VMS
Utility Routines Manual

Order Number: AA–LA67A–TE

April 1988

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

Revision/Update Information: This manual supersedes the *VAX/VMS Utility Routines Reference Manual*, Version 4.4.

Software Version: VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

ZK4493

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.
In New Hampshire, Alaska, and Hawaii call 603-884-6660.
In Canada call 800-267-6215.
*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).
Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript™ printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

EXAMPLES

FIGURES

TABLES

Preface

Intended Audience

This manual is intended for programmers who want to invoke and manipulate VMS utilities from a program.

Document Structure

This document contains 13 chapters. Chapter 1 introduces the utility routines and describes the documentation format used to describe each set of utility routines, as well as the individual routines in each set.

Chapters 2 through 13 each describe one set of utility routines. Each chapter contains an introduction to that set of utility routines, a programming example to illustrate the use of the routines in the set, and a detailed decription of each routine.

Associated Documents

The VAX Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines*, contains useful information for all programmers. The *Introduction to VMS System Routines* also describes in detail the documentation format of the routine descriptions.

Some sets of utility routines documented in this manual invoke and manipulate utilities that have a command level interface. Consult the following manuals for a description of the command level interface:

- *VMS Access Control List Editor Manual*
- *VMS Command Definition Utility Manual*
- *VMS Convert and Convert/Reclaim Utility Manual*
- *VAX EDT Reference Manual*
- *VMS File Definition Language Facility Manual*
- *VMS Librarian Utility Manual*
- *VMS Sort/Merge Utility Manual*
- *VAX Text Processing Utility Manual*
- *VMS National Character Set Utility Manual*

Conventions

The documentation template for utility routines, which is described in the *Introduction to VMS System Routines,* details the conventions used in this manual, as well as the organizational approach used to document each utility routine.

The following table describes additional conventions that may appear in this manual.

Convention	Meaning
RET	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

New Set of Utility Routines

The following set of utility routines is new for VMS Version 5.0:

National Character Set (NCS) Utility routines

New Utility Routines

A new routine has been added to the VAX Text Processing Utility routines:

TPU$CLOSE_TERMINAL

A new routine has been added to the SORT/MERGE Utility routines:

SOR$DTYPE

1 Introduction to Utility Routines

1.1 Overview

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

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

When using a set of utility routines that performs the same tasks as a VMS utility, you should read the documentation for that utility; doing so will provide additional information about the tasks the routines can perform as a set. Following is a list of VMS utilities that have corresponding utility routines.

Utility or Editor	Utility Routines
Access Control List Editor	ACL Editor routine
Command Definition Utility	CLI routines
Convert and Convert/Reclaim Utilities	CONV routines
EDT Editor	EDT routines
File Definition Language Facility	FDL routines
Library Utility	LBR routines
VMS National Character Set Utility	NCS routines
Sort/Merge Utility	SOR routines
VAX Text Processing Utility	VAXTPU routines

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

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

1.1 Overview

The utility routines described in this manual are called in the same way as all other system routines in the VMS operating system environment, which is to say that utility routines conform to the VAX Procedure Calling and Condition Handling Standard.

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

- An introduction to the set of utility routines. This component discusses the utility routines as a group and explains how to use them.
- A series of descriptions of each utility routine in the set.

Most of the chapters also include a programming example that illustrates how the utility routines are used.

2 Access Control List (ACL) Editor Routine

2.1 Introduction to the ACL Editor Routine

This chapter describes the Access Control List (ACL) routine, ACLEDIT$EDIT. User-written applications can use this callable interface of the ACL Editor to manipulate Access Control Lists.

The ACL Editor is a VMS utility that allows you to create and maintain access control lists. Using ACLs, you can fine-tune the type of access to files, devices, global sections, logical name tables, or mailboxes available to system users.

Currently, the ACL Editor provides one callable interface that allows the application program to define an object for editing.

2.2 Example of Using the ACL Editor Routine

Example 2–1 shows a VAX BLISS program that calls the ACL Editor routine.

Example 2–1 Calling the ACL Editor with a VAX BLISS Program

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

BEGIN

LIBRARY 'SYS$LIBRARY:LIB';

ROUTINE STARTUP =

BEGIN

LOCAL
 STATUS,       ! Routine return status
 ITMLST  : BLOCKVECTOR [6, ITM$S_ITEM, BYTE];
       ! ACL editor item list

EXTERNAL LITERAL
 ACLEDIT$V_JOURNAL,
 ACLEDIT$V_PROMPT_MODE,

 ACLEDIT$C_OBJNAM,
 ACLEDIT$C_OBJTYP,
 ACLEDIT$C_OPTIONS;

EXTERNAL ROUTINE
 ACLEDIT$EDIT : ADDRESSING_MODE (GENERAL), ! Main routine

 CLI$GET_VALUE,     ! Get qualifier value
 CLI$PRESENT,     ! See if qualifier present
 LIB$PUT_OUTPUT,     ! General output routine
 STR$COPY_DX;     ! Copy string by descriptor
```

Example 2–1 Cont'd. on next page

Example 2–1 (Cont.) Calling the ACL Editor with a VAX BLISS Program

```
! Set up the item list to pass back to TPU so it can figure out what to do.

CH$FILL (0, 6*ITM$S_ITEM, ITMLST);
ITMLST[0, ITM$W_ITMCOD] = ACLEDIT$C_OBJNAM;
ITMLST[0, ITM$W_BUFSIZ] = %CHARCOUNT ('YOUR_OBJECT_NAME');
ITMLST[0, ITM$L_BUFADR] = $DESCRIPTOR ('YOUR_OBJECT_NAME');
ITMLST[1, ITM$W_ITMCOD] = ACLEDIT$C_OBJTYP;
ITMLST[1, ITM$W_BUFSIZ] = 4;
ITMLST[1, ITM$L_BUFADR] = UPLIT (ACL$C_FILE);
ITMLST[2, ITM$W_ITMCOD] = ACLEDIT$C_OPTIONS;
ITMLST[2, ITM$W_BUFSIZ] = 4;
ITMLST[2, ITM$L_BUFADR] = UPLIT (1 ^ ACLEDIT$V_PROMPT_MODE OR
     1 ^ ACLEDIT$V_JOURNAL);

RETURN ACLEDIT$EDIT (ITMLST);
END;      ! End of routine STARTUP

END
ELUDOM
```

2.3 ACL Editor Routine

The following pages describe the ACL Editor routine.

ACLEDIT$EDIT Edit Access Control List

The ACLEDIT$EDIT routine is used to create and modify an Access Control List (ACL) associated with any system object.

FORMAT

ACLEDIT$EDIT *item_list*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

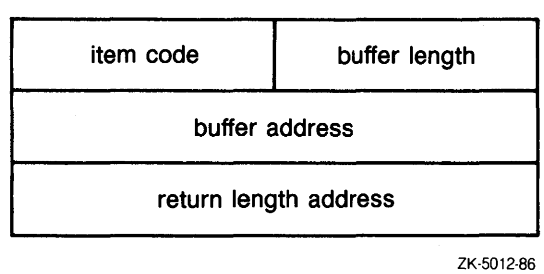
ARGUMENT

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

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

Each entry in an item list is in the standard format shown in Figure 2–1.

Figure 2–1 Item List



Following is a detailed description of each item list entry.

<table>
<tr><th>Item Identifier</th><th>Description</th></tr>
<tr><td>ACLEDIT$C_OBJNAM</td><td>Specifies the name of the object whose ACL is being edited.</td></tr>
<tr><td>ACLEDIT$C_OBJTYP</td><td>Specifies the type of the object whose ACL is being edited. These type codes are defined in $ACLDEF. The default object type is a file (ACL$C_FILE).</td></tr>
<tr><td>ACLEDIT$C_OPTIONS</td><td>Represents a longword bit mask of the various options available to control the editing session.
<table>
<tr><th>Flag</th><th>Function</th></tr>
<tr><td>ACLEDIT$V_JOURNAL</td><td>Indicates that the editing session is to be journaled.</td></tr>
<tr><td>ACLEDIT$V_RECOVER</td><td>Indicates that the editing session is to be recovered from an existing journal file.</td></tr>
<tr><td>ACLEDIT$V_KEEP_RECOVER</td><td>Indicates that the journal file used to recover the editing session is not to be deleted when the recovery is complete.</td></tr>
<tr><td>ACLEDIT$V_KEEP_JOURNAL</td><td>Indicates that the journal file used for the editing session is not to be deleted when the session ends.</td></tr>
<tr><td>ACLEDIT$V_PROMPT_MODE</td><td>Indicates that the session is to use automatic text insertion (prompting) to build new access control list entries (ACEs).</td></tr>
</table></td></tr>
<tr><td>ACLEDIT$C_BIT_TABLE</td><td>Specifies a vector of quadword descriptors to be used when parsing or formatting an ACE, which will define the names of the bits present in the access mask.</td></tr>
</table>

DESCRIPTION

You use the ACLEDIT$EDIT routine to create and modify an ACL associated with any system object.

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

If you also want to use a customized version of the ACL Editor section file, the logical name ACLEDT$SECTION should be defined. See the *VMS Access Control List Editor Manual* for more information.

CONDITION VALUES RETURNED

RMS$_xxx	See the *VMS Record Management Services Manual* for a description of RMS status codes.
TPU$_xxx	See Chapter 13 for a description of the TPU-specific condition values that may be returned by ACLEDIT$EDIT.

3 Command Language (CLI) Routines

3.1 Introduction to CLI Routines

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

The CLI routines include the following:

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

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

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

3.2 Example of Using the CLI Routines

Example 3–1 contains a command definition file (SUBCOMMANDS.CLD) and a FORTRAN program (INCOME.FOR). INCOME.FOR uses the command definitions in SUBCOMMANDS.CLD to process commands. To execute the example, enter the following commands:

```
$ SET COMMAND SUBCOMMANDS/OBJECT=SUBCOMMANDS
$ FORTRAN INCOME
$ LINK INCOME,SUBCOMMANDS
$ RUN INCOME
```

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

Example 3–1 Using the CLI Routines to Retrieve Information About Command Lines in a FORTRAN Program

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

MODULE INCOME_SUBCOMMANDS

DEFINE VERB ENTER
ROUTINE ENTER

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

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

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

! Write explanatory text
STATUS = LIB$PUT_OUTPUT
2 ('Subcommands: ENTER - FIX - REPORT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LIB$PUT_OUTPUT
2 ('Press CTRL/Z to exit')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get first subcommand
STATUS = CLI$DCL_PARSE (%VAL (0),
2                       INCOME_SUBCOMMANDS, ! CLD module
2                       LIB$GET_INPUT,      ! Parameter routine
2                       LIB$GET_INPUT,      ! Command routine
2                       'INCOME> ')         ! Command prompt
 ! Do it until user presses CTRL/Z
DO WHILE (STATUS .NE. RMS$_EOF)
! If no error on dcl_parse
IF (STATUS) THEN
! Dispatch depending on subcommand
STATUS = CLI$DISPATCH ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Do not signal warning again
ELSE IF (IBITS (STATUS, 0, 3) .NE. STS$K_WARNING) THEN
CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

Example 3–1 Cont'd. on next page

Example 3–1 (Cont.) Using the CLI Routines to Retrieve Information About Command Lines in a FORTRAN Program

```
! Get another subcommand
STATUS = CLI$DCL_PARSE (%VAL (0),
2                         INCOME_SUBCOMMANDS, ! CLD module
2                         LIB$GET_INPUT,      ! Parameter routine
2                         LIB$GET_INPUT,      ! Command routine
2                         'INCOME> ')         ! Command prompt
END DO
END

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

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

INTEGER FUNCTION REPORT ()
INTEGER STATUS,
2       CLI$GET_VALUE
CHARACTER*64 FILENAME
INTEGER*2    FN_SIZE
INCLUDE '($SSDEF)'
TYPE *, 'REPORT entered'
```

Example 3–1 Cont'd. on next page

Example 3–1 (Cont.) Using the CLI Routines to Retrieve Information About Command Lines in a FORTRAN Program

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

3.3 CLI Routines

The following pages describe the individual CLI routines.

CLI$DCL_PARSE Parse DCL Command String

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

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

command_string

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

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

If the command string contains a hyphen to indicate that the string is being continued, DCL uses the routine specified in the **prompt_routine** argument to obtain the rest of the string. The command string is limited to 256 characters. However, if the string is continued with a hyphen, CLI$DCL_PARSE can prompt for additional input until the total number of characters is 1024.

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

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

table

VMS usage: **char_string**
type: **unspecified**
access: **read only**
mechanism: **by reference**

Name of the module containing the command language description. The **table** argument is the address of the command table that describes the syntax by which the command line should be parsed. This is usually represented by a global symbol that is created by the Command Definition Utility when it processes the MODULE statement in the command definition file.

The command table is created with the DCL command SET COMMAND /OBJECT and is linked with your image.

param_routine

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

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

To specify the parameter routine, use the address of LIB$GET_INPUT or the address of a routine of your own that has the same three-argument calling format as LIB$GET_INPUT. See the description of LIB$GET_INPUT in the *VMS RTL Library (LIB$) Manual* for information about the calling format. The status returned by LIB$GET_INPUT must be success or the CLI$DCL_PARSE routine exits and propagates the error outward.

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

prompt_routine

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

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

To specify the prompt routine, use the address of LIB$GET_INPUT or the address of a routine of your own that has the same three-argument calling format as LIB$GET_INPUT. See the description of LIB$GET_INPUT in the *VMS RTL Library (LIB$) Manual* for information about the calling format. The status returned by LIB$GET_INPUT must be success or the CLI$DCL_PARSE routine exits and propagates the error outward.

prompt_string

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

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

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

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

DESCRIPTION

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

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

If you press CTRL/Z or if you return RMS$_EOF as a response to any prompt, CLI$DCL_PARSE immediately terminates and returns the status RMS$_EOF. If you enter a null string in response to a prompt for an entire or a continued command string (specified with the **prompt_routine** argument), CLI$DCL_PARSE terminates and returns the status CLI$_NOCOMD. If you enter a null string in response to a prompt for a required parameter, CLI$DCL_PARSE displays the prompt.

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

CONDITION VALUES RETURNED

CLI$_NORMAL	Normal successful completion.
CLI$_NOCOMD	Routine terminated. You entered a null string in response to a prompt from the **prompt_routine** argument, causing the CLI$DCL_PARSE routine to terminate.
RMS$_EOF	Routine terminated. You pressed CTRL/Z in response to a prompt, causing the CLI$DCL_PARSE routine to terminate.

CLI$DISPATCH Dispatch to Action Routine

The CLI$DISPATCH routine invokes the subroutine associated with the verb most recently parsed by a CLI$DCL_PARSE routine call.

FORMAT

CLI$DISPATCH *[userarg]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

ARGUMENT

userarg
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

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

DESCRIPTION

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

CONDITION VALUE RETURNED

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

CLI$GET_VALUE Get Value of Entity in Command String

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

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

entity_desc

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

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

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

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

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

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

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

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

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

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

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

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

If the **entity_desc** argument does not exist in the command table, CLI$GET_VALUE signals a syntax error (by means of the signaling mechanism described in the *VMS Run-Time Library Routines Volume*).

retdesc

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

Character string containing the value retrieved by CLI$GET_VALUE. The **retdesc** argument is the address of a string descriptor pointing to the buffer to receive the string value retrieved by CLI$GET_VALUE. The string is returned using the STR$COPY_DX VAX-11 Run-Time Library routine.

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

retlength

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

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

DESCRIPTION

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

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

$VERB	Describes the verb in the command string (the first four letters of the spelling as defined in the command table, instead of the string that was actually typed).
$LINE	Describes the entire command string as stored internally by DCL. In the internal representation of the command string, multiple spaces and tabs are removed, alphabetic characters are converted to uppercase, and comments are stripped. Integers are converted to decimal. If dates and times are specified in the command string, DCL fills in any defaulted fields. Also, if date-time strings (such as YESTERDAY) are used, DCL substitutes the corresponding absolute time value.

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

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

CONDITION VALUES RETURNED

CLI$_COMMA	Returned value terminated by a comma. This shows there are additional values in the list.
CLI$_CONCAT	Returned value concatenated to the next value with a plus sign. This shows there are additional values in the list.
SS$_NORMAL	Returned value terminated by a blank or an end-of-line. This shows that the value is the last, or only, value in the list.
CLI$_ABSENT	No value returned. The value is not present, or the last value in the list was already returned.

CLI$PRESENT Determine Presence of Entity in Command String

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

FORMAT

CLI$PRESENT *entity_desc*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

entity_desc
VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

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

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

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

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

If the **entity_desc** argument does not exist in the command table, CLI$PRESENT signals a syntax error (by means of the signaling mechanism described in the *VMS Run-Time Library Routines Volume*).

DESCRIPTION

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

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

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

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

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

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

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

CONDITION VALUES RETURNED

CLI$_PRESENT	Specified entity present in the command string. This status is returned for all entities except local qualifiers and positional qualifiers that are used locally.
CLI$_NEGATED	Specified qualifier present in negated form (with /NO) and used as a global qualifier.
CLI$_LOCPRES	Specified qualifier present and used as a local qualifier.
CLI$_LOCNEG	Specified qualifier present in negated form (with /NO) and used as a local qualifier.
CLI$_DEFAULTED	Specified entity not present, but there is a default value.
CLI$_ABSENT	Specified entity not present, and there is no default value.

4 Convert (CONV) Routines

4.1 Introduction to Convert Routines

This chapter describes the Convert routines. These routines perform the functions of both the VMS RMS Convert and Convert/Reclaim Utilities.

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

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

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

These routines cannot be called from AST level.

4.2 Examples of Using the Convert Routines

Example 4–1 shows how to use the Convert routines in a FORTRAN program.

Example 4–1 Using the Convert Routines in a FORTRAN Program

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

*           Program declarations

        IMPLICIT        INTEGER*4 (A - Z)

*           Set up parameter list: number of options, CREATE,
*           NOSHARE, FAST_LOAD, MERGE, APPEND, SORT, WORK_FILES,
*           KEY=0, NOPAD, PAD CHARACTER, NOTRUNCATE,
*           NOEXIT, NOFIXED_CONTROL, FILL_BUCKETS, NOREAD_CHECK,
*           NOWRITE_CHECK, FDL, and NOEXCEPTION.
*
        INTEGER*4       OPTIONS(19)
        1  /18,1,0,1,0,0,1,2,0,0,0,0,0,0,0,0,0,1,0/
```

Example 4–1 Cont'd. on next page

Example 4–1 (Cont.) Using the Convert Routines in a FORTRAN Program

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

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

*          Declare the file names.

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

*          Call the routines in their required order.

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

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

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

*          Display the statistics information.

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

        END
```

Example 4–2 shows how to use the Convert routines in a MACRO program.

Example 4–2 Using the Convert Routines in a MACRO Program

```
;
        .TITLE CONVSTAT.MAR
;
;  This module calls the routines that perform the functions
;  of the Convert Utility.  It creates an indexed output file
;  named CUSTDATA.DAT from the specifications in an FDL file
;  named INDEXED.FDL, and loads CUSTDATA.DAT with records from
;  the sequential file SEQ.DAT.  No exception file is created.
;  This module also returns all the CONVERT statistics.
;
;  Declare the file names.
;
FILEIN:         .ASCID          /SEQ.DAT/
FILEOUT:        .ASCID          /CUSTDATA.DAT/
FDLFILE:        .ASCID          /INDEXED.FDL/
```

Example 4–2 Cont'd. on next page

Example 4–2 (Cont.) Using the Convert Routines in a MACRO Program

```
;
;  Set up parameter list.
;
PARAM_LIST:     .LONG  18     ;NUMBER OF LONGWORDS FOLLOWING
                .LONG   1     ;CREATE
                .LONG   0     ;NOSHARE
                .LONG   1     ;FAST_LOAD
                .LONG   0     ;MERGE
                .LONG   0     ;APPEND
                .LONG   1     ;SORT
                .LONG   2     ;WORK_FILES
                .LONG   0     ;KEY=0
                .LONG   0     ;NOPAD
                .LONG   0     ;PAD CHARACTER
                .LONG   0     ;NOTRUNCATE
                .LONG   0     ;NOEXIT
                .LONG   0     ;NOFIXED_CONTROL
                .LONG   0     ;FILL_BUCKETS
                .LONG   0     ;NOREAD_CHECK
                .LONG   0     ;NOWRITE_CHECK
                .LONG   1     ;FDL
                .LONG   0     ;NOEXCEPTION
;
;  Have to use Formatted ASCII Output (FAO) conversion
;  Declare FAO info for statistics
;
FAO_DESC:       .LONG         132
                .LONG         FAO_BUFFER
FAO_BUFFER:     .BLKB         132
FAO_LEN:        .BLKL         1
OUTSTUFF:       .ASCID        #Number of files processed: !UL !/-
Number of records: !UL !/-
Number of exception records: !UL !/-
Number of valid records: !UL !/#
;
;  Have to pass a longword to the CONV$CONVERT ROUTINE with the
;  number of statistics that we want.  There are 4 -- number of
;  files, number of records, exception records, good records,
;  in that order.
;
STATSBLK:       .LONG 4       ;The value 4 is the number of statistics
                              ;that we want.  we pass this value to
                              ;the END_CONVERT routine.
;
STATS:          .BLKL 4       ;Where we place the statistics.  This block
                              ;must follow the longword that tells how
                              ;many stats we want.
;
TIMES:          .BLKL 5       ;Where we place the timing info.
;
;  Declare the external routines.
;
        .EXTRN  CONV$PASS_FILES,CONV$PASS_OPTIONS,CONV$CONVERT,-
                LIB$PUT_OUTPUT,LIB$INIT_TIMER,LIB$SYS_FAOL
;
        .ENTRY  CONV,^M<R2,R3,R4,R5,R6,R7>    ;SAVE THOSE REGISTERS;
```

Example 4–2 Cont'd. on next page

Example 4–2 (Cont.) Using the Convert Routines in a MACRO Program

```
;
;  Perform operations.  Push addresses on arg stack, call routines.
;
        PUSHAL  TIMES
        CALLS   #1,G^LIB$INIT_TIMER      ;Start the timer
;
        PUSHAL  FDLFILE
        PUSHAL  FILEOUT
        PUSHAL  FILEIN                   ;Push filenames on arg stack
        CALLS   #3,G^CONV$PASS_FILES     ;Pass filenames
        BLBC    R0,10$
;
        PUSHAL  PARAM_LIST               ;Push parameter list
        CALLS   #1,G^CONV$PASS_OPTIONS   ;Make the second call
        BLBC    R0,10$
;
        PUSHAL  STATSBLK                 ;Push address of the number of
                                         ;Statistics
        CALLS   #1,G^CONV$CONVERT        ;Perform conversion
        BLBC    R0,10$
;
;  Now need an FAO routine to format the counts
;

        $FAOL_S CTRSTR=OUTSTUFF,OUTLEN=FAO_LEN,OUTBUF=FAO_DESC,-
                PRMLST=STATS
        BLBC    R0,10$
;
        PUSHAL  FAO_DESC                 ;Push output buffer on stack
        CALLS   #1,G^LIB$PUT_OUTPUT      ;Send the output buffer to
                                         ;SYS$OUTPUT
        BLBC    R0,10$

;
;  Display times
;
        PUSHAL  TIMES
        CALLS   #1,G^LIB$SHOW_TIMER
        BLBC    R0,10$
        MOVL    #SS$_NORMAL,R0
;
10$:    RET
;
        .END    CONV
```

Example 4–3 shows how to use the CONV$RECLAIM routine in a FORTRAN program.

Example 4–3 Using the CONV$RECLAIM Routine in a FORTRAN Program

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

        IMPLICIT        INTEGER*4 (A - Z)

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

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

*          Declare the input file.

        CHARACTER       IN_FILE*9 /'PROL3.DAT'/

*          Call the routine.

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

*          Display the statistics.

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

        END
```

Example 4–4 shows how to use the CONV$RECLAIM routine in a MACRO program.

Example 4–4 Using the CONV$RECLAIM Routine in a MACRO Program

```
;
        .TITLE CONVREC.MAR
;
;
;  This module calls the routine that performs the
;  function of the CONVERT/RECLAIM Utility.  It reclaims
;  empty buckets from an indexed file named PROL3.DAT.
;
;  This module also returns all of the CONVERT/RECLAIM
;  statistics.
;
;  Declare the file name.
;
FILEIN:         .ASCID  /PROL3.DAT/
;
;  Declare statistics blocks
;
OUTSTATS:       .LONG   4
                .BLKL   4
;
;  Declare FAO info for statistics
;
;
FAO_DESC:       .LONG   132
                .LONG   FAO_BUFFER
FAO_BUFFER:     .BLKB   132
FAO_LEN:        .BLKL   1
OUTSTUFF:       .ASCID  #Data buckets scanned: !UL !/-
Data buckets reclaimed: !UL !/-
Index buckets reclaimed: !UL !/-
Total buckets reclaimed: !UL !/#
;
;  Looking for four statistics back from the end call.
;  Use FAO conversion.
;
;  Declare the external routines.
;
        .EXTRN  CONV$RECLAIM,LIB$PUT_OUTPUT
;
        .ENTRY  CONV,^M<>
;
;
;  Perform operations.  Push addresses on arg stack, call
;  routines.
;
        PUSHAL  OUTSTATS
        PUSHAL  FILEIN                  ;PUSH FILENAME ON ARG STACK
        CALLS   #2,G^CONV$RECLAIM       ;PASS FILENAME
        BLBC    R0,10$
```

Example 4–4 Cont'd. on next page

Example 4–4 (Cont.) Using the CONV$RECLAIM Routine in a MACRO Program

```
;
;
;  Now need an FAO routine to format the counts.
;
        $FAOL_S  CTRSTR=OUTSTUFF,OUTLEN=FAO_LEN,OUTBUF=FAO_DESC,-
                 PRMLST=OUTSTATS+4
        BLBC     RO,10$
;
        PUSHAL   FAO_DESC                ;PUSH OUTPUT BUFFER ON STACK
        CALLS    #1,G^LIB$PUT_OUTPUT     ;SEND THE OUTPUT BUFFER TO
                                         ;SYS$OUTPUT
        BLBC     RO,10$
        MOVL     #SS$_NORMAL,RO
10$:    RET
;
        .END CONV
```

4.3 Convert Routines

The following pages describe the individual Convert routines.

CONV$CONVERT Initiate Conversion

The CONV$CONVERT routine uses the Convert Utility to perform the actual conversion begun with CONV$PASS_FILES and CONV$PASS_OPTIONS. Optionally, the routine can return statistics about the conversion.

FORMAT

CONV$CONVERT *[status_block_address] [,flags]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

status_block_address

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

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

number of statistics
number of files
number of records
number of exception records
number of valid records

flags

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

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

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

This argument is optional. By default, an error status is returned rather than signalled.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
CONV$_BADBLK	Invalid option block.
CONV$_BADLOGIC	Internal logic error detected.
CONV$_BADSORT	Error trying to sort input file.
CONV$_CLOSEIN	Error closing file specification as input.
CONV$_CLOSEOUT	Error closing file specification as output.
CONV$_CONFQUAL	Conflicting qualifiers.
CONV$_CREA_ERR	Error creating output file.
CONV$_CREATEDSTM	File specification has been created in stream format.
CONV$_DELPRI	Cannot delete primary key.
CONV$_DUP	Duplicate key encountered.
CONV$_EXTN_ERR	Unable to extend output file.
CONV$_FATALEXC	Fatal exception encountered.
CONV$_FILLIM	Exceeded open file limit.
CONV$_IDX_LIM	Exceeded maximum index level.
CONV$_ILL_KEY	Illegal key or value out of range.
CONV$_INP_FILES	Too many input files.
CONV$_INSVIRMEM	Insufficient virtual memory.
CONV$_KEY	Invalid record key.
CONV$_LOADIDX	Error loading secondary index *n*.
CONV$_NARG	Wrong number of arguments.
CONV$_NOKEY	No such key.
CONV$_NOTIDX	File is not an indexed file.
CONV$_NOTSEQ	Output file is not a sequential file.
CONV$_NOWILD	No wildcard permitted.
CONV$_OPENEXC	Error opening exception file specification.
CONV$_OPENIN	Error opening file specification as input.
CONV$_OPENOUT	Error opening file specification as output.
CONV$_ORDER	Routine called out of order.

CONV$_PAD	PAD option ignored; output record format not fixed.
CONV$_PROERR	Error reading prolog.
CONV$_PROL_WRT	Prolog write error.
CONV$_READERR	Error reading file specification.
CONV$_REX	Record already exists.
CONV$_RMS	Record caused RMS severe error.
CONV$_RSK	Record shorter than primary key.
CONV$_RSZ	Record does not fit in block/bucket.
CONV$_RTL	Record longer than maximum record length.
CONV$_RTS	Record too short for fixed record format file.
CONV$_SEQ	Record not in order.
CONV$_UDF_BKS	Cannot convert UDF records into spanned file.
CONV$_UDF_BLK	Cannot fit UDF records into single block bucket.
CONV$_VALERR	Specified value is out of legal range.
CONV$_VFC	Record too short to fill fixed part of VFC record.
CONV$_WRITEERR	Error writing file specification.

CONV$PASS_FILES Specify Conversion Files

The CONV$PASS_FILES routine specifies a file to be converted using the CONV$CONVERT routine.

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

input_filespec

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

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

output_filespec

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

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

fdl_filespec

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

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

exception_filespec

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

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

flags

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

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

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

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

This argument is optional. By default, an error status is returned rather than signalled.

DESCRIPTION

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

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

The additional calls must immediately follow the original call that specified the output file specification. You may specify as many as 9 additional files for a maximum total of 10.

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

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
CONV$_INP_FILES	Too many input files.
CONV$_INSVIRMEM	Insufficient virtual memory.
CONV$_NARG	Wrong number of arguments.
CONV$_ORDER	Routine called out of order.

CONV$PASS_OPTIONS Specify Processing Options

The CONV$PASS_OPTIONS routine specifies which qualifiers are to be used by the Convert Utility (CONVERT).

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

parameter_list_address

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

Parameter list specifying information about the CONVERT qualifiers. The **parameter_list_address** argument is the address of a variable-length array of longwords. The first longword in the array is the number of parameters in the array. Each subsequent longword in the array (from the second one on) is associated with one of the CONVERT qualifiers. These functions are described in the *VMS Convert and Convert/Reclaim Utility Manual.*

To set one of the CONVERT qualifiers, you place a *1* in the longword associated with that qualifier. If you do not want to set one of the qualifiers (which has the same effect as using the negative form of the qualifier on the CONVERT command), you place a *0* in the correct longword.

If you do not specify **parameter_list_address**, then the following default values apply. You can also take all default values by passing the address of a longword that contains *0*, which means a parameter list of *0* longwords.

If you have specified all the values you want set, you may want to take the default values for all subsequent qualifiers in the list. You may omit the subsequent ones if you give the array length in the first longword. This is why the first longword contains a count of the qualifiers.

The qualifiers must appear in the following order.

Qualifier	Default Value (in Longwords)	Default CONVERT Value
CREATE	1	/CREATE
SHARE	0	/NOSHARE
FAST_LOAD	1	/FAST_LOAD
MERGE	0	/NOMERGE
APPEND	0	/NOAPPEND
SORT	1	/SORT
WORK_FILES	2	/WORK_FILES=2
KEY	0	/KEY=0
PAD	0	/NOPAD
Pad character	0	Pad character=0
TRUNCATE	0	/NOTRUNCATE
EXIT	0	/NOEXIT
FIXED_CONTROL	0	/NOFIXED_CONTROL
FILL_BUCKETS	0	/NOFILL_BUCKETS
READ_CHECK	0	/NOREAD_CHECK
WRITE_CHECK	0	/NOWRITE_CHECK
FDL	0	/NOFDL
EXCEPTION	0	/NOEXCEPTION
PROLOG	No default	System or process default

If you want to use the default null character for the PAD qualifier, you should specify *0* in the pad character longword. You can also specify the default null character by omitting the pad character longword. However, in this case, you must also take the default values for all subsequent qualifiers. To specify a pad character other than *0*, place the ASCII value of the character you want to use in the PAD qualifier longword.

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

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

If you specify the PROLOG longword, note that this overrides the KEY PROLOG attribute supplied by the FDL file. You must supply one of three values for the PROLOG longword if you use it. The three values are *0*, *2*, and *3*. The value *0* means that you want to use the system or process prolog type. The value *2* means that you want to create a Prolog 1 or 2 file in all instances, even when circumstances would allow you to create a Prolog 3 file. The value *3* means that you want to create a Prolog 3 file and, if circumstances do not allow you to, you want the conversion to fail.

flags

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

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

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

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

This argument is optional. By default, an error status is returned rather than signalled.

DESCRIPTION

The following example shows how to use the option array to reflect the CONVERT command:

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

A:	12	Specifies that 12 longwords follow
	1	Specifies the /CREATE option
	0	Specifies the /NOSHARE option
	1	Specifies the /FASTLOAD option
	0	Specifies the /NOMERGE option
	0	Specifies the /NOAPPEND option
	1	Specifies the /SORT option
	6	Specifies the /WORKFILES=6 option
	0	Specifies the /KEY=0 option
	0	Specifies the /NOPAD option
	0	Specifies the null pad character
	0	Specifies the /NOTRUNCATE option
	1	Specifies the /EXIT option

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
CONV$_BADBLK	Invalid option block.
CONV$_CONFQUAL	Conflicting qualifiers.
CONV$_INSVIRMEM	Insufficient virtual memory.
CONV$_NARG	Wrong number of arguments.
CONV$_OPENEXC	Error opening exception file *filespec.*
CONV$_ORDER	Routine called out of order.

CONV$RECLAIM Invoke Convert/Reclaim Utility

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

FORMAT

CONV$RECLAIM *input_filespec [,statistics_blk]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

input_filespec

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

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

statistics_blk

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

Bucket reclamation statistics. The **statistics_blk** argument is the address of a variable-length array of longwords to receive statistics on the bucket reclamation. You can choose which statistics you want returned by specifying a number in the first element of the array. This number determines how many of the four possible statistics the routine returns. Depending on the number chosen, the routine returns the statistics in the statistics array in the order specified by the following format:

A: Number of statistics
Data buckets scanned
Data buckets reclaimed
Index buckets reclaimed
Total buckets reclaimed

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
CONV$_BADLOGIC	Internal logic error detected.
CONV$_INSVIRMEM	Insufficient virtual memory.
CONV$_INVBKT	Invalid bucket at VBN *n*.
CONV$_NOTIDX	File is not an index file.
CONV$_OPENIN	Error opening *filespec* as input.
CONV$_PLV	Unsupported prolog version.
CONV$_PROERR	Error reading prolog.
CONV$_PROL_WRT	Prolog write error.
CONV$_READERR	Error reading *filespec*.
CONV$_NOWILD	No wildcard permitted.
CONV$_WRITEERR	Error writing output file.

5 Data Compression/Expansion (DCX) Routines

5.1 Introduction to DCX Routines

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

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

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

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

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

The procedure for using the DCX routines to perform data compression and expansion consists of three major steps. The list under each of the following steps shows the DCX routines used to perform that step:

1. Analyze some or all of the data records in the data file to produce a mapping function (or map).

   DCX$ANALYZE_INIT
   DCX$ANALYZE_DATA
   DCX$MAKE_MAP
   DCX$ANALYZE_DONE

2. Compress the data records in the file on the basis of the mapping function.

   DCX$COMPRESS_INIT
   DCX$COMPRESS_DATA
   DCX$COMPRESS_DONE

3. Expand the compressed data records on the basis of the mapping function.

   DCX$EXPAND_INIT
   DCX$EXPAND_DATA
   DCX$EXPAND_DONE

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

5.2 Examples of Using the DCX Routines

Examples 5–1 and 5–2 show how to use the DCX routines in VAX FORTRAN programs.

Example 5–1 Example of Compressing a File in a VAX FORTRAN Program

```
PROGRAM COMPRESS_FILES
! COMPRESSION OF FILES

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

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

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

! Logical end-of-file
LOGICAL EOF
! record buffers;  32767 is maximum record size
CHARACTER*32767 RECORD,
2               RECORD2
INTEGER RECORD_LEN,
2       RECORD2_LEN

! user routine
INTEGER GET_MAP,
2       WRITE_MAP
```

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Example of Compressing a File in a VAX FORTRAN Program

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

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


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

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Example of Compressing a File in a VAX FORTRAN Program

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

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

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

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Example of Compressing a File in a VAX FORTRAN Program

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

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

END

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

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Example of Compressing a File in a VAX FORTRAN Program

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

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

! Logical end-of-file
LOGICAL EOF
! record buffer;  32767 is the maximum record size
CHARACTER*32767 RECORD
INTEGER RECORD_LEN

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

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

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Example of Compressing a File in a VAX FORTRAN Program

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

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

END

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

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

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

END
```

Example 5–2 Example of Expanding a Compressed File in a VAX FORTRAN Program

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

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

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

! user routine
EXTERNAL EXPAND_DATA

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

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

Example 5–2 Cont'd. on next page

Example 5–2 (Cont.) Example of Expanding a Compressed File in a VAX FORTRAN Program

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

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

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

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

! context variable
INTEGER CONTEXT

! logical end_of_file
LOGICAL EOF
! record buffers
CHARACTER*32767 RECORD,
2               RECORD2
INTEGER RECORD_LEN,
2       RECORD2_LEN
```

Example 5–2 Cont'd. on next page

Example 5–2 (Cont.) Example of Expanding a Compressed File in a VAX FORTRAN Program

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

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

5.3 DCX Routines

The following pages describe the individual DCX routines.

DCX$ANALYZE_DATA Perform Statistical Analysis on a Data Record

The DCX$ANALYZE_DATA routine performs statistical analysis on a data record.

The results of the analysis are accumulated internally in the context area and are used by the DCX$MAKE_MAP routine to compute the mapping function.

FORMAT

DCX$ANALYZE_DATA *context ,record*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

record

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

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

DESCRIPTION

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

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

CONDITION VALUES RETURNED

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

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

DCX$ANALYZE_DONE Specify Analysis Completed

The DCX$ANALYZE_DONE routine deletes the context area and sets the context variable to zero, thus undoing the work of the DCX$ANALYZE_INIT routine.

You call DCX$ANALYZE_DONE after data records have been analyzed and the DCX$MAKE_MAP routine has created the map.

FORMAT

DCX$ANALYZE_DONE *context*

RETURNS

VMS usage: **cond_value**
type: **longword**
access: **write only**
mechanism: **by value**

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

context
VMS usage: **context**
type: **longword**
access: **write only**
mechanism: **by reference**

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

CONDITION VALUES RETURNED

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

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

DCX$ANALYZE_INIT Initialize Analysis Context

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

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

item_code

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

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

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

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

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

item_value

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

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

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

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

Item Code	Corresponding Item Value
DCX$C_BOUNDED	A Boolean variable. If bit <0> is true (equals *1*), you are stating your intention to submit for analysis all data records that will be compressed; doing so often enables DCX to compute a better compression algorithm. If bit <0> is false (equals *0*) or if the DCX$C_BOUNDED item code is not specified, DCX computes a compression algorithm without regard for whether all records to be compressed will also be submitted for analysis.
DCX$C_EST_BYTES	A longword value containing your estimate of the total number of data bytes that will be submitted for compression. This estimate is useful in those cases where fewer than the total number of bytes are presented for analysis. If you do not specify the DCX$C_EST_BYTES item code, DCX submits for compression the same number of bytes that was presented for analysis. Note that you may specify DCX$C_EST_RECORDS or DCX$C_EST_BYTES, or both.
DCX$C_EST_RECORDS	A longword value containing your estimate of the total number of data records that will be submitted for compression. This estimate is useful in those cases where fewer than the total number of records are presented for analysis. If you do not specify the DCX$C_EST_RECORDS item code, DCX submits for compression the same number of bytes that was presented for analysis.

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

DESCRIPTION

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

CONDITION VALUES RETURNED

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

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

DCX$COMPRESS_DATA Compress a Data Record

The DCX$COMPRESS_DATA routine compresses a data record. You call this routine for each data record to be compressed.

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

in_rec

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

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

out_rec

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

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

out_length

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

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

DESCRIPTION

The DCX$COMPRESS_DATA routine compresses a data record. You call this routine for each data record to be compressed. As you compress each record, write the compressed record to the file containing the compression/expansion map. For each record, write the length of the record and substring string containing the record to the same file. See the COMPRESS DATA section in Example 5–1.

