tr>
<td>CVT$M_RESULT_INEXACT</td>
<td>Conversion resulted in a loss of precision.4</td>
</tr>
</tbody>
</table>

4For all data type conversions.

Return status values are in the $CVTDEF module in the system symbol libraries.

Arguments

**input_value**

OpenVMS usage: varying_arg
type: unspecified
access: read only
mechanism: by reference

The address of a data area containing a floating-point number to be converted. The number can be floating-point data in one of the following formats:

- P_Floating
- D_Floating
- G_Floating
- H_Floating
- IEEE_S_Floating
- IEEE_T_Floating
- IEEE_X_Floating

The value of the **input_type_code** argument determines the format and size of the **input_value** argument.

**input_type_code**

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

The value of a longword bit mask specifying the type of floating-point data being passed in the **input_value** argument. Valid type codes are:

<table>
<thead>
<tr>
<th>Input_type_code</th>
<th>Format</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$K_VAX_F</td>
<td>F_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_VAX_D</td>
<td>D_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_VAX_G</td>
<td>G_Floating</td>
<td>8</td>
</tr>
</tbody>
</table>
Declarations for the **input_type_code** argument are in the $CVTDEF module found in the system symbol libraries.

**output_value**

OpenVMS usage: varying_arg  
type: unspecified  
access: write only  
mechanism: by reference

The address of a data area that receives the converted floating-point number. The number can be floating-point data in F_Floating, D_Floating, G_Floating, H_Floating, IEEE_S_Floating, IEEE_T_Floating, IEEE_X_Floating, Big_Endian_IEEE_S_Floating, Big_Endian_IEEE_T_Floating, Big_Endian_IEEE_X_Floating, IBM_Long_Floating, IBM_Short_Floating, or CRAY_Floating_Single format. The value of the **output_type_code** argument determines the size and format of the converted floating-point number.

**output_type_code**

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

The value of a longword bit mask specifying the type of floating-point data that the **input_value** argument will be converted into and returned in the **output_value** argument. Valid type codes are:
### CVT$ Routine

#### CVT$FTOF

<table>
<thead>
<tr>
<th>Output_type_code</th>
<th>Format</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT$K_BIG_ENDIAN_IEEE_T</td>
<td>Big_Endian_IEEE_T_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_BIG_ENDIAN_IEEE_X</td>
<td>Big_Endian_IEEE_X_Floating</td>
<td>16</td>
</tr>
<tr>
<td>CVT$K_IBM_LONG</td>
<td>IBM_Long_Floating</td>
<td>8</td>
</tr>
<tr>
<td>CVT$K_IBM_SHORT</td>
<td>IBM_Short_Floating</td>
<td>4</td>
</tr>
<tr>
<td>CVT$K_CRAY_SINGLE</td>
<td>CRAY_Floating</td>
<td>8</td>
</tr>
</tbody>
</table>

Declarations for the **output_type_code** argument are in the $CVTDEF module found in the system symbol libraries.

**options**

OpenVMS usage: mask_longword  
Type: longword (unsigned)  
Access: read only  
Mechanism: by value

Conversion option specifier. The **options** argument is the address of a longword bit mask in which each option bit set causes the corresponding option to be used during the conversion. Provide a zero (0) value to the **options** argument to select default behavior or choose one or more options (status condition option, rounding options, "FORCE" options, CRAY and IBM options) from the following tables. Specify only the options that apply to your conversion. A conflicting or incompatible **options** argument is reported as an error (CVT$M_INVALID_OPTION).

<table>
<thead>
<tr>
<th>Applicable Conversion</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>CVT$M_REPORT_ALL</td>
<td>Report all applicable status conditions as the default. The reporting of recoverable status conditions is disabled by default when this option is not used.</td>
</tr>
</tbody>
</table>

**Rounding Options**

<table>
<thead>
<tr>
<th>Applicable Conversion</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>CVT$M_ROUND_TO_NEAREST</td>
<td>The default rounding option for conversions to IEEE data types. This IEEE Std. 754 rounding mode results in the representable output value nearest to the infinitely precise result. If the two nearest representable values are equally near, the one whose least significant bit is 0 is the result.</td>
</tr>
</tbody>
</table>
### Rounding Options