CONDITION VALUES RETURNED

DCX$_INVCTX	Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.
DCX$_INVDATA	Error. You specified the item value DCX$C_BOUNDED in the DCX$ANALYZE_INIT routine and attempted to compress a data record (using DCX$COMPRESS_DATA) that was not presented for analysis (using DCX$ANALYZE_DATA). Specifying the DCX$C_BOUNDED item value means that you must analyze all data records that are to be compressed.
DCX$_INVMAP	Error; invalid map. The **map** argument was not specified correctly or the context area is invalid.
DCX$_NORMAL	Successful completion.
DCX$_TRUNC	Error. The compressed data record has been truncated because the **out_rec** descriptor did not specify enough memory to accommodate the record.

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

DCX$COMPRESS_DONE Specify Compression Complete

The DCX$COMPRESS_DONE routine deletes the context area and sets the context variable to zero.

FORMAT

DCX$COMPRESS_DONE *context*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

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

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

DESCRIPTION

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

CONDITION VALUES RETURNED

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

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

DCX$COMPRESS_INIT Initialize Compression Context

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

FORMAT

DCX$COMPRESS_INIT *context ,map*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

map

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

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

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

DESCRIPTION

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

You call the DCX$COMPRESS_INIT routine after the call to DCX$ANALYZE_DONE.

CONDITION VALUES RETURNED

DCX$_INVMAP	Error; invalid map. The map argument was not specified correctly, or the context area is invalid.
DCX$_NORMAL	Successful completion.

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

DCX$EXPAND_DATA Expand a Compressed Data Record

The DCX$EXPAND_DATA routine expands (or restores) a compressed data record to its original state.

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

in_rec

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

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

out_rec

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

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

out_length

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

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

DESCRIPTION

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

CONDITION VALUES RETURNED

DCX$_INVCTX	Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.
DCX$_INVDATA	Error. A compressed data record is invalid (probably truncated) and therefore cannot be expanded.
DCX$_INVMAP	Error; invalid map. The map argument was not specified correctly, or the context area is invalid.
DCX$_NORMAL	Successful completion.
DCX$_TRUNC	Warning. The expanded data record has been truncated because the **out_rec** descriptor did not specify enough memory to accommodate the record.

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

DCX$EXPAND_DONE Specify Expansion Complete

The DCX$EXPAND_DONE routine deletes the context area and sets the context variable to zero.

FORMAT

DCX$EXPAND_DONE *context*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

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

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

DESCRIPTION

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

CONDITION VALUES RETURNED

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

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

DCX$EXPAND_INIT Initialize Expansion Context

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

FORMAT

DCX$EXPAND_INIT *context ,map*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

map

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

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

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

DESCRIPTION

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

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

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

CONDITION VALUES RETURNED

DCX$_INVMAP	Error; invalid map. The map argument was not specified correctly, or the context area is invalid.
DCX$_NORMAL	Successful completion.

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

DCX$MAKE_MAP Compute the Compression/Expansion Function

The DCX$MAKE_MAP routine uses the statistical information gathered by DCX$ANALYZE_DATA to compute the compression/expansion function.

FORMAT

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

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

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

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

map_addr

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

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

map_size

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

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

DESCRIPTION

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

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

The map is stored with the compressed data records, because the compressed data records are indecipherable without the map. When compressed data records have been expanded to their original state and no further compression is desired, you should delete the map using the LIB$FREE_VM routine.

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

CONDITION VALUES RETURNED

DCX$_AGAIN	Informational. The map has not been created and the **map_addr** and **map_size** arguments have not been written because further analysis is required. The data records must be analyzed (using DCX$ANALYZE_DATA) again, and DCX$MAKE_MAP must be called again before DCX$MAKE_MAP will create the map and return the DCX$_NORMAL status code.
DCX$_INVCTX	Error. The context variable is invalid, or the context area is invalid or corrupted. This may be caused by a failure to call the appropriate routine to initialize the context variable or by an application program error.
DCX$_NORMAL	Successful completion.

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

6 EDT Routines

6.1 Introduction to EDT Routines

On VMS operating systems, the EDT editor can be called from a program. Calling programs can be written in any VAX language that generates calls using the VAX Procedure Calling and Condition Handling Standard.

You can set up your call to EDT so that the program handles all the editing work, or you can make EDT run interactively so that you can edit a file while the program is running.

This chapter on callable EDT assumes that you know how to call an external facility from the language you are using. Callable EDT is a shareable image, which means that you save physical memory and disk space by having all processes access a single copy.

You must include a statement in your program accessing the EDT entry point. This reference statement is similar to a library procedure reference statement. The EDT entry point is referenced as EDT$EDIT. You can pass arguments to EDT$EDIT; for example, you can pass EDT$FILEIO or your own routine. When you refer to the routines you pass, call them FILEIO, WORKIO, and XLATE. Therefore, FILEIO can be either a routine provided by EDT (named EDT$FILEIO) or a routine that you write.

6.2 Example of Using EDT Routines

Example 6–1 shows a VAX BASIC program that calls EDT. All three routines (FILEIO, WORKIO, and XLATE) are called. Note the reference to the entry point EDT$EDIT in line number 500.

Example 6–1 Using the EDT Routines in a VAX BASIC Program

```
100  EXTERNAL INTEGER EDT$FILEIO  ➊
200  EXTERNAL INTEGER EDT$WORKIO
250  EXTERNAL INTEGER AXLATE
300  EXTERNAL INTEGER FUNCTION EDT$EDIT
400  DECLARE INTEGER RESULT

450  DIM INTEGER PASSFILE(1%)  ➋
460  DIM INTEGER PASSWORK(1%)
465  DIM INTEGER PASSXLATE(1%)
470  PASSFILE(0%) = LOC(EDT$FILEIO)
480  PASSWORK(0%) = LOC(EDT$WORKIO)
485  PASSXLATE(0%) = LOC(AXLATE)
```

Example 6–1 Cont'd. on next page

Example 6–1 (Cont.) Using the EDT Routines in a VAX BASIC Program

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

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

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

❸ Here is the call to EDT. The input file is FILE.BAS, the output and journal files are defaulted, and the command file is EDTINI. A *0* is passed for the options word to get the default EDT options.

❹ The array PASSFILE points to the entry point for all file I/O, which is set up in this example to be the EDT-supplied routine with the entry point EDT$FILEIO. Similarly, the array PASSWORK points to the entry point for all work I/O, which is the EDT-supplied routine with the entry point EDT$WORKIO.

❺ PASSXLATE points to the entry point that EDT will use for all XLATE processing. PASSXLATE points to a user-supplied routine with the entry point AXLATE.

6.3 EDT Routines

The following pages describe the individual EDT routines.

EDT$EDIT Edit a File

The EDT$EDIT routine invokes the EDT editor.

FORMAT

EDT$EDIT *in_file [,out_file] [,com_file] [,jou_file] [,options] [,fileio] [,workio] [,xlate]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

in_file

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

File specification of the input file that EDT$EDIT is to edit. The **in_file** argument is the address of a descriptor pointing to this file specification. The string that you enter in this calling sequence is passed to the FILEIO routine to open the primary input file. This is the only required argument.

out_file

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

File specification of the output file that EDT$EDIT creates. The **out_file** argument is the address of a descriptor pointing to this file specification. The default is that the input file specification is passed to the FILEIO routine to open the output file for the EXIT command.

com_file

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

File specification of the startup command file to be executed when EDT is invoked. The **com_file** argument is the address of a descriptor pointing to this file specification. The **com_file** string is passed to the FILEIO routine to open the command file. The default is the same as that for EDT command file defaults.

jou_file

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

File specification of the journal file to be opened when EDT is invoked. The **jou_file** argument is the address of a descriptor pointing to this file specification. The **jou_file** string is passed to the FILEIO routine to open the journal file. The default is to use the same file name as **in_file**.

options

VMS usage: **mask_longword**
type: **aligned bit string**
access: **read only**
mechanism: **by reference**

Bit vector specifying options for the edit operation. The **options** argument is the address of an aligned bit string containing this bit vector. Only bits <5:0> are currently defined; all others must be *0*. The default options have all bits set to *0*. This is the same as the default setting when you invoke EDT to edit a file from DCL.

Symbols and their descriptions follow:

EDT$M_RECOVER	If set, bit <0> causes EDT to read the journal file and execute the commands in it, except for the EXIT or QUIT commands, which are ignored. After the journal file commands are processed, editing continues normally. If bit <0> is set, the FILEIO routine is asked to open the journal file for both input and output; otherwise FILEIO is asked only to open the journal file for output. Bit <0> corresponds to the /RECOVER qualifier on the EDT command line.
EDT$M_COMMAND	If set, bit <1> causes EDT to signal if the startup command file cannot be opened. When bit <1> is *0*, EDT intercepts the signal from the FILEIO routine indicating that the startup command file could not be opened. Then, EDT proceeds with the editing session without reading any startup command file. If no command file name is supplied with the call to the EDT$EDIT routine, EDT tries to open SYS$LIBRARY:EDTSYS.EDT or, if that fails, EDTINI.EDT. Bit <1> corresponds to the /COMMAND qualifier on the EDT command line. If EDT$M_NOCOMMAND (bit <4>) is set, bit <1> is overridden because bit <4> prevents EDT from trying to open a command file.
EDT$M_NOJOURNAL	If set, bit <2> prevents EDT from opening the journal file. Bit <2> corresponds to the /NOJOURNAL or /READ_ONLY qualifier on the EDT command line.
EDT$M_NOOUTPUT	If set, bit <3> prevents EDT from using the input file name as the default output file name. Bit <3> corresponds to the /NOOUTPUT or /READ_ONLY qualifier on the EDT command line.

EDT$M_NOCOMMAND	If set, bit <4> prevents EDT from opening a startup command file. Bit <4> corresponds to the /NOCOMMAND qualifier on the EDT command line.
EDT$M_NOCREATE	If set, bit <5> causes EDT to return to the caller if the input file is not found. The status returned is the error code EDT$_INPFILNEX.

fileio

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **function call**
mechanism: **by reference**

User-supplied routine called by EDT to perform file I/O functions. The **fileio** argument is the address of a bound procedure value containing the user-supplied routine. When you do not need to intercept any file I/O, either use the entry point EDT$FILEIO for this argument or omit it. When you only need to intercept some amount of file I/O, call the EDT$FILEIO routine for the other cases.

To avoid confusion, note that EDT$FILEIO is a routine provided by EDT whereas FILEIO is a routine that you provide.

In order to accommodate routines written in high-level languages that do up-level addressing, this argument must have a data type of BPV (bound procedure value). BPV is a two-longword entity in which the first longword contains the address of a procedure entry mask and the second longword is the environment value. When the bound procedure is called, EDT loads the second longword into R1. If you use EDT$FILEIO for this argument, set the second longword to <0> . You can pass a <0> for the argument, and EDT will set up EDT$FILEIO as the default and set the environment word to *0*.

workio

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **function call**
mechanism: **by reference**

User-supplied routine called by EDT to perform I/O between the work file and EDT. The **workio** argument is the address of a bound procedure value containing the user-supplied routine. Work file records are addressed only by number and are always 512 bytes long. If you do not need to intercept work file I/O, you can either use the entry point EDT$WORKIO for this argument or omit it.

In order to accommodate routines written in high-level languages that do up-level addressing, this argument must have a data type of BPV (bound procedure value). This means that EDT loads R1 with the second longword addressed before calling it. If EDT$WORKIO is used for this argument, set the second longword to *0*. You can pass a *0* for this argument, and EDT will set up EDT$WORKIO as the default and set the environment word to *0*.

xlate

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **function call**
mechanism: **by reference**

User-supplied routine that EDT calls when it encounters the nokeypad command XLATE. The **xlate** argument is the address of a bound procedure value containing the user-supplied routine. The XLATE routine allows you to gain control of your EDT session. If you do not need control of EDT during the editing session, you can either use the entry point EDT$XLATE for this argument or omit it.

In order to accommodate routines written in high-level languages that do up-level addressing, this argument must have a data type of BPV (bound procedure value). This means that EDT loads R1 with the second longword addressed before calling it. If EDT$XLATE is used for this argument, set the second longword to *0*. You can pass a *0* for this argument, and EDT will set up EDT$XLATE as the default and set the environment word to *0*.

DESCRIPTION

If the EDT session is terminated by EXIT or QUIT, the status will be a successful value (bit <0> = 1). If the session is terminated because the file was not found and if the /NOCREATE qualifier was in effect, the failure code EDT$_INPFILNEX is returned. In an unsuccessful termination caused by an EDT error, a failure code corresponding to that error is returned. Each error status from the FILEIO and WORKIO routines is explained separately.

Three of the arguments to the EDT$EDIT routine, **fileio**, **workio**, and **xlate** are the entry point names of user-supplied routines.

CONDITION VALUES RETURNED

SS$_NORMAL	Successful completion.
EDT$_INPFILNEX	/NOCREATE specified and input file does not exist.

This routine also returns any condition values returned by user-supplied routines.

FILEIO

The user-supplied FILEIO routine performs file I/O functions. You call it by specifying it as an argument in the EDT$EDIT routine. It cannot be called independently.

FORMAT

FILEIO *code ,stream ,record ,rhb*

RETURNS

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

A VMS status code that your FILEIO routine returns to EDT$EDIT. The **fileio** argument is a longword containing the status code. The only failure code that is normally returned is RMS$_EOF from a GET call. All other VMS RMS errors are signaled, not returned. The VMS RMS signal should include the file name and both longwords of the RMS status. Any errors detected with the FILEIO routine can be indicated by setting status to an error code. That special error code will be returned to the program by the EDT$EDIT routine. There is a special status value EDT$_NONSTDFIL for nonstandard file opening.

Condition values are returned in R0.

ARGUMENTS

code

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

A code from EDT that specifies what function the FILEIO routine is to perform. The **code** argument is the address of a longword integer containing this code. Following are the valid function codes:

Function Code	Description
EDT$K_OPEN_INPUT	The **record** argument names a file to be opened for input. The **rhb** argument is the default file name.
EDT$K_OPEN_OUTPUT_SEQ	The **record** argument names a file to be opened for output as a sequenced file. The **rhb** argument is the default file name.
EDT$K_OPEN_OUTPUT_NOSEQ	The **record** argument names a file to be opened for output. The **rhb** argument is the default file name.

Function Code	Description
EDT$K_OPEN_IN_OUT	The **record** argument names a file to be opened for both input and output. The **rhb** argument is the default file name.
EDT$K_GET	The **record** argument is to be filled with data from the next record of the file. If the file has record prefixes, **rhb** is filled with the record prefix. If the file has no record prefixes, **rhb** is not written. When you attempt to read past the end of file, **status** is set to RMS$_EOF.
EDT$K_PUT	The data in the **record** argument is to be written to the file as its next record. If the file has record prefixes, the record prefix is taken from the **rhb** argument. For a file opened for both input and output, EDT$K_PUT is valid only at the end of the file, indicating that the **record** is to be appended to the file.
EDT$K_CLOSE_DEL	The file is to be closed and then deleted. The **record** and **rhb** arguments are not used in the call.
EDT$K_CLOSE	The file is to be closed. The **record** and **rhb** arguments are not used in the call.

stream

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

A code from EDT that indicates which file is being used. The **stream** argument is the address of a longword integer containing the code. Following are the valid codes.

Function Code	Description
EDT$K_COMMAND_FILE	The command file.
EDT$K_INPUT_FILE	The primary input file.
EDT$K_INCLUDE_FILE	The secondary input file. Such a file is opened in response to an INCLUDE command. It is closed when the INCLUDE command is complete and will be reused for subsequent INCLUDE commands.
EDT$K_JOURNAL_FILE	The journal file. If bit 0 of the options is set, it is opened for both input and output and is read completely. Otherwise, it is opened for output only. After it is read or opened for output only, it is used for writing. On a successful termination of the editing session, the journal file is closed and deleted. EXIT/SAVE and QUIT/SAVE close the journal file without deleting it.

Function Code	Description
EDT$K_OUTPUT_FILE	The primary output file. It is not opened until you enter the EXIT command.
EDT$K_WRITE_FILE	The secondary output file. Such a file is opened in response to a WRITE or PRINT command. It is closed when the command is complete and will be reused for subsequent WRITE or PRINT commands.

record

VMS usage: **char_string**
type: **character-coded text string**
access: **modify**
mechanism: **by descriptor**

Text record passed by descriptor from EDT to the user-supplied FILEIO routine; the **code** argument determines how the **record** argument is used. The **record** argument is the address of a descriptor pointing to this argument. When the **code** argument starts with EDT$K_OPEN, the **record** is a file name. When the **code** argument is EDT$K_GET, the **record** is a place to store the record that was read from the file. For **code** argument EDT$K_PUT, the **record** is a place to find the record to be written to the file. This argument is not used if the **code** argument starts with EDT$K_CLOSE.

Note that for EDT$K_GET, EDT uses a dynamic or varying string descriptor; otherwise, EDT has no way of knowing the length of the record being read. EDT uses only string descriptors that can be handled by the Run-Time Library (RTL) routine STR$COPY_DX.

rhb

VMS usage: **char_string**
type: **character-coded text string**
access: **modify**
mechanism: **by descriptor**

Text record passed by descriptor from EDT to the user-supplied FILEIO routine; the **code** argument determines how the **rhb** argument is used. When the **code** argument starts with EDT$K_OPEN, the **rhb** argument is the default file name. When the **code** is EDT$K_GET and the file has record prefixes, the prefixes are put in this argument. When the **code** is EDT$K_PUT and the file has record prefixes, the prefixes are taken from this argument. Like the **record** argument, EDT uses a dynamic or varying string descriptor for EDT$K_GET and uses only string descriptors that can be handled by the RTL routine STR$COPY_DX.

DESCRIPTION

If you do not need to intercept any file I/O, you can use the entry point EDT$FILEIO for this argument or you can omit it. If you need to intercept only some file I/O, call the EDT$FILEIO routine for the other cases.

When you use EDT$FILEIO as a value for the **fileio** argument, files are opened as follows:

- The **record** argument is always the RMS file name.
- The **rhb** argument is always the RMS default file name.
- There is no related name for the input file.
- The related name for the output file is the input file with OFP (output file parse). EDT passes the input file name, the output file name, or the name from the EXIT command in the **record** argument.
- The related name for the journal file is the input file name with the output file parse (OFP) RMS bit set.
- The related name for the INCLUDE file is the input file name with the OFP set. This is unusual because the file is being opened for input.

CONDITION VALUES RETURNED

SS$_NORMAL	Successful completion.
EDT$_NONSTDFIL	File is not in standard text format.
RMS$_EOF	End of file on a GET.

WORKIO

The user-supplied WORKIO routine is called by EDT when it needs temporary storage for the file being edited. You call it by specifying it as an argument in the EDT$EDIT routine. It cannot be called independently.

FORMAT

WORKIO *code ,recordno ,record*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by immediate value**

Longword value returned as a VMS status code. It is generally a success code, because all VMS RMS errors should be signaled. The signal should include the file name and both longwords of the VMS RMS status. Any errors detected within work I/O can be indicated by setting status to an error code, which will be returned by the EDT$EDIT routine.

The condition value is returned in R0.

ARGUMENTS

code

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

A code from EDT that specifies the operation to be performed. The **code** argument is the address of a longword integer containing this argument. The valid function codes are as follows:

Function Code	Description
EDT$K_OPEN_IN_OUT	Open the work file for both input and output. Neither the **record** nor **recordno** argument is used.
EDT$K_GET	Read a record. The **recordno** argument is the number of the record to be read. The **record** argument gives the location where the record is to be stored.
EDT$K_PUT	Write a record. The **recordno** argument is the number of the record to be written. The **record** argument tells the location of the record to be written.
EDT$K_CLOSE_DEL	Close the work file. After a successful close, the file is deleted. Neither the **record** nor **recordno** argument is used.

recordno

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

Number of the record to be read or written. The **recordno** argument is the address of a longword integer containing this argument. EDT always writes a record before reading that record. This argument is not used for open or close calls.

record

VMS usage: **char_string**
type: **character string**
access: **modify**
mechanism: **by descriptor**

Location of the record to be read or written. This argument always refers to a 512-byte string during GET and PUT calls. This argument is not used for open or close calls.

DESCRIPTION

Work file records are addressed only by number and are always 512 bytes long. If you do not need to intercept work file I/O, you can use the entry point EDT$WORKIO for this argument or you can omit it.

CONDITION VALUE RETURNED

SS$_NORMAL	Successful completion.

XLATE

The user-supplied XLATE routine is called by EDT when it encounters the nokeypad command XLATE. You cause it to be called by specifying it as an argument in the EDT$EDIT routine. It cannot be called independently.

FORMAT

XLATE *string*

RETURNS

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

Longword value returned as a VMS status code. It is generally a success code. If the XLATE routine cannot process the passed string for some reason, it sets status to an error code. Returning an error code from the XLATE routine aborts the current key execution and displays the appropriate error message.

The condition value is returned in R0.

ARGUMENT

string
VMS usage: **char_string**
type: **character-coded text string**
access: **modify**
mechanism: **by descriptor**

Text string passed to the nokeypad command XLATE. You can use the nokeypad command XLATE by defining a key to include the following command in its definition:

```
XLATEtext^Z
```

The text is passed by the **string** argument. The **string** argument is one that can be handled by the Run-Time Library (RTL) routine STR$COPY_DX.

This argument is also a text string returned to EDT. The string is made up of nokeypad commands that EDT is to execute.

DESCRIPTION

The nokeypad command XLATE allows you to gain control of the EDT session. (See the *VAX EDT Reference Manual* for more information about the XLATE command.) If you do not need to gain control of EDT during the editing session, you can use the entry point EDT$XLATE for this argument or you can omit it.

CONDITION VALUE RETURNED

SS$_NORMAL	Successful completion.

7 File Definition Language (FDL) Routines

7.1 Introduction to FDL Routines

This chapter describes the File Definition Language (FDL) routines. These routines perform many of the functions of the RMS File Definition Language.

The FDL$CREATE routine is the one most likely to be called from a high-level language. It creates a file from an FDL specification and then closes the file.

The following three FDL routines provide a way to specify all the options RMS allows when it executes create, open, or connect operations. They also allow you to specify special processing options required for your applications.

The FDL$GENERATE routine produces an FDL specification by interpreting a set of RMS control blocks. It then writes the FDL specification either to an FDL file or to a character string.

The FDL$PARSE routine parses an FDL specification, allocates RMS control blocks, and fills in the relevant fields.

The FDL$RELEASE routine deallocates the virtual memory used by the RMS control blocks created by FDL$PARSE.

These routines cannot be called from AST level.

An FDL specification can be either in a file or in a character string. When specifying an FDL specification in a character string, delimit the statements of the FDL specification with semicolons.

7.2 Examples of Using the FDL Routines

Example 7–1 shows how to use the FDL$CREATE routine in a FORTRAN program.

Example 7–1 Using FDL$CREATE in a FORTRAN Program

```
*       This program calls the FDL$CREATE routine.  It
*       creates an indexed output file named NEW_MASTER.DAT
*       from the specifications in the FDl file named
*       INDEXED.FDL.  You can also supply a default filename
*       and a result name (that receives the name of the
*       created file).  The program also returns all the
*       statistics.
*
        IMPLICIT        INTEGER*4       (A - Z)
        EXTERNAL        LIB$GET_LUN,    FDL$CREATE
        CHARACTER       IN_FILE*11      /'INDEXED.FDL'/,
        1               OUT_FILE*14     /'NEW_MASTER.DAT'/,
        1               DEF_FILE*11     /'DEFAULT.FDL'/,
        1               RES_FILE*50
        INTEGER*4       FIDBLK(3)       /0,0,0/
        I = 1
        STATUS = FDL$CREATE (IN_FILE,OUT_FILE,
                 DEF_FILE,RES_FILE,FIDBLK,,)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        STATUS=LIB$GET_LUN(LOG_UNIT)
        OPEN (UNIT=LOG_UNIT,FILE=RES_FILE,STATUS='OLD')
        CLOSE (UNIT=LOG_UNIT, STATUS='KEEP')

        WRITE (6,1000) (RES_FILE)
        WRITE (6,2000) (FIDBLK (I), I=1,3)

1000    FORMAT  (1X,'The result filename is: ',A50)

2000    FORMAT  (/1X,'FID-NUM: ',I5/,
        1         1X,'FID-SEQ: ',I5/,
        1         1X,'FID-RVN: ',I5)

        END
```

Example 7–2 shows how to use the FDL$PARSE and FDL$RELEASE routines in a MACRO program.

Example 7–2 Using FDL$PARSE and FDL$RELEASE in a MACRO Program

```
;
;  This program calls the FDL utility routines FDL$PARSE and
;  FDL$RELEASE.  First, FDL$PARSE parses the FDL specification
;  PART.FDL.  Then the data file named in PART.FDL is accessed
;  using the primary key.  Last, the control blocks allocated
;  by FDL$PARSE are released by FDL$RELEASE.
;
              .TITLE  FDLEXAM
;
              .PSECT  DATA,WRT,NOEXE
;
MY_FAB:       .LONG   0
MY_RAB:       .LONG   0
FDL_FILE:     .ASCID  /PART.FDL/          ; Declare FDL file
REC_SIZE=80
LF=10
REC_RESULT:   .LONG   REC_SIZE
              .ADDRESS REC_BUFFER
REC_BUFFER:   .BLKB   REC_SIZE
HEADING:      .ASCID /ID   PART   SUPPLIER        COLOR /<LF>
;
              .PSECT  CODE
;
;  Declare the external routines
;
.EXTRN        FDL$PARSE, -
              FDL$RELEASE
;
.ENTRY        FDLEXAM,^M<>                ; Set up entry mask
              PUSHAL  MY_RAB              ; Get set up for call with
              PUSHAL  MY_FAB              ; addresses to receive the
              PUSHAL  FDL_FILE            ; FAB and RAB allocated by
              CALLS   #3,G^FDL$PARSE      ; FDL$PARSE
              BLBS    R0,KEY0
              BRW     ERROR
;
KEY0:         MOVL    MY_FAB,R10          ; Move address of FAB to R10
              MOVL    MY_RAB,R9           ; Move address of RAB to R9
              MOVL    #REC_SIZE,RAB$W_USZ(R9)
              MOVAB   REC_BUFFER,RAB$L_UBF(R9)
              $OPEN   FAB=(R10)           ; Open the file
              BLBC    R0,ERROR
              $CONNECT RAB=(R9)           ; Connect to the RAB
              BLBC    R0,ERROR
              PUSHAQ  HEADING             ; Display the heading
              CALLS   #1,G^LIB$PUT_OUTPUT
              BLBC    R0,ERROR
```

Example 7–2 Cont'd. on next page

Example 7–2 (Cont.) Using FDL$PARSE and FDL$RELEASE in a MACRO Program

```
;
GET_REC:        $GET    RAB=(R9)                ; Get a record
                CMPL    #RMS$_EOF,RO            ; If not end of file,
                BEQLU   CLEAN                   ; continue
                BLBC    RO,ERROR
                MOVZWL  RAB$W_RSZ(R9),REC_RESULT  ; Move a record into
                PUSHAL  REC_RESULT                ; the buffer
                CALLS   #1,G^LIB$PUT_OUTPUT       ; Display the record
                BLBC    RO,ERROR
                BRB     GET_REC                 ; Get another record
CLEAN:          $CLOSE  FAB=(R10)               ; Close the FAB
                BLBC    RO,ERROR
                PUSHAL  MY_RAB                  ; Push RAB addr on stack
                PUSHAL  MY_FAB                  ; Push FAB addr on stack
                CALLS   #2,G^FDL$RELEASE        ; Release control blocks
                BLBC    RO,ERROR
                BRB     FINI
;
ERROR:          PUSHL   RO
                CALLS   #1,G^LIB$SIGNAL
                $CLOSE  FAB=(R10)
;
RAB_ERROR:      PUSHL   RAB$L_STV(R9)
                PUSHL   RAB$L_STS(R9)
                BRB     RMS_ERR
;
FAB_ERROR:      PUSHL   FAB$L_STV(R10)
                PUSHL   FAB$L_STS(R10)
;
RMS_ERR:        CALLS   #2,G^LIB$SIGNAL
                BRB     FINI
;
FINI:           RET
                .END FDLEXAM
```

Example 7–3 shows how to use the FDL$GENERATE routine in a VAX Pascal program.

Example 7–3 Using FDL$PARSE and FDL$GENERATE in a VAX Pascal Program

```
[INHERIT ('SYS$LIBRARY:STARLET')]
PROGRAM FDLexample (input,output,order_master);

(* This program fills in its own FAB, RAB, and       *)
(* XABs by calling FDL$PARSE and then generates      *)
(* an FDL specification describing them.             *)
(* It requires an existing input FDL file            *)
(* (TESTING.FDL) for FDL$PARSE to parse.             *)
TYPE
(*+                                                  *)
(* FDL CALL INTERFACE CONTROL FLAGS                  *)
(*-                                                  *)
        $BIT1 = [BIT(1),UNSAFE] BOOLEAN;

        FDL2$TYPE = RECORD CASE INTEGER OF
        1: (FDL$_FDLDEF_BITS : [BYTE(1)] RECORD END;
            );
        2: (FDL$V_SIGNAL : [POS(0)] $BIT1;
              (* Signal errors; don't return         *)
            FDL$V_FDL_STRING : [POS(1)] $BIT1;
              (* Main FDL spec is a char string      *)
            FDL$V_DEFAULT_STRING : [POS(2)] $BIT1;
              (* Default FDL spec is a char string   *)
            FDL$V_FULL_OUTPUT : [POS(3)] $BIT1;
              (* Produce a complete FDL spec         *)
            FDL$V_$CALLBACK : [POS(4)] $BIT1;
              (* Used by EDIT/FDL on input (DEC only) *)
            )
        END;

    mail_order =  RECORD
                  order_num : [KEY(0)] INTEGER;
                  name : PACKED ARRAY[1..20] OF CHAR;
                  address : PACKED ARRAY[1..20] OF CHAR;
                  city : PACKED ARRAY[1..19] OF CHAR;
                  state : PACKED ARRAY[1..2] OF CHAR;
                  zip_code : [KEY(1)] PACKED ARRAY[1..5]
                       OF CHAR;
                  item_num : [KEY(2)] INTEGER;
                  shipping : REAL;
                  END;

    order_file  = [UNSAFE] FILE OF mail_order;
    ptr_to_FAB  = ^FAB$TYPE;
    ptr_to_RAB  = ^RAB$TYPE;
    byte = 0..255;

VAR
    order_master : order_file;
    flags        : FDL2$TYPE;
    order_rec    : mail_order;
    temp_FAB     : ptr_to_FAB;
    temp_RAB     : ptr_to_RAB;
    status       : integer;
```

Example 7–3 Cont'd. on next page

Example 7–3 (Cont.) Using FDL$PARSE and FDL$GENERATE in a VAX Pascal Program

```
FUNCTION FDL$PARSE
    (%STDESCR FDL_FILE : PACKED ARRAY [L..U:INTEGER]
         OF CHAR;
    VAR FAB_PTR : PTR_TO_FAB;
    VAR RAB_PTR : PTR_TO_RAB) : INTEGER; EXTERN;

FUNCTION FDL$GENERATE
    (%REF FLAGS : FDL2$TYPE;
    FAB_PTR : PTR_TO_FAB;
    RAB_PTR : PTR_TO_RAB;
    %STDESCR FDL_FILE_DST : PACKED ARRAY [L..U:INTEGER]
         OF CHAR) : INTEGER;
    EXTERN;

BEGIN

    status := FDL$PARSE ('TESTING',TEMP_FAB,TEMP_RAB);
    flags::byte := 0;
    status := FDL$GENERATE (flags,
                            temp_FAB,
                            temp_RAB,
                            'SYS$OUTPUT:');

END.
```

7.3 FDL Routines

The following pages describe the individual FDL routines.

FDL$CREATE Create a File from an FDL Specification and Close the File

The FDL$CREATE routine creates a file from an FDL specification and then closes the file.

FORMAT

FDL$CREATE *fdl_desc [,filename] [,default_name] [,result_name] [,fid_block] [,flags] [,stmnt_num] [,retlen] [,sts] [,stv]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

fdl_desc

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

Name of a file that contains the FDL specification or the actual FDL specification to be parsed. The **fdl_desc** argument is the address of a character string descriptor pointing to this information.

If the FDL$V_FDL_STRING flag is set in the mask argument, FDL$CREATE interprets this argument as an FDL specification in string form. Otherwise, FDL$CREATE interprets this argument as a file name.

filename

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

Name of the VMS RMS file to be created using the FDL specification. The **filename** argument is the address of a character string descriptor pointing to the VMS RMS file name. This name overrides the *default_name* parameter given in the FDL specification.

This argument is optional.

default_name

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

Default name of the file to be created using the FDL specification. The **default_name** argument is the address of a character string descriptor pointing to the default file name. This name overrides any name given in the FDL specification.

This argument is optional.

result_name

VMS usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed-length string descriptor**

Resultant name of the file created by FDL$CREATE. The **result_name** argument is the address of a character string descriptor that receives the resultant file name.

This argument is optional.

fid_block

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

File identification of the VMS RMS file created by FDL$CREATE. The **fid_block** argument is the address of an array of longwords that receives the VMS RMS file identification information. The first longword contains the FID_NUM; the second contains the FID_SEQ; and the third contains the FID_RVN. They have the following definitions:

FID_NUM	The location of the file on the disk. Its value can range from *1* up to the number of files the disk can hold.
FID_SEQ	The file sequence number, which is the number of times the file number has been used.
FID_RVN	The relative volume number, which is the volume number of the volume on which the file is stored. If the file is not stored on a volume set, the relative volume number is *0*.

This argument is optional.

flags

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

Flags (or masks) that control how the **fdl_desc** argument is interpreted and how errors are signaled. The **flags** argument is the address of a longword containing the control flags (or a mask). If you omit this argument or specify it as zero, no flags are set. The flags and their meanings are as follows:

Flag	Description
FDL$V_FDL_STRING	Interprets the **fdl_desc** argument as an FDL specification in string form. By default, the **fdl_desc** argument is interpreted as the file name of an FDL file.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

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

stmnt_num

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

FDL statement number. The **stmnt_num** argument is the address of a longword that receives the FDL statement number. If the routine completes successfully, the **stmnt_num** argument is the number of statements in the FDL specification. If the routine does not complete successfully, the **stmnt_num** argument receives the number of the statement that caused the error. In general, however, line numbers and statement numbers are not the same. Null statements (blank lines) are not counted. Also, an FDL specification in string form has no "lines."

This argument is optional.

retlen

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

Number of characters returned in the **result_name** argument. The **retlen** argument is the address of a longword that receives this number.

This argument is optional.

sts

VMS usage: **longword_unsigned**
type: **longword_unsigned**
access: **write only**
mechanism: **by reference**

VMS RMS status value FAB$L_STS. The **sts** argument is the address of a longword that receives the VMS RMS status value FAB$L_STS from SYS$CREATE.

stv

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

VMS RMS status value FAB$L_STV. The **stv** argument is the address of a longword that receives the VMS RMS status value FAB$L_STV from SYS$CREATE.

DESCRIPTION

FDL$CREATE calls the FDL$PARSE routine to parse the FDL specification. The FDL specification can be either in a file or a character string. FDL$CREATE opens (creates) the specified VMS RMS file, and then closes it without putting any data in it.

FDL$CREATE does not create the output file if an error status is either returned or signaled.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
FDL$_ABKW	Ambiguous keyword in statement *n*.
FDL$_ABPRIKW	Ambiguous primary keyword in statement *n*.
FDL$_BADLOGIC	Internal logic error detected.
FDL$_CLOSEIN	Error closing file specification as input.
FDL$_CLOSEOUT	Error closing file specification as output.
FDL$_CREATE	Error creating file specification.
FDL$_CREATED	File specification created.
FDL$_CREATED_STM	File specification created in stream format.
FDL$_FDLERROR	Error parsing FDL file.
FDL$_ILL_ARG	Wrong number of arguments.
FDL$_INSVIREM	Insufficient virtual memory.
FDL$_INVBLK	Invalid VMS RMS control block at virtual address *n*.
FDL$_MULPRI	Multiple primary definition in statement *n*.
FDL$_OPENFDL	Error opening file specification.
FDL$_OPENIN	Error opening file specification as input.
FDL$_OPENOUT	Error opening file specification as output.
FDL$_OUTORDER	Key or area primary defined out of order in statement *n*.
FDL$_READERR	Error reading file specification.
FDL$_RFLOC	Unable to locate related file.
FDL$_SYNTAX	Syntax error in statement *n*.
FDL$_UNPRIKW	Unrecognized primary keyword in statement *n*.
FDL$_UNQUAKW	Unrecognized qualifier keyword in statement *n*.
FDL$_UNSECKW	Unrecognized secondary keyword in statement *n*.
FDL$_VALERR	Specified value is out of legal range.
FDL$_VALPRI	Value required on primary in statement *n*.
FDL$_WARNING	Parsed with warnings.
FDL$_WRITEERR	Error writing file specification.
RMS$_ACT	File activity precludes operation.
RMS$_CRE	ACP file create failed.
RMS$_CREATED	File was created, not opened.

RMS$_DNF	Directory not found.
RMS$_DNR	Device not ready or not mounted.
RMS$_EXP	File expiration date not yet reached.
RMS$_FEX	File already exists, not superseded.
RMS$_FLK	File currently locked by another user.
RMS$_PRV	Insufficient privilege or file protection violation.
RMS$_SUPERSEDE	Created file superseded existing version.
RMS$_WLK	Device currently write locked.

FDL$GENERATE Generate an FDL Specification

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

FORMAT

FDL$GENERATE *flags ,fab_pointer ,rab_pointer [,fdl_file_dst] [,fdl_file_resnam] [,fdl_str_dst] [,bad_blk_addr] [,retlen]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

flags

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

Flags (or masks) that control how the **fdl_str_dst** argument is interpreted and how errors are signalled. The **flags** argument is the address of a longword containing the control flags (or a mask). If you omit this argument or specify it as zero, no flags are set. The flags and their meanings are as follows:

Flag	Description
FDL$V_FDL_STRING	Interprets the **fdl_str_dst** argument as an FDL specification in string form. By default, the **fdl_str_dst** argument is interpreted as a file name of an FDL file.
FDL$V_FULL_OUTPUT	Includes the FDL attributes to describe all the bits and fields in the VMS RMS control blocks, including run-time options. If this flag is set, every VMS RMS field is inspected before being written. By default, only the FDL attributes that describe permanent file attributes are included (producing a much shorter FDL specification).
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

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

fab_pointer

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

VMS RMS file access block (FAB). The **fab_pointer** argument is the address of a longword containing the address of a VMS RMS file access block (FAB).

rab_pointer

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

VMS RMS record access block (RAB). The **rab_pointer** argument is the address of a longword containing the address of a VMS RMS record access block (RAB).

fdl_file_dst

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

Name of the FDL file to be created. The **fdl_file_dst** argument is the address of a character string descriptor containing the file name of the FDL file to be created. If the FDL$V_FDL_STRING flag is set in the **flags** argument, this argument is ignored; otherwise, it is required. The FDL specification is written to the file named in this argument.

fdl_file_resnam

VMS usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed-length string descriptor**

Resultant name of the FDL file created. The **fdl_file_resnam** argument is the address of a variable character string descriptor that receives the resultant name of the FDL file created (if FDL$GENERATE is directed to create an FDL file).

This argument is optional.

fdl_str_dst

VMS usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor—fixed-length string descriptor**

FDL specification. The **fdl_str_dst** argument is the address of a variable character string descriptor that receives the FDL specification created. If the FDL$V_FDL_STRING bit is set in the **flags** argument, this argument is required; otherwise, it is ignored.

bad_blk_addr

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

Address of an invalid VMS RMS control block. The **bad_blk_addr** argument is the address of a longword that receives the address of an invalid VMS RMS control block. If an invalid control block (a fatal error) is detected, this argument is returned; otherwise, it is ignored.

This argument is optional.

retlen

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

Number of characters received in either the **fdl_file_resnam** or the **fdl_str_dst** argument. The **retlen** argument is the address of a longword that receives this number.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
FDL$_INVBLK	Invalid VMS RMS control block at virtual address *n*.
RMS$_ACT	File activity precludes operation.
RMS$_CONTROLC	Operation completed under CTRL/C.
RMS$_CONTROLO	Output completed under CTRL/O.
RMS$_CONTROLY	Operation completed under CTRL/Y.
RMS$_DNR	Device not ready or mounted.
RMS$_EXT	ACP file extend failed.
RMS$_OK_ALK	Record already locked.
RMS$_OK_DUP	Record inserted had duplicate key.
RMS$_OK_IDX	Index update error occurred.
RMS$_PENDING	Asynchronous operation pending completion.
RMS$_PRV	Insufficient privilege or file protection violation.
RMS$_REX	Record already exists.
RMS$_RLK	Target record currently locked by another stream.
RMS$_RSA	Record stream currently active.
RMS$_WLK	Device currently write locked.
SS$_ACCVIO	Access violation.
STR$_FATINERR	Fatal internal error in Run-Time Library.
STR$_ILLSTRCLA	Illegal string class.
STR$_INSVIRMEM	Insufficient virtual memory.