<table>
<thead>
<tr>
<th>Applicable Conversion</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>CVT$M_BIASED_ROUNDING</td>
<td>The default rounding option for conversions to non-IEEE data types. Performs &quot;traditional&quot; style rounding. This mode results in the representable output value nearest to the infinitely precise result. If the two nearest representable values are equally near, the output value is the closest to either positive infinity or negative infinity depending on the sign of the input value.</td>
</tr>
<tr>
<td>All</td>
<td>CVT$M_ROUND_TO_ZERO</td>
<td>Round the output value toward zero (truncate).</td>
</tr>
<tr>
<td>All</td>
<td>CVT$M_ROUND_TO_POS</td>
<td>Round the output value toward positive infinity.</td>
</tr>
<tr>
<td>All</td>
<td>CVT$M_ROUND_TO_NEG</td>
<td>Round the output value toward negative infinity.</td>
</tr>
</tbody>
</table>

### "FORCE" Options

<table>
<thead>
<tr>
<th>Applicable Conversion</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>CVT$M_FORCE_ALL_SPECIAL_VALUES</td>
<td>Apply all applicable &quot;FORCE&quot; options for the current conversion.</td>
</tr>
<tr>
<td>IEEE</td>
<td>CVT$M_FORCE_DENORM_TO_ZERO&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Force a denormalized IEEE output value to zero.</td>
</tr>
<tr>
<td>IEEE</td>
<td>CVT$M_FORCE_INF_TO_MAX_FLOAT&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Force a positive IEEE infinite output value to +max_float and force a negative IEEE infinite output value to –max_float.</td>
</tr>
<tr>
<td>IEEE or VAX</td>
<td>CVT$M_FORCE_INVALID_TO_ZERO&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Force an invalid IEEE NaN (not a number) output value or a VAX ROP (reserved operand) output value to zero.</td>
</tr>
</tbody>
</table>

### CRAY Format Conversion Options

| CRAY                  | CVT$M_ALLOW_OVRFLW_RANGE_VALUES            | Allow an input/output exponent value > 60000 (8).                                                                                                 |
| CRAY                  | CVT$M_ALLOW_UDRFLW_RANGE_VALUES            | Allow an input/output exponent value < 20000 (8).                                                                                                 |

### IBM Format Conversion Option

| IBM                   | CVT$M_ALLOW_UNNORMALIZED_VALUES            | Allow unnormalized input arguments. Allow an unnormalized output value for a small value that would normalize to zero. |

<sup>1</sup>This option is valid only for conversions to IEEE output values.

<sup>2</sup>This option is valid only for conversions to IEEE or VAX output values.

The maximum representable floating-point values (max_float) for the IEEE_S_Floating, IEEE_T_Floating, IEEE_X_Floating, Big_Endian_IEEE_S_Floating, Big_Endian_IEEE_T_Floating, and Big_Endian_IEEE_X_Floating formats are:
CVT$ Routine
CVT$FTOF

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value for: max_float</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Decimal: 3.402823e38</td>
</tr>
<tr>
<td>T</td>
<td>Decimal: 1.797693134862316e308</td>
</tr>
<tr>
<td>X</td>
<td>Decimal: 1.189731495357231765085759326628007016196477e4932</td>
</tr>
</tbody>
</table>

Declarations for the options argument are in the $CVTDEF module found in the system symbol libraries.

Description

CVT$FTOF functionality is available on all HP platforms and is the floating-point conversion routine of choice for portability. When compared with the standard CVT$CONVERT_FLOAT routine, CVT$FTOF includes additional functionality and increased performance.

CVT$FTOF is a general-purpose floating-point conversion routine that converts any input_type_code floating-point data type into any output_type_code floating-point data type. The conversion is subject to the options specified in the options argument.

Note

OpenVMS compilers do not support arithmetic operations for all of the floating-point data types described here. Additional floating-point data types are supported by this routine for data conversion purposes only.
A

Addition
quadword times, lib–8
two's complement, lib–5

ASCII character set
EBCDIC translation from, lib–401, lib–636
EBCDIC translation to, lib–403, lib–640

$ASCTIM system service
RTL jacket routine, lib–561

ASTs (asynchronous system traps)
progress check, lib–17

B

Binary numbers, multiple precision, lib–5
Binary subtraction, lib–557
Binary tree
balanced, lib–341, lib–350
Bit fields
replace field, lib–371
return sign extended to longword, lib–178

C

CALLG (Call Routine with General Argument List) instruction, RTL routine to access, lib–28, lib–29
Character sets
translation of, lib–399
Character string routines, lib–30
CLI symbols
deleting, lib–143
getting value of, lib–287
RTL routines, lib–143, lib–287, lib–491
setting value of, lib–491
Condition handlers
establishing, lib–173
Condition values, lib–392
Conversions
general
descriptor to descriptor, lib–88
numeric text to binary, lib–101 to lib–103
of data type
descriptor to descriptor, lib–88

Converting
floating-point data types, cvt–3, cvt–9
CVT$CONVERT_FLOAT routine, cvt–3
CVT$FTOF routine, cvt–9
Cyclic redundancy check, lib–41, lib–43

D

Data types
conversion of
descriptor to descriptor, lib–88
Date/Time routines
LIB$CONVERT_DATE_STRING, lib–37
LIB$DATE_TIME, lib–105
LIB$DAY, lib–107
LIB$DAY_OF_WEEK, lib–109
LIB$FORMAT_DATE_TIME, lib–212
LIB$GET_DATE_FORMAT, lib–261
LIB$GET_MAXIMUM_DATE_LENGTH, lib–283
LIB$INIT_DATE_TIME_CONTEXT, lib–335
Decimal overflow detection, lib–130
Decimal text, converting to binary, lib–101, lib–103
DECnet
full name routines
LIB$BUILD_NODENAMES routine, lib–25
LIB$COMPARE_NODENAMES routine, lib–32
LIB$COMPRESS_NODENAMES routine, lib–34
LIB$EXPAND_NODENAMES routine, lib–175
LIB$FIT_NODENAMES routine, lib–205
LIB$GETFULLNAMES OFFSET routine, lib–268
LIB$GET_HOSTNAME routine, lib–270
LIB$TRIMFULLNAMES routine, lib–646
DECnet for OpenVMS
full name routines
DECnet-Plus for OpenVMS
full name routines
LIB$BUILD_NODENAMES routine, lib–25
DECnet-Plus for OpenVMS
full name routines (cont’d)
LIB$COMPARE_NODENAME routine, lib–32
LIB$COMPRESSION_NODENAME routine, lib–34
LIB$EXPAND_NODENAME routine, lib–175
LIB$FIT_NODENAME routine, lib–205
LIB$GET_FULLNAME_OFFSET routine, lib–268
LIB$GET_HOSTNAME routine, lib–270
LIB$TRIM_FULLNAME routine, lib–646
Descriptors
analysis of, lib–10
Directories
creating, lib–46
Division, extended precision, lib–155
Dynamic length strings
allocating, lib–497 to lib–499
deallocating, lib–494, lib–495
E
EBCDIC character set
ASCII translation from, lib–403, lib–640
ASCII translation to, lib–401, lib–636
EDIV (Extended Divide) instructions, RTL routine to access, lib–155
EMODD instructions, RTL routine to access, lib–157
EMODF instructions, RTL routine to access, lib–159
EMODG instructions, RTL routine to access, lib–161
EMODH instructions, RTL routine to access, lib–163
EMODS instructions, RTL routine to access, lib–165
EMODT instructions, RTL routine to access, lib–167
EMUL (Extended Multiply) instructions, RTL routine to access, lib–169
Event flags
reserve, lib–471
RTL routine to free, lib–218
status, lib–263
F
$FAOL system service
RTL jacket routine for, lib–565
$FAOL_64 system service
RTL jacket routine for, lib–567
$FO system service
RTL jacket routine for, lib–563
Faults, fix floating reserved operands, lib–208
FFx instructions, RTL routine to access, lib–183
Floating-point number conversions, cvt–3
BigEndian_IEEE_S_Floating, cvt–9
BigEndian_IEEE_T_Floating, cvt–9
BigEndian_IEEE_X_Floating, cvt–9
Cray_Floating, cvt–9
D_Floating, cvt–9
F_Floating, cvt–9
G_Floating, cvt–9
H_Floating, cvt–9
IBM_Long_Floating, cvt–9
IBM_Short_Floating, cvt–9
IEEE_S_Floating, cvt–9
IEEE_T_Floating, cvt–9
IEEE_X_Floating, cvt–9
Floating-point underflow detection, lib–210
Full name routines
See DECnet full name routines
G
$GETDVI system service
RTL jacket routine for, lib–231
$GETJPI system service
RTL jacket routine for, lib–237
$GETMSG system service
RTL jacket routine for, lib–569
$GETQUI system service
RTL jacket routine for, lib–242
$GETSYI system service
RTL jacket routine for, lib–247
H
Hexadecimal text, converting to binary, lib–101, lib–103
Hibernation
LIB$WAIT, lib–652
I
Integer overflow, lib–373
Invocation context
obtaining handle, lib–277
updating, lib–323, lib–444
Invocation handle
access routines, lib–305, lib–314
functions, lib–305, lib–314
K
Keywords, in keyword table, lib–380