FDL$PARSE Parse an FDL Specification

The FDL$PARSE routine parses an FDL specification, allocates VMS RMS control blocks (FABs, RABs, or XABs), and fills in the relevant fields.

FORMAT

FDL$PARSE *fdl_spec ,fdl_fab_pointer ,fdl_rab_pointer [,flags] [,dflt_fdl_spc] [,stmnt_num]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

fdl_spec

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

Name of the FDL file or the actual FDL specification to be parsed. The **fdl_spec** argument is the address of a character string descriptor pointing to either the name of the FDL file or the actual FDL specification to be parsed. If the FDL$V_FDL_STRING flag is set in the **flags** argument, FDL$PARSE interprets this argument as an FDL specification in string form. Otherwise, FDL$PARSE interprets this argument as a file name of an FDL file.

fdl_fab_pointer

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

Address of an RMS file access block (FAB). The **fdl_fab_pointer** argument is the address of a longword that receives the address of an RMS file access block (FAB). FDL$PARSE both allocates the FAB and fills in its relevant fields.

fdl_rab_pointer

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

Address of an RMS record access block (RAB). The **fdl_rab_pointer** argument is the address of a longword that receives the address of an RMS record access block (RAB). FDL$PARSE both allocates the RAB and fills in its relevant fields.

flags

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

Flags (or masks) that control how the **dflt_fdl_spc** argument is interpreted and how errors are signaled. The **flags** argument is the address of a longword containing the control flags. If you omit this argument or specify it as zero, no **flags** are set. The **flags** and their meanings are as follows:

Flag	Description
FDL$V_DEFAULT_STRING	Interprets the **dflt_fdl_spc** argument as an FDL specification in string form. By default, the **dflt_fdl_spc** argument is interpreted as a file name of an FDL file.
FDL$V_FDL_STRING	Interprets the **fdl_spec** argument as an FDL specification in string form. By default, the **fdl_spec** argument is interpreted as a file name of an FDL file.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

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

dflt_fdl_spc

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

Name of the default FDL file or the default FDL specification itself. The **dflt_fdl_spc** argument is the address of a character string descriptor pointing to either the default FDL file or the default FDL specification. If the FDL$V_DEFAULT_STRING flag is set in the **flags** argument, FDL$PARSE interprets this argument as an FDL specification in string form. Otherwise, FDL$PARSE interprets this argument as a file name of an FDL file.

This argument allows you to specify default FDL attributes. In other words, FDL$PARSE processes the attributes specified in this argument, unless you override them with the attributes you specify in the **fdl_spec** argument.

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

This argument is optional.

stmnt_num

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

FDL statement number. The **stmnt_num** argument is the address of a longword that receives the FDL statement number. If the routine completes successfully, the **stmnt_num** argument is the number of statements in the FDL specification. If the routine does not complete successfully, the **stmnt_num** argument receives the number of the statement that caused the error. In general, however, line numbers and statement numbers are not the same.

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

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
LIB$_BADBLOADR	Bad block address.
LIB$_BADBLOSIZ	Bad block size.
LIB$_INSVIRMEM	Insufficient virtual memory.
RMS$_DNF	Directory not found.
RMS$_DNR	Device not ready or not mounted.
RMS$_WCC	Invalid wildcard context (WCC) value.

FDL$RELEASE Free Virtual Memory Obtained By FDL$PARSE

The FDL$RELEASE routine deallocates the virtual memory used by the VMS RMS control blocks created by FDL$PARSE. You must use FDL$PARSE to populate the control blocks if you plan to deallocate memory with FDL$RELEASE later.

FORMAT

FDL$RELEASE *[fab_pointer] [,rab_pointer] [,flags] [,badblk_addr]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

fab_pointer

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

File access block (FAB) to be deallocated using the LIB$FREE_VM system service. The **fab_pointer** argument is the address of a longword containing the address of the file access block (FAB). The FAB must be the same one returned by the FDL$PARSE routine. Any name blocks (NAMs) and extended attribute blocks (XABs) connected to the FAB are also released.

This argument is optional. If you omit this argument or specify it as zero, the FAB (and any associated NAM blocks and XABs) is not released.

rab_pointer

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

Record access block (RAB) to be deallocated using the LIB$FREE_VM system service. The **rab_pointer** argument is the address of a longword containing the address of the record access block (RAB). The address of the RAB must be the same one returned by the FDL$PARSE routine. Any XABs connected to the RAB are also released.

This argument is optional. If you omit this argument or specify it as zero, the RAB (and any associated XABs) is not released.

flags

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

Flag (or mask) that controls how errors are signalled. The **flags** argument is the address of a longword containing the control flag (or a mask). If you omit this argument or specify it as zero, no flag is set. The flag is defined as follows:

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

This argument is optional.

badblk_addr

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

Address of an invalid VMS RMS control block. The **badblk_addr** argument is the address of a longword that receives the address of an invalid VMS RMS control block. If an invalid control block (a fatal error) is detected, this argument is returned; otherwise, it is ignored.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
FDL$_INVBLK	Invalid VMS RMS control block at virtual address *n*.
LIB$_BADBLOADR	Bad block address.
RMS$_ACT	File activity precludes operation.
RMS$_RNL	Record not locked.
RMS$_RSA	Record stream currently active.
SS$_ACCVIO	Access violation.

8 Librarian (LBR) Routines

8.1 Introduction to LBR Routines

Libraries are files that provide a convenient way to organize frequently used modules of code or text. The librarian routines allow you to create and maintain libraries and their modules and to use the data stored in library modules.

You can also create and maintain libraries at the DCL level, using the DCL command LIBRARY. For details, see the *VMS DCL Dictionary*.

8.1.1 Types of Library

You can use the librarian routines to maintain the following types of library:

- Object libraries, which contain the object modules of frequently called routines. The VMS Linker Utility searches specified object module libraries when it encounters a reference it cannot resolve in one of its input files. For more information about how the linker uses libraries, see the description of the VMS Linker Utility in the *VMS Linker Utility Manual*.

  An object library has a default file type of OLB and defaults the file type of input files to OBJ.

- Macro libraries, which contain macro definitions used as input to the assembler. The assembler searches specified macro libraries when it encounters a macro that is not defined in the input file. See the *VAX MACRO and Instruction Set Reference Manual* for information about defining macros.

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

- Help libraries, which contain modules of help messages that provide user information about a program. You can retrieve help messages at DCL level by executing the DCL command HELP, or in your program by calling the appropriate librarian routines. For information about creating help modules for insertion into help libraries, see the description of the Librarian Utility in the *VMS Librarian Utility Manual*.

  A help library has a default file type of HLB and defaults the file type of input files to HLP.

- Text libraries, which contain any sequential record files that you want to retrieve as data for a program. For example, some compilers can retrieve program source code from text libraries. Each text file inserted into the library corresponds to one library module. Your programs can retrieve text from text libraries by calling the appropriate librarian routines.

  A text library has a default file type of TLB and defaults the file type of input files to TXT.

- Shareable image libraries, which contain the symbol tables of shareable images used as input to the linker. For information about how to create a shareable image library, see the descriptions of the Librarian and Linker Utilities in the *VMS Librarian Utility Manual* and the *VMS Linker Utility Manual.*

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

- User-developed libraries, which have characteristics specified when you call the LBR$OPEN routine to create a new library. User-developed libraries allow you to use the librarian routines to create and maintain libraries that are not structured in the form assigned by default to the other library types. Note that you cannot use the DCL command LIBRARY to access user-developed libraries.

8.1.2 Structure of Libraries

You create libraries by executing the DCL command LIBRARY or by calling the LBR$OPEN routine. When object, macro, text, help, or shareable image libraries are created, the Librarian Utility structures them as described in Figures 8–1 and 8–2. You can create user-developed libraries only by calling LBR$OPEN; they are structured as described in Figure 8–3.

8.1.2.1 Library Headers

Every library contains a library header that describes the contents of the library, for example, its type, size, version number, creation date, and number of indexes. You can retrieve data from a library's header by calling the LBR$GET_HEADER routine.

8.1.2.2 Modules

Each library module consists of a header and data. The data is the data you inserted into the library; the header associated with the data is created by the librarian routine and provides information about the module, including its type, attributes, and date of insertion into the library. You can read and update a module's header by calling the LBR$SET_MODULE routine.

8.1.2.3 Indexes and Keys

Libraries contain one or more indexes, which can be thought of as directories of the library's modules. The entries in each index are keys, and each key consists of a key name and a module reference. The module reference is a pointer to the module's header record and is called that record's file address (RFA). Macro, text, and help libraries (see Figure 8–1) contain only one index, called the module name table. The names of the keys in the index are the names of the modules in the library.

Object and shareable image libraries (see Figure 8–2) contain two indexes: the module name table and a global symbol table. The global symbol table consists of all the global symbols defined in the modules in the library. Each global symbol is a key in the index and points to the module in which it was defined.

If you need to point to the same module with several keys, you should create a user-developed library, which can have up to eight indexes (see Figure 8–3). Each index consists of keys that point to the library's modules.

The librarian routines differentiate library indexes by numbering them, starting with 1. For all but user-developed libraries, the module name table is index number 1 and the global symbol table, if present, is index number 2. You number the indexes in user-developed libraries. When you access libraries that contain more than one index, you may have to call LBR$SET_INDEX to tell the librarian routines which index to use.

Figure 8–1 Structure of a Macro, Text, or Help Library

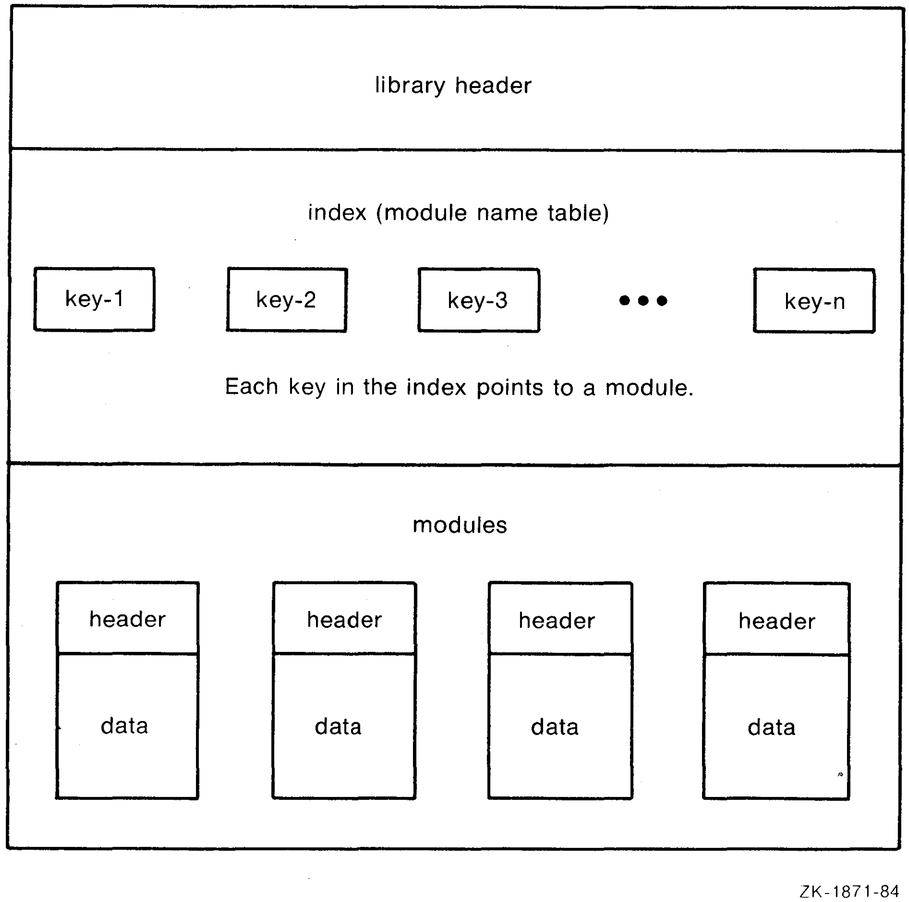


Figure 8–2 Structure of an Object or Shareable Image Library

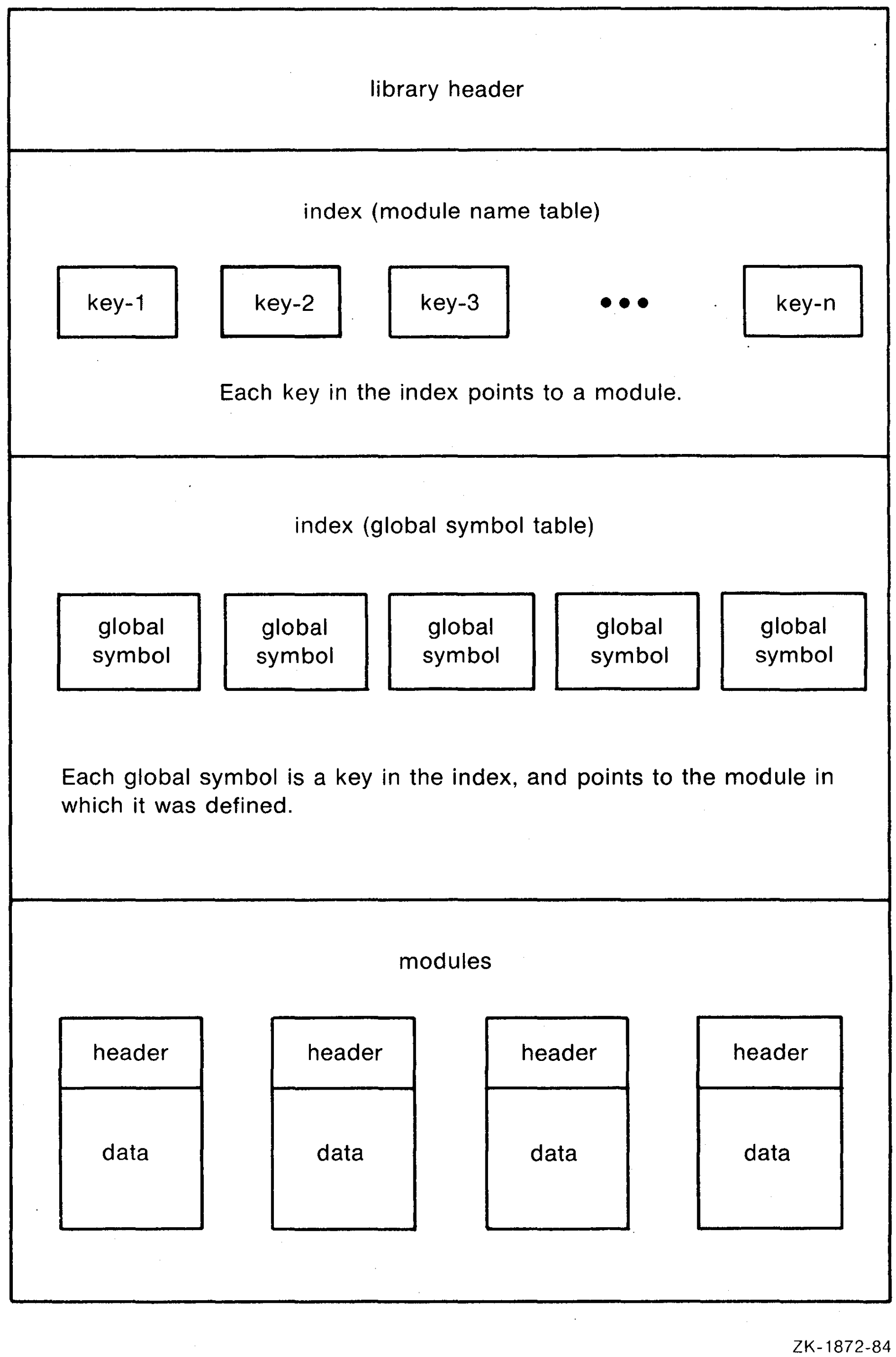
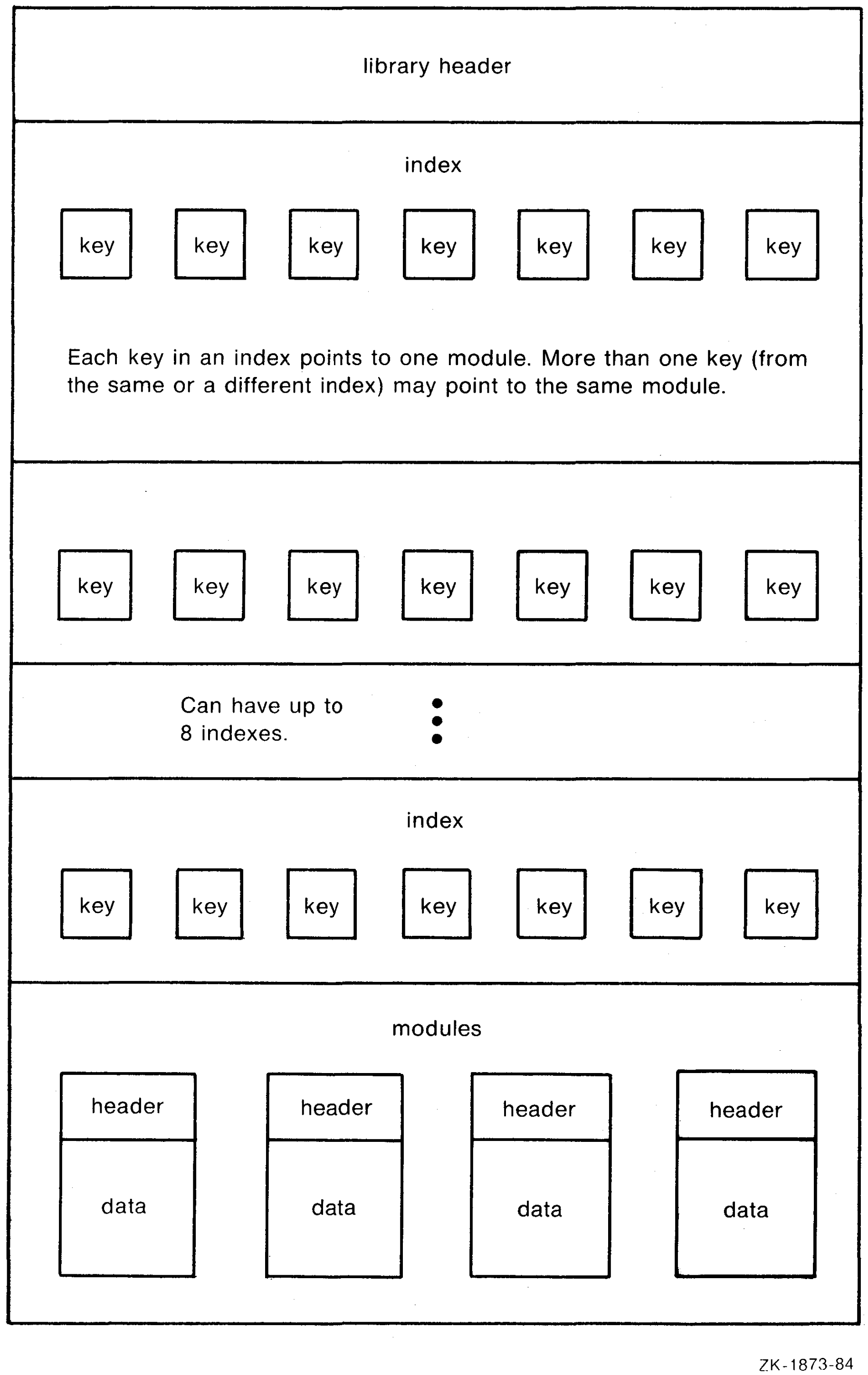


Figure 8–3 Structure of a User-Developed Library



8.1.2.4 Summary of Routines

All the librarian routines begin with the characters LBR$. Your programs can call these routines by using the VMS Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines*. When you call a librarian routine, you must provide whatever arguments the routine requires; when the routine completes execution, it returns a status value to your program. In addition to the condition values listed with the descriptions of each routine, some routines may return the success code SS$_NORMAL as well as various RMS or SS error codes. When you link programs that contain calls to librarian routines, the linker locates the routines during its default search of SYS$SHARE:LBRSHR.

The following table lists the routines and summarizes their functions.

Routine Name	Function
LBR$CLOSE	Closes an open library.
LBR$DELETE_DATA	Deletes a specified module's header and data.
LBR$DELETE_KEY	Deletes a key from a library index.
LBR$FIND	Finds a module by using an address returned by a preceding call to LBR$LOOKUP_KEY.
LBR$FLUSH	Writes the contents of modified blocks to the library file and returns the virtual memory that contained those blocks.
LBR$GET_HEADER	Retrieves information from the library header.
LBR$GET_HELP	Retrieves help text from a specified library.
LBR$GET_HISTORY	Retrieves library update history records and calls a user-supplied routine with each record returned.
LBR$GET_INDEX	Calls a routine to process modules associated with some or all of the keys in an index.
LBR$GET_RECORD	Reads a data record from the module associated with a specified key.
LBR$INI_CONTROL	Initializes a control index that the librarian uses to identify a library.
LBR$INSERT_KEY	Inserts a new key in the current library index.
LBR$LOOKUP_KEY	Looks up a key in the current index.
LBR$OPEN	Opens an existing library or creates a new one.
LBR$OUTPUT_HELP	Retrieves help text from an explicitly named library or from user-supplied default libraries, and optionally prompts you for additional help queries.
LBR$PUT_END	Terminates a sequence of records written to a module with LBR$PUT_RECORD.
LBR$PUT_HISTORY	Inserts a library update history record.
LBR$PUT_RECORD	Writes a data record to the module associated with the specified key.
LBR$REPLACE_KEY	Replaces an existing key in the current library index.
LBR$RET_RMSSTV	Returns the last VMS RMS status value.

Routine Name	Function
LBR$SEARCH	Finds index keys that point to specified data.
LBR$SET_INDEX	Sets the index number to be used during processing of the library.
LBR$SET_LOCATE	Sets librarian subroutine record access to locate mode.
LBR$SET_MODULE	Reads and optionally updates a module header.
LBR$SET_MOVE	Sets librarian subroutine record access to move mode.

8.2 Examples of Using the LBR Routines

This section provides programming examples that show how to call LBR$ routines to create a library, insert a module into a library, extract a module from a library, and delete a module from a library. Although the examples do not use all of the librarian routines, they do provide an introduction to the data structures needed and the calling syntax required to use any of the routines.

For each library you want to work with, you must call LBR$INI_CONTROL and LBR$OPEN before calling any other routine (except LBR$OUTPUT_HELP).

When you call LBR$INI_CONTROL, this routine sets up a control index (do not confuse this with a library index) that is used, in the calls to the other librarian routines, to identify the library to which the routine applies (because you may want your program to work with more than one library at a time). LBR$INI_CONTROL also specifies whether you want to create, read, or modify the library.

After you call LBR$INI_CONTROL, you call LBR$OPEN to open the library and specify its type. When you finish working with a library, you should call LBR$CLOSE to close it. Remember to call LBR$INI_CONTROL again, if you want to reopen the library. LBR$CLOSE deallocates all the memory associated with the library including the control index. The order in which you call the routines between LBR$OPEN and LBR$CLOSE depends upon the library operations you need to perform. You may want to call LBR$LOOKUP_KEY or LBR$GET_INDEX to find a key, then perform some operation on the module associated with the key. You can think of a module as being both the module itself and its associated keys. To access a module, you first need to access a key that points to it; to delete a module, you first need to delete any keys that point to it.

The examples are written in VAX Pascal. In VAX Pascal, all data items, functions (such as the librarian routines), and procedures must be declared at the beginning of the program. Following the declarations is the executable section, which performs the actions of the program. The executable section makes extensive use of the structured control constructs IF-THEN-ELSE and WHILE-condition-DO. Note that code between a BEGIN END pair is treated as a unit.

The listing of each example contains many comments (any code between a pair of asterisks (* *) is a comment), and each listing is followed by notes about the program. The highlighted numbers in the notes are keyed to the highlighted numbers in the examples.

Example 8–1 illustrates the use of LBR routines to create a new library.

Example 8–1 Creating a New Library Using VAX Pascal

```
PROGRAM createlib(INPUT,OUTPUT);
                (*This program creates a text library*)
TYPE                                             (*Data type of*)
    Create_Array = ARRAY [1..20] OF INTEGER;     (*create options array*)
VAR                                      (*Constants and return status error
                                         codes for LBR$_OPEN & LBR$INI_CONTROL.
                                         These are defined in $LBRDEF macro*)
    LBR$C_CREATE,LBR$C_TYP_TXT,LBR$_ILLCREOPT,LBR$_ILLCTL,     ❶
    LBR$_ILLFMT,LBR$_NOFILNAM,LBR$_OLDMISMCH,LBR$_TYPMISMCH :
                                 [EXTERNAL] INTEGER;
                                         (*Create options array codes.  These
                                         are defined in $CREDEF macro*)
    CRE$L_TYPE,CRE$L_KEYLEN,CRE$L_ALLOC,CRE$L_IDXMAX,CRE$L_ENTALL,  ❷
    CRE$L_LUHMAX,CRE$L_VERTYP,CRE$L_IDXOPT,CRE$C_MACTXTCAS,
    CRE$C_VMSV3 :                [EXTERNAL]INTEGER;
    Lib_Name : VARYING [128] OF CHAR;    (*Name of library to create*)
    Options : Create_Array;              (*Create options array*)
    File_Type : PACKED ARRAY [1..4]      (*Character string that is default*)
                OF CHAR := '.TLB';         (*file type of created lib file*)
    lib_index_ptr : UNSIGNED;            (*Value returned in library init*)
    status : UNSIGNED;                   (*Return Status for function calls*)
        (*-*-*-*-Function and Procedure Definitions-*-*-*-*)
                                         (*Function that returns library
                                         control index used by librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;     ❸
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[1..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                         (*Function that creates/opens library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                   fns: [class_s]PACKED ARRAY[1..u:INTEGER] OF CHAR;
                   create_options: Create_Array;
                   dns: [CLASS_S] PACKED ARRAY [13..u3:INTEGER] OF CHAR;
                   rlfna: ARRAY [14..u4:INTEGER] OF INTEGER := %IMMED 0;
                   rns: [CLASS_S] PACKED ARRAY [15..u5:INTEGER] OF CHAR :=
                          %IMMED 0;
                   VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                         (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;
                                         (*Error handler to check error codes
                                         if open/create not successful*)
```

Example 8–1 Cont'd. on next page

Example 8–1 (Cont.) Creating a New Library Using VAX Pascal

```
PROCEDURE Open_Error;       ❹
    BEGIN
        WRITELN('Open Not Successful'); (*Now check specific error codes*)
        IF status = IADDRESS(LBR$_ILLCREOPT) THEN
            WRITELN('    Create Options Not Valid Or Not Supplied');
        IF status = IADDRESS(LBR$_ILLCTL) THEN
            WRITELN('    Invalid Library Index');
        IF status = IADDRESS(LBR$_ILLFMT) THEN
            WRITELN('    Library Not In Correct Format');
        IF status = IADDRESS(LBR$_NOFILNAM) THEN
            WRITELN('    Library Name Not Supplied');
        IF status = IADDRESS(LBR$_OLDMISMCH) THEN
            WRITELN('    Old Library Conflict');
        IF status = IADDRESS(LBR$_TYPMISMCH) THEN
            WRITELN('    Library Type Mismatch')
    END; (*of procedure Open_Error*)
BEGIN  (* *************** DECLARATIONS COMPLETE *************************
        *************** MAIN PROGRAM BEGINS HERE ********************** *)
                                        (*Prompt for Library Name*)
    WRITE('Library Name: '); READLN(Lib_Name);
                                        (*Fill Create Options Array. Divide
                                       by 4 and add 1 to get proper subscript*)
    Options[IADDRESS(CRE$L_TYPE) DIV 4 + 1]   := IADDRESS(LBR$C_TYP_TXT);
    Options[IADDRESS(CRE$L_KEYLEN) DIV 4 + 1] := 31;      ❺
    Options[IADDRESS(CRE$L_ALLOC) DIV 4 + 1]  := 8;
    Options[IADDRESS(CRE$L_IDXMAX) DIV 4 + 1] := 1;
    Options[IADDRESS(CRE$L_ENTALL) DIV 4 + 1] := 96;
    Options[IADDRESS(CRE$L_LUHMAX) DIV 4 + 1] := 20;
    Options[IADDRESS(CRE$L_VERTYP) DIV 4 + 1] := IADDRESS(CRE$C_VMSV3);
    Options[IADDRESS(CRE$L_IDXOPT) DIV 4 + 1] := IADDRESS(CRE$C_MACTXTCAS);
                                        (*Initialize library control index*)
    status := LBR$INI_CONTROL   (lib_index_ptr,      ❻
                                IADDRESS(LBR$C_CREATE),    (*Create access*)
                                IADDRESS(LBR$C_TYP_TXT));  (*Text library*)
    IF NOT ODD(status) THEN             (*Check return status*)
        WRITELN('Initialization Failed')
    ELSE                                (*Initialization was successful*)
        BEGIN                           (*Create and open the library*)
            status := LBR$OPEN  (lib_index_ptr,
                                Lib_Name,
                                Options,       ❼
                                File_Type);
                IF NOT ODD(status) THEN (*Check return status*)
                    Open_Error          (*Call error handler*)      ❽
                ELSE                    (*Open/create was successful*)
                    BEGIN               (*Close the library*)
                        status := LBR$CLOSE(lib_index_ptr);
                        IF NOT ODD(status) THEN (*Check return status*)
                            WRITELN('Close Not Successful')
                    END
        END
END. (*of program creatlib*)
```

Each item in the following list corresponds to a number highlighted in Example 8–1.

❶ To gain access to these LBR$ symbols in your program, write the following two-line MACRO program:

```
$LBRDEF GLOBAL
.END
```

Then assemble the program into an object module by executing the command:

```
MACRO program-name
```

Finally, link the resultant object module with the object module created when your source program is compiled or assembled. (Note: Pascal programmers alternatively may use the INHERIT attribute to include these symbols from SYS$LIBRARY:STARLET.PEN.)

❷ To gain access to the CRE$ symbols, write a two-line MACRO program as described in item 1, substituting $CREDEF for $LBRDEF.

❸ Start the declarations of the librarian routines that are used by the program. Each argument to be passed to the librarian is specified on a separate line and includes the name (which just acts as a placeholder) and data type (for example: UNSIGNED, which means an unsigned integer value, and PACKED ARRAY OF CHAR, which means a character string). If the argument is preceded by VAR, then a value for that argument is returned by the librarian to the program.

❹ Declare the procedure Open_Error, which is called in the executable section if the librarian returns an error when LBR$OPEN is called. Open_Error checks the librarian's return status value to determine the specific cause of the error. The return status values for each routine are listed in the descriptions of the routines.

❺ Initialize the array called Options with the values the librarian needs to create the library.

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

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

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

Example 8–2 illustrates the use of LBR routines to insert a new module into a library.

Example 8–2 Inserting a Module Into a Library Using VAX Pascal

```
PROGRAM insertmod(INPUT,OUTPUT);
                (*This program inserts a module into a library*)
TYPE
    Rfa_Ptr = ARRAY [0..1] OF INTEGER;  (*Data type of RFA of module*)
VAR
    LBR$C_UPDATE,                       (*Constants for LBR$INI_CONTROL*)
    LBR$C_TYP_TXT,                        (*Defined in $LBRDEF macro*)
    LBR$_KEYNOTFND : [EXTERNAL] INTEGER;(*Error code for LBR$LOOKUP_KEY*)
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library receiving module*)
    Module_Name : VARYING [31] OF CHAR; (*Name of module to insert*)
    Text_Data_Record : VARYING [255] OF CHAR; (*Record in new module*)
    Textin : FILE OF VARYING [255] OF CHAR; (*File containing new module*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
    status : UNSIGNED;                  (*Return status for function calls*)
    txtrfa_ptr : Rfa_Ptr;               (*For key lookup and insertion*)
    Key_Not_Found : BOOLEAN := FALSE;   (*True if new mod not already in lib*)
                (*-*-*-*-Function Definitions-*-*-*-*)
                                        (*Function that returns library
                                        control index used by librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[1..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                        (*Function that creates/opens library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                   fns: [class_s]PACKED ARRAY[1..u:INTEGER] OF CHAR;
                   create_options: ARRAY [12..u2:INTEGER] OF INTEGER :=
                        %IMMED 0;
                   dns: [CLASS_S] PACKED ARRAY [13..u3:INTEGER] OF CHAR
                        := %IMMED 0;
                   rlfna: ARRAY [14..u4:INTEGER] OF INTEGER := %IMMED 0;
                   rns: [CLASS_S] PACKED ARRAY [15..u5:INTEGER] OF CHAR :=
                        %IMMED 0;
                   VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                        (*Function that finds a key in index*)
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [1..u:INTEGER] OF
                               CHAR;
                         VAR txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                        (*Function that inserts key in index*)
FUNCTION LBR$INSERT_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [1..u:INTEGER] OF
                               CHAR;
                         txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                        (*Function that writes data records*)
```

Example 8–2 Cont'd. on next page

Example 8–2 (Cont.) Inserting a Module Into a Library Using VAX Pascal

```
FUNCTION LBR$PUT_RECORD (library_index: UNSIGNED;             (*to modules*)
                         textline:[CLASS_S] PACKED ARRAY [1..u:INTEGER] OF
                               CHAR;
                        txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                        (*Function that marks end of a module*)
FUNCTION LBR$PUT_END (library_index: UNSIGNED):
        INTEGER; EXTERN;
                                        (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;
BEGIN  (* *************** DECLARATIONS COMPLETE *************************
        *************** MAIN PROGRAM BEGINS HERE ********************** *)
                                        (*Prompt for library name and
                                        module to insert*)
    WRITE('Library Name: '); READLN(Lib_Name);
    WRITE('Module Name:  '); READLN(Module_Name);
                                        (*Initialize lib for update access*)
    status := LBR$INI_CONTROL   (lib_index_ptr,      ❶
                                IADDRESS(LBR$C_UPDATE),    (*Update access*)
                                IADDRESS(LBR$C_TYP_TXT));  (*Text library*)
    IF NOT ODD(status) THEN             (*Check error status*)
        WRITELN('Initialization Failed')
    ELSE                                (*Initialization was successful*)
        BEGIN
            status := LBR$OPEN  (lib_index_ptr, (*Open the library*)
                                Lib_Name);
            IF NOT ODD(status) THEN     (*Check error status*)
                WRITELN('Open Not Successful')
            ELSE                        (*Open was successful*)
                BEGIN                   (*Is module already in the library?*)
                    status := LBR$LOOKUP_KEY    (lib_index_ptr,      ❷
                                                Module_Name,
                                                txtrfa_ptr);
                    IF ODD(status) THEN (*Check status.  Should not be odd*)
                        WRITELN('Lookup key was successful.',
                                'The module is already in the library.')
                    ELSE   (*Did lookup key fail because key not found?*)
                            IF status = IADDRESS(LBR$_KEYNOTFND) THEN ❸
                                Key_Not_Found := TRUE
                END
        END;
```

Example 8–2 Cont'd. on next page

Example 8–2 (Cont.) Inserting a Module Into a Library Using VAX Pascal

```
        (******If LBR$LOOKUP_KEY failed because the key was not found
        (as expected), we can open the file containing the new module,
        and write the module's records to the library file*******)
    IF Key_Not_Found THEN
        BEGIN
            OPEN(Textin,Module_Name,old);
            RESET(Textin);
            WHILE NOT EOF(Textin) DO            (*Repeat until end of file*)
                BEGIN                     ❹
                    READ(Textin,Text_Data_Record);      (*Read record from
                                                         external file*)
                    status := LBR$PUT_RECORD    (lib_index_ptr,    (*Write*)
                                                Text_Data_Record, (*record to*)
                                                txtrfa_ptr);      (*library*)
                    IF NOT ODD(status) THEN
                        WRITELN('Put Record Routine Not Successful')
                END; (*of WHILE statement*)
            IF ODD(status) THEN (*True if all the records have been
                                successfully written into the library*)
                BEGIN
                    status := LBR$PUT_END (lib_index_ptr); (*Write end of
                                                            module record*)
                    IF NOT ODD(status) THEN
                        WRITELN('Put End Routine Not Successful')
                    ELSE                (*Insert key for new module*)
                        BEGIN                  ❺
                            status := LBR$INSERT_KEY    (lib_index_ptr,
                                                         Module_Name,
                                                         txtrfa_ptr);
                            IF NOT ODD(status) THEN
                                WRITELN('Insert Key Not Successful')
                        END
                END
        END;
    status := LBR$CLOSE(lib_index_ptr);
    IF NOT ODD(status) THEN
        WRITELN('Close Not Successful')
END. (*of program insertmod*)
```

Each item in the following list corresponds to a number highlighted in Example 8–2.

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

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

❸ Call LBR$LOOKUP_KEY to see whether the lookup key failed because the key was not found. (In this case, the status value is LBR$_KEYNOTFND.)

❹ Read a record from the input file, then use LBR$PUT_RECORD to write the record to the library. When all the records have been written to the library, use LBR$PUT_END to write an end of module record.

❺ Use LBR$INSERT_KEY to insert a key for the module into the current index.

Example 8–3 illustrates the use of LBR routines to extract a module from a library.

Example 8–3 Extracting a Module from a Library Using VAX Pascal

```
PROGRAM extractmod(INPUT,OUTPUT,Textout);
                (*This program extracts a module from a library*)
TYPE
    Rfa_Ptr = ARRAY [0..1] OF INTEGER;  (*Data type of RFA of module*)
VAR
    LBR$C_UPDATE,                       (*Constants for LBR$INI_CONTROL*)
    LBR$C_TYP_TXT,                        (*Defined in $LBRDEF macro*)
    RMS$_EOF : [EXTERNAL] INTEGER;      (*RMS return status; defined in
                                        $RMSDEF macro*)
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library receiving module*)
    Module_Name : VARYING [31] OF CHAR; (*Name of module to insert*)
    Extracted_File : VARYING [31] OF CHAR;      (*Name of file to hold
                                                extracted module*)
    Outtext : PACKED ARRAY [1..255] OF CHAR;    (*Extracted mod put here,*)
    Outtext2 : VARYING [255] OF CHAR;           (*     then moved to here*)
    i : INTEGER;                        (*For loop control*)
    Textout : FILE OF VARYING [255] OF CHAR;    (*File containing extracted
                                                module*)
    nullstring : CHAR;                  (*nullstring, pos, and len used to*)
    pos, len : INTEGER;                 (*find string in extracted file recd*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
    status : UNSIGNED;                  (*Return status for function calls*)
    txtrfa_ptr : Rfa_Ptr;               (*For key lookup and insertion*)
                (*-*-*-*-Function Definitions-*-*-*-*)
                        (*Function that returns library
                        control index used by librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[1..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                        (*Function that creates/opens library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                   fns: [class_s]PACKED ARRAY[1..u:INTEGER] OF CHAR;
                   create_options: ARRAY [12..u2:INTEGER] OF INTEGER :=
                        %IMMED 0;
                   dns: [CLASS_S] PACKED ARRAY [13..u3:INTEGER] OF CHAR
                        := %IMMED 0;
                   rlfna: ARRAY [14..u4:INTEGER] OF INTEGER := %IMMED 0;
                   rns: [CLASS_S] PACKED ARRAY [15..u5:INTEGER] OF CHAR :=
                        %IMMED 0;
                   VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                        (*Function that finds a key in an index*)
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [1..u:INTEGER] OF
                               CHAR;
                         VAR txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
```

Example 8–3 Cont'd. on next page

Example 8–3 (Cont.) Extracting a Module from a Library Using VAX Pascal

```
                        (*Function that retrieves records from modules*)
FUNCTION LBR$GET_RECORD (library_index: UNSIGNED;
                         var textline:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR):
                        INTEGER;
EXTERN;
                        (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;
BEGIN  (* *************** DECLARATIONS COMPLETE *************************
        *************** MAIN PROGRAM BEGINS HERE ********************** *)
 (* Get Library Name, Module To Extract, And File To Hold Extracted Module *)
    WRITE('Library Name: '); READLN(Lib_Name);
    WRITE('Module Name:  '); READLN(Module_Name);
    WRITE('Extract Into File: '); READLN(Extracted_File);

    status := LBR$INI_CONTROL   (lib_index_ptr,      ❶
                                 IADDRESS(LBR$C_UPDATE),
                                 IADDRESS(LBR$C_TYP_TXT));
    IF NOT ODD(status) THEN
        WRITELN('Initialization Failed')
    ELSE
        BEGIN
            status := LBR$OPEN  (lib_index_ptr,
                                 Lib_Name);
            IF NOT ODD(status) THEN
                WRITELN('Open Not Successful')
            ELSE
                BEGIN                          ❷
                    status := LBR$LOOKUP_KEY    (lib_index_ptr,
                                                 Module_Name,
                                                 txtrfa_ptr);
                    IF NOT ODD(status) THEN
                        WRITELN('Lookup Key Not Successful')
                    ELSE
                        BEGIN                  ❸
                            OPEN(Textout,Extracted_File,new);
                            REWRITE(Textout)
                        END
                END
        END;
    WHILE ODD(status) DO
        BEGIN
            nullstring := ''(0);
            FOR i := 1 TO 255 DO               ❹
                Outtext[i] := nullstring;
            status := LBR$GET_RECORD    (lib_index_ptr,
                                         Outtext);
            IF NOT ODD(status) THEN
                BEGIN                          ❺
                    IF status = IADDRESS(RMS$_EOF) THEN
                        WRITELN('  RMS end of file')
                END
```

Example 8–3 Cont'd. on next page

Example 8–3 (Cont.) Extracting a Module from a Library Using VAX Pascal

```
          ELSE
              BEGIN                        ❻
                  pos := INDEX(Outtext, nullstring);  (*find first null
                                                       in Outtext*)
                  len := pos - 1;     (*length of Outtext to first null*)
                  IF len >= 1 THEN
                      BEGIN
                          Outtext2 := SUBSTR(Outtext,1,LEN);
                          WRITE(Textout,Outtext2)
                      END
              END
      END; (*of WHILE*)
    status := LBR$CLOSE(lib_index_ptr);
    IF NOT ODD(status) THEN
        WRITELN('Close Not Successful')
END. (*of program extractmod*)
```

Each item in the following list corresponds to a number highlighted in Example 8–3.

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

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

❸ Open an output file to receive the extracted module.

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

❺ Call LBR$GET_RECORD to see if there are more records in the file (module). A failure indicates that the end of the file has been reached.

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

Example 8–4 illustrates the use of LBR routines to delete a library.

Example 8–4 Deleting a Module from a Library Using VAX Pascal

```
PROGRAM deletemod(INPUT,OUTPUT);
        (*This program deletes a module from a library*)
TYPE
    Rfa_Ptr = ARRAY [0..1] OF INTEGER;  (*Data type of RFA of module*)
VAR
    LBR$C_UPDATE,                       (*Constants for LBR$INI_CONTROL*)
    LBR$C_TYP_TXT,                        (*Defined in $LBRDEF macro*)
    LBR$_KEYNOTFND : [EXTERNAL] INTEGER;(*Error code for LBR$LOOKUP_KEY*)
    Lib_Name : VARYING [128] OF CHAR;   (*Name of library receiving module*)
    Module_Name : VARYING [31] OF CHAR; (*Name of module to insert*)
    Text_Data_Record : VARYING [255] OF CHAR; (*Record in new module*)
    Textin : FILE OF VARYING [255] OF CHAR; (*File containing new module*)
    lib_index_ptr : UNSIGNED;           (*Value returned in library init*)
    status : UNSIGNED;                  (*Return status for function calls*)
    txtrfa_ptr : Rfa_Ptr;               (*For key lookup and insertion*)
    Key_Not_Found : BOOLEAN := FALSE;   (*True if new mod not already in lib*)
                (*-*-*-*-Function Definitions-*-*-*-*)
                                (*Function that returns library
                                control index used by librarian*)
FUNCTION LBR$INI_CONTROL (VAR library_index: UNSIGNED;
                          func: UNSIGNED;
                          typ: UNSIGNED;
                          VAR namblk: ARRAY[l..u:INTEGER]
                                OF INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                (*Function that creates/opens library*)
FUNCTION LBR$OPEN  (library_index: UNSIGNED;
                    fns: [class_s]PACKED ARRAY[l..u:INTEGER] OF CHAR;
                    create_options: ARRAY [l2..u2:INTEGER] OF INTEGER :=
                         %IMMED 0;
                    dns: [CLASS_S] PACKED ARRAY [l3..u3:INTEGER] OF CHAR
                         := %IMMED 0;
                    rlfna: ARRAY [l4..u4:INTEGER] OF INTEGER := %IMMED 0;
                    rns: [CLASS_S] PACKED ARRAY [l5..u5:INTEGER] OF CHAR :=
                         %IMMED 0;
                    VAR rnslen: INTEGER := %IMMED 0):
        INTEGER; EXTERN;
                                (*Function that finds a key in index*)
FUNCTION LBR$LOOKUP_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR;
                         VAR txtrfa: Rfa_Ptr):
        INTEGER; EXTERN;
                                (*Function that removes a key from an index*)
FUNCTION LBR$DELETE_KEY (library_index: UNSIGNED;
                         key_name:[CLASS_S] PACKED ARRAY [l..u:INTEGER] OF
                               CHAR):
                         INTEGER;
EXTERN;
```

Example 8–4 Cont'd. on next page

Example 8–4 (Cont.) Deleting a Module from a Library Using VAX Pascal

```
                              (*Function that deletes all the records
                              associated with a module*)
FUNCTION LBR$DELETE_DATA (library_index: UNSIGNED;
                         txtrfa: Rfa_Ptr):
                         INTEGER;
EXTERN;
                              (*Function that closes library*)
FUNCTION LBR$CLOSE (library_index: UNSIGNED):
        INTEGER; EXTERN;

BEGIN  (* *************** DECLARATIONS COMPLETE *************************
        *************** MAIN PROGRAM BEGINS HERE ********************** *)
                              (* Get Library Name and Module to Delete *)
    WRITE('Library Name: '); READLN(Lib_Name);
    WRITE('Module Name:  '); READLN(Module_Name);
                                      (*Initialize lib for update access*)
    status := LBR$INI_CONTROL   (lib_index_ptr,         ❶
                                IADDRESS(LBR$C_UPDATE),     (*Update access*)
                                IADDRESS(LBR$C_TYP_TXT));  (*Text library*)
    IF NOT ODD(status) THEN           (*Check error status*)
        WRITELN('Initialization Failed')
    ELSE                              (*Initialization was successful*)
        BEGIN
            status := LBR$OPEN  (lib_index_ptr, (*Open the library*)
                                Lib_Name);
            IF NOT ODD(status) THEN     (*Check error status*)
                WRITELN('Open Not Successful')
            ELSE                      (*Open was successful*)
                BEGIN        ❷  (*Is module in the library?*)
                    status := LBR$LOOKUP_KEY    (lib_index_ptr,
                                                Module_Name,
                                                txtrfa_ptr);
                    IF NOT ODD(status) THEN (*Check status*)
                        WRITELN('Lookup Key Not Successful')
                END
        END;
    IF ODD(status) THEN               (*Key was found; delete it*)
        BEGIN
            status := LBR$DELETE_KEY    (lib_index_ptr,    ❸
                                        Module_Name);
            IF NOT ODD(status) THEN
                WRITELN('Delete Key Routine Not Successful')
            ELSE                      (*Delete key was successful*)
                BEGIN                 (*Now delete module's data records*)
                    status := LBR$DELETE_DATA   (lib_index_ptr,    ❹
                                                txtrfa_ptr);
                    IF NOT ODD(status) THEN
                        WRITELN('Delete Data Routine Not Successful')
                END
        END;
    status := LBR$CLOSE(lib_index_ptr); (*Close the library*)
    IF NOT ODD(status) THEN
        WRITELN('Close Not Successful');
END. (*of program deletemod*)
```

Each item in the following list corresponds to a number highlighted in Example 8–4.

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

❷ Call LBR$LOOKUP_KEY to find the key associated with the module you want to delete.

❸ Call LBR$DELETE_KEY to delete the key associated with the module you want to delete. If more than one key points to the module, you need to call LBR$LOOKUP_KEY and LBR$DELETE_KEY for each key.

❹ Call LBR$DELETE_DATA to delete the module (the module header and data) from the library.

8.3 LBR Routines

The following pages describe the individual LBR routines.

LBR$CLOSE Close a Library

The LBR$CLOSE routine closes an open library.

FORMAT

LBR$CLOSE *library_index*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

DESCRIPTION

When you are finished working with a library, you should call LBR$CLOSE to close it. Upon successful completion, LBR$CLOSE closes the open library and deallocates all of the memory used for processing it.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.

LBR$DELETE_DATA Delete a Module's Data

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

FORMAT

LBR$DELETE_DATA *library_index ,txtrfa*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

txtrfa

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

Record's file address (RFA) of the module header for the module you want to delete. The **txtrfa** argument is the address of the 2-longword array that contains the RFA. You can obtain the RFA of a module header by calling LBR$LOOKUP_exit KEY or LBR$PUT_RECORD.

DESCRIPTION

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

Note that other librarian routines may reuse data blocks that contain no data.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_INVRFA	Specified RFA not valid.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_STILLKEYS	Keys in other indexes still point at the module header. Therefore, the specified module was not deleted.

LBR$DELETE_KEY Delete a Key

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

FORMAT

LBR$DELETE_KEY *library_index ,key_name*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of a longword containing the index.

key_name

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

Key to be deleted from the library index. For libraries with binary keys, the **key_name** argument is the address of an unsigned longword containing the key number.

For libraries with ASCII keys, the **key_name** argument is the address of the string descriptor pointing to the key with the following argument characteristics.

Argument Characteristics	Entry
VMS Usage	Char_string
Type	Character string
Access	Read only
Mechanism	By descriptor

DESCRIPTION

If LBR$DELETE_KEY finds the key specified by **key_name** in the current index, it deletes the key. Note that, if you want to delete a library module, you should first use LBR$DELETE_KEY to delete any keys that point to it, then use LBR$DELETE_DATA to delete the module's header and associated data.

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

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_KEYNOTFND	Specified key not found.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_UPDURTRAV	Specified index update not valid in a user-supplied routine specified in LBR$SEARCH or LBR$GET_INDEX.

LBR$FIND Look Up a Module by Its RFA

The LBR$FIND routine sets the current internal read context for the library to the library module specified.

FORMAT

LBR$FIND *library_index ,txtrfa*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

txtrfa

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

RFA (record's file address) of the module header for the module you want to access. The **txtrfa** argument is the address of a 2-longword array containing the RFA. You can obtain the RFA of a module header by calling LBR$LOOKUP_KEY or LBR$PUT_RECORD.

DESCRIPTION

You use the LBR$FIND routine to access a module that you had accessed earlier in your program. For example, if you look up several keys with LBR$LOOKUP_KEY, you can save the RFAs returned by LBR$LOOKUP_KEY and later use LBR$FIND to reaccess the modules. Thus, you do not have to look up the module header's key every time you want to access the module. If the specified RFA is valid, LBR$FIND initializes internal tables so that you can read the associated data.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_INVRFA	Specified RFA not valid.
LBR$_LIBNOTOPN	Specified library not open.

LBR$FLUSH Recover Virtual Memory

The LBR$FLUSH routine writes modified blocks back to the library file and frees the virtual memory the blocks had been using.

FORMAT

LBR$FLUSH *library_index ,block_type*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

block_type

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

Extent of the flush operation. The **block_type** argument contains the longword value that indicates how the flush operation proceeds. If you specify LBR$C_FLUSHDATA, the data blocks are flushed. If you specify LBR$C_FLUSHALL, first the data blocks and then the current library index are flushed.

The LBR$ symbols LBR$C_FLUSHDATA and LBR$C_FLUSHALL are defined in the macro $LBRDEF (found in SYS$LIBRARY:STARLET.MLB), which must be assembled and then linked with your program.

DESCRIPTION

LBR$FLUSH cannot be called from other librarian routines that reference cache addresses or by routines called by librarian routines.

CONDITION VALUES RETURNED

LBR$_NORMAL	Operation completed successfully.
LBR$_BADPARAM	Error. A value passed to the LBR$FLUSH routine was either out of range or an illegal value.
LBR$_WRITERR	Error. An error occurred during the writing of the cached update blocks to the library file.

LBR$GET_HEADER Retrieve Library Header Information

The LBR$GET_HEADER routine returns information from the library's header to the caller.

FORMAT

LBR$GET_HEADER *library_index ,retary*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

retary

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

Array of 128 longwords that receives the library header. The **retary** argument is the address of the array that contains the header information. The information returned in the array is listed in the following table (the symbols are defined by the $LHIDEF macro in SYS$LIBRARY:STARLET.MLB).

Offset in Longwords	Symbolic Name	Contents
0	LHI$L_TYPE	Library type (see LBR$OPEN for possible values)
1	LHI$L_NINDEX	Number of indexes
2	LHI$L_MAJORID	Library format major identification
3	LHI$L_MINORID	Library format minor identification

Offset in Longwords	Symbolic Name	Contents
4	LHI$T_LBRVER	ASCIC version of Librarian
12	LHI$L_CREDAT	Creation date/time
14	LHI$L_UPDTIM	Date/time of last update
16	LHI$L_UPDHIS	VBN of start of update history
17	LHI$L_FREEVBN	First logically deleted block
18	LHI$L_FREEBLK	Number of deleted blocks
19	LHI$B_NEXTRFA	Record's File Address (RFA) of end of library
21	LHI$L_NEXTVBN	Next VBN to allocate at end of file
22	LHI$L_FREIDXBLK	Number of free preallocated index blocks
23	LHI$L_FREEIDX	Listhead for preallocated index blocks
24	LHI$L_HIPREAL	VBN of highest preallocated block
25	LHI$L_IDXBLKS	Number of index blocks in use
26	LHI$L_IDXCNT	Number of index entries (total)
27	LHI$L_MODCNT	Number of entries in index 1 (module names)
28	LHI$L_MHDUSZ	Number of bytes of additional information reserved in module header
29	LHI$L_MAXLUHREC	Maximum number of library update history records maintained
30	LHI$L_NUMLUHREC	Number of library update history records in history
31	LHI$L_LIBSTATUS	Library status (false if there was an error closing the library)
32-128		Reserved by DIGITAL

DESCRIPTION

On successful completion, LBR$GET_HEADER places the library header information into the array of 128 longwords.

Note that the offset is the byte offset of the value into the header structure. You can convert the offset to a longword subscript by dividing the offset by 4 and adding 1 (assuming that subscripts in your programming language begin with 1).

CONDITION VALUES RETURNED

LBR$_LIBNOTOPN	Specified library not open.
LBR$_ILLCTL	Specified library control index not valid.

LBR$GET_HELP Retrieve Help Text

The LBR$GET_HELP routine retrieves help text from a help library, displaying it on SYS$OUTPUT or calling your routine for each record returned.

FORMAT

LBR$GET_HELP *library_index [,line_width] [,routine] [,data] [,key_1] [,key_2 . . . ,key_10]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

line_width

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

Width of the help text line. The **line_width** argument is the address of a longword containing the width of the listing line. If you do not supply a line width or if you specify 0, the line width defaults to 80 characters per line.

routine

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

Routine called for each line of text you want output. The **routine** argument is the address of the entry mask for this user-written routine.

If you do not supply a **routine** argument, LBR$GET_HELP calls the Run-Time Library procedure LIB$PUT_OUTPUT to send the help text lines to the current output device (SYS$OUTPUT). However, if you want SYS$OUTPUT for your program to be a disk file rather than the terminal, you should supply a routine to output the text.

The routine you specify is called with an argument list of four longwords:

1. The first argument is the address of a string descriptor for the output line.
2. The second argument is the address of an unsigned longword containing flag bits that describe the contents of the text being passed. The possible flags are as follows:

HLP$M_NOHLPTXT	Specified help text cannot be found.
HLP$M_KEYNAMLIN	Text contains key names of the printed text.
HLP$M_OTHERINFO	Text is part of the information provided on additional help available.

   (The $HLPDEF macro in SYS$LIBRARY:STARLET.MLB defines these flag symbols.)

   Note that, if no flag bit is set, help text is passed.
3. The third argument is the address stipulated in the data argument specified in the call to LBR$GET_HELP (or the address of a 0 constant if the data argument is zero or was omitted).
4. The fourth argument is a longword containing the current key level.

The routine you specify must return with success or failure status. A failure status (low bit = 0) terminates the current call to LBR$GET_HELP.

data

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

Data passed to the routine specified in the **routine** argument. The **data** argument is the address of data for the routine. The address is passed to the routine specified in the **routine** argument. If you omit this argument or specify it as zero, then the argument passed in your routine will be the address of a zero constant.

key_1,key_2, . . . ,key_10

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by descriptor**

Level of the help text to be output. Each **key_1,key_2, . . . ,key_10** argument is the address of a descriptor pointing to the key for that level.

If the **key_1** descriptor is 0 or if it is not present, LBR$GET_HELP assumes that the **key_1** name is HELP, and it ignores all the other keys. For **key_2** through **key_10**, a descriptor address of 0, or a length of 0, or a string address of 0 terminates the list.

The **key** argument may contain any of the following special character strings:

String	Meaning
*	Return all level 1 help text in the library.
KEY . . .	Return all help text associated with the specified key and its subkeys (valid for level 1 keys only).
* . . .	Return all help text in the library.

DESCRIPTION

LBR$GET_HELP returns all help text in the same format as the output returned by the DCL command HELP; that is, it indents two spaces for every key level of text displayed. (Because of this formatting, you may want to make your help messages shorter than 80 characters, so they fit on one line on terminal screens with the width set to 80.) If you do not want the help text indented to the appropriate help level, you must supply your own routine to change the format.

Note that most application programs use LBR$OUTPUT_HELP instead of LBR$GET_HELP.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_NOTHLPLIB	Specified library not a help library.

LBR$GET_HISTORY Retrieve a Library Update History Record

The LBR$GET_HISTORY routine returns each library update history record to a user-specified action routine.

FORMAT

LBR$GET_HISTORY *library_index ,action_routine*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

action_routine

VMS usage: **procedure**
type: **procedure entry mask**
access: **modify**
mechanism: **by reference**

User-supplied routine for processing library update history records. The **action_routine** argument is the address of the entry mask of this user-supplied routine. The routine is invoked once for each update history record in the library. One argument is passed to the routine, namely, the address of a descriptor pointing to a history record.

DESCRIPTION

This routine retrieves the library update history records written by the routine LBR$PUT_HISTORY.

CONDITION VALUES RETURNED

LBR$_NORMAL	Normal exit from the routine.
LBR$_EMPTYHIST	History empty. This is an informational code, not an error code.
LBR$_NOHISTORY	No update history. This is an informational code, not an error code.
LBR$_INTRNLERR	Internal librarian routine error occurred.

LBR$GET_INDEX Call a Routine for Selected Index Keys

The LBR$GET_INDEX routine calls a user-supplied routine for selected keys in an index.

FORMAT

LBR$GET_INDEX *library_index ,index_number ,routine_name [,match_desc]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

index_number

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

Number of the library index. The **index_number** argument is the address of a longword containing the index number. This is the index number associated with the keys you want to use as input to the user-supplied routine (see Section 8.1.2.3).

routine_name

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

User-supplied routine called for each of the specified index keys. The **routine_name** argument is the address of the entry mask for this user-supplied routine.

LBR$GET_INDEX passes two arguments to the routine:

- A key name.
  - For libraries with ASCII keys, the **key_name** argument is the address of a string descriptor pointing to the key. Note that the string and the string descriptor passed to the routine are valid only for the duration of that call. The string must be privately copied if you need it again for more processing.
  - For libraries with binary keys, the **key_name** argument is the address of an unsigned longword containing the key number.
- The record's file address (RFA) of the module's header for this key name. The RFA argument is the address of a 2-longword array that contains the RFA.

The routine must return a value to indicate success or failure. If the routine returns a false value (low bit = 0), LBR$GET_INDEX stops searching the index and returns the status value of the user-specified routine to the calling program.

The routine cannot contain calls to either LBR$DELETE_KEY or LBR$INSERT_KEY.

match_desc

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

Key matching identifier. The **match_desc** argument is the address of a string descriptor pointing to a string used to identify which keys result in calls to the user-supplied routine. Wildcard characters are allowed in this string. If you omit this argument, the routine is called for every key in the index. The **match_desc** argument is valid only for libraries that have ASCII keys.

DESCRIPTION

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

For example, if you call LBR$GET_INDEX with **match_desc** equal to TR* and **index_number** set to 1 (module name table), then LBR$GET_INDEX calls **routine_name** for each module whose name begins with TR.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_ILLIDXNUM	Specified index number not valid.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_NULIDX	Specified library empty.

LBR$GET_RECORD Read a Data Record

The LBR$GET_RECORD routine returns the next data record in the module associated with a specified key.

FORMAT

LBR$GET_RECORD *library_index [,inbufdes] [,outbufdes]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index. The library must be open and LBR$LOOKUP_KEY or LBR$FIND must have been called to find the key associated with the module whose records you want to read.

inbufdes

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

User buffer to receive the record. The **inbufdes** argument is the address of a string descriptor that points to the buffer that receives the record from LBR$GET_RECORD. This argument is required when the librarian subroutine record access is set to move mode (which is the default). This argument is not used if the record access mode is set to locate mode. The DESCRIPTION section contains more information about the locate and move modes.

outbufdes

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

String descriptor that receives the actual length and address of the data for the record returned. The **outbufdes** argument is the address of the string descriptor for the returned record. The length and address fields of the string descriptor are filled in by the LBR$GET_RECORD routine. This parameter must be specified when Librarian subroutine record access is set to locate mode. This parameter is optional if record access mode is set to move mode. The DESCRIPTION section contains more information about the locate and move modes.

DESCRIPTION

Before calling LBR$GET_RECORD, you must first call LBR$LOOKUP_KEY or LBR$FIND to set the internal library read context to the record's file address (RFA) of the module header of the module whose records you want to read.

LBR$GET_RECORD uses two record access modes: locate mode and move mode. Move mode is the default. The LBR$SET_LOCATE and LBR$SET_MOVE subroutines set these modes. The record access modes are mutually exclusive; that is, when one is set the other is turned off. If move mode is set, LBR$GET_RECORD copies the record to the user-specified buffer described by **inbufdes**. If you have optionally specified the output buffer string descriptor, **outbufdes**, the librarian fills it with the actual length and address of the data. If locate mode is set, LBR$GET_RECORD returns the record by way of an internal subroutine buffer, pointing the **outbufdes** descriptor to the internal buffer. The second parameter, **inbufdes**, is not used when locate mode is set.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_LKPNOTDON	Requested key lookup not done.
RMS$_EOF	Error. An attempt has been made to read past the logical end of the data in the module.

LBR$INI_CONTROL Initialize a Library Control Structure

The LBR$INI_CONTROL routine initializes a control structure, called a library control index, to identify the library for use by other Librarian routines.

FORMAT

LBR$INI_CONTROL *library_index ,func [,type] [,namblk]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of a longword that is to receive the index.

func

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

Library function to be performed. The **func** argument is the address of the longword that contains the library function. Valid functions are LBRC_CREATE, LBRC_READ, and LBR$C_UPDATE. (These symbols are defined by the $LBRDEF macro in SYS$LIBRARY:STARLET.MLB.)

type

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

Library type. The **type** argument is the address of the longword containing the library type. Valid library types are LBR$C_TYP_OBJ (object or shareable image), LBR$C_TYP_MLB (macro), LBR$C_TYP_HLP (help), LBR$C_TYP_TXT (text), LBR$C_TYP_UNK (unknown), or, for user-developed libraries, a type in the range of LBR$C_TYP_USRLW through LBR$C_TYP_USRHI.

namblk

VMS usage: **nam**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

VMS RMS name block (NAM). The **namblk** argument is the address of a variable-length data structure containing an RMS NAM block. The LBR$OPEN routine fills in the information in the NAM block so that it can be used later to open the library. If the NAM block has this file identification in it from previous use, the LBR$OPEN routine uses the VMS RMS open-by-NAM block option. This argument is optional and should be used if the library will be opened many times during a single run of the program. For a detailed description of VMS RMS NAM blocks, see the *VMS Record Management Services Manual.*

DESCRIPTION

Except for the LBR$OUTPUT_HELP routine, you must call LBR$INI_CONTROL before calling any other librarian routine. After you initialize the library control index, you must open the library or create a new one using the LBR$OPEN routine. You can then call other librarian routines that you need. After you finish working with a library, close it with the LBR$CLOSE routine.

LBR$INI_CONTROL initializes a library by filling the longword referenced by the **library_index** argument with the control index of the library. Upon completion of the call, the index can be used to refer to the current library in all future routine calls. Therefore, your program must not alter this value.

You can have up to 16 libraries open simultaneously in your program.

CONDITION VALUES RETURNED

LBR$_NORMAL	Library control index initialized successfully.
LBR$_ILLFUNC	Requested function not valid.
LBR$_ILLTYP	Specified library type not valid.
LBR$_TOOMNYLIB	Error. An attempt was made to allocate more than 16 control indexes.

LBR$INSERT_KEY Insert a New Key

The LBR$INSERT_KEY routine inserts a new key in the current library index.

FORMAT

LBR$INSERT_KEY *library_index ,key_name ,txtrfa*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

key_name

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

Name of the new key you are inserting.

If the library uses binary keys, the **key_name** argument is the address of an unsigned longword containing the value of the key.

If the library uses ASCII keys, the **key_name** argument is the address of a string descriptor of the key with the following argument characteristics.

Argument Characteristics	Entry
VMS Usage	Char_string
Type	Character string
Access	Write only
Mechanism	By descriptor

txtrfa

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

Record file address (RFA) of the module associated with the new key you are inserting. The **txtrfa** argument is the address of a 2-longword array containing the RFA. You can use the RFA returned by the first call to LBR$PUT_RECORD.

DESCRIPTION

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

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_INVRFA	Specified RFA does not point to valid data.
LBR$_DUPKEY	Index already contains the specified key.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_UPDURTRAV	LBR$INSERT_KEY was called by the user-defined routine specified in LBR$SEARCH or LBR$GET_INDEX.

LBR$LOOKUP_KEY Look Up a Library Key

The LBR$LOOKUP_KEY routine looks up a key in the library's current index and prepares to access the data in the module associated with the key.

FORMAT

LBR$LOOKUP_KEY *library_index ,key_name ,txtrfa*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

key_name

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

Name of the library key. If the library uses binary keys, the **key_name** argument is the address of the unsigned longword value of the key.

If the library uses ASCII keys, the **key_name** argument is the address of a string descriptor for the key with the following argument characteristics.

Argument Characteristics	Entry
VMS Usage	Char_string
Type	Character string
Access	Read only
Mechanism	By descriptor

txtrfa
VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

The record's file address (RFA) of the library module header. The **txtrfa** argument is the address of the 2-longword array that receives the RFA of the module header.

DESCRIPTION

If LBR$LOOKUP_KEY finds the specified key, it initializes internal tables so that you can access the associated data.

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

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_INVRFA	RFA obtained not valid.
LBR$_KEYNOTFND	Specified key not found.
LBR$_LIBNOTOPN	Specified library not open.

LBR$OPEN Open or Create a Library

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

FORMAT

LBR$OPEN *library_index [,fns] [,create_options] [,dns] [,rlfna] [,rns] [,rnslen]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of a longword containing the index.

fns

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

File specification of the library. The **fns** argument is the address of a string descriptor pointing to the file specification. Unless the VMS RMS NAM block address was previously supplied in the LBR$INI_CONTROL routine and contained a file specification, this argument must be included. Otherwise, the librarian returns an error (LBR$_NOFILNAM).

create_options

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

Library characteristics. The **create_options** argument is the address of an array of 20 longwords that define the characteristics of the library you are creating. If you are creating a library with LBR$C_CREATE, you must include the **create-options** argument. The following table shows the entries that the array must contain (the $LBRDEF and $CREDEF macros in SYS$LIBRARY:STARLET.MLB define the symbols listed).

Offset in Longwords	Symbolic Name	Contents
0	CRE$L_TYPE	Library type:
	LBR$C_TYP_UNK (0)	Unknown/unspecified
	LBR$C_TYP_OBJ (1)	Object and/or shareable image
	LBR$C_TYP_MLB (2)	Macro
	LBR$C_TYP_HLP (3)	Help
	LBR$C_TYP_TXT (4)	Text
	(5-127)	Reserved by DIGITAL
	LBR$C_TYP_USR (128-255)	User-defined
1	CRE$L_KEYLEN	Maximum length of ASCII keys or, if 0, indicates 32-bit unsigned keys (binary keys)
2	CRE$L_ALLOC	Initial library file allocation
3	CRE$L_IDXMAX	Number of library indexes (maximum of 8)
4	CRE$L_UHDMAX	Number of additional bytes to reserve in module header
5	CRE$L_ENTALL	Numb of index entries to preallocate
6	CRE$L_LUHMAX	Maximum number of library update history records to maintain
7	CRE$L_VERTYP	Format of library to create:
	CRE$C_VMSV2	VMS Version 2.0
	CRE$C_VMSV3	VMS Version 3.0
8	CRE$L_IDXOPT	Index key casing option:
	CRE$C_HLPCASING	Treat character case as it is for help libraries
	CRE$C_OBJCASING	Treat character case as it is for object libraries
	CRE$C_MACTXTCAS	Treat character case as it is for macro and text libraries
9-20		Reserved by DIGITAL

The input of uppercase and lowercase characters is treated differently for help, object, macro, and text libraries. For details, see the *VMS Librarian Utility Manual*.

dns

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

Default file specification. The **dns** argument is the address of the string descriptor that points to the default file specification. See the *VMS Record Management Services Manual* for details about how defaults are processed.

rlfna

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

Related file name. The **rlfna** argument is the address of a VMS RMS NAM block pointing to the related file name. If you do not specify **rlfna**, no related file name processing occurs. If a related file name is specified, only the file name, type, and version fields of the NAM block are used for related name block processing. The device and directory fields are not used. See the *VMS Record Management Services Manual* for details on processing related file names.

rns

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

Resultant file specification returned. The **rns** argument is the address of a string descriptor pointing to a buffer that is to receive the resultant file specification string. If an error occurs during an attempt to open the library, the expanded name string is returned instead.

rnslen

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by reference**

Length of the resultant or expanded file name. The **rnslen** argument is the address of a longword receiving the length of the resultant file specification string (or the length of the expanded name string if there was an error in opening the library).

DESCRIPTION

You can call this routine only after you call LBR$INI_CONTROL and before you call any other librarian routine except LBR$OUTPUT_HELP.

When the library is successfully opened, the librarian routine reads the library header into memory and sets the default index to 1.

If the library cannot be opened because it is already open for a write operation, LBR$OPEN retries the open operation every second for a maximum of 30 seconds before returning the VMS RMS error, RMS$_FLK, to the caller.

CONDITION VALUES RETURNED

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

LBR$OUTPUT_HELP Output Help Messages

The LBR$OUTPUT_HELP routine outputs help text to a user-supplied output routine. The text is obtained from an explicitly named help library or, optionally, from user-specified default help libraries. An optional prompting mode is available that enables LBR$OUTPUT_HELP to interact with you and continue to provide help information after the initial help request has been satisfied.

FORMAT

LBR$OUTPUT_HELP *output_routine [,output_width] [,line_desc] [,library_name] [,flags] [,input_routine]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

output_routine

VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**

Name of a routine that writes help text a line at a time. The **output_routine** argument is the address of the entry mask of the routine to call. You should specify either the address of LIB$PUT_OUTPUT or a routine of your own that has the same calling format as LIB$PUT_OUTPUT.

output_width

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

Width of the help-text line to be passed to the user-supplied output routine. The **output_width** argument is the address of a longword containing the width of the text line to be passed to the user-supplied output routine. If you omit **output_width** or specify it as 0, the default output width is 80 characters per line.

line_desc

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

Contents of the help request line. The **line_desc** argument is the address of a string descriptor pointing to a character string containing one or more help keys defining the help requested, for example, the HELP command line minus the HELP command and HELP command qualifiers. The default is a string descriptor for an empty string.

library_name

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

Name of the main library. The **library_name** argument is the address of a string descriptor pointing to the main library file specification string. The default is a null string, which means you should use the default help libraries. If you omit the device and directory specifications, the default is SYS$HELP. The default file type is HLB.

flags

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

Flags specifying help output options. The **flags** argument is the address of an unsigned longword that contains the following flags.

Flag	Description
HLP$M_PROMPT	When set, interactive help prompting is in effect.
HLP$M_PROCESS	When set, the process logical name table is searched for default help libraries.
HLP$M_GROUP	When set, the group logical name table is searched for group default help libraries.
HLP$M_SYSTEM	When set, the system logical name table is searched for system default help libraries.
HLP$M_LIBLIST	When set, the list of default libraries available is output with the list of topics available.
HLP$M_HELP	When set, the list of topics available in a help library is preceded by the major portion of the text on HELP.

(The $HLPDEF macro in SYS$LIBRARY:STARLET.MLB defines these flag symbols.)

If you omit this longword, the default is for prompting and all default library searching to be enabled, but no library list will be generated and no help text will precede the list of topics.

input_routine

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

Routine used for prompting. The **input_routine** argument is the address of the entry mask of the prompting routine. You should specify either the address of LIB$GET_INPUT or a routine of your own that has the same calling format as LIB$GET_INPUT. This argument must be supplied when the HELP command is run in prompting mode (that is, HLP$M_PROMPT is set or defaulted).

DESCRIPTION

The LBR$OUTPUT_HELP routine provides a simple, one-call method to initiate an interactive help session. Help library bookkeeping functions, such as LBR$INI_CONTROL and LBR$OPEN, are handled internally. You should not call LBR$INI_CONTROL or LBR$OPEN before you issue a call to LBR$OUTPUT_HELP.

LBR$OUTPUT_HELP accepts help keys in the same format as LBR$GET_HELP, with the following qualifications:

- If the keyword HELP is supplied, help text on HELP is output, followed by a list of HELP subtopics available.

  If no help keys are provided or if the **line_desc** argument is 0, a list of topics available in the root library is output.
- If the **line_desc** argument contains a list of help keys, then each key must be separated from its predecessor by a slash (/) or by one or more spaces.
- The first key can specify a library to replace the main library as the root library (the first library searched) in which LBR$OUTPUT_HELP searches for help. A key used for this purpose must have the form *<@filespec>* , where *filespec* is subject to the same restrictions as the **library_name** argument. If the specified library is an enabled user-defined default library, then *filespec* can be abbreviated as any unique substring of that default library's logical name translation.

In default library searches, you can define one or more default libraries for LBR$OUTPUT_HELP to search for help information not contained in the root library. You do this by equating logical names (HLP$LIBRARY, HLP$LIBRARY_1, . . . ,HLP$LIBRARY_999) to the file specifications of the default help libraries. You can define these logical names in the process, group, or system logical name table.

If default library searching is enabled by the **flags** argument, LBR$OUTPUT_HELP uses those flags to determine which logical name tables are enabled and then automatically searches any user default libraries that have been defined in those logical name tables. The library search order proceeds as follows: root library, main library (if specified and different from the root library), process libraries (if enabled), group libraries (if enabled), system libraries (if enabled). If the requested help information is not found in any of these libraries, LBR$OUTPUT_HELP returns to the root library and issues a "help not found" message.

To enter an interactive help session (after your initial request for help has been satisfied), you must set the HLP$M_PROMPT bit in the **flags** argument.

You can encounter four different types of prompt in an interactive help session. Each type represents a different level in the hierarchy of help available to you:

1. If the root library is the main library and you are not currently examining HELP for a particular topic, the prompt *Topic?* is output.

2 If the root library is a library other than the main library and if you are not currently examining HELP for a particular topic, a prompt of the form @ <*library-spec*> *Topic?* is output.

3 If you are currently examining HELP for a particular topic (and subtopics), a prompt of the form <*keyword...*> *subtopic?* is output.

4 A combination of 2 and 3.

When you encounter one of these prompt messages, you can respond in any one of several ways. Each type of response and its effect on LBR$OUTPUT_HELP in each prompting situation is described in the following table.

Response	Action in the Current Prompt Environment[1]
keyword [. . .]	(1,2) Search all enabled libraries for these keys.
	(3,4) Search additional help for the current topic (and subtopic) for these keys.
@filespec [keyword[. . .]]	(1,2) Same as above, except that the root library is the library specified by *filespec*. If the specified library does not exist, treat *@filespec* as a normal key.
	(3,4) Same as above; treat *@filespec* as a normal key.
?	(1,2) Display a list of topics available in the root library.
	(3,4) Display a list of subtopics of the current topic (and subtopics) for which help exists.
Carriage Return	(1) Exit from LBR$OUTPUT_HELP.
	(2) Change root library to main library.
	(3,4) Strip the last keyword from a list of keys defining the current topic (and subtopic) environment.
CTRL/Z	(1,2,3,4) Exit from LBR$OUTPUT_HELP.

[1]Keyed to the prompt in the preceding list.

CONDITION VALUES RETURNED

LBR$_ILLINROU	Input routine improperly specified or omitted.
LBR$_ILLOUTROU	Output routine improperly specified or omitted.
LBR$_NOHLPLIS	Error. No default help libraries can be opened.
LBR$_TOOMNYARG	Error. Too many arguments were specified.
LBR$_USRINPERR	Error. An error status was returned by the user-supplied input routine.

LBR$PUT_END Write an End-of-Module Record

The LBR$PUT_END routine marks the end of a sequence of records written to a library by the LBR$PUT_RECORD routine.

FORMAT

LBR$PUT_END *library_index*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of a longword containing the index.

DESCRIPTION

Call LBR$PUT_END after you write data records to the library with the LBR$PUT_RECORD routine. LBR$PUT_END terminates a module by attaching a 3-byte logical end-of-file record (hexadecimal 77,00,77) to the data.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.

LBR$PUT_HISTORY Write an Update History Record

The LBR$PUT_HISTORY routine adds an update history record to the end of the update history list.

FORMAT

LBR$PUT_HISTORY *library_index ,record_desc*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

record_desc

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

Library history record. The **record_desc** argument is the address of a string descriptor pointing to the record to be added to the library update history.

DESCRIPTION

LBR$PUT_HISTORY writes a new update history record. If the library already contains the maximum number of history records (as specified at creation time by CRE$L_LUHMAX, see LBR$OPEN for details), the oldest history record is deleted before the new record is added.

CONDITION VALUES RETURNED

LBR$_NORMAL	Normal exit from the routine.
LBR$_NOHISTORY	No update history. This is an informational code, not an error code.
LBR$_INTRNLERR	Internal librarian error.
LBR$_RECLNG	Record length greater than that specified by LBR$C_MAXRECSIZ. The record was not inserted or truncated.

LBR$PUT_RECORD Write a Data Record

The LBR$PUT_RECORD routine writes a data record beginning at the next free location in the library.

FORMAT

LBR$PUT_RECORD *library_index ,bufdes ,txtrfa*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

bufdes

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

Record to be written to the library. The **bufdes** argument is the address of a string descriptor pointing to the buffer containing the output record.

txtrfa

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by descriptor**

Record's file address (RFA) of the module header. The **txtrfa** argument is the address of a 2-longword array receiving the RFA of the newly created module header upon the first call to LBR$PUT_RECORD.

DESCRIPTION

If this is the first call to LBR$PUT_RECORD, this routine first writes a module header and returns its RFA to the 2-longword array pointed to by **txtrfa**. LBR$PUT_RECORD then writes the supplied data record to the library.
On subsequent calls to LBR$PUT_RECORD, this routine writes the data record beginning at the next free location in the library (after the previous record). The last record written for the module should be followed by a call to LBR$PUT_END.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.

LBR$REPLACE_KEY Replace a Library Key

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

FORMAT

LBR$REPLACE_KEY *library_index ,key_name ,oldrfa ,newrfa*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

key_name

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

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

key_name

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

Library key (for libraries with binary keys). The **key_name** argument is the address of an unsigned longword value for the key.

oldrfa

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

Old RFA. The **oldrfa** argument is the address of a 2-longword array containing the original RFA (returned by LBR$LOOKUP_KEY) of the module header associated with the key you are replacing.

newrfa

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

New RFA. The **newrfa** argument is the address of a 2-longword array containing the RFA (returned by LBR$PUT_RECORD) of the module header associated with the new key.

DESCRIPTION

If LBR$REPLACE_KEY does not find the key in the current index, it calls the LBR$INSERT_KEY routine to insert the key. If LBR$REPLACE_KEY does find the key, it modifies the key entry in the index so that it points to the new module header.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.
LBR$_INVRFA	Specified RFA not valid.

LBR$RET_RMSSTV Return VMS RMS Status Value

The LBR$RET_RMSSTV routine returns the status value of the last VMS Record Management Services (RMS) function performed by any librarian subroutine.

FORMAT

LBR$RET_RMSSTV

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

None.

DESCRIPTION

The LBR$RET_RMSSTV routine returns, as the status value, the status of the last RMS operation performed by the librarian. RMS status codes are defined by the $RMSDEF macro in SYS$LIBRARY:STARLET.MLB.