L
Language, user’s choice of natural, lib–292
LIB$ADAIIW routine, lib–3
LIB$ADDX routine, lib–5
LIB$ADDTIMES routine, lib–8
LIB$ANALYZE_SDESC routine, lib–10
LIB$ANALYZE_SDESC_64 routine, lib–12
LIB$ASN_WTH_MBX routine, lib–14
LIB$AST_IN_PROG routine, lib–17
LIB$ATTACH routine, lib–19
LIB$BBCCI routine, lib–21
LIB$BBSSI routine, lib–23
LIB$BUILD_NODESPEC routine, lib–25
LIB$CALLG routine, lib–28
LIB$CALLG_64 routine, lib–29
LIB$CHAR routine, lib–30
LIB$COMPARE_NODENAME routine, lib–32
LIB$COMPRESS_NODENAME routine, lib–34
LIB$CONVERT_DATE_STRING routine, lib–37
LIB$CRC routine, lib–41
LIB$CRC_TABLE routine, lib–43
LIB$CREATE_DIR routine, lib–46
LIB$CREATE_USER_VM_ZONE routine, lib–50
LIB$CREATE_USER_VM_ZONE_64 routine, lib–54
LIB$CREATE_VM_ZONE routine, lib–57
LIB$CREATE_VM_ZONE_64 routine, lib–63
LIB$CRF_INS_KEY routine, lib–69
LIB$CRF_INS_REF routine, lib–71
LIB$CRF_OUTPUT routine, lib–74
LIB$CVT_FROM_INTERNAL_TIME routine, lib–80
LIB$CVT_TO_INTERNAL_TIME routine, lib–84
LIB$CVTS_FROM_INTERNAL_TIME routine, lib–82
LIB$CVTS_TO_INTERNAL_TIME routine, lib–86
LIB$CVT_DTB routine, lib–101, lib–103
LIB$CVT_DX_DX routine, lib–88
LIB$CVT_FROM_INTERNAL_TIME routine, lib–94
LIB$CVT_HTB routine, lib–101, lib–103
LIB$CVT_OTB routine, lib–101, lib–103
LIB$CVT_TO_INTERNAL_TIME routine, lib–97
LIB$CVT_VECTIM routine, lib–99
LIB$DATE_TIME routine, lib–105
LIB$DAY routine, lib–107
LIB$DAY_OF_WEEK routine, lib–109
LIB$DECIMAL_FAULT routine, lib–111
LIB$DEC_OVER routine, lib–130
LIB$DELETE_FILE routine, lib–132
LIB$DELETE_LOGICAL routine, lib–141
LIB$DELETE_SYMBOL routine, lib–143
LIB$DELETE_VM_ZONE routine, lib–145
LIB$DELETE_VM_ZONE_64 routine, lib–147
LIB$DIGIT_SEP routine, lib–149
LIB$DISABLE_CTRL routine, lib–151
LIB$DO_COMMAND routine, lib–153
LIB$DT_FORMAT, lib–213 to lib–214
LIB$DT_INPUT_FORMAT, lib–39, lib–261
LIB$DIV routine, lib–155
LIB$EMODD routine, lib–157
LIB$EMODF routine, lib–159
LIB$EMODG routine, lib–161
LIB$EMODH routine, lib–163
LIB$EMODS routine, lib–165
LIB$EMODT routine, lib–167
LIB$EMUL routine, lib–169
LIB$ENABLE_CTRL routine, lib–171
LIB$ESTABLISH routine, lib–173
LIB$EXPAND_NODESPEC routine, lib–175
LIB$EXTV routine, lib–178
LIB$EXTZV routine, lib–181
LIB$FFC routine, lib–183
LIB$FFS routine, lib–183
LIB$FID_TO_NAME routine, lib–185
LIB$FILE_SCAN routine, lib–188
LIB$FILE_SCAN_END routine, lib–190
LIB$FIND_FILE routine, lib–192
LIB$FIND_FILE_END routine, lib–196
LIB$FIND_IMAGE_SYMBOL routine, lib–197
LIB$FIND_VM_ZONE routine, lib–201
LIB$FIND_VM_ZONE_64 routine, lib–203
LIB$FIT_NODENAME routine, lib–205
LIB$FIXUP_FLT routine, lib–208
LIB$FLT_UNDER routine, lib–210
LIB$FORMAT_DATE_TIME routine, lib–212
LIB$FORMAT_SOGW PROT routine, lib–215
LIB$FREE_DATE_TIME_CONTEXT routine, lib–217
LIB$FREE_EF routine, lib–218
LIB$FREE_LUN routine, lib–219
LIB$FREE_TIMER routine, lib–220
LIB$FREE_VM routine, lib–221
LIB$FREE_VM_64 routine, lib–224
LIB$FREE_VM_PAGE routine, lib–227
LIB$GETDVI routine, lib–231
LIB$GETJPI routine, lib–237
LIB$GETQUI routine, lib–242
LIB$GETSYI routine, lib–247
LIB$GET_ACCCNAM routine, lib–251
LIB$GET_ACCCNAM_BY_CONTEXT routine, lib–253
LIB$GET_COMMAND routine, lib–255
LIB$GET_COMMON routine, lib–258