CONDITION VALUES RETURNED

This routine returns any condition values returned by VMS RMS routines.

LBR$SEARCH Search an Index

The LBR$SEARCH routine finds index keys that point to specified data.

FORMAT

LBR$SEARCH *library_index ,index_number ,rfa_to_find ,routine_name*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

index_number

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

Library index number. The **index_number** argument is the address of a longword containing the number of the index you want to search (see Section 8.1.2.3).

rfa_to_find

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

Record's file address (RFA) of the module whose keys you are searching for. The **rfa_to_find** argument is the address of a 2-longword array containing the RFA (returned earlier by LBR$LOOKUP_KEY or LBR$PUT_RECORD) of the module header.

routine_name

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

Name of a user-supplied routine to process the keys. The **routine_name** argument is the address of the entry mask of a user-supplied routine to call for each key entry containing the RFA (in other words, for each key that points to the same module header).

This user-supplied routine cannot contain any calls to LBR$DELETE_KEY or LBR$INSERT_KEY.

DESCRIPTION

Use LBR$SEARCH to find index keys that point to the same module header. Generally, in index number 1 (the module name table), just one key points to any particular module; thus, you would probably use this routine only to search library indexes where more than one key points to a module. For example, you might call LBR$SEARCH to find all the global symbols associated with an object module in an object library.

If LBR$SEARCH finds an index key associated with the specified RFA, it calls a user-supplied routine with two arguments:

- The key argument, which is the address of either of the following:
  - A string descriptor for the keyname (libraries with ASCII keynames).
  - An unsigned longword for the key value (libraries with binary keys).
- The RFA argument, which is the address of a 2-longword array containing the RFA of the module header.

The routine must return a value to indicate success or failure. If the specified routine returns a false value (low bit = 0), then the index search terminates.

Note that the key found by LBR$SEARCH is valid only during the call to the user-supplied routine. If you want to use the key later, you must copy it.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_ILLIDXNUM	Specified library index number not valid.
LBR$_KEYNOTFND	Librarian did not find any keys with the specified RFA.
LBR$_LIBNOTOPN	Specified library not open.

LBR$SET_INDEX Set the Current Index Number

The LBR$SET_INDEX routine sets the index number to use when processing libraries that have more than one index.

FORMAT

LBR$SET_INDEX *library_index ,index_number*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

index_number

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

Index number you want to establish as the current index number. The **library_index** is the address of the longword that contains the number of the index you want to establish as the current index. (See Section 8.1.2.3.)

DESCRIPTION

When you call LBR$INI_CONTROL, the librarian sets the current library index to 1 (the module name table, unless the library is a user-developed library). If you need to process another library index, you must use LBR$SET_INDEX to change the current library index.

Note that macro, help, and text libraries contain only one index; therefore, you do not need to call LBR$SET_INDEX. Object libraries contain two indexes. If you want to access the global symbol table, you must call the LBR$SET_INDEX routine to set the index number. User-developed libraries can contain more than one index; therefore, you may need to call LBR$SET_INDEX to set the index number.

Upon successful completion, LBR$SET_INDEX sets the current library index to the requested index number. The librarian routines number indexes starting with 1.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_ILLIDXNUM	Library index number specified not valid.
LBR$_LIBNOTOPN	Specified library not open.

LBR$SET_LOCATE Set Record Access to Locate Mode

The LBR$SET_LOCATE routine sets the record access of librarian subroutines to locate mode.

FORMAT

LBR$SET_LOCATE *library_index*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

DESCRIPTION

Librarian record access may be set to move mode (the default set by LBR$SET_MOVE) or locate mode. The setting affects the operation of the LBR$GET_RECORD routine.

If move mode is set (the default), LBR$GET_RECORD copies the requested record to the specified user buffer. If locate mode is set, the record is not copied. Instead, the **outbufdes** descriptor is set to reference the internal librarian subroutine buffer that contains the record.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.

LBR$SET_MODULE Read or Update a Module Header

The LBR$SET_MODULE routine reads, and optionally updates, the module header associated with a given record's file address (RFA).

FORMAT

LBR$SET_MODULE *library_index ,rfa [,bufdesc] [,buflen] [,updatedesc]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

library_index

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

rfa

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

Record's file address (RFA) associated with the module header. The **rfa** argument is the address of a 2-longword array containing the RFA returned by LBR$PUT_RECORD or LBR$LOOKUP_KEY.

bufdesc

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

Buffer that receives the module header. The **bufdesc** argument is the address of a string descriptor pointing to the buffer that receives the module header. The buffer must be the size specified by the symbol MHD$B_USRDAT plus the value of the CRE$L_UHDMAX create option. The MHD$ and CRE$ symbols are defined in the modules $MHDDEF and $CREDEF, which are stored in SYS$LIBRARY:STARLET.MLB.

buflen

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by reference**

Length of the module header. The **buflen** argument is the address of a longword receiving the length of the returned module header.

updatedesc

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

Additional information to be stored with the module header. The **updatedesc** argument is the address of a string descriptor pointing to additional data that the librarian stores with the module header. If you include this argument, the librarian updates the module header with the additional information.

DESCRIPTION

If you specify **bufdesc**, the librarian routine returns the module header into the buffer. If you specify **buflen**, the librarian routine also returns the buffer's length. If you specify **updatedesc**, the routine updates the header information.

You define the maximum length of the update information (by specifying a value for CRE$L_UHDMAX) when you create the library. The librarian zero-fills the information if it is less than the maximum length or truncates it if it exceeds the maximum length.

CONDITION VALUES RETURNED

LBR$_HDRTRUNC	Buffer supplied to hold the module header was too small.
LBR$ILLCTL	Specified library control index not valid.
LBR$_ILLOP	Error. The **updatedesc** argument was supplied and the library was a Version 1.0 library or the library was opened only for read access.
LBR$_INVRFA	Specified RFA does not point to a valid module header.
LBR$_LIBNOTOPN	Specified library not open.

LBR$SET_MOVE Set Record Access to Move Mode

The LBR$SET_MOVE routine sets the record access of Librarian subroutines to move mode.

FORMAT

LBR$SET_MOVE *library_index*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

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

Library control index returned by the LBR$INI_CONTROL routine. The **library_index** argument is the address of the longword that contains the index.

DESCRIPTION

Librarian record access may be set to move mode (the default, set by LBR$SET_MOVE) or locate mode. The setting affects the operation of the LBR$GET_RECORD routine. If move mode is set, LBR$GET_RECORD copies the requested record to the specified user buffer. For details, see the description of LBR$GET_RECORD.

CONDITION VALUES RETURNED

LBR$_ILLCTL	Specified library control index not valid.
LBR$_LIBNOTOPN	Specified library not open.

9 National Character Set (NCS) Utility Routines

9.1 Introduction to NCS Routines

This section describes the National Character Set (NCS) Utility routines. The NCS Utility provides a common facility for defining and accessing collating sequences and conversion functions. Collating sequences are used to compare strings for sorting purposes. Conversion functions are used to derive an altered form of an input string based on an appropriate conversion algorithm.

Using NCS, you can formulate collating sequences and conversion functions, and register them in an NCS library. The NCS routines provide a programming interface to NCS that allows you to access the collating sequences and conversion functions from an NCS library for doing string comparisons.

Typically, NCS collating sequences are selective subsets of the multinational character set. They are used extensively in programming applications involving various national character sets. For example, a program may use the Spanish collating sequence to assign appropriate collating weight to characters from the Spanish national character set. Another program may use the French collating sequence to assign appropriate collating weight to characters in the French national character set.

In addition to providing program access to collating sequences and conversion functions in an NCS library, the NCS routines provide a means for saving definitions in a local file for subsequent use by the comparison and conversion routines.

Table 9–1 lists the NCS routines.

Table 9–1 NCS Routines

Routine	Description
NCS$COMPARE	Compares two strings using specified collating sequence as comparison basis
NCS$CONVERT	Converts a string using the specified conversion function
NCS$END_CS	Terminates the use of a collating sequence by the calling program
NCS$END_CF	Terminates the use of a conversion function by the calling program
NCS$GET_CF	Retrieves the definition of the named conversion function from the NCS library
NCS$GET_CS	Retrieves the definition of the named collating sequence from the NCS library
NCS$RESTORE_CF	Permits the calling program to restore the definition of a "saved" conversion function from a data base or an RMS file
NCS$RESTORE_CS	Permits the calling program to restore the definition of a "saved" collating sequence from a data base or an RMS file
NCS$SAVE_CF	Provides the calling program with information that permits the application to store the definition of a conversion function in a local data base or an RMS file
NCS$SAVE_CS	Provides the calling program with information that permits the application to store the definition of a collating sequence in a local data base or an RMS file

In a typical application, the program does the following:

1. Prepares a string for comparison
2. Makes a call to the NCS$GET routine specifying the appropriate collating sequence
3. Makes one or more calls to the NCS$COMPARE routine which does the actual comparison
4. Terminates the comparison with a call to the NCS$END routine

The program may also include the use of conversion functions in preparation for the comparison routines.

9.2 Examples of How to Use NCS Utility Routines

This section includes two examples of how to use NCS utility routines in program applications. Example 9–1 illustrates the use of NCS utility routines in a FORTRAN program.

Example 9–1 Using NCS Routines in a FORTRAN Program

```
        CSSTRING,STRING1,STRING2
        CSLENGTH,LENGTH1,LENGTH2,CSID,STATUS,RESULT
        NCS$GET_CS,NCS$COMPARE,NCS$END_CS

        CMP(3)

        CMP(1) = '<'
        CMP(2) = '='
        CMP(3) = '>'
C
C       Read the name of the collating sequence.
C
        READ (5,15,END=999) CSLENGTH,CSSTRING
30      FORMAT(' Collating Sequence: ')
C
C       Get the collating sequence from the NCS library.
C
        CSID = 0
        STATUS = NCS$GET_CS (CSID, CSSTRING(1:CSLENGTH))
        IF ((STATUS .AND. 1) .NE. 1) THEN
            CALL LIB$SIGNAL (%VAL(STATUS))
            ENDIF

C
C       Read two strings to be compared according to the collating sequence.
C
        READ (5,15,END=999) LENGTH1,STRING1
        WRITE (6,20)
        READ (5,15,END=999) LENGTH2,STRING2

        IF (LENGTH1 .EQ. 0 .AND. LENGTH2 .EQ. 0) THEN
            GOTO 200
            ENDIF

10      FORMAT(' String1: ')
20      FORMAT(' String2: ')
15      FORMAT (Q,A80)
C
C       Compare the strings.
C
        RESULT = NCS$COMPARE (CSID, STRING1(1:LENGTH1), STRING2(1:LENGTH2))
C
C       Display the results of the comparison.
C

40      FORMAT(' ',A,' ',A,' ',A)

        GOTO 100
```

Example 9–1 Cont'd. on next page

Example 9–1 (Cont.) Using NCS Routines in a FORTRAN Program

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

999     CONTINUE
        END
```

Example 9–2 illustrates the use of NCS routines from a MACRO-32 program.

Example 9–2 Using NCS Routines in a MACRO-32 Program

```
        .TITLE /NCS Conversion Function Example/

        $NAMDEF

        .PSECT  DATA  LONG,NOEXE,WRT

CFID:   .LONG
LENGTH: .WORD

CFNAME_D:
        .ASCID  /EDT_VT2xx/
PROMPT_D:
        .ASCID  /_File:    /

SIZE = 1024

        .ALIGN LONG
INFAB:  $FAB    FNA=FILE,FNS=NAM$C_MAXRSS
INRAB:  $RAB    FAB=INFAB,UBF=REC,USZ=SIZE

FILE:   .BLKB   NAM$C_MAXRSS
FILE_D: .LONG   NAM$C_MAXRSS
        .ADDRESS -
                FILE

REC:    .BLKB   SIZE
REC_D:
        .LONG   SIZE
        .ADDRESS -
                REC
DEST:
        .BLKB   SIZE
DEST_D: .LONG   SIZE
        .ADDRESS -
                DEST

        .PSECT  CODE EXE,NOWRT

        .ENTRY  NCS$EXAMPLE, ^M<>
```

Example 9–2 Cont'd. on next page

Example 9–2 (Cont.) Using NCS Routines in a MACRO-32 Program

```
;
; Get the EDT_VT2xx conversion function from default library.
;
        PUSHAL  CFNAME_D
        PUSHAL  CFID
        CALLS   #2,G^NCS$GET_CF
        BSBW    ERROR
;
; Get the file to be converted.
;
        PUSHAL  LENGTH
        PUSHAL  PROMPT_D
        PUSHAL  FILE_D
        CALLS   #3,G^LIB$GET_INPUT
        BSBW    ERROR
        MOVW    LENGTH,FILE_D
;
; Open the file to be converted.
;
        $OPEN   FAB=INFAB
        BSBW    ERROR
        $CONNECT RAB=INRAB
        BSBW    ERROR
;
; Read each record from the file.
;
LOOP:   $GET    RAB=INRAB
        BLBC    R0,STATUS
;
; Call NCS$CONVERT to convert the input string to EDT fallback.
;
; (e.g., Convert form feed to <FF>, escape to <ESC>, etc...)
;
        MOVW    INRAB+RAB$W_RSZ,REC_D
        PUSHAL  LENGTH
        PUSHAL  DEST_D
        PUSHAL  REC_D
        PUSHAL  CFID
        CALLS   #4,G^NCS$CONVERT
        BSBW    ERROR
        MOVW    LENGTH,DEST_D
;
; Write result to SYS$OUTPUT.
;
        PUSHAL  DEST_D
        CALLS   #1,G^LIB$PUT_OUTPUT
        MOVW    #SIZE,DEST_D
        BRB     LOOP
```

Example 9–2 Cont'd. on next page

Example 9–2 (Cont.) Using NCS Routines in a MACRO-32 Program

```
STATUS: CMPL    RO,#RMS$_EOF
        BEQL    DONE
        BSBW    ERROR
;
; Call NCS$END_CF to free any storage used to hold the conversion function.
;
DONE:   PUSHAL  CFID
        CALLS   #1,G^NCS$END_CF
        BSBW    ERROR
        RET
;
; Error handling.
;
ERROR:  BLBC    RO,10$
        RSB

10$:    $EXIT_S RO

        .END NCS$EXAMPLE
```

9.3 NCS Routines

The following pages describe the NCS routines.

NCS$COMPARE Compare Strings

The NCS$COMPARE routine compares two strings using a specified collating sequence as a comparison basis.

FORMAT

NCS$COMPARE *cs_id, string_1, string_2*

RETURNS

VMS usage: **integer**
type: **longword integer (signed)**
access: **write only**
mechanism: **by value**

Longword condition value. Most routines return a condition value in R0 but the NCS$COMPARE routine uses R0 to return the result of the comparison, as shown in the following table.

Returned Value	Comparison Result
–1	**string_1** is less than **string_2**
0	**string_1** is equal to **string_2**
1	**string_1** is greater than **string_2**

The NCS$COMPARE routine uses the VAX Signaling Mechanism to indicate completion status as described under CONDITION VALUE SIGNALED.

ARGUMENTS

cs_id

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

Address of a longword that NCS uses to identify a collating sequence. The **cs_id** argument is required, and can be obtained by a call to the NCS$GET_CS routine.

All calls to the NCS$COMPARE routine and the call to the NCS$END_CS routine that terminates the comparison must pass this longword identifier. Upon completion, the NCS$END_CS routine releases the memory used to store the collating sequence and sets the value of the longword identifier to zero.

string_1

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

Descriptor (length and address) of the first string.

string_2

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

Descriptor of the second string.

DESCRIPTION

The NCS$COMPARE routine compares two strings using the specified collating sequence as the comparison basis. The routine indicates whether the value of the first string is greater than, less than, or equal to the value of the second string.

CONDITION VALUE SIGNALED

STR$_ILLSTRCLA	Illegal string class. Severe error. The descriptor of **string_1** or **string_2**, or both, contains a class code not supported by the VAX Procedure Calling and Condition Handling Standard.

NCS$CONVERT Convert String

The NCS$CONVERT routine converts a string using the specified conversion function.

FORMAT

NCS$CONVERT *cf_id, source, dest [,ret_length]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cf_id

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

Address of a longword that NCS uses to identify a conversion function. The **cf_id** argument is required, and can be obtained by a call to the NCS$GET_CF routine.

All calls to the NCS$CONVERT routine and the call to the NCS$END_CF routine that terminates the conversion must pass this longword identifier. Upon completion, the NCS$END_CF routine releases the memory used to store the conversion function and sets the value of the longword identifier to zero.

source

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

Descriptor of source string.

dest

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

Descriptor of destination string.

ret_length

VMS usage: **word unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Length of converted string.

DESCRIPTION

Using the specified conversion function, the NCS$CONVERT routine converts the source string and stores the result in the specified destination. Optionally, the calling program may request that the routine return the length of the converted string.

CONDITION VALUES RETURNED

NCS$_NOT_CF	Name of identifier does not refer to a conversion function.
SS$_NORMAL	Successful completion.
LIB$_STRTRU	Successful completion. However, the resultant string was truncated because the storage allocation for the destination string was inadequate.

NCS$END_CF End Conversion Function

The NCS$END_CF routine terminates a conversion function.

FORMAT

NCS$END_CF *cf_id*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

cf_id
VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **modify**
mechanism: **by reference**

Address of a longword that NCS uses to store a nonzero value identifying a conversion function.

The **cf_id** argument is required.

DESCRIPTION

The NCS$END_CF routine indicates to NCS that the calling program no longer needs the conversion function. NCS releases the memory space allocated for the coversion function and sets the value of the longword identifier to zero.

CONDITION VALUES RETURNED

NCS$_NORMAL	Normal successful completion. The longword identifier value is set to zero.
NCS$_NOT_CF	Name of identifier does not refer to a conversion function.

NCS$END_CS End Collating Sequence

The NCS$END_CS routine terminates a collating sequence.

FORMAT

NCS$END_CS *cs_id*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

cs_id
VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **modify**
mechanism: **by reference**

Address of a longword that NCS uses to store a nonzero value identifying a collating sequence.

The **cs_id** argument is required.

DESCRIPTION

The NCS$END_CS routine indicates to NCS that the calling program no longer needs the collating sequence. NCS releases the memory space allocated for the collating sequence and sets the value of the longword identifier to zero.

CONDITION VALUES RETURNED

NCS$_NORMAL	Normal successful completion. The longword identifier value is set to zero.
NCS$_NOT_CS	Name of identifier does not refer to a collating sequence.

NCS$GET_CF Get Conversion Function

The NCS$GET_CF routine retrieves the definition of the named conversion function from the NCS library.

FORMAT

NCS$GET_CF *cf_id[,cfname][,librar]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cf_id

VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **modify**
mechanism: **by reference**

Address of a longword used by NCS to identify a conversion function. The calling program must ensure that the longword contains zero before invoking the NCS$GET_CF routine because the routine stores a nonzero value in the longword. The nonzero value identifies the conversion function. All subsequent calls to the NCS$CONVERT routine and the call to the NCS$END_CF routine to terminate the conversion function pass the longword identifier. When it completes the conversion, the NCS$END_CF routine releases the memory used to store the conversion function and sets the value of the longword identifier to zero.

The conversion function identifier enhances modular programming and permits concurrent use of multiple conversion functions within a program.

The calling program should not attempt to interpret the contents of the longword identifier.

The **cf_id** argument is required.

cfname

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

Name of the conversion function being retrieved.

librar

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

Name of the library where the conversion function is stored.

DESCRIPTION

The NCS$GET_CF routine extracts the named conversion function from the specified NCS library.

If the calling program omits the **cfname** argument, an "identity" conversion function padded with NUL characters (hex 0) is provided. The identity conversion function effectively leaves each character unchanged by converting each character to itself. For example, *A* becomes *A*, *B* becomes *B*, *C* becomes *C*, and so forth.

If the calling program omits the **librar** argument, NCS accesses the default NCS library.

CONDITION VALUES RETURNED

NCS$_DIAG	Operation completed with signalled diagnostics.
NCS$_NOT_CF	Name of identifier does not refer to a conversion function.
NCS$_NOT_FOUND	Name of identifier not found in the NCS library.

NCS$GET_CS Get Collating Sequence

The NCS$GET_CS routine retrieves the definition of the named collating sequence from the NCS library.

FORMAT

NCS$GET_CS *cs_id[,csname][,librar]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cs_id

VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **modify**
mechanism: **by reference**

Address of a longword that NCS uses to store a nonzero value identifying a collating sequence. The calling program must ensure that the longword identifier contains zero before invoking the NCS$GET_CS routine.

All subsequent calls to the NCS$COMPARE routine and the call to the NCS$END_CS routine that terminates the use of the collating sequence must pass this longword identifier. Upon completion of the comparisons, the NCS$END_CS routine releases the memory used to store the collating sequence and sets the value of the longword identifier to zero.

The collating sequence identifier enhances modular programming and permits concurrent use of multiple collating sequences within a program.

The calling program should not attempt to interpret the contents of the longword identifier.

The **cs_id** argument is required.

csname

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

Name of the collating sequence being retrieved.

librar

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

File specification of the library where the collating sequence is stored.

DESCRIPTION

The NCS$GET_CS routine extracts the named collating sequence from the specified NCS library. If the calling program omits the **csname** argument, NCS creates a collating sequence that uses the "native" collating sequence as a basis for the comparisons. This collating sequence is padded with NUL characters (hex 0).

If the calling program omits the **librar** argument, NCS accesses the default NCS library.

CONDITION VALUES RETURNED

NCS$_DIAG	Operation completed with signalled diagnostics.
NCS$_NOT_CS	Name of identifier does not refer to a collating sequence.
NCS$_NOT_FOUND	Name of identifier not found in the NCS library.

NCS$RESTORE_CF Restore Conversion Function

The NCS$RESTORE_CF routine permits the calling program to restore the definition of a saved conversion function from a database or a file.

FORMAT

NCS$RESTORE_CF *cf_id[,length][,address]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cf_id

VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**

Address of a longword that NCS uses to identify a conversion function.

The **cf_id** argument is required.

length

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

Longword that the calling program uses to indicate the length of the conversion function being restored.

address

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

Longword that the calling program uses as a pointer to the conversion function being restored.

DESCRIPTION

The NCS$RESTORE_CF routine, used in conjunction with the NCS$SAVE_CF routine, permits the application program to keep a local copy of the conversion function. The NCS$SAVE_CF routine obtains the length and location of the conversion function and returns it to the application program. The application program subsequently provides this information to the NCS$RESTORE_CF routine, which uses it to access the conversion function.

This routine also does some integrity checking on the conversion function as it is being processed.

CONDITION VALUE RETURNED

NCS$_NOT_CF	Name of identifier does not refer to a conversion function.

NCS$RESTORE_CS Restore Collating Sequence

The NCS$RESTORE_CS routine permits the calling program to restore the definition of a "saved" collating sequence from a database or a file.

FORMAT

NCS$RESTORE_CS *cs_id[,length][,address]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cs_id

VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**

Address of a longword that NCS uses to identify a collating sequence.

The **cs_id** argument is required.

length

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

Longword that the calling program uses to indicate the length of the collating sequence being restored.

address

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

Longword that the calling program uses as a pointer to the collating sequence being restored.

DESCRIPTION

The NCS$RESTORE_CS routine, used in conjunction with the NCS$SAVE_CS routine, permits the application program to keep a local copy of the collating sequence. The NCS$SAVE_CS routine obtains the length and location of the collating sequence and returns it to the application program. The application program subsequently provides this information to the NCS$RESTORE_CS routine, which uses it to access the collating sequence.

This routine also does some integrity checking on the collating sequence as it is being processed.

CONDITION VALUE RETURNED

NCS$_NOT_CS	Name of identifier does not refer to a collating sequence.

NCS$SAVE_CF Save Conversion Function

The NCS$SAVE_CF routine provides the calling program with information that permits the application to store the definition of a conversion function in a local database or a file rather than in the NCS library.

FORMAT

NCS$SAVE_CF *cf_id[,length][,address]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cf_id

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

Address of a longword that NCS uses to identify a conversion function.

The **cf_id** argument is required.

length

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

Longword used to store the length of the specified conversion function.

address

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

Longword used to store the address of the specified conversion function.

DESCRIPTION

The NCS$SAVE_CF routine, used in conjunction with the NCS$_RESTORE_CF routine, permits the application program to store a conversion function definition in a local file or in a database. When the calling program specifies the conversion function identifier, NCS returns the location of the definition and its length in bytes, permitting the calling program to store the definition locally, rather than in an NCS library. Subsequently, the application supplies this information to the NCS$RESTORE_CF routine, which restores the conversion function to a form that can be used by the NCS$CONVERT routine.

This routine also does some integrity checking on the conversion function as it is being processed.

CONDITION VALUE RETURNED

NCS$_NOT_CF	Name of identifier does not refer to a conversion function.

NCS$SAVE_CS Save Collating Sequence

The NCS$SAVE_CS routine provides the calling program with information that permits the application program to store the definition of a collating sequence in a database or a file rather than in the NCS library.

FORMAT

NCS$SAVE_CS *cs_id[,length][,address]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

LBR messages (prefaced by an NCS message) may signal errors detected while the process is accessing the NCS library.

ARGUMENTS

cs_id

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

Address of a longword that NCS uses to identify a collating sequence.

The **cs_id** argument is required.

length

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

Longword that NCS uses to indicate the length of the specified collating sequence to the calling program.

address

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

Longword that NCS uses to indicate the address of the specified collating sequence to the calling program.

DESCRIPTION

The NCS$SAVE_CS routine, used in conjunction with the NCS$_RESTORE_CS routine, permits the application program to store a collating sequence definition in a local file or in a database. When the calling program specifies the collating sequence identifier, NCS returns the location of the definition sequence and its length in bytes, permitting the calling program to store the definition locally, rather than in a library. Subsequently, the application supplies this information to the NCS$RESTORE_CS routine, which restores the collating sequence to a form that can be used by the NCS$COMPARE routine.

This routine also does some integrity checking on the collating sequence as it is being processed.

CONDITION VALUE RETURNED

NCS$_NOT_CS	Name of identifier does not refer to a collating sequence.

10 Print Symbiont Modification (PSM) Routines

10.1 Introduction to PSM Routines

The print symbiont modification (PSM) routines allow you to modify the behavior of the print symbiont supplied with the VMS operating system.

The VMS print symbiont processes data for output to standard line printers and printing terminals by performing the following functions:

- Reading the data from disk
- Formatting the data
- Sending the data to the printing device
- Composing separation pages (flag, burst, and trailer pages) and inserting them into the data stream for printing

Some of the reasons for modifying the print symbiont include the following:

- To include additional information on the separation pages (flag, burst, and trailer) or to format them differently
- To filter and modify the data stream sent to the printer
- To change some of the ways that the symbiont controls the printing device

You may not always be able to modify the print symbiont to suit your needs. For example, you cannot do the following:

- Modify the VMS symbiont's control logic or the sequence in which the symbiont calls routines.
- Modify the interface between the VMS symbiont and the job controller.

If you cannot modify the VMS print symbiont to suit your needs, you may want to write your own symbiont. Section 10.3 describes how to write your own symbiont and integrate it with the VMS operating system. However, you should modify the VMS print symbiont, when possible, rather than write your own symbiont.

The rest of this chapter contains the following information about PSM routines:

- Section 10.2 contains an overview of the VMS print symbiont and of symbionts in general. It explains concepts such as "symbiont streams"; describes the relationship between a symbiont, a device driver, and the job controller; and gives an overview of the VMS print symbiont's internal logic.

  This section is recommended for those who want to either modify the VMS print symbiont or write a new symbiont.

- Section 10.3 details the procedure for modifying the VMS print symbiont. It includes an overview of the entire procedure, followed by a detailed description of each step.
- Section 10.4 contains an example of a simple modification to the VMS print symbiont.
- Section 10.5 describes each PSM routine and the interface used by the routines you substitute for the standard PSM routines.

10.2 VMS Print Symbiont Overview

The VMS operating system supplies two symbionts: a print symbiont, which is an *output* symbiont, and a card reader, which is an *input* symbiont. An *output* symbiont receives tasks from the job controller, whereas an *input* symbiont sends jobs to the job controller. The card reader symbiont cannot be modified. You can modify the print symbiont, described in this section, using PSM routines.

There are two types of output symbiont: device and server. A device symbiont processes data for output to a device, for example, a printer. A server symbiont also processes data but not necessarily for output to a device, for example, a symbiont that copies files across a network. The VMS operating system supplies no server symbionts.

10.2.1 Components of the VMS Print Symbiont

The VMS print symbiont includes the following major components:

- PSM routines that are used to modify the print symbiont
- Routines that implement input, format, and output services in the VMS print symbiont
- Routines that implement the internal logic of the VMS print symbiont

The VMS print symbiont is implemented using the Symbiont Services Facility. This facility provides communication and control between the job controller and symbionts through a set of Symbiont/Job-Controller Interface Routines (SMB routines), which are documented in Chapter 11.

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

10.2.2 Creation of the Print Symbiont Process

The print symbiont is a device symbiont, receiving tasks from the job controller and processing them for output to a printing device. In the VMS operating system, the existence of a print symbiont process is linked to the existence of at least one print execution queue that is started.

The job controller creates the print symbiont process by calling the Create Process ($CREPRC) system service; it does this whenever either of the following conditions occur:

- A print execution queue is started (from the stopped state) and no symbiont process is running the image specified with the START/QUEUE command.

  A print execution queue is started by means of the DCL command START /QUEUE. You can use the /PROCESSOR qualifier with the START /QUEUE command to specify the name of the symbiont image that is to service an execution queue; if you omit /PROCESSOR, then the default symbiont image is PRTSMB.

- Currently existing symbiont processes suited to a print execution queue cannot accept additional devices; that is, the symbionts have no more available streams. In such a case, the job controller creates another print symbiont process. The next section discusses symbiont streams.

The print symbiont process runs as a detached process.

10.2.3 Symbiont Streams

A *stream* is a logical link between a print execution queue and a printing device. When the queue is started (by means of START/QUEUE), the job controller creates a stream linking the queue with a symbiont process. Because each print execution queue has a single associated printing device (specified with the /ON=device_name qualifier in the INITIALIZE/QUEUE or START/QUEUE command), each stream created by the job controller links a print execution queue, a symbiont process, and the queue's associated printer.

A symbiont that can support multiple streams simultaneously (that is, multiple print execution queues and multiple devices) is termed a multithreaded symbiont. The job controller enforces an upper limit of 16 on the number of streams that any symbiont can service simultaneously.

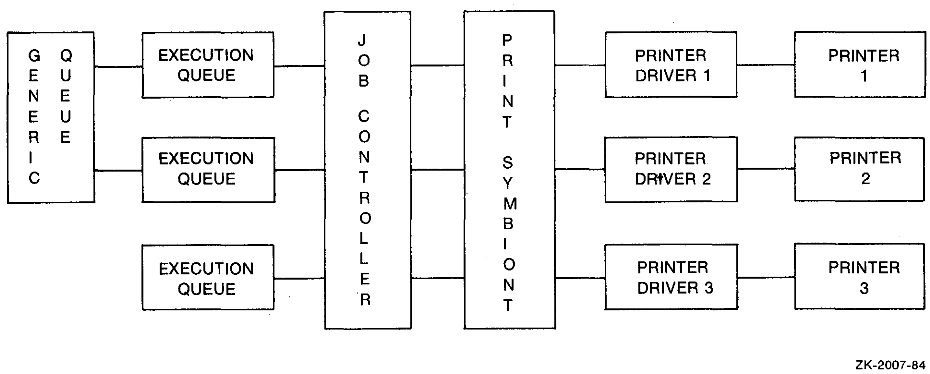
Therefore, in the VMS operating system environment, only one print symbiont process is needed as long as the number of print execution queues (and associated printers) does not exceed 16. If there are more than 16 print execution queues, the job controller creates another print symbiont process.

The VMS print symbiont is, therefore, a multithreaded symbiont that can service as many as 16 queues and devices, but you can modify it to service any number of queues and devices as long as the number is less than or equal to 16.

A symbiont stream is said to be "active" when a queue is started on that stream. The print symbiont maintains a count of active streams. It increments this count each time a queue is started and decrements it when a queue is stopped with the DCL command STOP/QUEUE/NEXT or STOP/QUEUE /RESET. When the count falls to zero, the symbiont process exits. The symbiont does not decrement the count when the queue is paused by STOP /QUEUE.

Figure 10–1 shows the relationship of generic print queues, execution print queues, the job controller, the print symbiont, printer device drivers, and printers. The dotted lines connecting the boxes denote streams.

Figure 10–1 Multithreaded Symbiont



10.2.4 Symbiont and Job Controller Functions

This section compares the roles of the symbiont and job controller in the execution of print requests. You issue print requests using the PRINT command.

The job controller uses the information specified on the PRINT command line to determine the following:

- Which queue to place the job in (/QUEUE, /REMOTE, /LOWERCASE, and /DEVICE)
- How many copies to print (/COPIES and /JOB_COUNT)
- Scheduling constraints for the job (/PRIORITY, /AFTER, /BLOCK_LIMIT, /HOLD, /FORM, /CHARACTERISTICS, and /RESTART)
- How and whether to display the status of jobs and queues (/NOTIFY, /OPERATOR, and /IDENTIFY)

The print symbiont, on the other hand, interprets the information supplied with the qualifiers that specify this information:

- Whether to print file separation pages (/BURST, /FLAG, and /TRAILER)
- Information to include when printing the separation pages (/NAME and /NOTE)
- Which pages to print (/PAGES)
- How to format the print job (/FEED, /SPACE, and /PASSALL)
- How to set up the job (/SETUP)

The print symbiont, not the job controller, performs all necessary device-related functions. It communicates with the printing device driver. For example, when a print execution queue is started (by means of START /QUEUE/ON=device_name) and the stream is established between the queue and the symbiont, the symbiont parses the device name specified by the /ON qualifier in the START/QUEUE command, allocates the device, assigns a channel to it, obtains the device characteristics, and determines the device class. In versions of the VMS operating system prior to Version 4.0, the job controller performed these functions.

The print symbiont's output routine returns an error to the job controller if the device class is neither printer nor terminal.

10.2.5 Print Symbiont Internal Logic

The job controller deals with units of work called jobs, while the print symbiont deals with units of work called tasks. A print job can consist of several print tasks. Thus, in the processing of a print job, the job controller's role is to divide a print job into one or more print tasks, which the symbiont can process. The symbiont reports the completion of each task to the job controller, but the symbiont contains no logic to determine that the print job as a whole is complete.

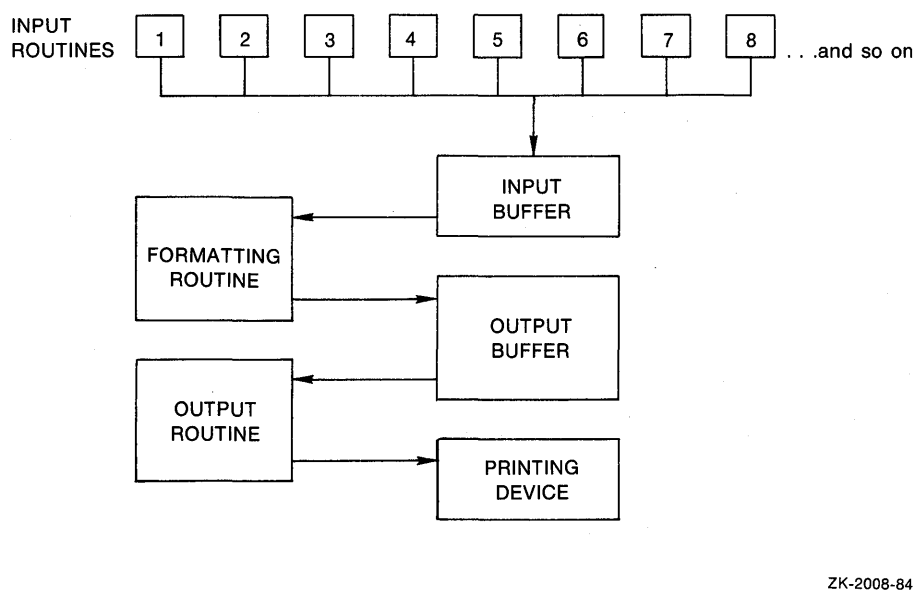
In the processing of a print task, the symbiont performs three basic functions: input, format, and output. The symbiont performs these functions by calling routines to perform each function.

The following steps describe the action taken by the symbiont in processing a task:

1. The symbiont receives the print request from the job controller and stores it in a message buffer.
2. The symbiont searches its list of input routines and selects the first input routine that is applicable to the print task.
3. The input routine returns a data record to the symbiont's input buffer or in a buffer supplied by the input routine.
4. Data in the input buffer is moved to the symbiont's output buffer by the formatting routines, which format it in the process.
5. Data in the output buffer is sent to the printing device by the output routine.
6. When an input routine completes execution, that is, when it has no more input data to process, the symbiont selects another applicable input routine. Steps 3, 4, and 5 are repeated until all applicable input routines have executed.
7. The symbiont informs the job controller that the task is complete.

Figure 10–2 illustrates the steps taken by the symbiont in the processing of a print task.

Figure 10–2 Symbiont Execution Sequence or Flow of Control



As Figure 10–2 shows, most of the input routines execute in a specified sequence. This sequence is defined by the symbiont's main control routine. You cannot modify this main control routine; thus, you cannot modify the sequence in which symbiont routines are called.

The input routines that do not execute in sequence are called "demand input routines." These routines are called whenever the service they provide is required and include the page header, page setup, and library module input routines.

The symbiont can perform input, formatting, and output functions asynchronously; that is, the order in which the symbiont calls the input, formatting, and output routines can vary. For example, the symbiont can call an input routine, which returns a record to the input buffer; it can then call the format routine, which moves that record to the output buffer; and then it can call the output routine to move that data to the printing device. This sequence results in the movement of a single data record from disk to printing device.

On the other hand, the symbiont can call the input and formatting routines several times before calling the output routine for a single buffer. The buffer can contain one or more formatted input records. In some cases an output buffer might contain only a portion of an input record.

In this way the symbiont can buffer input records; then call the format routine, which moves one of those records to the output buffer; and finally call the output routine, which moves that data to the printing device. Note, however, that the formatting routine must be called once for each input record.

Similarly, the symbiont can buffer several formatted records before calling the output routine to move them to the printing device.

The symbiont requires this flexibility in altering the sequence in which input, format, and output routines are called for reasons of efficiency (high rate of throughput) and adaptability to various system parameters and system events.

The value specified with the call to PSM$PRINT determines the maximum size of the symbiont's output buffer, which cannot be larger than the value of the SYSGEN parameter MAXBUF. If the buffer is very small, the symbiont might need to call its output routine one or more times for each record formatted. If the buffer is large, the symbiont will buffer several formatted records before calling the output routine to move them to the printing device.

10.3 Symbiont Modification Procedure

To modify the VMS print symbiont, perform the following steps. These steps are described in more detail in the sections that follow.

1. Determine the modification needed. The modification might involve changing the way the symbiont performs a certain function, or it might involve adding a new function.
2. Determine where to make the modification. This involves selecting a function and determining where that function is performed within the symbiont's execution sequence. You specify a function by calling the PSM$REPLACE routine and specifying the code that identifies the function.

   Some codes correspond to symbiont-supplied routines. When you specify one of these codes, you replace that routine with your routine. Other codes do not correspond to symbiont-supplied routines. When you specify one of these codes, you add your routine to the set of routines the symbiont executes. Table 10–1 lists these codes.
3. Write the routine. Because the symbiont calls your routine, your routine must have one of three call interfaces, depending on whether it is an input, format, or output routine. See the descriptions of the USER-INPUT-ROUTINE, USER-FORMAT-ROUTINE, and USER-OUTPUT-ROUTINE routines, which follow the descriptions of the PSM routines.
4. Write the symbiont-initialization routine. This routine executes when the symbiont is first activated by the job controller. It initializes the symbiont's internal database; specifies, by calling PSM$REPLACE, the routines you have supplied; activates the symbiont by calling PSM$PRINT; and performs any necessary cleanup operations when PSM$PRINT completes.
5. Construct the modified symbiont. This involves compiling your routines, then linking them.
6. Integrate the modified symbiont with the system. This involves placing the executable image in SYS$SYSTEM, identifying the symbiont image to the job controller, and debugging the symbiont.

As mentioned previously, you identify each routine you write for the symbiont by calling the PSM$REPLACE routine. The **code** argument for this routine specifies the point within the symbiont's execution sequence at which you want your routine to execute. You should know which code you will use to identify your routine before you begin to write the routine. Section 10.3.6 provides more information about these codes.

10.3.1 Guidelines and Restrictions

The following guidelines and restrictions apply to the writing of any symbiont routine:

- Do not use the process-permanent files identified by the logical names SYS$INPUT, SYS$OUTPUT, SYS$ERROR, and SYS$COMMAND.
- Do not use the system services SYS$HIBER and SYS$WAKE.
- Use the following two Run-Time-Library routines for allocation and deallocation of memory: LIB$GET_VM and LIB$FREE_VM.
- Minimize the amount of time that your routine spends executing at AST level. The job controller sends messages to the symbiont by means of user-mode ASTs; the symbiont cannot receive these ASTs while your user routine is executing at AST level.
- The symbiont can call your routines at either AST level or non-AST level.
- If your routine returns any error-condition value (low bit clear), the symbiont aborts the current task and notifies the job controller. Note that, by default, an error-condition value returned during the processing of a task causes the job controller to abort the entire job. However, this default behavior may be overridden. See the description of the /RETAIN qualifier of the DCL commands START/QUEUE, INITIALIZE/QUEUE, and SET QUEUE in the *VMS DCL Dictionary*.

  The symbiont stores the first error-condition value (low bit clear) returned during the processing of a task. The symbiont's file-errors routine, an input routine (code PSM$K_FILE_ERRORS), places the message text associated with this condition value in the symbiont's input stream. The symbiont prints this text at the end of the listing, immediately before the trailer pages.

  The symbiont sends this error-condition value to the job controller; the job controller then stores this condition value with the job record in the job controller's queue file. The job controller also writes this condition value in the accounting record for the job.

  If you choose to return a condition value when an error occurs, you should choose one from the system message file. This allows system programs to access the message text associated with the condition value. Specifically, the Accounting and SHOW/QUEUE utilities and the job controller will be able to translate the condition value to its corresponding message text and to display this message text as appropriate.

  This guideline applies to input, input-filter, and output-filter routines, and to the symbiont's use of dynamic string descriptors in these routines.

The simplest way for an input routine to pass the data record to the symbiont is for it to use an RTL string-handling routine (for example, STR$COPY_R). These routines use dynamic string descriptors to point to the record they have handled to copy the record from your input buffer to the symbiont-supplied buffer specified in the **funcdesc** argument to the input routine.

By default, the symbiont initializes a dynamic string descriptor that your input routine can use to describe the data record it returns. Specifically, the symbiont initializes the DSC$B_DTYPE field of the string descriptor with the value DSC$K_DTYPE_T (which indicates that the data to which the descriptor points is a string of characters) and initializes the DSC$B_CLASS field with the value DSC$K_CLASS_D (which indicates that the descriptor is dynamic).

Alternatively, the input routine can pass a data record to the symbiont by providing its own buffer and passing a static string descriptor that describes the buffer. To do this, you must redefine, using the following steps, the fields of the descriptor to which the **funcdesc** argument points:

1. Initialize the field DSC$B_CLASS with the value DSC$K_CLASS_S (which indicates that the descriptor points to a scalar value or a fixed-length string).
2. Initialize the field DSC$A_POINTER with the address of the buffer that contains the data record.
3. Initialize the field DSC$W_LENGTH with the length, in bytes, of the data record.

Each time the symbiont calls the routine to read some data, the symbiont reinitializes the descriptor to make it a dynamic descriptor. Consequently, if you want to use the descriptor as a static descriptor, your input routine must initialize the descriptor as described previously every time it is called to perform a reading operation.

Input-filter routines and output-filter routines return a data record to the symbiont by means of the **func_desc_2** argument. The symbiont initializes a descriptor for this argument the same way it does for descriptors used by input routine described previously. Thus, the guidelines described for the input routine apply to the input-filter routine and output-filter routine.

10.3.2 Writing an Input Routine

This section provides additional information to that given in Section 10.3 on writing an input routine. It provides an overview of the logic used in the VMS print symbiont's main input routine, and it discusses the way in which the VMS print symbiont handles carriage-control effectors.

The print symbiont calls your input routine, supplying it with arguments. Your routine must return arguments and condition values to the print symbiont. For this reason, your input routine must use the interface described in the description of the USER-INPUT-ROUTINE.

When the print symbiont calls your routine, it specifies a particular request in the **func** argument. Each function has a corresponding code.

Your routine must provide the functions identified by the codes PSMK_OPEN, PSMK_READ, and PSM$K_CLOSE. Your routine need not respond to the other function codes, but it can if you want it to. If your routine does not provide a function that the symbiont requests, it must return the condition value PSM$_FUNNOTSUP to the symbiont.

The description of the **func** argument of the USER-INPUT-ROUTINE describes the codes that the symbiont can send to an input routine.

See Section 10.3.5 for additional information about other function codes used in the user-written input routine.

For each task that the symbiont processes, it calls some input routines only once, and some more than once; it always calls some routines, and calls others only when needed.

Table 10–1 lists the codes that you can specify when you call the PSM$REPLACE routine to identify your input routine to the symbiont. The description of the PSM$REPLACE routine describes these routines.

10.3.2.1 Internal Logic of the Symbiont's Main Input Routine

The internal logic of the symbiont's main input routine, as described in this section, is subject to change without notice. This logic is summarized here. This summary is not intended as a tutorial on the writing of a symbiont's main input routine, although it does provide insight into such a task.

A main input routine is one that the symbiont calls to read data from the file that is to be printed. A main input routine must perform three sets of tasks: one set when the symbiont calls the routine with an OPEN request, one set when the symbiont calls with a READ request, and one set when the symbiont calls with a CLOSE request.

The following table names the codes that identify each of these three requests and describes the tasks that the VMS symbiont's main input routine performs for each of these requests.

Code	Action Taken by the Input Routine
PSM$K_OPEN	An OPEN request. When the main input routine receives this request code, it does the following: **1** Opens the input file. **2** Stores information about the input file. **3** Returns the type of carriage control used in the input file. If this routine cannot open the file, it returns an error.

Code	Action Taken by the Input Routine
	Note: **The VMS print symbiont's main input routine performs these tasks when it receives the PSM$K_START_TASK function code, rather than the PSM$K_OPEN function code.** **This atypical behavior occurs because some of the information stored by the main input routine must be available for other input routines that execute before the main input routine. For example, information about file attributes and record formats is needed by the symbiont's separation-page routines, which print flag and burst pages.** **Consequently, if you supply your own main input routine, some of the information about the file being printed that appears on the standard separation pages is not available, and the symbiont prints a message on the separation page stating so.** The symbiont receives the file-identification number from the job controller in the SMBMSG$K_FILE_IDENTIFICATION item of the requesting message and uses this value rather than the file specification to open the main input file.
PSM$K_READ	A READ request. When the main input routine receives this request, it returns the next record from the file. In addition, when the carriage control used by the data file is PSM$K_CC_PRINT, the main input routine returns the associated record header.
PSM$K_CLOSE	A CLOSE request. When the main input routine receives this request, it closes the input file.

10.3.2.2 Symbiont Processing of Carriage Control

Each input record can be thought of as consisting of three parts: leading carriage control, data, and trailing carriage control. Taken together, these three parts are called the composite data record.

Leading and trailing carriage control are determined by the type of carriage control used in the file and explicit carriage-control information returned with each record. For embedded carriage control, however, leading and trailing carriage control is always null.

The type of carriage control returned by the main input routine on the PSM$K_OPEN request code determines, for that invocation of the input routine, how the symbiont applies carriage control to each record that the main input routine returns on the PSM$K_READ request code.

Note that, for all four carriage control types, the first character returned on the first PSM$K_READ call to an input routine receives special processing. If that character is a linefeed or a formfeed and if the symbiont is currently at line 1, column 1 of the current page, then the symbiont discards that linefeed or formfeed.

The Four Types of Carriage Control

The following table briefly describes each type of carriage control and how the symbiont's main input routine processes it. For a detailed explanation of each of these types of carriage control, refer to the description of the FAB$B_RAT field of the FAB block in the *VMS Record Management Services Manual*.

Type of Carriage Control	Symbiont Processing
Embedded	Leading and trailing carriage control are embedded in the data portion of the input record. Therefore, the symbiont supplies no special carriage control processing; it assumes that leading and trailing carriage control are null.
FORTRAN	The first byte of each data record contains a FORTRAN carriage-control character. This character specifies both the leading and trailing carriage control for the data record. The symbiont extracts the first byte of each data record and interprets that byte as a FORTRAN carriage-control character. If the data record is empty, the symbiont generates a leading carriage control of linefeed and a trailing carriage control of carriage return.
PRN	Each data record contains a two-byte header that contains the carriage-control specifier. The first byte specifies the carriage control to apply before printing the data portion of the record. The second byte specifies the carriage control to apply after printing the data portion. The abbreviation PRN stands for print-file format. Unlike other types of carriage control, PRN carriage control information is returned through the **funcarg** argument of the main input routine; this occurs with the PSM$K_READ request. The **funcarg** argument specifies a longword; your routine writes the 2-byte PRN carriage control specifier into the first two bytes of this longword.
Implied	The symbiont provides a leading linefeed and a trailing carriage return. But if the data record consists of a single formfeed, the symbiont sets to null the leading and trailing carriage control for that record, and the leading carriage control for the record that follows it.

10.3.3 Writing a Format Routine

To write a format routine, follow the modification procedure described in Section 10.3. Do not replace the VMS symbiont's main format routine. Instead, modify its action by writing input and output filter routines. These execute immediately before and after the main format routine, respectively. The main formatting routine uses an undocumented and nonpublic interface; you cannot replace the main formatting routine. The DCL command PRINT /PASSALL bypasses the main format routine of the print symbiont.

See Section 10.3.5 for additional information about other function codes used in the user-written formatting routine.

10.3.3.1 Internal Logic of the Symbiont's Main Format Routine

The main format routine contains all the logic necessary to convert composite data records to a data stream for output. Actions taken by the format routine include the following:

- Tracking the current column and line
- Implementing the special processing of the first character of the first record
- Implementing the alignment data mask specified by the DCL command START/QUEUE/ALIGN=MASK
- Handling margins as specified by the forms definition
- Initiating processing of page headers when specified by the DCL command PRINT/HEADER
- Expanding leading and trailing carriage control
- Handling line overflow
- Handling page overflow
- Expanding tab characters to spaces for some devices
- Handling escape sequences
- Accumulating accounting information
- Implementing double-spacing when specified by the DCL command PRINT/SPACE
- Implementing automatic page ejection when specified by the DCL command PRINT/FEED

The symbiont's main format routine uses a special rule when processing the first character of the first composite data record returned by an input routine. (A composite data record is the input data record and a longword that contains carriage-control information for the input data record.) This rule is that if the first character is a vertical format effector (formfeed or linefeed) and if the symbiont has processed no printable characters on the current page (that is, the current position is column 1, line 1), then that vertical format effector is discarded.

10.3.4 Writing an Output Routine

To write an output routine, follow the modification procedure described in Section 10.3.

The print symbiont calls your output routine. Input arguments are supplied by the print symbiont; output arguments and status values are returned by your routine to the print symbiont. For this reason, your output routine must have the call interface that is described in the USER-OUTPUT-ROUTINE routine.

When the print symbiont calls your routine, it specifies in one of the input arguments—the **func** argument—the reason for the call. Each reason has a corresponding function code.

There are several function codes that the print symbiont can supply when it calls your output routine. Your routine must contain the logic to respond to the following function codes: PSMK_OPEN, PSMK_WRITE, PSM$K_WRITE_NOFORMAT, and PSM$K_CLOSE.

It is not required that your output routine contain the logic to respond to the other function codes, but you can provide this logic if you want to.

A complete list and description of all relevant function codes for output routines is provided in the description of the **func** argument of the USER-OUTPUT-ROUTINE routine.

See Section 10.3.5 for additional information about other function codes.

10.3.4.1 Internal Logic of the Symbiont's Main Output Routine

When the symbiont calls the main output routine with the PSM$K_OPEN function code, the main output routine takes the following steps:

1. Allocates the print device
2. Assigns a channel to the device
3. Obtains the device characteristics
4. Returns the device-status longword in the **funcarg** argument (for more information, see the description of the SMBMSG$K_DEVICE_STATUS message item in Section 10, Symbiont/Job-Controller Interface (SMB) Routines)
5. Returns an error if the device is not a terminal or a printer

When this routine receives a PSM$K_WRITE service request code, it sends the contents of the symbiont output buffer to the device for printing.

When this routine receives a PSM$K_WRITE_NOFORMAT service request code, it sends the contents of the symbiont output buffer to the device for printing and suppresses device drive formatting as appropriate for the device in use.

When this routine receives a PSM$K_CANCEL service request code, it requests the device driver to cancel any outstanding output operations.

When this routine receives a PSM$K_CLOSE service request code, it deassigns the channel to the device and deallocates the device.

10.3.5 Other Function Codes

A status PSM$_PENDING may not be returned whenever the symbiont notifies user-written input, output, and format routines using the following message function codes.

Function Code	Description
PSM$K_START_STREAM	Job controller sends a message to the symbiont to start a queue
PSM$K_START_TASK	Symbiont parses a message from job controller directing it to start a queue
PSM$K_PAUSE_TASK	Job controller sends a message to the symbiont to suspend processing of the current task
PSM$K_STOP_STREAM	Job controller sends a message to the symbiont to stop the queue
PSM$K_STOP_TASK	Job controller sends a message to the symbiont to stop the task
PSM$K_RESUME_TASK	Job controller sends a message to the symbiont to resume processing of the current task
PSM$K_RESET_STREAM	Same as PSM$K_STOP_STREAM

10.3.6 Writing a Symbiont Initialization Routine

Writing a symbiont initialization routine involves writing a program that calls the following:

1 PSM$REPLACE once for each routine (input, output, or format) that you have written. PSM$REPLACE identifies your routines to the symbiont.

2 PSM$PRINT exactly once after you have identified all your service routines using PSM$REPLACE.

Table 10–1 lists all routine codes that you can specify in the PSM$REPLACE routine. Choosing the correct routine code for your routine is important because the routine code specifies when the symbiont will call your routine. The functions of these routines are described further in the description of the PSM$REPLACE routine.

Column one in Table 10–1 lists each routine code.

For those input routines that execute in a predefined sequence, the second column contains a number showing the order in which that input routine is called relative to the other input routines for a single file job. If the routine does not execute in a predefined sequence, the second column contains the character *x*.

Column three specifies whether the routine is an input, format, or output routine; this information directs you to the section describing how to write a routine of that type.

Column four specifies whether there is a symbiont-supplied routine corresponding to that routine code. The codes for the input-filter and output-filter routines, which have no corresponding routines in the VMS symbiont, allow you to specify new routines for inclusion in the symbiont.

Table 10–1 Routine Codes for Specification to PSM$REPLACE

Routine Code	Sequence	Function	Supplied
PSM$K_JOB_SETUP	1	Input	Yes
PSM$K_FORM_SETUP	2	Input	Yes
PSM$K_JOB_FLAG	3	Input	Yes
PSM$K_JOB_BURST	4	Input	Yes
PSM$K_FILE_SETUP	5	Input	Yes
PSM$K_FILE_FLAG	6	Input	Yes
PSM$K_FILE_BURST	7	Input	Yes
PSM$K_FILE_SETUP_2	8	Input	Yes
PSM$K_MAIN_INPUT	9	Input	Yes
PSM$K_FILE_INFORMATION	10	Input	Yes
PSM$K_FILE_ERRORS	11	Input	Yes
PSM$K_FILE_TRAILER	12	Input	Yes
PSM$K_JOB_RESET	13	Input	Yes
PSM$K_JOB_TRAILER	14	Input	Yes
PSM$K_JOB_COMPLETION	15	Input	Yes
PSM$K_PAGE_SETUP	x	Input	Yes
PSM$K_PAGE_HEADER	x	Input	Yes
PSM$K_LIBRARY_INPUT	x	Input	Yes
PSM$K_INPUT_FILTER	x	Formatting	No
PSM$K_MAIN_FORMAT	x	Formatting	Yes
PSM$K_OUTPUT_FILTER	x	Formatting	No
PSM$K_OUTPUT	x	Output	Yes

10.3.7 Integrating a Modified Symbiont

To integrate your user routine and the symbiont initialization routine, perform the following steps; note that the sequence of steps described here assumes that you will be debugging the modified symbiont:

1. Compile or assemble the user routine and the symbiont initialization routine into an object module.
2. Enter the following DCL command:

   ```
   $ LINK/DEBUG your-symbiont
   ```

   The file name *your-symbiont* is the object module built in step 1. Symbols necessary for this link operation are located in the shareable images SYS$SHARE:SMBSRVSHR.EXE and SYS$LIBRARY:IMAGELIB.EXE. The linker automatically searches these shareable images and extracts the necessary information.
3. Place the resulting executable symbiont image in SYS$SYSTEM.
4. Locate two unallocated terminals: one at which to issue DCL commands and one at which to debug the symbiont image.

5 Log in on one of the terminals under UIC [1,4], which is the system manager's account. This terminal is the one at which you enter DCL commands. Do not log in at the other terminal.

6 Enter the following DCL command:

```
$ SET TERMINAL/NODISCONNECT/PERMANENT _TTcu:
```

The variable *_TTcu:* is the physical terminal name of the terminal at which you want to debug (the terminal you are not logged in at). The underscore (_) and colon (:) characters must be specified.

7 Enter the following DCL commands:

```
$ DEFINE/GROUP DBG$INPUT  _TTcu:
$ DEFINE/GROUP DBG$OUTPUT _TTcu:
```

The variable *_TTcu:* specifies the physical terminal name of the terminal at which you will be debugging. Note that other users having a UIC with group number 1 should not use the debugger at the same time.

8 Initialize the queue by entering the following DCL command:

```
$ INITIALIZE/QUEUE/PROCESSOR= your-symbiont /ON= printer_name
```

The symbiont image specified by the file name *your-symbiont* must reside in SYS$SYSTEM. Note too that the /PROCESSOR qualifier accepts only a file name; the device, directory, and file type default to SYS$SYSTEM:.EXE.

The /ON qualifier specifies the device that will be served by the symbiont while you debug the symbiont.

9 Enter the following DCL command to execute the modified symbiont routine:

```
$ PRINT/HEADER/QUEUE=queue-id
```

Enter the following DCL command to start the queue and invoke the debugger:

```
$ START/QUEUE queue-name
```

10 After you debug your symbiont, relink the symbiont by entering the following DCL command:

```
$ LINK/NOTRACEBACK/NODEBUG your-symbiont
```

11 Deassign the logical names DBG$INPUT and DBG$OUTPUT so that they will not interfere with other users in UIC group 1.

10.4 Example of Using the PSM Routines

Example 10–1 shows how to use PSM routines to supply a page header routine in a MACRO program.

Example 10–1 Using PSM Routines to Supply a Page Header Routine in a MACRO Program

```
        .TITLE  EXAMPLE - Example user modified symbiont
        .IDENT  'V03-000'

;++
;       THIS PROGRAM SUPPLIES A USER WRITTEN PAGE HEADER
;       ROUTINE TO THE STANDARD SYMBIONT.  THE PAGE HEADER
;       INCLUDES THE SUBMITTER'S ACCOUNT NAME AND USER NAME,
;       THE FULL FILE SPECIFICATION, AND THE PAGE NUMBER.
;       THE HEADER LINE IS UNDERLINED BY A ROW OF DASHES
;       PRINTED ON A SECOND HEADER LINE.
;--

;
; System definitions
;
        $PSMDEF                                 ; Symbiont definitions
        $SMBDEF                                 ; Message item definitions
        $DSCDEF                                 ; Descriptor definitions

;
; Define argument offsets for user supplied services called by symbiont
;
        CONTEXT         = 04                    ; symbiont context
        WORK_AREA       = 08                    ; user context
        FUNC            = 12                    ; function code
        FUNC_DESC       = 16                    ; function dependent descriptor
        FUNC_ARG        = 20                    ; function dependent argument

;
; Macro to create dynamic descriptors
;
        .MACRO  D_DESC
                .WORD   0                       ; DSC$W_LENGTH = 0
                .BYTE   DSC$K_DTYPE_T           ; DSC$B_DTYPE = STRING
                .BYTE   DSC$K_CLASS_D           ; DSC$B_CLASS = DYNAMIC
                .LONG   0                       ; DSC$A_POINTER = 0
        .ENDM

;
; Storage for page header information
;
        FILE:           D_DESC                  ; file name descriptor
        USER:           D_DESC                  ; user name descriptor
        ACCOUNT:        D_DESC                  ; account name descriptor

        PAGE:           .LONG   0               ; page number
        LINE:           .LONG   0               ; line number

;
; FAO control string and work buffer.  Header format:
;       "[account,name] filename ........ Page 9999"
;
        FAO_CTRL:       .ASCID  /!71<[!AS, !AS] !AS!>Page 9999/
        FAO_CTRL_2:     .ASCID  /!4UL/
        FAO_DESC:       .LONG   80              ; work buffer descriptor
                        .ADDRESS FAO_BUFF
        FAO_BUFF:       .BLKB   80              ; work buffer
```

Example 10–1 Cont'd. on next page

Example 10–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a MACRO Program

```
;
; Own storage for values passed by reference
;
        CODE:           .LONG   0               ; service or item code
        STREAMS:        .LONG   1               ; number of simultaneous streams
        BUFSIZ:         .LONG   2048            ; output buffer size
        LINSIZ:         .WORD   81              ; line size for underlines

;
; Main routine -- invoked at image startup
;
START:  .WORD   0       ; save nothing because this routine uses only R0 and R1


;
; Supply private page header routine
;
        MOVZBL  #PSM$K_PAGE_HEADER,CODE         ; set the service code
        PUSHAL  HEADER                          ; address of modified routine
        PUSHAL  CODE                            ; address of service code
        CALLS   #2,G^PSM$REPLACE                ; replace the routine
        BLBC    R0,10$                          ; exit if any errors

;
; Transfer control to the standard symbiont
;
        PUSHAL  BUFSIZ                          ; address of output buffer size
        PUSHAL  STREAMS                         ; address of number of streams
        CALLS   #2,G^PSM$PRINT                  ; invoke standard symbiont
10$:    RET


;
; Page header routine
;
HEADER: .WORD   0                               ; save nothing

;
; Check function code
;
        CMPL    #PSM$K_START_TASK,@FUNC(AP)     ; new task?
        BEQL    20$                             ; branch if so
        CMPL    #PSM$K_READ,@FUNC(AP)           ; READ function?
        BNEQ    15$
        BEQL    50$                             ; branch if so
15$:    CMPL    #PSM$K_OPEN, @FUNC(AP)          ; OPEN function?
        BNEQ    16$
        BEQL    66$                             ; branch if so
16$:    MOVL    #PSM$_FUNNOTSUP,R0              ; unsupported function
        RET                                     ; return to symbiont
;
; Starting a new file
;
20$:
        CLRL    PAGE                            ; reset the page number
        MOVZBL  #2,LINE                         ; and the line number
```

Example 10–1 Cont'd. on next page

Example 10–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a MACRO Program

```
;
; Get the account name
;
        MOVZBL  #SMBMSG$K_ACCOUNT_NAME,CODE     ; set item code
        PUSHAL  ACCOUNT                         ; address of descriptor
        PUSHAL  CODE                            ; address of item code
        PUSHAL  @CONTEXT(AP)                    ; address of symbiont ctx value
        CALLS   #3,G^PSM$READ_ITEM_DX           ; read it
        BLBC    R0,40$                          ; branch if any errors

;
; Get the file name
;
        MOVZBL  #SMBMSG$K_FILE_SPECIFICATION,CODE       ; set item code
        PUSHAL  FILE                            ; address of descriptor
        PUSHAL  CODE                            ; address of item code
        PUSHAL  @CONTEXT(AP)                    ; address of symbiont ctx value
        CALLS   #3,G^PSM$READ_ITEM_DX           ; read it
        BLBC    R0,40$                          ; branch if any errors

;
; Get the user name
;
        MOVZBL  #SMBMSG$K_USER_NAME,CODE        ; set item code
        PUSHAL  USER                            ; address of descriptor
        PUSHAL  CODE                            ; address of item code
        PUSHAL  @CONTEXT(AP)                    ; address of symbiont ctx value
        CALLS   #3,G^PSM$READ_ITEM_DX           ; read it
        BLBC    R0,40$                          ; branch if any errors

;
; Set up the static header information that is constant for the task
;
        $FAO_S  CTRSTR  = FAO_CTRL, -           ; FAO control string desc
                OUTBUF  = FAO_DESC, -           ; output buffer descriptor
                P1      = #ACCOUNT, -           ; account name descriptor
                P2      = #USER, -              ; user name descriptor
                P3      = #FILE                 ; file name descriptor
40$:    RET                                     ; return success or any error
;
; Read a page header
;
50$:
        DECL    LINE                            ; decrement the line number
        BEQL    60$                             ; branch if second read
        BLSS    70$                             ; branch if third read

;
; Insert the page number into the header
;
        INCL    PAGE                            ; increment the page number
        MOVAB   FAO_BUFF+76,FAO_DESC+4          ; point to page number buffer
        $FAO_S  CTRSTR  = FAO_CTRL_2, -         ; FAO control string desc
                OUTBUF  = FAO_DESC, -           ; output buffer descriptor
                P1      = PAGE                  ; page number
        MOVAB   FAO_BUFF,FAO_DESC+4             ; point to work buffer
        BLBC    R0,55$                          ; return if error
```

Example 10–1 Cont'd. on next page

Example 10–1 (Cont.) Using PSM Routines to Supply a Page Header Routine in a MACRO Program

```
;
; Copy the line to the symbiont's buffer
;
        PUSHAB  FAO_DESC                        ; work buffer descriptor
        PUSHL   FUNC_DESC(AP)                   ; symbiont descriptor
        CALLS   #2,G^STR$COPY_DX                ; copy to symbiont buffer
55$:    RET                                     ; return success or any error

;
; Second line -- underline header
;
60$:
        PUSHL   FUNC_DESC(AP)                   ; symbiont descriptor
        PUSHAL  LINSIZ                          ; number of bytes to reserve
        CALLS   #2,G^STR$GET1_DX                ; reserve the space
        BLBC    R0,67$                          ; exit if error
        MOVL    FUNC_DESC(AP),R1                ; get address of descriptor
        MOVL    4(R1),R1                        ; get address of buffer
        MOVAB   80(R1),R0                       ; set up transfer limit
65$:    MOVB    #^A/-/,(R1)+                    ; fill with dashes
        CMPL    R0,R1                           ; reached limit?
        BGTRU   65$                             ; branch if not
        MOVB    #10,(R1)+                       ; extra line feed
66$:    MOVZBL  #SS$_NORMAL,R0                  ; set success
67$:    RET                                     ; return