Index–3
Memory
allocation (cont’d)
freeing dynamic length strings, lib–494, lib–495
MOVC3 (Move Character 3 Operand) instruction,
RTL routine to access, lib–395
MOVC5 (Move Character 5 Operand) instruction,
RTL routine to access, lib–397
Multiplication, lib–157, lib–159, lib–161, lib–163,
lib–165, lib–167, lib–169

N
NBDS (numeric byte data string), lib–89
Node names
See DECnet full name routines

O
Octal text, converting to binary, lib–101, lib–103
Overflow detection
integer, lib–373

P
Pause program execution, lib–428
Polynomials
evaluating, lib–429, lib–431, lib–434, lib–436,
lib–438, lib–440

Q
Queue information, getting, lib–242
Queues
inserting an entry, lib–359 to lib–368
removing an entry, lib–450 to lib–459

R
Radix point symbol, returning the system’s,
lib–448
Reserved operands, fix floating-point faults,
lib–208
Run-time library routines
conversion, 1–10
CVT$, 1–10
library, 1–1
translated, 1–9

S
SCANC (SCAN Characters) instruction, RTL
routine to access, lib–480
Shareable images, activating, lib–197
Sign-extended longword fields, lib–178
Spawning a subprocess, lib–540

M
Mailboxes, lib–14
MATCHC (Match Characters) instruction, RTL
routine to access, lib–390
max_float boundary value, cvt–14
Memory
allocation
for dynamic length strings, lib–497, lib–499
String descriptors, lib–10, lib–12
Strings
  copying by descriptor, lib–482
  copying by reference, lib–484 to lib–486
dynamic length
  allocating, lib–497 to lib–499
  deallocating, lib–494, lib–495
position of substring in, lib–333
skipping characters in, lib–537
Subtraction
  quadword times, lib–559
  two’s complement, lib–558
Symbols
  See also CLI symbols
SYS$LANGUAGE logical name, lib–39 to lib–40,
  lib–213 to lib–214
Systemwide information, getting, lib–247

T
Time conversion routine, lib–94
  F_Floating value, lib–80
  S_Floating value, lib–82
Translation
  of character sets, lib–399
Tree, balanced binary, lib–341, lib–350

V
Virtual memory statistics
  returning, lib–551, lib–553
  showing, lib–505, lib–508