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

        .END    START
```

10.5 PSM Routines

The following pages describe the individual PSM routines.

PSM$PRINT Invoke VMS-Supplied Print Symbiont

The PSM$PRINT routine invokes the VMS-supplied print symbiont.

PSM$PRINT must be called exactly once after all user service routines have been specified using PSM$REPLACE.

FORMAT

PSM$PRINT *[streams] [,bufsiz] [,worksiz]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

streams

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

Maximum number of streams that the print symbiont is to support. The **streams** argument is the address of a longword containing this number, which must be in the range *1* to *16*. If you do not specify **streams**, a default value of *1* is used. Thus, by default, a user-modified print symbiont supports one stream, which is to say that it is a single-threaded symbiont.

A stream (or thread) is a logical link between a print execution queue and a printing device. When a symbiont process can accept simultaneous links to more than one queue, that is, when it can service multiple queues simultaneously, the symbiont is said to be multithreaded.

bufsiz

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

Maximum buffer size in bytes that the print symbiont is to use for output operations. The **bufsiz** argument is the address of a longword containing the specified number of bytes.

The print symbiont actually uses a buffer size that is the smaller of (1) the value specified by **bufsiz** or (2) the SYSGEN parameter MAXBUF. If you do not specify **bufsiz**, then the print symbiont uses the value of MAXBUF.

The print symbiont uses this size limit only for output operations. Output operations involve the placing of processed or formatted pages into a buffer that will be passed to the output routine.

The print symbiont uses the value specified by **bufsiz** only as an upper limit; most buffers that it writes will be smaller than this value.

worksiz

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

Size in bytes of a work area to be allocated for the use of user routines. The **worksiz** argument is the address of a longword containing this size in bytes. If you do not specify **worksiz**, no work area is allocated.

A separate area of the specified size is allocated for each active symbiont stream.

DESCRIPTION

The PSM$PRINT routine must be called exactly once after all user routines have been specified to the print symbiont. Each user routine is specified to the symbiont in a call to the PSM$REPLACE routine.

The PSM$PRINT routine allows you to specify whether the print symbiont is to be single-threaded or multithreaded, and if multithreaded, how many streams or threads it can have. In addition, this routine allows you to control the maximum size of the output buffer.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.

This routine also returns any condition values returned by the $SETPRV, $GETSYI, $PURGWS, and $DCLAST system services, as well as any condition values returned by the SMB$INITIALIZE routine documented in Chapter 11.

PSM$READ_ITEM_DX Obtain Value of Message Items

The PSM$READ_ITEM_DX routine obtains the value of message items that are sent by the job controller and stored by the VMS symbiont.

FORMAT

PSM$READ_ITEM_DX *request_id ,item ,buffer*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

request_id

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

Request identifier supplied by the symbiont to the user routine currently calling PSM$READ_ITEM_DX. The symbiont always supplies a request identifier when it calls a user routine with a service request. The **request_id** argument is the address of a longword containing this request identifier value.

Your user routine must copy the request identifier value that the symbiont supplies (in the **request_id** argument) when it calls your user routine. Then, when your user routine calls PSM$READ_ITEM_DX, it must supply (in the **request_id** argument) the address of the request identifier value that it copied.

item

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

Item code that identifies the message item that PSM$READ_ITEM_DX is to return. The **item** argument is the address of a longword that specifies the item's code.

For a complete list and description of each item code, refer to the documentation of the **item** argument in the SMB$READ_MESSAGE_ITEM routine in Chapter 11.

buffer

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

Buffer into which PSM$READ_ITEM_DX returns the specified informational item. The **buffer** argument is the address of a descriptor pointing to this buffer.

The PSM$READ_ITEM_DX routine returns the specified informational item by copying that item to the buffer using one of the STR$COPY_xx routines documented in the *VMS Run-Time Library Routines Volume.*

DESCRIPTION

The PSM$READ_ITEM_DX routine obtains the value of message items that are sent by the job controller and stored by the VMS symbiont. You use PSM$READ_ITEM_DX to obtain information about the task currently being processed, for example, the name of the file being printed (SMBMSG$K_FILE_SPECIFICATION) or the name of the user who submitted the job (SMBMSG$K_USER_NAME).

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
PSM$_INVITMCOD	Invalid item code specified in the **item** argument.

This routine also returns any condition values returned by any of the STR$COPY_xx routines documented in the *VMS Run-Time Library Routines Volume.*

PSM$REPLACE Declare User Service Routine

The PSM$REPLACE routine substitutes a user service routine for a symbiont routine or adds a user service routine to the set of symbiont routines.

You must call PSM$REPLACE once for each routine that you replace or add.

FORMAT

PSM$REPLACE *code ,routine*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

ARGUMENTS

code

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

Routine code that identifies the symbiont routine to be replaced by a user service routine. The **code** argument is the address of a longword containing the routine code.

Some routine codes identify routines that are supplied with the VMS symbiont; when you specify such a routine code, you replace the symbiont-supplied routine with your service routine.

Two routine codes identify routines that are not supplied with the VMS symbiont; when you specify such a routine code, your service routine is added to the set of symbiont routines.

Table 10–1 lists each routine code in the order in which it is called within the symbiont execution stream; this table also specifies whether a routine code identifies an input, formatting, or output routine, and whether the routine is supplied with the VMS symbiont.

The routine codes are defined by the $PSMDEF macro. The following pages list each routine code in alphabetical order; the description of each code includes the following information about its corresponding routine:

- Whether the routine is supplied by the VMS symbiont
- Whether the routine is an input, formatting, or output routine

- Under what conditions the routine is called
- What task the routine performs

Routine Codes

PSM$K_FILE_BURST

This code identifies a symbiont-supplied input routine; it is called whenever a file burst page is requested. This routine obtains information about the job, formats the file burst page, and returns the contents of the page to the input buffer. A file burst page follows a file flag page and precedes the contents of the file.

PSM$K_FILE_ERRORS

This code identifies a symbiont-supplied input routine; it is called when errors have occurred during the job. This routine places the error message text in the input buffer.

PSM$K_FILE_FLAG

This code identifies a symbiont-supplied input routine; it is called whenever a file flag page is requested. This routine obtains information about the job, formats the file flag page, and returns the contents of the page to the input buffer. A flag page follows the job burst page (if any) and precedes the file burst page (if any). It contains such information as the file specification of the file and the name of the user issuing the print request.

PSM$K_FILE_INFORMATION

This code identifies a symbiont-supplied input routine; it is called when the file information item has been specified by the job controller. This routine expands the file information item to text and returns it to the input buffer.

PSM$K_FILE_SETUP

This code identifies a symbiont-supplied input routine; it is always called. This routine queues any specified file-setup modules for insertion in the input stream when the PSM$K_FILE_SETUP routine closes.

PSM$K_FILE_SETUP_2

This code identifies a symbiont-supplied input routine; it is always called. This routine returns a formfeed to ensure that printing of the file begins at the top of the page. This routine is called just before the main input routine.

PSM$K_FILE_TRAILER

This code identifies a symbiont-supplied input routine; it is called whenever a file trailer page is requested. This routine obtains information about the job, formats the file trailer page, and returns the contents of the page to the input buffer. A trailer page follows the last page of the file contents.

PSM$K_MAIN_FORMAT

This code identifies the symbiont-supplied formatting routine; it is always called. This routine performs numerous formatting functions. You cannot replace this routine.

PSM$K_FORM_SETUP

This code identifies a symbiont-supplied input routine; it is always called. This routine queues any specified form-setup modules for insertion in the input stream when the PSM$K_FORM_SETUP routine closes.

PSM$K_INPUT_FILTER
This code identifies a format routine that is not supplied by the VMS symbiont. If the routine is supplied by the user, it is always called immediately prior to the symbiont-supplied formatting routine (routine code PSM$K_MAIN_FORMAT). An input-filter service routine is useful for modifying input data records and their carriage control before they are formatted by the symbiont.

PSM$K_JOB_BURST
This code identifies a symbiont-supplied input routine; it is called whenever a job burst page is requested. This routine obtains information about the job, formats the job burst page, and returns the contents of the page to the input buffer. A job burst page follows the job flag page and precedes the file flag page (if any) of the first file in the job. It is similar to a file burst page except that it appears only once per job and only at the beginning of the job.

PSM$K_JOB_COMPLETION
This code identifies a symbiont-supplied input routine; it is always called. This routine returns a formfeed, which causes any output buffered by the device to be printed.

PSM$K_JOB_FLAG
This code identifies a symbiont-supplied input routine; it is called whenever a job flag page is requested. This routine obtains information about the job, formats the job flag page, and returns the contents of the page to the input buffer. A job flag page is similar to a file flag page except that it appears only once per job, preceding the job burst page (if any).

PSM$K_JOB_RESET
This code identifies a symbiont-supplied input routine; it is always called. This routine queues any specified job-reset modules for insertion in the input stream when the PSM$K_JOB_RESET routine closes.

PSM$K_JOB_SETUP
This code identifies a symbiont-supplied input routine; it is always called. This routine checks to see if this is the first job to be printed on the device, and if so, it issues a formfeed and then performs a job reset. See the description of the PSM$K_JOB_RESET routine for information about job reset.

PSM$K_JOB_TRAILER
This code identifies a symbiont-supplied input routine; it is called whenever a job trailer page is requested. This routine obtains information about the job, formats the job trailer page, and returns the contents of the page to the input buffer. A job trailer page is similar to a file trailer page except that it appears only once per job, as the last page in the job.

PSM$K_MAIN_INPUT
This code identifies a symbiont-supplied input routine; it is always called. This routine opens the file to be printed, returns input records to the input buffer, and closes the file.

PSM$K_LIBRARY_INPUT
This code identifies a symbiont-supplied input routine; it is called when an input routine closes and when modules have been requested for insertion in the input stream. This routine returns the contents of the specified modules, one record per call. You cannot replace this routine.

PSM$K_OUTPUT_FILTER
This code identifies a formatting routine that is not supplied by the VMS symbiont. If the routine is supplied by the user, it is always called. This routine executes prior to the symbiont output routine (routine code PSM$K_OUTPUT). An output-filter service routine is useful for modifying output data buffers before they are passed to the output routine.

At the point where the output-filter routine executes within the symbiont execution stream, the input data is no longer in record format; instead, the data exists as a stream of characters. The carriage control, for example, is embedded in the data stream. Thus, the output buffer may contain what was once a complete record, part of a record, or several records.

PSM$K_PAGE_HEADER
This code identifies a symbiont-supplied input routine; it is called once at the beginning of each page if page headers are requested. This routine returns to the input buffer one or more lines containing information about the file being printed and the current page number. This routine is called only while the main input routine is open.

PSM$K_PAGE_SETUP
This code identifies a symbiont-supplied routine; it is called at the beginning of each page if page-setup modules were specified. This routine queues any specified page-setup modules for insertion in the input stream when the PSM$K_PAGE_SETUP routine closes. This routine is called only while the main input routine is open.

PSM$K_OUTPUT
This code identifies the symbiont-supplied output routine; it is always called. This routine writes the contents of the output buffer to the printing device, but it also performs many other functions.

routine

VMS usage:	**procedure**
type:	**procedure entry mask**
access:	**read only**
mechanism:	**by reference**

User service routine that is to replace a symbiont routine or to be included. The **routine** argument is the address of the user routine entry point.

DESCRIPTION

The PSM$REPLACE routine must be called each time a user service routine replaces a symbiont routine or is added to a set of symbiont routines.

The code argument specifies the symbiont routine to be replaced. The routine codes that can be specified in the **code** argument are of two types: those that identify existing print symbiont routines and those that do not. All the routine codes are similar, however, in the sense that each supplies a location within the print symbiont execution stream where your routine can execute.

By selecting a routine code that identifies an existing symbiont routine, you effectively disable that symbiont routine. The service routine that you specify may or may not perform the function that the disabled symbiont routine performs. If it does not, the net effect of the replacement is to eliminate that function from the list of functions performed by the print symbiont. Exactly what your service routine does is up to you.

By selecting a routine code that does not identify an existing symbiont routine (those that identify the input-filter and output-filter routines), your service routine has a chance to execute at the location signified by the routine code. Because the service routine you specify to execute at this location does not replace another symbiont routine, your service routine is an addition to the set of symbiont routines.

As mentioned, each routine code identifies a location in the symbiont execution stream, whether or not it identifies a symbiont routine. Table 10–1 lists each routine code in the order in which the location it identifies is reached within the symbiont execution stream.

CONDITION VALUE RETURNED

SS$_NORMAL	Normal successful completion.

PSM$REPORT Report Completion Status

The PSM$REPORT routine reports to the print symbiont the completion status of an asynchronous operation initiated by a user routine.

Such a user routine must return the completion status PSM$_PENDING. PSM$REPORT must be called exactly once for each time a user routine returns the status PSM$_PENDING.

FORMAT

PSM$REPORT *request_id [,status]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

ARGUMENTS

request_id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Request identifier supplied by the symbiont to the user routine at the time the symbiont called the user routine with the service request. The user routine must return the completion status PSM$_PENDING on the call for this service request. The **request_id** argument is the address of a longword containing the request identifier value.

The symbiont calls the user routine with a request code that specifies the function that the symbiont expects the user routine to perform. In the call, the symbiont also supplies a request identifier, which serves to identify the request. If the user routine initiates an asynchronous operation, a mechanism is required for notifying the symbiont that the asynchronous operation has completed and for providing the completion status of the operation.

The PSM$REPORT routine conveys the above two pieces of information. In addition, PSM$REPORT returns to the symbiont (in the **request_id** argument) the same request identifier value as that supplied by the symbiont to the user routine that initiated the operation. In this way, the symbiont synchronizes the completion status of an asynchronous operation with that invocation of the user routine that initiated the operation.

Any user routine that initiates an asynchronous operation must, therefore, copy the request identifier value that the symbiont supplies (in the **request_id** argument) when it calls the user routine. The user routine will later need to supply this value to PSM$REPORT.

In addition, when the user routine returns, which it does before the asynchronous operation has completed, the user routine must return the status PSM$_PENDING.

status

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Completion status of the asynchronous operation that has completed. The **status** argument is the address of a longword containing this completion status. The **status** argument is optional; if it is not specified, the symbiont assumes the completion status SS$_NORMAL.

The user routine that initiates the asynchronous operation must test for the completion of the operation and must supply the operation's completion status as the **status** argument to the PSM$REPORT routine. The DESCRIPTION section describes this procedure in greater detail.

If the completion status specified by **status** has the low bit clear, the symbiont aborts the task.

DESCRIPTION

An asynchronous operation is an operation that, once initiated, executes "off to the side" and need not be completed before other operations can begin to execute. Asynchronous operations are common in symbiont applications because a symbiont, if it is multithreaded, must handle concurrent I/O operations.

One example of a user routine that performs an asynchronous operation is an output routine that calls the $QIO system service to write a record to the printing device. When the user output routine completes execution, the I/O request queued by $QIO may not have completed. In order to synchronize this I/O request, that is, to associate the I/O request with the service request that initiated it, you should use the following mechanism:

1. In making the call to $QIO, specify the **astadr** and **iosb** arguments. The **astadr** argument specifies an AST routine to execute when the queued output request has completed, and the **iosb** argument specifies an I/O status block to receive the completion status of the I/O operation. Item 3 describes some functions that your AST routine will need to do.
2. Have the user output routine return the status PSM$_PENDING.
3. Write the AST routine to perform the following functions:
   a. Copy the completion status word from the I/O status block to a longword location that you will specify as the **status** argument in the call to PSM$REPORT.
   b. Call PSM$REPORT. Specify as the **request_id** argument the request identifier that was supplied by the print symbiont in the original call to the user output routine.

CONDITION VALUE RETURNED

SS$_NORMAL	Normal successful completion.

USER-FORMAT-ROUTINE Invoke User-Written Format Routine

The user-written USER-FORMAT-ROUTINE performs format operations. The symbiont's control logic routine calls your format routine at one of two possible points within the symbiont's execution stream. You select this point by specifying one of two routine codes when you call the PSM$REPLACE routine.

A user format routine may be an input filter routine (routine code PSM$K_INPUT_FILTER) or an output filter routine (routine code PSM$K_OUTPUT_FILTER). The main format routine (routine code PSM$K_MAIN_FORMAT) may not be replaced.

A user format routine must use the call interface described here.

FORMAT

USER-FORMAT-ROUTINE *request_id ,work_area ,func ,func_desc_1 ,func_arg_1 ,func_desc_2 ,func_arg_2*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

request_id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Request identifier supplied by the symbiont when it calls your format routine. The **request_id** argument is the address of a longword containing this request identifier value.

work_area

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

Work area supplied by the symbiont for the use of your format routine. The symbiont supplies the address of this area when it calls your routine. The **work_area** argument is a longword containing the address of the work area.

The work area is a section of memory that your format routine can use for buffering and other internal operations.

The size of the work area allocated is specified by the **work_size** argument in the PSM$PRINT routine. If you do not specify **work_size** in the call to PSM$PRINT, no work area is allocated.

In a multithreaded symbiont, a separate work area is allocated for each thread. This work area is shared by all user routines. The work area is initialized to zero when the symbiont is first started.

func

VMS usage: **function_code**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Function code specifying the service that the symbiont expects your format routine to perform. The **func** argument is the address of a longword into which the symbiont writes this function code.

The function code specifies the reason the symbiont is calling your format routine or, in other words, the service that the symbiont expects your routine to perform at this time.

The PSM$K_FORMAT function code is the only one to which your format routine must respond. When the symbiont calls your format routine with this function code, your routine must move a record from the input buffer to the output buffer.

The symbiont may call your format routine with other function codes. Your routine should return the status PSM$_FUNNOTSUP (function not supported) when it is called with any of the following function codes or with any undocumented function code:

PSM$K_START_STREAM	PSM$K_STOP_STREAM
PSM$K_START_TASK	PSM$K_PAUSE_TASK
PSM$K_RESUME_TASK	PSM$K_STOP_TASK
PSM$K_RESET_STREAM	

These function codes correspond to message items, which are discussed in more detail in Section 9.3.6, sent by the job controller to the symbiont.

Other function codes correspond to internal symbiont mechanisms that are not part of the public interface to the print symbiont.

Your format routine should return the status PSM$_FUNNOTSUP or SS$_NORMAL when it is called with a message function code or with a private function code.

func_desc_1

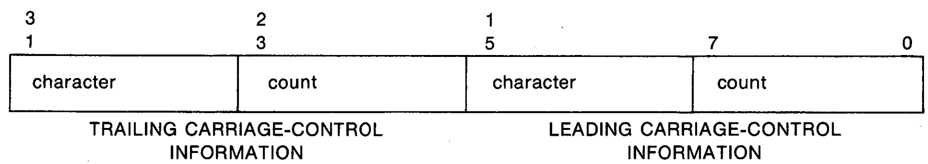
VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Descriptor supplying an input record to be processed by the format routine. The **func_desc_1** argument is the address of a string descriptor. By using this argument, the symbiont supplies the input record that your format routine is to process. Because this descriptor may be of any valid string type, your format routine should use the Run-Time Library string routines to analyze this descriptor and to manipulate the input record.

func_arg_1

VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Carriage control for the input record supplied by **func_desc_1**. The **func_arg_1** argument is the address of a 4-byte vector that specifies the carriage control for the input record. The following diagram depicts the format of this 4-byte vector.



ZK-2009-84

Bytes 0 and 1 describe the leading carriage control to apply to the input data record; bytes 2 and 3 describe the trailing carriage control.

Byte 0 is a number specifying the number of times the carriage control specifier in byte 1 is to be repeated preceding the input data record. Byte 2 is a number specifying the number of times the carriage control specifier in byte 3 is to be repeated following the input data record.

For values of the carriage control specifier from 1 to 255, the specifier is the ASCII character to be used as carriage control. Value *0* represents the ASCII "newline" sequence. Newline consists of a carriage return followed by a linefeed.

The **func_arg_1** argument is not used if your format routine is an output filter routine (routine code PSM$K_OUTPUT_FILTER). See the DESCRIPTION section for more information.

func_desc_2

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

Descriptor of a buffer to which your format routine writes the formatted output record. The **func_desc_2** argument is the address of a string descriptor.

Your format routine must return the formatted data record by using the **func_desc_2** argument.

Your format routine should use the Run-Time Library string routines to write into the buffer specified by this descriptor.

func_arg_2

VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Carriage control for the output record returned in **func_desc_2**. The **func_arg_2** argument is the address of a 4-byte vector that specifies the carriage control for the output record. See the description of **func_arg_1** for the contents and format of this 4-byte vector.

If you do not process the carriage-control information supplied in **func_arg_1** , then you should copy that value into **func_arg_2**. Otherwise, the carriage-control information will be lost.

The **func_arg_2** argument is not used if your format routine is an output filter routine (routine code PSM$K_OUTPUT_FILTER). See the DESCRIPTION section for more information.

DESCRIPTION

When used, the **func_arg_1** argument describes carriage-control information for the input data record, and the **func_arg_2** argument describes carriage-control information for the output data record.

The input data record is passed to the format routine (input filter or output filter) for processing, and the output data record is returned by the format routine (input filter or output filter).

One of the tasks performed by the main format routine (routine code PSM$K_MAIN_FORMAT) is that of embedding the carriage-control information (specified by **func_arg_1**) into the data record (specified by **func_desc_1**). Thus, the output data (specified by **func_desc_2**) contains embedded carriage control and is thus no longer in record format; it is, therefore, properly referred to as an output data stream rather than an output data record.

Similarly, the output filter routine (routine code PSM$K_OUTPUT_FILTER), which executes after the main format routine, uses neither the **func_arg_1** nor **func_arg_2** argument; the data it receives (via **func_desc_1**) and the data it returns (via **func_desc_2**) are data streams, not data records.

However, the input filter routine (routine code PSM$K_INPUT_FILTER), which executes before the main format routine, uses both **func_arg_1** and **func_arg_2**. This is so because the main format routine has not yet executed, and so the carriage control information has not yet been embedded in the data record.

CONDITION VALUES RETURNED

SS$_NORMAL	Successful completion. The user format routine has completed the function that the symbiont requested.
PSM$_FUNNOTSUP	Function not supported. The user format routine does not support or does not recognize the function code supplied by the symbiont. To ensure future compatibility, your format routine should return this status for any unrecognized status codes.

This routine also returns any error condition values that you have coded your format routine to return. Refer to Section 10.3.1 for more information about error condition values.

USER-INPUT-ROUTINE Invoke User-Written Input Routine

The user-written USER-INPUT-ROUTINE performs input operations. The symbiont calls your routine at a specified point in its execution stream; you specify this point using the PSM$REPLACE routine.

FORMAT

USER-INPUT-ROUTINE *request_id ,work_area ,func ,funcdesc ,funcarg*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

request_id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Request identifier value supplied by the symbiont when it calls your input routine. The **request_id** argument is the address of a longword containing this request identifier value.

If your input routine initiates an asynchronous operation (for example, a call to the $QIO system service), your input routine must copy the request identifier value specified by **request_id** because this value must later be passed to the PSM$REPORT routine. See the description of the PSM$REPORT routine for more information.

work_area

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

Work area supplied by the symbiont for the use of your input routine. The symbiont supplies the address of this area when it calls your routine. The **work_area** argument is a longword into which the symbiont writes the address of the work area. The work area is a section of memory that your input routine can use for buffering and for other internal operations.

The size of the work area allocated is specified by the **work_size** argument in the PSM$PRINT routine. If you do not specify **work_size** in the call to PSM$PRINT, no work area is allocated.

In a multithreaded symbiont, a separate work area is allocated for each thread. This work area is shared by all user routines. The work area is initialized to zero when the symbiont is first started.

func

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

Function code supplied by the symbiont when it calls your input routine. The **func** argument is the address of a longword containing this code.

The function code specifies the reason the symbiont is calling your input routine or, in other words, the function that the symbiont expects your routine to perform at this time.

Most function codes require or allow additional information to be passed in the call by means of the **funcdesc** and **funcarg** arguments. The description of each input function code, therefore, includes a description of how these two arguments are used with that function code.

Following is a list of all the function codes that the symbiont can specify when it calls your input routine (function codes applicable only to format and output routines are explained in the descriptions of the USER-FORMAT-ROUTINE and USER-OUTPUT-ROUTINE, respectively); all function codes are defined by the $PSMDEF macro.

Function Codes for Input Routines

PSM$K_CLOSE

When the symbiont calls your routine with this function code, your routine must terminate processing by releasing any resources it may have allocated.

The symbiont calls your routine with PSM$K_CLOSE when (1) your routine returns from a PSM$K_READ function call with the status PSM$_EOF (end of input) or with any error condition, or (2) the symbiont receives a task-abortion request from the job controller.

In any event, the symbiont always calls your input routine with PSM$K_CLOSE if your routine returns successfully from a PSM$K_OPEN function call. This guaranteed behavior ensures that any resources your routine may have allocated on the OPEN will be released on the CLOSE.

PSM$K_GET_KEY

Typically, the use of both the PSM$K_GET_KEY and PSMK$K_POSITION_TO_KEY function codes is appropriate only for a main input routine (routine code PSM$K_MAIN_INPUT).

When the symbiont calls your routine with this function code, your routine may do one of two things: (1) return PSM$_FUNNOTSUP (function not supported) or (2) return an input marker string to the symbiont.

If your routine returns PSM$_FUNNOTSUP to this function code, then your routine must also return PSM$_FUNNOTSUP if the symbiont subsequently calls your routine with the PSM$K_POSITION_TO_KEY function code. By returning PSM$_FUNNOTSUP, your routine is choosing not to respond to the symbiont request.

If your routine chooses to respond to the PSM$K_GET_KEY function code, your routine must return an input marker string to the symbiont; this input marker string identifies the input record that your input routine most recently returned to the symbiont. Subsequently, when the symbiont calls your input routine with the PSM$K_POSITION_TO_KEY function code, the symbiont passes your input routine one of the input marker strings that your input routine has returned on a previous PSM$K_GET_KEY function call. Using this marker string, your input routine must position itself so that, on the next PSM$K_READ call from the symbiont, your input routine will return (or reread) the input record identified by the marker string.

Coding your input routine to respond to PSM$K_GET_KEY and PSM$K_POSITION_TO_KEY allows the modified symbiont to perform the file-positioning functions specified by the DCL commands START/QUEUE /FORWARD, START/QUEUE/ALIGN, START/QUEUE/TOP_OF_FILE, START/QUEUE/SEARCH, and START/QUEUE/BACKWARD. These file positioning functions also depend on the job controller's checkpointing capability for print jobs.

Note that your input routine might be called with a marker string that was originally returned in a different process context from the current one. This can occur because marker strings are sometimes stored in the queue-data file across system shutdowns or different invocations of your symbiont.

The **funcdesc** argument specifies the address of a string descriptor. Your routine must return the marker string by way of this argument. DIGITAL recommends that you use one of the Run-Time Library string routines to copy the marker string to the descriptor.

The symbiont periodically calls your input routine with the PSM$K_GET_KEY function code when the symbiont wants to save a marker to a particular input record.

PSM$K_OPEN

When the symbiont calls your routine with this function code, your routine should prepare for input operations by performing such tasks as allocating necessary resources, initializing storage areas, opening an input file, and so on. Typically, the next time the symbiont calls your input routine, the symbiont will specify the PSM$K_READ function code. Note, however, that under some circumstances the symbiont might follow an OPEN call immediately with a CLOSE call.

The **funcdesc** argument points to the name of the file to be opened. Your routine can use this file specification or the file identifcation to open the file.

The **funcarg** argument specifies the address of a longword. Your input routine must return, in this longword, the carriage control type that is to be applied to the input records that your input routine will provide.

The symbiont formatting routine requires this information to determine where to apply leading and trailing carriage control characters to the input records that your input routine will provide.

The $PSMDEF macro defines the following four carriage control types.

Carriage Type	Description
PSM$K_CC_IMPLIED	Implied carriage control. For this type, the symbiont inserts a leading linefeed (LF) and trailing carriage return (CR) in each input record. This is the default carriage control type; it is used if your routine does not supply a carriage control type in the **funcarg** argument in response to the PSM$K_OPEN function call.
PSM$K_CC_FORTRAN	FORTRAN carriage control. For this type, the symbiont extracts the first byte of each input record and interprets the byte as a FORTRAN carriage control character, which it then applies to the input record.
PSM$K_CC_PRINT	PRN carriage control. For this type, the symbiont generates carriage control from a 2-byte record header that your input routine supplies, with each READ call, in the **funcarg** argument. The **funcarg** argument specifies the address of a longword to receive this two-byte header record, which appears only in PRN print files.
PSM$K_CC_INTERNAL	Embedded carriage control. For this type, the symbiont supplies no carriage control to input records. Carriage control is assumed to be embedded in the input records.

PSM$K_POSITION_TO_KEY

When the symbiont calls your routine with this function code, your routine must locate the point in the input stream designated by the marker string that your routine returned to the symbiont on the PSM$K_GET_KEY function call.

The next time the symbiont calls your routine, the symbiont specifies the PSM$K_READ function call, expecting to receive the next sequential input record. After rereading this record, subsequent READ calls proceed from this new position of the file. This is not a one-time rereading of a single record but a repositioning of the file. The symbiont calls your routine with this function code when the job controller receives a request to resume printing at a particular page.

Refer to the description of the PSM$K_GET_KEY for more information.

PSM$K_READ

When the symbiont calls your routine with this function code, your routine must return an input record. The symbiont repeatedly calls your input routine with the PSM$K_READ function code until (1) your routine indicates end of input by returning the status PSM$_EOF, (2) your routine or another routine returns an error status, or (3) the symbiont receives an asynchronous task-abortion request from the job controller.

The **funcdesc** argument specifies the address of a string descriptor. Your routine must return the input record by using this argument. DIGITAL recommends that you use one of the Run-Time Library string routines to copy the input record to the descriptor.

The **funcarg** argument specifies the address of a longword. This argument is used only if the carriage control type returned by your input routine on the PSM$K_OPEN function call was PSM$K_CC_PRINT. In this case, your input routine must supply, in the **funcarg** argument, the 2-byte record header found at the beginning of each input record.

PSM$K_REWIND

When the symbiont calls your routine with this function code, your routine must do one of two things: (1) return PSM$_FUNNOTSUP (function not supported) or (2) locate the point in the input stream designated as the beginning of the file.

If your routine returns PSM$_FUNNOTSUP to this function code, then the symbiont subsequently calls your input routine with a PSM$K_CLOSE function call followed by a PSM$K_OPEN function call. By returning PSM$K_FUNNOTSUP, your routine is choosing not to support the repositioning of the input service to the beginning of the file. The symbiont, therefore, performs the desired function by closing and then reopening the input routine.

You cannot use the **funcdesc** and the **funcarg** arguments with this function code.

This function call allows the modified symbiont to perform the file-positioning functions specified by the DCL commands START/QUEUE/TOP_OF_FILE, START/QUEUE/FORWARD, START/QUEUE/BACKWARD, START/QUEUE /SEARCH, and START/QUEUE/ALIGN. This is a required repositioning of the file.

Other Input Function Codes

The symbiont may call your input routine with other function codes. Your routine *must* return the status PSM$_FUNNOTSUP (function not supported) when it is called with any of the following function codes or with any undocumented function code:

PSM$K_START_STREAM	PSM$K_STOP_STREAM
PSM$K_START_TASK	PSM$K_PAUSE_TASK
PSM$K_RESUME_TASK	PSM$K_STOP_TASK
PSM$K_RESET_STREAM	

These function codes correspond to message items, which are discussed in detail in Section 10.3.5, sent by the job controller to the symbiont.

Other function codes correspond to internal symbiont mechanisms that are not part of the public interface to the print symbiont.

Your input routine should return the status PSM$K_FUNNOTSUP or SS$_NORMAL when it is called with a message function code or with a private function code.

funcdesc

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

Function descriptor supplying information related to the function specified by the **func** argument. The **funcdesc** argument is the address of this descriptor.

The contents of the function descriptor vary for each function. Refer to the description of each function code to determine the contents of the function descriptor. In some cases, the function descriptor is not used at all.

funcarg

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

Function argument supplying information related to the function specified by the **func** argument. The **funcarg** argument is the address of a longword containing this function argument. This argument can be an input or an output argument, depending on the function request, but is usually used as an output argument.

CONDITION VALUES RETURNED

SS$_NORMAL	Successful completion. The user input routine has completed the function that the symbiont requested.
PSM$_FLUSH	Flush output stream. The user input routine can return this status only when called with the PSM$K_READ function code. When this status is returned to the symbiont, the symbiont stops calling the input routine with the PSM$K_READ function code until all outstanding format and output operations have completed.
PSM$_FUNNOTSUP	Function not supported. The user input routine does not support or does not recognize the function code supplied by the symbiont. To ensure future compatibility, your input routine should return this status for any unrecognized status codes.
PSM$_PENDING	Requested function accepted but not completed. Your input routine can return this status only with the PSM$K_READ function call. Further, if your routine returns PSM$_PENDING, your routine must eventually signal completion via the PSM$REPORT routine. Refer to the description of the PSM$REPORT routine for more information about asynchronous operations and the PSM$_PENDING condition value.

This routine also returns any error condition values that you have coded your format routine to return. Refer to Section 10.3.1 for more information about error condition values.

USER-OUTPUT-ROUTINE Invoke User-Written Output Routine

The user-written USER-OUTPUT-ROUTINE performs output operations. You supply a user output routine by calling the PSM$REPLACE routine with the routine code PSM$K_OUTPUT.

FORMAT

USER-OUTPUT-ROUTINE *request_id ,work_area ,func ,funcdesc ,funcarg*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

request_id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Request identifier value supplied by the symbiont when it calls your output routine. The **request_id** argument is the address of a longword containing this value.

If your output routine initiates an asynchronous operation (for example, a call to the Queue I/O Request (SYS$QIO) system service), you must save the **request_id** argument because you will need to store the request identifier value for later use with the PSM$REPORT routine. See the description of the PSM$REPORT routine for more information.

work_area

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

Work area supplied by the symbiont for the use of your format routine. The symbiont supplies the address of this area when it calls your routine. The **work_area** argument is a longword containing the address of the work area. The work area is a section of memory that your format routine can use for buffering and other internal operations.

The size of the work area allocated is specified by the **work_size** argument in the PSM$PRINT routine. If you do not specify **work_size** in the call to PSM$PRINT, no work area is allocated.

In a multithreaded symbiont, a separate work area is allocated for each thread. This work area is shared by all user routines. The work area is initialized to zero when the symbiont is first started.

func

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

Function code supplied by the symbiont when it calls your output routine. The **func** argument is the address of a longword containing this code.

The function code specifies the reason the symbiont is calling your output routine or, in other words, the function that the symbiont expects your routine to perform at this time.

Most function codes require or allow additional information to be passed in the call via the **funcdesc** and **funcarg** arguments. The description of each output function code, therefore, includes a description of how these two arguments are used for that function code.

The following list describes all the function codes that the symbiont may supply when it calls your output routine (function codes applicable only to input and formatting routines are explained in the descriptions of the user input routine and user formatting routine, respectively); all function codes are defined by the $PSMDEF macro.

Function Codes for Output Routines

PSM$K_OPEN
When the symbiont calls your output routine with this function code, your routine should prepare to move data to the device by performing such tasks as allocating the device, assigning a channel to the device, and so on. The next time the symbiont calls your output routine, the symbiont specifies one of the WRITE function codes (PSM$K_WRITE or PSM$K_WRITE_NOFORMAT).

The symbiont calls your output routine with the PSM$K_OPEN function code when the symbiont receives the SMBMSG$K_START_STREAM message from the job controller.

If your output routine returns an error condition value (low bit clear) to the PSM$K_OPEN function call, the job controller stops processing on the stream and reports the error to whomever entered the DCL command START/QUEUE.

The **funcdesc** argument is the address of a descriptor that identifies the name of the device the output routine is to write to. This device name is established by the DCL command INITIALIZE/QUEUE/ON=device-name.

The **funcarg** argument is the address of a longword into which the user output routine returns the device status longword. For the contents of the device status longword, refer to the description of the SMBMSG$K_DEVICE_STATUS item in the SMB$READ_MESSAGE_ITEM routine, which is documented in Chapter 11.

Your output routine sets bits in the device status longword to indicate to the job controller whether the device falls into one of the following categories:

- Can print lowercase letters
- Is a terminal
- Is connected to the CPU by means of a modem (remote)

If your output routine does not set any of these bits in the device status longword, the job controller assumes, by default, that the device is a line printer that prints only uppercase letters.

PSM$K_WRITE

When the symbiont calls your routine with this function code, your routine must write data to the device. The symbiont supplies the data to be written in the **funcdesc** argument. DIGITAL recommends that you use one of the Run-Time Library string routines to access the data in the buffer described by the **funcdesc** argument.

PSM$K_WRITE_NOFORMAT

When the symbiont calls your routine with this function code, your routine must write data to the device and must indicate to the device driver that the data is not to be formatted.

The symbiont calls your routine with this function code when (1) the print request specifies the PASSALL option or (2) data is introduced by the ANSI DCS (device control string) escape sequence.

The symbiont supplies the data to be written in the **funcdesc** argument. DIGITAL recommends that you use one of the Run-Time Library string routines to move the data from the descriptor to the device.

The output routine of the symbiont informs the device driver not to format the data in the following way:

- When the device is a line printer, the symbiont's output routine specifies the IO$_WRITEPBLK function code when it calls the $QIO system service.
- When the device is a terminal, the symbiont's output routine specifies the IO$M_NOFORMAT function modifier when it calls the $QIO system serivce.

PSM$K_CANCEL

When the symbiont calls your routine with this function code, your routine must abort any outstanding asynchronous I/O requests.

The output routine supplied by the symbiont aborts outstanding I/O requests by calling the $QIO system service with the IO$_CANCEL function code.

If your output routine returned the condition value PSM$_PENDING to one or more previous write requests that are still outstanding (that is, PSM$REPORT has not yet been called to report completion), then your output routine must call PSM$REPORT one time for each outstanding write request that is canceled with this call. That is, canceling an asynchronous write request does not relieve the user output routine of the requirement to call PSM$REPORT once for each asynchronous write request.

You cannot use the **funcdesc** and **funcarg** arguments with this function code.

PSM$K_CLOSE

When the symbiont calls your routine with this function code, your output routine must terminate processing and release any resources it allocated (for example, channels assigned to the device).

You cannot use the **funcdesc** and **funcarg** arguments with this function code.

Other Output Function Codes

The symbiont may call your output routine with other function codes. Your routine should return the status PSM$_FUNNOTSUP (function not supported) when it is called with any of the following function codes or with any undocumented function code:

PSM$K_START_STREAM	PSM$K_STOP_STREAM
PSM$K_START_TASK	PSM$K_PAUSE_TASK
PSM$K_RESUME_TASK	PSM$K_STOP_TASK
PSM$K_RESET_STREAM	

These function codes correspond to message items, which are discussed in more detail in Section 11.1.6, sent by the job controller to the symbiont.

Other function codes correspond to internal symbiont mechanisms that are not part of the public interface to the print symbiont.

Your output routine should return the status PSM$K_FUNNOTSUP or SS$_NORMAL when it is called with a message function code or with a private function code.

funcdesc

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

Function descriptor supplying information related to the function specified by the **func** argument. The **funcdesc** argument is the address of this descriptor.

The contents of the function descriptor vary for each function. Refer to the description of each function code to determine the contents of the function descriptor. In some cases, the function descriptor is not used at all.

funcarg

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

Function argument supplying information related to the function specified by the **func** argument. The **funcarg** argument is the address of a longword containing this function argument.

The contents of the function argument vary for each function. Refer to the description of each function code to determine the contents of the function argument. In some cases, the function argument is not used.

CONDITION VALUES RETURNED

SS$_NORMAL	Successful completion. The user output routine has completed the function that the symbiont requested.
PSM$_FUNNOTSUP	Function not supported. The user output routine does not support or does not recognize the function code supplied by the symbiont. To ensure future compatibility, your output routine should return this status for any unrecognized status codes.
PSM$_PENDING	Requested function accepted but not completed. Your output routine can return this status only with PSM$K_WRITE and PSM$K_WRITE_NOFORMAT function calls. Further, if your routine returns PSM$_PENDING, your routine must eventually signal completion by way of the PSM$REPORT routine. Refer to the description of the PSM$REPORT routine for more information about asynchronous write operations and the PSM$_PENDING condition value.

This routine also returns any error condition values that you have coded your output routine to return. Refer to Section 10.3.1 for more information about error condition values.

11 Symbiont/Job Controller Interface (SMB) Routines

11.1 Introduction to SMB Routines

The Symbiont/Job Controller Interface (SMB) routines provide the interface between the job controller and symbiont processes. A user-written symbiont must use these routines to communicate with the VMS job controller.

Always use the SMB interface routines or SYS$SNDJBC system service to communicate with the job controller. You need not and should not attempt to communicate directly with the job controller.

To write your own symbiont, you need to understand how symbionts work and, in particular, how the standard VMS print symbiont behaves.

11.1.1 Types of Symbiont

There are two types of symbiont:

- Device symbiont, either an input symbiont or an output symbiont. An input symbiont is one that transfers data from a slow device to a fast device, for example, from a card reader to a disk. A card-reader symbiont is an input symbiont. An output symbiont is one that transfers data from a fast device to a slow device, for example, from a disk to a printer or terminal. A print symbiont is an output symbiont.
- Server symbiont, a symbiont that processes or transfers data but is not associated with a particular device; one example is a symbiont that transfers files across a network.

The VMS operating system does not supply any server symbionts.

11.1.2 Symbionts Supplied with the VMS Operating System

The VMS operating system supplies two symbionts:

- SYS$SYSTEM:PRTSMB.EXE (PRTSMB for short), an output symbiont for use with printers and printing terminals.

  PRTSMB performs such functions as inserting flag, burst, and trailer pages into the output stream; reading and formatting input files; and writing formatted pages to the printing device.

  You can modify PRTSMB using the Print Symbiont Modification (PSM) routines.
- SYS$SYSTEM:INPSMB.EXE (INPSMB for short) an input symbiont for use with card readers.

  This symbiont handles the transferring of data from a card reader to a disk file. You cannot modify INPSMB, nor can you write an input symbiont using the SMB routines.

11.1.3 Symbiont Behavior in the VMS Environment

In the VMS environment, a symbiont is a process under the control of the VMS job controller that transfers or processes data.

Figure 11–1 depicts the VMS components that take part in the handling of user requests that involve symbionts. This figure shows two symbionts: (1) the print symbiont supplied by the VMS operating system, PRTSMB, and (2) a user-written symbiont, GRAPHICS.EXE, which services a graphics plotter. The numbers in the figure correspond to the numbers in the list that follows.

This list does not reflect the activities that must be performed by the hypothetical, user-written symbiont, GRAPHICS.EXE. This symbiont is represented in the figure to illustrate the correspondence between a user-written symbiont and the print symbiont supplied by the VMS operating system.

Although SMB routines can be used for a different kind of symbiont, many of their arguments and associated symbols have names related to the print symbiont. The print symbiont is presented here as an example of a typical symbiont and illustrates points that are generally true for symbionts.

❶ You request a printing job with the DCL command PRINT. DCL calls the Send to Job Controller (SYS$SNDJBC) system service, passing the name of the file to be printed to the job controller, along with any other information specified by qualifiers for the PRINT command.

❷ The job controller places the print request in the appropriate queue and assigns the request a job number.

❸ The job controller breaks the print job into a number of tasks (for example, printing three copies of the same file is three separate tasks). The job controller makes a separate request to the symbiont for each task.

Each request that the job controller makes consists of a message. Each message consists of a code that indicates what the symbiont is to do and a number of items of information that the symbiont needs to carry out the task (the name of the file, the name of the user, and so on.)

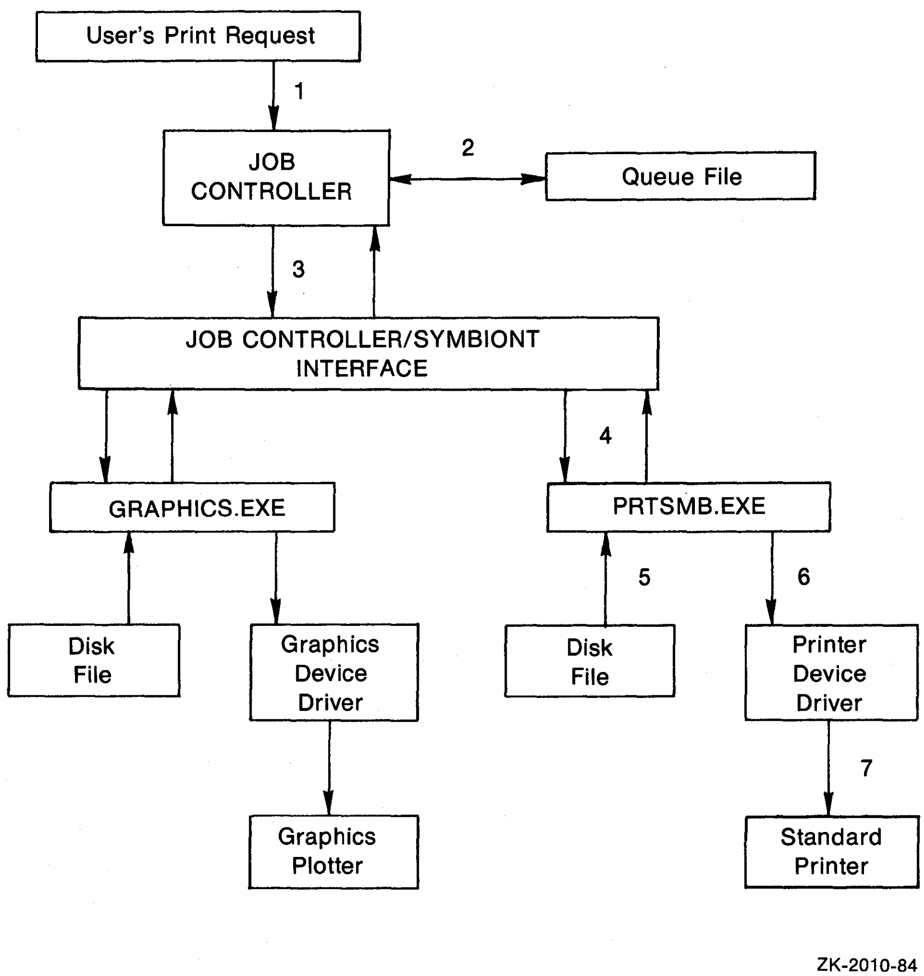
❹ PRTSMB interprets the information it receives from the job controller.

❺ PRTSMB locates and opens the file it is to print by using the file-identification number the job controller specified in the start-task message.

❻ PRTSMB sends the data from the file to the printer's driver.

❼ The device driver sends the data to the printer.

Figure 11–1 Symbionts in the VMS Operating System Environment



11.1.4 Writing a Symbiont

Writing your own symbiont permits you to use the queueing mechanisms and control functions of the VMS job controller. You might want to do this if you need a symbiont for a device that cannot be served by PRTSMB (or a modified form of PRTSMB) or if you need a server symbiont. The interface between the job controller and the symbiont permits the symbiont you write to use the many features of the job controller.

For example, when you use the DCL command PRINT to print a file, the job controller sends a message to the print symbiont telling it to print the file. However, when a user-written symbiont receives the same message (caused by entering a PRINT command), it might interpret it to mean something quite different. A robot symbiont, for example, might interpret the message as a command for movement and the file specification (specified with the PRINT command) might be a file describing the directions in which the robot is to move.

Note: Modifying PRTSMB is easier than writing your own symbiont; choose this option if possible. The Print Symbiont Modification (PSM) routines describe how to modify PRTSMB to suit your needs.

11.1.5 Guidelines for Writing a Symbiont

Although you can write a symbiont to use the queuing mechanisms and other features of the job controller in whatever way you want, you must follow these guidelines to ensure that your symbiont works correctly:

- The symbiont must not use any of the process-permanent channels, which are assigned to the following logical names:
  - SYS$INPUT
  - SYS$OUTPUT
  - SYS$ERROR
  - SYS$COMMAND
- The symbiont must allocate and deallocate memory using the RTL routines LIB$GET_VM and LIB$FREE_VM.
- To be compatible with future releases of the VMS operating system, you should write the symbiont to ignore unknown message-item codes and unknown message-request codes. (See the SMB$READ_ITEM_MESSAGE routine.)
- The symbiont must communicate with the job controller by using the Job-Controller/Symbiont Interface (SMB) routines and the $SNDJBC system service.
- The symbiont should not perform lengthy operations within the context of an AST routine. The symbiont can receive messages only from the job controller when it is not executing within the context of an AST routine.
- To assign a symbiont to a queue after it is compiled and linked, the executable image of the symbiont must reside in SYS$SYSTEM, and you must enter either of the following commands:

  ```
  $ INITIALIZE/QUEUE/PROCESSOR=symbiont_filename
  ```

  ```
  $ START/QUEUE/PROCESSOR=symbiont_filename
  ```

  You should specify only the file name in the command. The disk and directory default to SYS$SYSTEM, and all fields except the file name are ignored.

11.1.6 The Symbiont/Job-Controller Interface Routines

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

SMB$INITIALIZE	Initializes the SMB facility's internal database, establishes the interface to the job controller, and defines whether • Messages from the job controller are to be delivered to the symbiont synchronously or asynchronously with respect to execution of the symbiont. • The symbiont is to be single-threaded or multithreaded; these concepts are described in the sections that follow.
SMB$CHECK_FOR_MESSAGE	Checks to see if a message from the job controller to the symbiont has arrived
SMB$READ_MESSAGE	Reads the job controller's message into a buffer
SMB$READ_MESSAGE_ITEM	Returns one item of information from the job controller's message (which can have several informational items)
SMB$SEND_TO_JOBCTL	Sends a message from the symbiont to the job controller

The following sections discuss how to use the SMB routines when writing your symbiont.

11.1.7 Choosing the Symbiont Environment

The first SMB routine that a symbiont must call is the SMB$INITIALIZE routine. In addition to allocating and initializing the SMB facility's internal database, it offers you two options for your symbiont environment: synchronous or asynchonous delivery of messages from the job controller, and single streaming or multistreaming the symbiont.

11.1.7.1 Synchronous Versus Asynchronous Delivery of Requests

When you initialize your job controller/symbiont interface, the symbiont has the option of accepting requests from the job controller sychronously or asynchronously.

Synchronous Environment

The address of an AST routine is an optional argument to the SMB$INITIALIZE routine; if it is not specified, the symbiont receives messages from the job controller synchronously. A symbiont that receives messages synchronously must call SMB$CHECK_FOR_MESSAGE periodically during the processing of tasks in order to ensure the timely delivery of STOP_TASK, PAUSE_TASK, and RESET_STREAM requests.

SMB$CHECK_FOR_MESSAGE checks to see if a message from the job controller is waiting. If a message is waiting, SMB$CHECK_FOR_MESSAGE returns a success code. The caller of SMB$CHECK_FOR_MESSAGE can then call SMB$READ_MESSAGE to read the message and take the appropriate action.

If no message is waiting, SMB$CHECK_FOR_MESSAGE returns a zero in R0. The caller of SMB$CHECK_FOR_MESSAGE can continue to process the task at hand.

Figure 11–2 is a flowchart for a synchronous, single-threaded symbiont. The flowchart does not show all the details of the logic the symbiont needs and does not show how the symbiont handles pause-task, resume-task, or reset-stream requests.

Asynchronous Environment

To receive messages asynchronously, a symbiont specifies a message-handling AST routine as the second argument to the SMB$INITIALIZE routine. In this scheme, whenever the job controller sends messages to the symbiont, the AST routine is called.

The AST routine is called with no arguments and returns no value. You have the option of having the AST routine read the message within the context of its execution or of having the AST routine wake a suspended process to read the message outside the context of the execution of the AST routine.

Be aware that an AST can be delivered only while the symbiont is not executing within the context of an AST routine. Thus, in order to ensure delivery of messages from the job controller, the symbiont should not perform lengthy operations at AST level.

This is particularly important to the execution of STOP_TASK, PAUSE_TASK, and RESET_STREAM requests. If a STOP_TASK request cannot be delivered during the processing of a task, for example, it is useless.

One technique that ensures delivery of STOP and PAUSE requests in an asynchronous environment is to have the AST routine set a flag if it reads a PAUSE_TASK, STOP_TASK, or a RESET_STREAM request and to have the symbiont's main routine periodically check the flag.

Figure 11–3 shows a logic chart for a single-threaded, asynchronous symbiont. The figure does not show many details that your symbiont might include, such as a call to the $QIO system service.

Note that the broken lines that connect the calls to SYS$HIBER with the AST routine's calls to SYS$WAKE show that the next action to take place is the call to SYS$WAKE. They do not accurately represent the flow of control within the symbiont but represent the action of the job controller in causing the AST routine to execute.

Figure 11–2 Flowchart for a Single-Threaded, Synchronous Symbiont

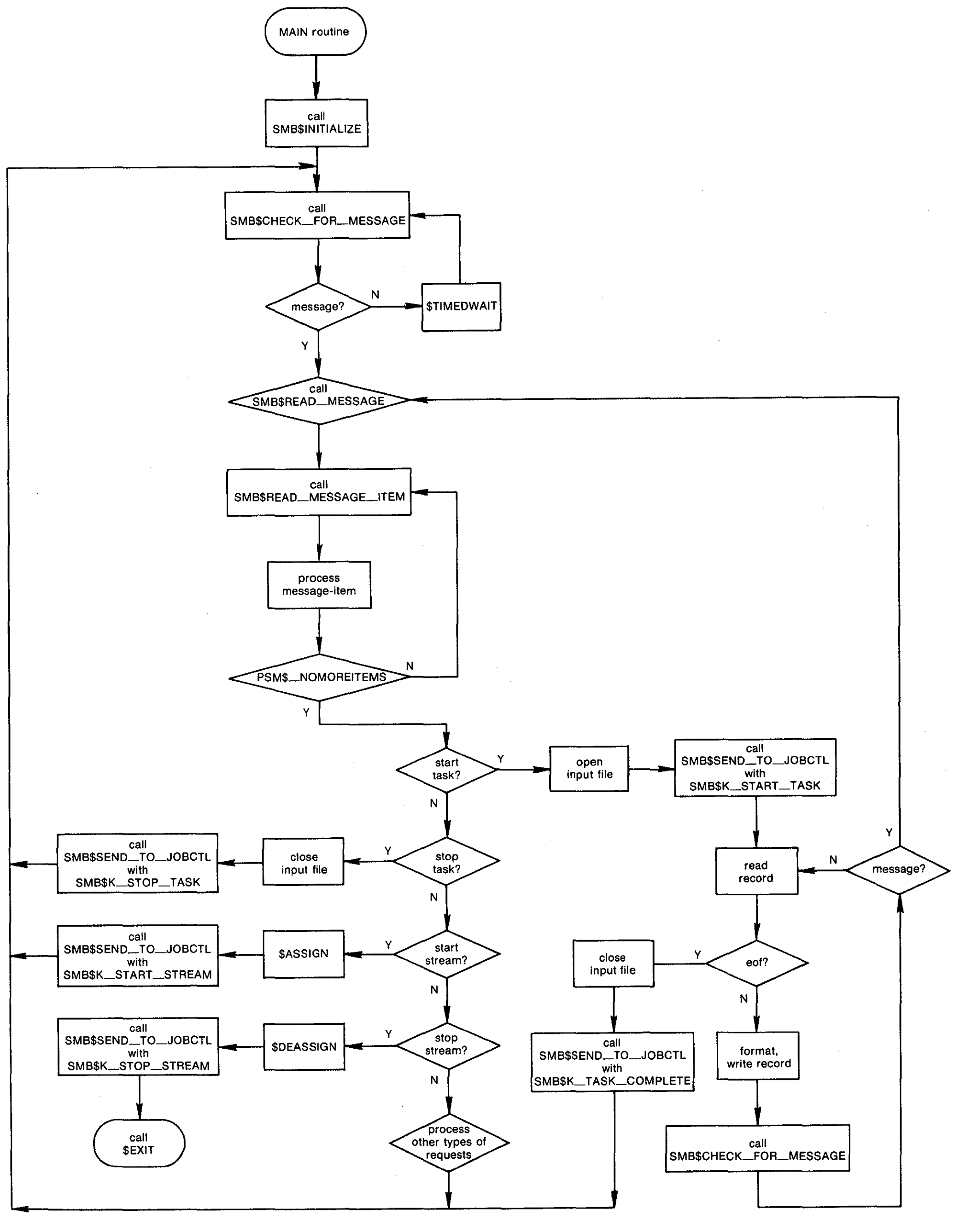


Figure 11–3 Flow Chart for a Single-Threaded, Asynchronous Symbiont

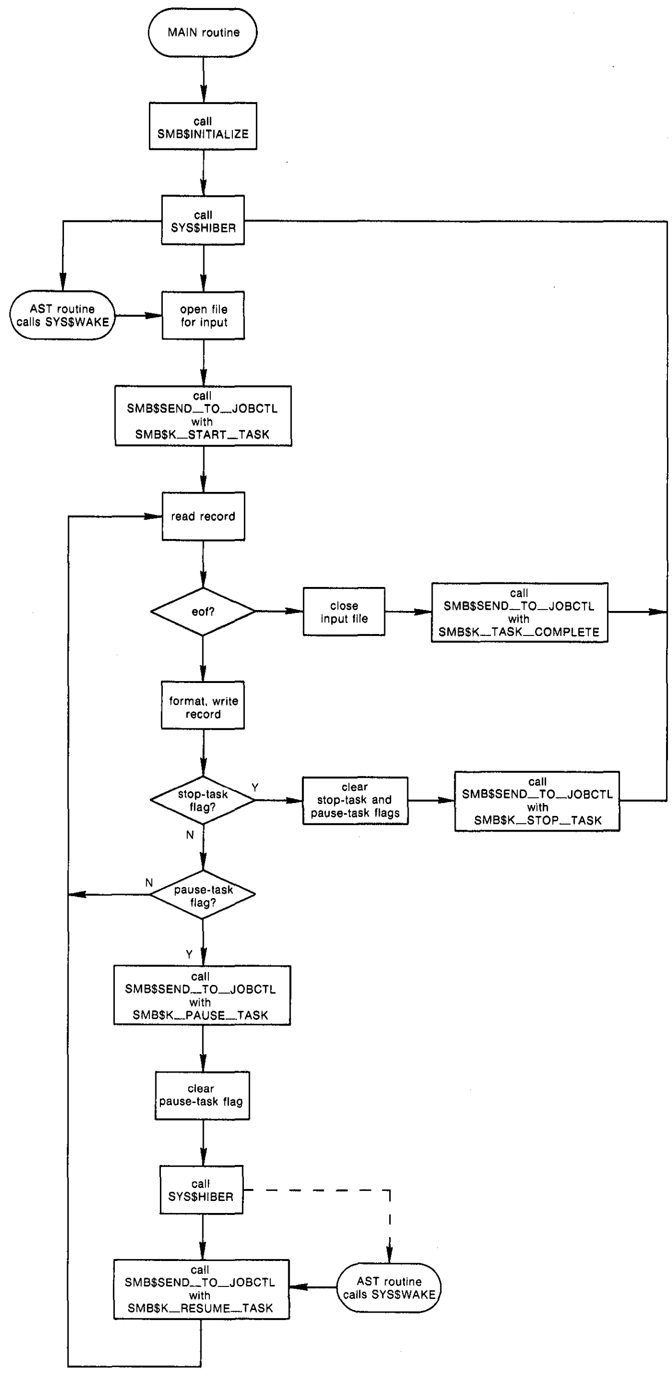
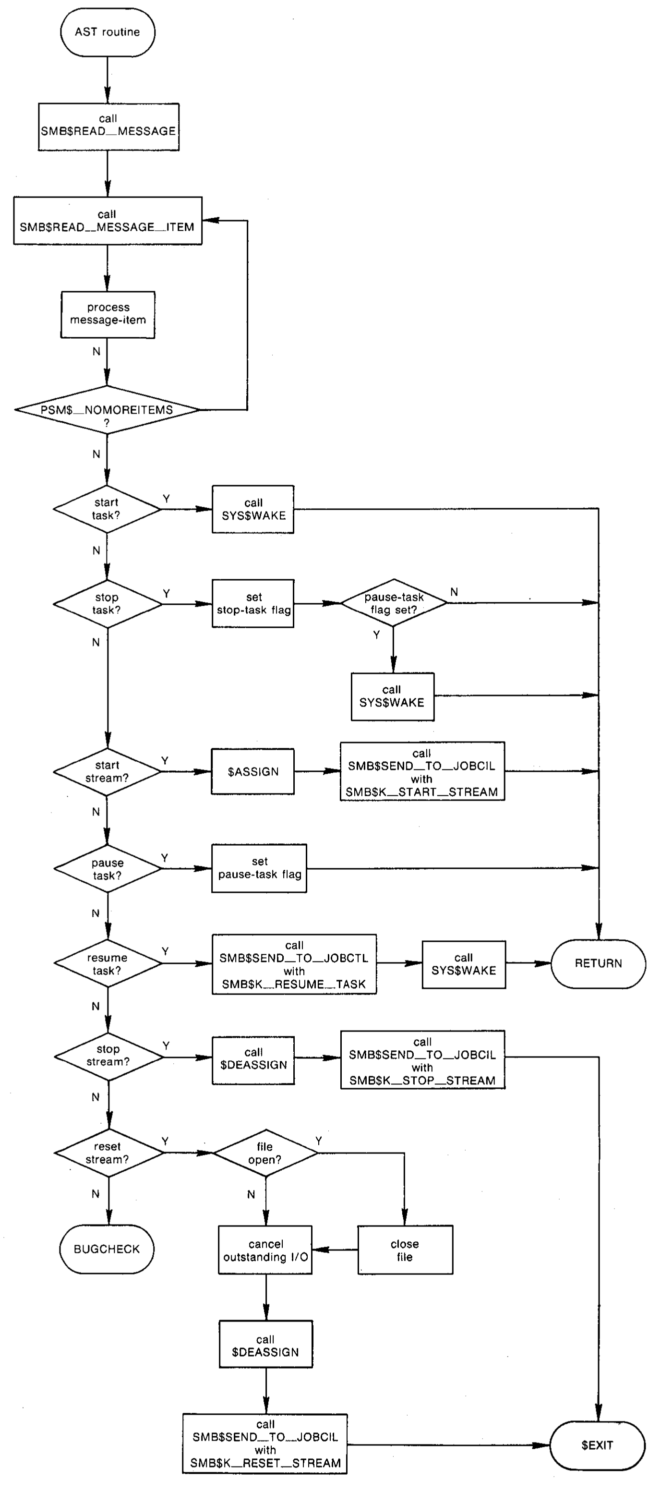


Figure 11–3 Cont'd. on next page

Figure 11–3 (Cont.) Flow Chart for a Single-Threaded, Asynchronous Symbiont



11.1.7.2 Single-Streaming Versus Multistreaming

A single-stream (or thread) is a logical link between a queue and a symbiont process. When a symbiont process is linked to more than one queue and serves those queues simultaneously, it is called a *multithreaded* symbiont.

The argument to the SMB$READ_MESSAGE routine provides a way for a multithreaded symbiont to keep track of the stream referred to by a request. Writing your own multithreaded symbiont, however, can be a complex undertaking.

11.1.8 Reading Job Controller Requests

There are seven general functions that the job controller can request of the symbiont:

SMBMSG$K_START_STREAM	SMBMSG$K_STOP_STREAM
SMBMSG$K_START_TASK	SMBMSG$K_PAUSE_TASK
SMBMSG$K_RESUME_TASK	SMBMSG$K_STOP_TASK
SMBMSG$K_RESET_STREAM	

The job controller passes these requests to the symbiont in a structure that contains (1) a code that identifies the requested function and (2) optional items of information that the symbiont might need to perform the requested function.

By calling SMB$READ_MESSAGE, the symbiont reads the function code and writes the associated items of information, if any, into a buffer. The symbiont then parses the message items stored in the buffer by calling the SMB$READ_MESSAGE_ITEM routine. SMB$READ_MESSAGE_ITEM reads one message item each time it is called.

Each message item consists of a code that identifies the type of information the item contains, and the information itself. For example, the SMB$K_JOB_NAME code tells the symbiont that the item contains a string, which is the name of a job.

The number of message items in a request message varies with each type of request. Therefore, to ensure that all message items are read, SMB$READ_MESSAGE_ITEM must be called repeatedly for each request. SMB$READ_MESSAGE_ITEM returns status SMB$_NOMOREITEMS after it has read the last message item in a given request.

Typically, a symbiont checks the code of a message item against a case table and stores the message string in an appropriate variable until all the message items are read and the processing of the request can begin.

See the description of the SMB$READ_MESSAGE_ITEM routine for a table that shows the message items that make up each type of request.

11.1.9 Processing Job Controller Requests

After a request is read, it must be processed. The way a request is processed depends on the type of request. The following section lists, for each request that the job controller sends to the print symbiont, the actions that the standard symbiont (PRTSMB) takes when the message is received. These actions are oriented toward print symbionts in particular but can serve as a guideline for other kinds of symbionts as well.

The symbiont you write can respond to requests in a similar way or in a different way appropriate to the function of your symbiont. DIGITAL suggests that your routines follow the guidelines described in this document. (Note that the behavior of the standard symbiont is subject to change without notice in future versions of the VMS operating system.)

SMBMSG$K_START_STREAM

- Reset all stream-specific information that might have been altered by previous START_STREAM requests on this stream (for multithreaded symbionts).
- Read and store the message items associated with the request.
- Allocate the device specified by the SMBMSG$K_DEVICE_NAME item.
- Assign a channel to the device.
- Obtain the device characteristics.
- If the device is neither a terminal nor a printer, then abort processing and return an error to the job controller by means of the SMB$SEND_TO_JOBCTL routine. Note that, even though an error has occurred, the stream is still considered started. The job controller detects the error and sends a STOP_STREAM request to the symbiont.
- Set temporary device characteristics suited to the way the symbiont will use the device.
- For remote devices (devices connected to the system by means of a modem) establish an AST to report loss of the carrier signal.
- Report to the job controller that the request has been completed and that the stream is started, by specifying SMBMSG$K_START_STREAM in the call to SMB$SEND_TO_JOBCTL.

SMBMSG$K_START_TASK

- Reset all task-specific information that might have been altered by previous START_TASK requests on this stream number.
- Read and store the message items associated with the request.
- Open the main input file.
- Report to the job controller that the task has been started by specifying SMBSMG$K_START_TASK in the call to the SMB$SEND_TO_JOBCTL routine.
- Begin processing the task.
- When the task is complete, notify the job controller by specifying SMBMSG$K_TASK_COMPLETE in the call to the SMB$SEND_TO_JOBCTL routine.

SMBMSG$K_PAUSE_TASK

- Read and store the message items associated with the request.
- Set a flag that will cause the main processing routine to pause at the beginning of the next output page.
- When the main routine pauses, notify the job controller by specifying SMBMSG$K_PAUSE_TASK in the call to the SMB$SEND_TO_JOBCTL routine.

SMBMSG$K_RESUME_TASK

- Read and store the message items associated with the request.
- Perform any positioning functions specified by the message items.
- Clear the flag that causes the main input routine to pause, and resume processing the task.
- Notify the job controller that the task has been resumed by specifying SMBMSG$K_RESUME_TASK in the call to the SMB$SEND_TO_JOBCTL routine.

SMBMSG$K_STOP_TASK

- Read and store the message items associated with the request.
- If processing of the current task has paused, then resume it.
- Cancel any outstanding I/O operations.
- Close the input file.
- If the job controller specified, in the START_TASK message, that a trailer page should be printed when the task is stopped or if it specified that the device should be reset when the task is stopped, then perform those functions.
- Notify the job controller that the task has been stopped abnormally by specifying SMBMSG$K_STOP_TASK and by specifying an error vector in the call to SMB$SEND_TO_JOBCTL. PRTSMB specifies the value passed by the job controller in the SMBMSG$K_STOP_CONDITION item as the error condition in the error vector.

SMBMSG$K_STOP_STREAM

- Read and store the message items associated with the request.
- Release any stream-specific resources: (1) deassign the channel to the device, and (2) deallocate the device.
- Notify the job controller that the stream has been stopped by specifying SMBMSG$K_STOP_STREAM in the call to SMB$SEND_TO_JOBCTL.
- If this is a single-threaded symbiont or if this is a multithreaded symbiont but all other streams are currently stopped, then call the SYS$EXIT system service with the condition code SS$_NORMAL.

SMBMSG$K_RESET_STREAM

- Read and store the message items associated with the request.
- Abort any task in progress—you do not need to notify the job controller that the task has been aborted, but you can do so if you want.
- If the job controller specified, in the START_TASK message, that a trailer page should be printed when the task is stopped or if it specified that the device should be reset when the task is stopped, then suppress those functions.

  The job controller sends the symbiont a RESET_STREAM request to regain control of a queue or a device that has failed to respond to a STOP_TASK request. The RESET_STREAM request should avoid any further I/O activity if possible. The printer might be disabled, for example, and requests for output on that device will never be completed.
- Continue as if this were a STOP_STREAM request.

Note: **A STOP_STREAM request and a RESET_STREAM request each stop the queue; but a RESET_STREAM request is an emergency stop and is used, for example, when the device has failed. A RESET_STREAM request should prevent any further I/O activity because the printer might not be able to complete it.**

11.1.10 Responding to Job Controller Requests

The symbiont uses the SMB$SEND_TO_JOBCTL routine to send messages to the job controller.

Most messages that the symbiont sends to the job controller are responses to requests made by the job controller. Such messages inform the job controller that the request has been completed successfully or unsuccessfully. The function code that the symbiont returns to the controller in the call to SMB$SEND_TO_JOBCTL indicates what request has been completed.

For example, if the job controller sends a START_TASK request using the SMBMSG$K_START_TASK code, the symbiont responds by calling SMB$SEND_TO_JOBCTL using SMBMSG$K_START_TASK as the **request** argument to indicate that task processing has begun. Until the symbiont responds, the DCL command SHOW QUEUE indicates that the queue is starting.

The responses to some requests use additional arguments to send more information than just the request code. See the SMB$SEND_TO_JOBCTL routine for a table showing the additional arguments allowed in response to each request.

In addition to sending messages in response to requests, the symbiont can send other messages to the job controller. In these messages the symbiont sends either the SMBMSG$K_TASK_COMPLETE code, indicating that it has completed a task, or SMBMSG$K_TASK_STATUS, indicating that the message contains information on the status of a task.

Note that, when a START_TASK request is delivered, the symbiont responds with a SMB$SEND_TO_JOBCTL message with the SMBSMG$K_START_TASK code. This response means the task has been started. It does not mean the task has been completed. When the symbiont completes the task, it calls SMB$SEND_TO_JOBCTL with the SMBSMG$K_TASK_COMPLETE code.

11.2 SMB Routines

The following pages describe the individual SMB routines.

SMB$CHECK_FOR_MESSAGE Check for Message from Job Controller

The SMB$CHECK_FOR_MESSAGE routine determines whether a message sent from the job controller to the symbiont is waiting to be read.

FORMAT

SMB$CHECK_FOR_MESSAGE

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

None.

DESCRIPTION

When your symbiont calls the SMB$INITIALIZE routine to initialize the interface between the symbiont and the job controller, you can choose to have requests from the job controller delivered by means of an AST. If you choose not to use ASTs, your symbiont must call SMB$CHECK_FOR_MESSAGE during the processing of tasks in order to see if a message from the job controller is waiting to be read. If a message is waiting, SMB$CHECK_FOR_MESSAGE returns a success code; if not, it returns a zero.

If a message is waiting, the symbiont should call SMB$READ_MESSAGE to read it and determine if immediate action should be taken (as in the case of STOP_TASK, RESET_STREAM or PAUSE_TASK).

If a message is not waiting, SMB$CHECK_MESSAGE returns a zero. If this condition is detected, the symbiont should continue processing the request at hand.

CONDITION VALUES RETURNED

SS$_NORMAL	One or more messages waiting.
0	No messages waiting.

SMB$INITIALIZE Initialize User-Written Symbiont

The SMB$INITIALIZE routine initializes the user-written symbiont and the interface between the symbiont and the job controller. It allocates and initializes the internal databases of the interface and sets up the mechanism that is to wake up the symbiont when a message is received.

FORMAT

SMB$INITIALIZE *structure_level [,ast_routine] [,streams]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

structure_level

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

Version of the job-controller/symbiont interface. The **structure_level** argument is the address of a longword containing the version of the job-controller/symbiont interface used when the symbiont was compiled. Always place the value of the symbol SMBMSG$K_STRUCTURE_LEVEL in the longword addressed by this argument. This symbol is defined by the $SMBDEF macro. The $SMBDEF macro is defined in the macro library SYS$LIBRARY:LIB.MLB.

ast_routine

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference**

Message-handling routine called at AST level. The **ast_routine** argument is the address of the entry point of the message-handling routine to be called at AST level when the symbiont receives a message from the job controller. The AST routine is called with no parameters and returns no value. If an AST routine is specified, the routine is called once each time the symbiont receives a message from the job controller.

The AST routine typically reads the message and determines if immediate action must be taken. Be aware that an AST can be delivered only while the symbiont is operating at non-AST level. Thus, to ensure delivery of messages from the job controller, the symbiont should not perform lengthy operations at AST level.

The **ast_routine** argument is optional. If you do not specify it, the symbiont must call the SMB$CHECK_FOR_MESSAGE routine to check for waiting messages.

streams

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

Maximum number of streams the symbiont is to support. The **streams** argument is the address of a longword containing the number of streams that the symbiont is to support. The number must be in the range *1* to *16*.

If you do not specify this argument, a default value of *1* is used. Thus, by default, a symbiont supports one stream. Such a symbiont is called a single-threaded symbiont.

A stream (or thread) is a logical link between a queue and a symbiont. When a symbiont is linked to more than one queue, and serves those queues simultaneously, it is called a multithreaded symbiont.

DESCRIPTION

Your symbiont must call SMB$INITIALIZE before calling any other SMB$ routines. It calls SMB$INITIALIZE in order to do the following:

- Allocate and initialize the SMB$ facility's internal database.
- Establish the interface between the job controller and the symbiont.
- Determine the threading scheme of the symbiont.
- Set up the mechanism to wake your symbiont when a message is received.

After the symbiont calls SMB$INITIALIZE, it can communicate with the job controller using the other SMB$ services.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SMB$_INVSTRLEV	Invalid structure level.

This routine also returns any codes returned by $ASSIGN and LIB$GET_VM.

SMB$READ_MESSAGE Obtain Message Sent by Job Controller

The SMB$READ_MESSAGE routine copies a message that the job controller has sent into the caller's specified buffer.

FORMAT

SMB$READ_MESSAGE *stream ,buffer ,request*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

stream

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

Stream number specifying the stream to which the message refers. The **stream** argument is the address of a longword into which the job controller writes the number of the stream referred to by the message. In single-threaded symbionts, the stream number is always 0.

buffer

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

Address of the descriptor that points to the buffer into which the job controller writes the message. SMB$READ_MESSAGE uses the RTL STR$ string-handling routines to copy the message into the buffer you supply. The buffer should be specified by a dynamic string descriptor.

request

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Code that identifies the request. The **request** argument is the address of a longword into which SMB$READ_MESSAGE writes the code that identifies the request.

There are seven request codes. Each code is interpreted as a message by the symbiont. The codes and their descriptions follow:

SMBMSG$K_START_STREAM	Initiates processing on an inactive symbiont stream. The job controller sends this message when a START/QUEUE or an INITIALIZE /QUEUE/START command is issued on a stopped queue.
SMBMSG$K_STOP_STREAM	Stops processing on a started queue. The job controller sends this message when a STOP /QUEUE/NEXT command is issued, after the symbiont completes any currently active task.
SMBMSG$K_RESET_STREAM	Aborts all processing on a started stream and requeues the current job. The job controller sends this message when a STOP/QUEUE /RESET command is issued.
SMBMSG$K_START_TASK	Requests that the symbiont begin processing a task. The job controller sends this message when a file is pending on an idle, started queue.
SMBMSG$K_STOP_TASK	Requests that the symbiont abort the processing of a task. The job controller sends this message when a STOP/QUEUE/ABORT or STOP/QUEUE/REQUEUE command is issued. The item SMBMSG$K_STOP_CONDITION identifies whether this is an abort or a requeue request.
SMBMSG$K_PAUSE_TASK	Requests that the symbiont pause in the processing of a task but retain the resources necessary to continue. The job controller sends this message when a STOP/QUEUE command is issued without the /ABORT, /ENTRY, /REQUEUE, or /NEXT qualifier for a queue that is currently printing a job.
SMBMSG$K_RESUME_TASK	Requests that the symbiont continue processing a task that has been stopped with a PAUSE_TASK request. This message is sent when a START/QUEUE command is issued for a queue served by a symbiont that has paused in processing the current task.

DESCRIPTION

Your symbiont calls SMB$READ_MESSAGE to read a message that the job controller has sent to the symbiont.

Each message from the job controller consists of a code identifying the function the symbiont is to perform and a number of message items. There are seven codes. Message items are pieces of information that the symbiont needs to carry out the requested function.

For example, when you enter the DCL command PRINT, the job controller sends a message containing a START_TASK code and a message item containing the specification of the file to be printed.

SMB$READ_MESSAGE writes the code into a longword (specified by the **request** argument) and writes the accompanying message items, if any, into a buffer (specified by the **buffer** argument).

See the description of the SMB$READ_MESSAGE_ITEM routine for information about processing the individual message items.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine completed successfully.
LIB$_INVARG	Routine completed unsuccessfully because of an invalid argument.

This routine also returns any of the condition codes returned by the Run-Time Library string-handling (STR$) routines.

SMB$READ_MESSAGE_ITEM Parse Next Item from Message Buffer

The SMB$READ_MESSAGE_ITEM routine reads a buffer that was filled in by the SMB$READ_MESSAGE routine, parses one message item from the buffer, writes the item's code into a longword, and writes the item into a buffer.

FORMAT

SMB$READ_MESSAGE_ITEM *message ,context ,item_code ,buffer [,size]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

message

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

Message items that SMB$READ_MESSAGE_ITEM is to read. The **message** argument is the address of a descriptor of a buffer. The buffer is the one that contains the message items that SMB$READ_MESSAGE_ITEM is to read. The buffer specified here must be the same as that specified with the call to the SMB$READ_MESSAGE routine, which fills the buffer with the contents of the message.

context

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

Value initalized to *0* specifying the first message item in the buffer to be read. The **context** argument is the address of a longword that the SMB$READ_MESSAGE_ITEM routine uses to determine the next message item to be returned. When this value is *0*, it indicates that SMB$READ_MESSAGE_ITEM is to return the first message item.

The SMB$READ_MESSAGE_ITEM routine updates this value each time it reads a message item. SMB$READ_MESSAGE_ITEM sets the value to *0* when it has returned all the message items in the buffer.

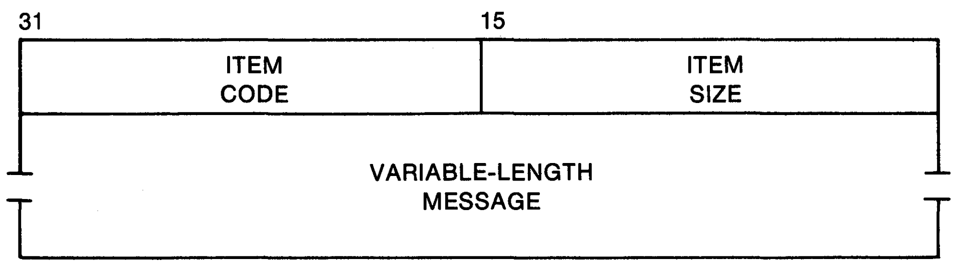
item_code

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

Item code specified in the message item that identifies its type. The **item_code** argument is the address of a longword into which SMB$READ_MESSAGE_ITEM writes the code that identifies what item it is returning.

The codes that identify message items are defined at the end of the Description section for this routine.

The following diagram depicts the format of a single message item.



ZK-2037-84

SMB$READ_MESSAGE_ITEM copies the code from the second word in the message item to the longword specified by the **item_code** argument.

SMB$READ_MESSAGE_ITEM uses the item-size field in the message item to determine the length, in bytes, of the variable-length message.

buffer

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

Message item. The **buffer** argument is the address of a descriptor of a buffer. The buffer is the one in which the SMB$READ_MESSAGE_ITEM routine is to place the message item. SMB$READ_MESSAGE_ITEM uses the RTL STR$ string-handling routines to copy the message item into the buffer.

size

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

Size of the message item. The **size** argument is the address of a word in which the SMB$READ_MESSAGE_ITEM is to place the size, in bytes, of the item.

DESCRIPTION

The job controller can request seven functions from the symbiont. They are identified by the following codes:

SMBMSG$K_START_STREAM	SMBMSG$K_STOP_STREAM
SMBMSG$K_START_TASK	SMBMSG$K_PAUSE_TASK
SMBMSG$K_RESUME_TASK	SMBMSG$K_STOP_TASK
SMBMSG$K_RESET_STREAM	

The job controller passes the symbiont a request containing a code and, optionally, a number of message items containing information the symbiont might need to perform the function. The code specifies what function the request is for, and the message items contain information that the symbiont needs to carry out the function.

By calling SMB$READ_MESSAGE, the symbiont reads the request and writes the message items into the specified buffer. The symbiont then obtains the individual message items by calling the SMB$READ_MESSAGE_ITEM routine.

Each message item consists of a code that identifies the information the item represents, and the item itself. For example, the SMB$K_JOB_NAME code tells the symbiont that the item specifies a job's name.

The number of items in a request varies with each type of request. Therefore, you must call SMB$READ_MESSAGE_ITEM repeatedly for each request to ensure that all message items are read. Each time SMB$READ_MESSAGE_ITEM reads a message item, it updates the value in the longword specified by the **context** argument. SMB$READ_MESSAGE_ITEM returns the code SMB$_NOMOREITEMS after it has read the last message item.

The following table shows the message items that can be delivered with each request.

Request	Message Item
SMBMSG$K_START_TASK	SMBMSG$K_ACCOUNT_NAME
	SMBMSG$K_AFTER_TIME
	SMBMSG$K_BOTTOM_MARGIN
	SMBMSG$K_CHARACTERISTICS
	SMBMSG$K_CHECKPOINT_DATA
	SMBMSG$K_ENTRY_NUMBER
	SMBMSG$K_FILE_COPIES
	SMBMSG$K_FILE_COUNT
	SMBMSG$K_SETUP_MODULES
	SMBMSG$K_FIRST_PAGE
	SMBMSG$K_FORM_LENGTH
	SMBMSG$K_FORM_NAME
	SMBMSG$K_FORM_SETUP_MODULES
	SMBMSG$K_FORM_WIDTH
	SMBMSG$K_FILE_IDENTIFICATION

Request	Message Item
	SMBMSG$K_MESSAGE_VECTOR
	SMBMSG$K_FILE_SPECIFICATION
	SMBMSG$K_JOB_COPIES
	SMBMSG$K_JOB_COUNT
	SMBMSG$K_JOB_NAME
	SMBMSG$K_JOB_RESET_MODULES
	SMBMSG$K_LAST_PAGE
	SMBMSG$K_LEFT_MARGIN
	SMBMSG$K_NOTE
	SMBMSG$K_PAGE_SETUP_MODULES
	SMBMSG$K_PARAMETER_1
	SMBMSG$K_SEPARATION_CONTROL
	SMBMSG$K_REQUEST_CONTROL
	SMBMSG$K_PRIORITY
	SMBMSG$K_QUEUE
	SMBMSG$K_TIME_QUEUED
	SMBMSG$K_TOP_MARGIN
	SMBMSG$K_UIC
	SMBMSG$K_USER_NAME
	SMBMSG$K_RIGHT_MARGIN
SMBMSG$K_STOP_TASK	SMBMSG$K_STOP_CONDITION
SMBMSG$K_PAUSE_TASK	None
SMBMSG$K_RESUME_TASK	SMBMSG$K_ALIGNMENT_PAGES
	SMBMSG$K_RELATIVE_PAGE
	SMBMSG$K_REQUEST_CONTROL
	SMBMSG$K_SEARCH_STRING
SMBMSG$K_START_STREAM	SMBMSG$K_DEVICE_NAME
	SMBMSG$K_EXECUTOR_QUEUE
	SMBMSG$K_JOB_RESET_MODULES
	SMBMSG$K_LIBRARY_SPECIFICATION
SMBMSG$K_STOP_STREAM	None
SMBMSG$K_RESET_STREAM	None

Following are the message items that the symbiont can send to the job controller:

- SMBMSG$K_ACCOUNTING_DATA
- SMBMSG$K_CHECKPOINT_DATA
- SMBMSG$K_CONDITION_VECTOR
- SMBMSG$K_DEVICE_STATUS
- SMBMSG$K_REQUEST_RESPONSE

The following list enumerates each item code. For each code, the list describes the contents of the message item identified by the code and whether the code identifies an item sent from the job controller to the symbiont or from the symbiont to the job controller.

Many of the codes described are specifically oriented toward print symbionts. The symbiont you implement, which might not print files or serve an output device, need not recognize all these codes. In addition, it need not respond in the same way as the VMS print symbiont to the codes it recognizes. The descriptions in the list describe how the standard VMS print symbiont (PRTSMB.EXE) processes these items.

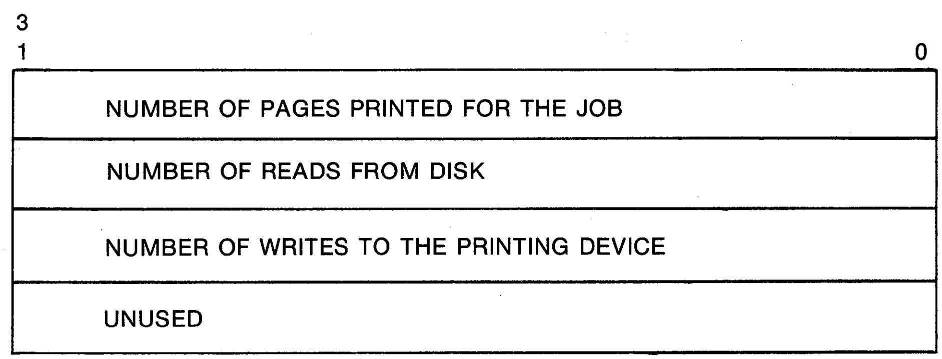
Note: Because new codes might be added in the future, you should write your symbiont so that it ignores codes it does not recognize.

Codes for Message Items

SMBMSG$K_ACCOUNTING_DATA

This code identifies a 16-byte structure that the symbiont sends to the job controller. This structure contains accounting statistics that the symbiont has accumulated for the task. The job controller accumulates task statistics into a job-accounting record, which it writes to the accounting file when the job is completed.

The following diagram depicts the contents of the 16-byte structure.



ZK-2011-84

SMBMSG$K_ACCOUNT_NAME

This code identifies a string containing the name of the account to be charged for the job, that is, the account of the process that submitted the print job. The job controller sends this item to the symbiont.

SMBMSG$K_AFTER_TIME

This code identifies a 64-bit, absolute-time value specifying the system time after which the job controller can process this job. The job controller sends this item to the symbiont.

SMBMSG$K_ALIGNMENT_PAGES

This code identifies a longword specifying the number of alignment pages that the symbiont is to print. The job controller sends this item to the symbiont.

SMBMSG$K_BOTTOM_MARGIN

This code identifies a longword containing the number of lines to be left blank at the bottom of a page. The job controller sends this item to the symbiont.

The symbiont inserts a formfeed character into the output stream if it determines the following:

- That the number of lines left at the bottom of the page is equal to the value in SMBMSG$K_BOTTOM_MARGIN
- That sending more data to the printer to be output on this page would cause characters to be printed within this bottom margin of the page
- That the /FEED qualifier was specified with the PRINT command that caused the symbiont to perform this task

(Linefeed, formfeed, carriage-return, and vertical-tab characters in the output stream are collectively known as embedded carriage control.)

SMBMSG$K_CHARACTERISTICS

This code identifies a 32-byte structure specifying characteristics of the job. A detailed description of the format of this structure is contained in the description of the SJC$_CHARACTERISTIC code in the SYS$SNDJBC system service in the *VMS System Services Reference Manual*. The job controller sends this item to the symbiont.

SMBMSG$K_CHECKPOINT_DATA

This code identifies a user-defined structure containing checkpointing information. The symbiont sends this item to the job controller, which saves it in the queue's data file.

When a restart-from-checkpoint request is executed for the queue, the job controller retrieves the checkpointing information from the queue's data file and sends it to the symbiont with a SMBMSG$K_START_TASK request. The symbiont uses the checkpointing information to reposition the input file to the point corresponding to the last page output at the time of the checkpoint.

SMBMSG$K_CONDITION_VECTOR

This code identifies an array of longwords, each longword containing a code that specifies a termination status for the current request. The symbiont sends this item to the job controller. For example, the STS and STV values from an RMS control block might be two longwords in the array.

SMBMSG$K_DEVICE_NAME

This code identifies a string that is the name of the device to which the symbiont is to send data. The symbiont interprets this information. The name need not be the name of a physical device, and the symbiont can interpret this string as something other than the name of a device.

SMBMSG$K_DEVICE_STATUS

This code identifies a longword bit vector, each bit of which specifies device-status information. The symbiont sends this item to the job controller. The $SMBDEF macro defines these device-status bits. The following describes the effect of setting each bit in the longword.

Device Status Bit	Description
SMBMSG$V_LOWERCASE	The device to which the symbiont is connected supports lowercase characters.
SMBMSG$V_PAUSE_TASK	Informs the job controller that the symbiont has paused on its own initiative.
SMBMSG$V_REMOTE	The device is connected to the symbiont by means of a modem.
SMBMSG$V_SERVER	The symbiont is not connected to a device.
SMBMSG$V_STALLED	Symbiont processing is temporarily stalled.
SMBMSG$V_STOP_STREAM	The symbiont requests that the job controller stop the queue.
SMBMSG$V_TERMINAL	The symbiont is connected to a terminal.
SMBMSG$V_UNAVAILABLE	The device to which the symbiont is assigned is not available.

SMBMSG$K_ENTRY_NUMBER

This code identifies a longword containing the number that the job controller assigned to the job. The job controller sends this item to the symbiont.

SMBMSG$K_EXECUTOR_QUEUE

This code identifies a string that is the name of the queue on which the currently executing job is listed. The job controller sends this item to the symbiont.

SMBMSG$K_FILE_COPIES

This code identifies a longword containing the number of copies of the file that were requested.

SMBMSG$K_FILE_COUNT

This code identifies a longword that specifies, out of the number of copies requested for this job (SMBMSG$K_FILE_COPIES), the number of the copy of the file currently printing. The job controller sends this item to the symbiont.

SMBMSG$K_FILE_IDENTIFICATION

This code identifies a 28-byte structure identifying the file to be processed. This structure consists of the following three file-identification fields in the RMS NAM block:

1 The 16-byte NAM$T_DVI field
2 The 6-byte NAM$W_FID field
3 The 6-byte NAM$W_DID field

These fields occur consecutively in the NAM block in the order listed. The job controller sends this item to the symbiont.

SMBMSG$K_FILE_SETUP_MODULES

This code identifies a string specifying the names of one or more text modules that the symbiont should copy from the library into the output stream before processing the file. When you specify more than one name, you must separate the names with commas. The job controller sends this item to the symbiont.

SMBMSG$K_FILE_SPECIFICATION

This code identifies a string specifying the name of the file that the symbiont is to process. This file name is formatted as a standard RMS file specification. The job controller sends this item to the symbiont.

SMBMSG$K_FIRST_PAGE

This code identifies a longword containing the number of the page at which the symbiont should begin printing. The job controller sends this item to the symbiont. When not specified, the symbiont begins processing at page 1.

SMBMSG$K_FORM_LENGTH

This code identifies a longword value specifying the length (in lines) of the physical form (the paper). The job controller sends this item to the symbiont.

SMBMSG$K_FORM_NAME

This code identifies a string specifying the name of the form. The job controller sends this item to the symbiont.

SMBMSG$K_FORM_SETUP_MODULES

This code identifies a string consisting of the names of one or more modules that the symbiont should copy from the device-control library before processing the file. When you specify more than one name, you must separate the names with commas. The job controller sends this item to the symbiont.

SMBMSG$K_FORM_WIDTH

This code identifies a longword specifying the width (in characters) of the print area on the physical form (the paper). The symbiont sends this item to the job controller.

SMBMSG$K_JOB_COPIES

This code identifies a longword specifying the requested number of copies of the job. The job controller sends this item to the symbiont.

SMBMSG$K_JOB_COUNT

This code identifies a longword specifying, out of the number of copies requested (SMBMSG$K_JOB_COPIES), the number of the copy of the job currently printing. The job controller sends this item to the symbiont.

SMBMSG$K_JOB_NAME

This code identifies a string specifying the name of the job. The job controller sends this item to the symbiont.

SMBMSG$K_JOB_RESET_MODULES
This code identifies a string specifying the names of one or more modules that the symbiont should copy from the device-control library after processing the task. These modules can be used to reset programmable devices to a known state. When you specify more than one name, you must separate the names with commas. The job controller sends this item to the symbiont.

SMBMSG$K_LAST_PAGE
This code identifies a longword specifying the number of the last page that the symbiont is to print. The job controller sends this item to the symbiont. When not specified, the symbiont attempts to print all the pages in the file.

SMBMSG$K_LEFT_MARGIN
This code identifies a longword specifying the number of spaces to be inserted at the beginning of each line. The job controller sends this item to the symbiont.

SMBMSG$K_LIBRARY_SPECIFICATION
This code identifies a string specifying the name of the device-control library. The job controller sends this item to the symbiont.

SMBMSG$K_MESSAGE_VECTOR
This code identifies a vector of longword condition codes, each of which contains information about the job to be printed. The job controller sends this item to the symbiont.

When LOGINOUT cannot open a log file for a batch job, a code in the message vector specifies the reason for the failure. The job controller does not send the SMBMSG$K_FILE_IDENTIFICATION item if it has detected such a failure but instead sends the message vector, which the symbiont prints, along with a message stating that there is no file to print.

SMBMSG$K_NOTE
This code identifies a user-supplied string that the symbiont is to print on the job-flag page and on the file-flag page. The job controller sends this item to the symbiont.

SMBMSG$K_PAGE_SETUP_MODULES
This code identifies a string consisting of the names of one or more modules that the symbiont should copy from the device-control library before printing each page. When you specify more than one name, you must separate the names with commas. The job controller sends this item to the symbiont.

SMBMSG$K_PARAMETER_1 through SMBMSG$K_PARAMETER_8
Each of these eight codes identifies a user-supplied string. Both the semantics and syntax of each string are determined by your symbiont. The VMS-supplied symbiont makes no use of these eight items. The job controller sends these items to the symbiont.

SMBMSG$K_PRINT_CONTROL
This code identifies a longword bit vector, each bit of which supplies information that the symbiont is to use in controlling the printing of the file. The job controller sends this item to the symbiont.

The $SMBDEF macro defines the following symbols for each bit in the bit vector.

Symbol	Description
SMBMSG$V_DOUBLE_SPACE	The symbiont uses a double-spaced format; it skips a line after each line it prints.
SMBMSG$V_NORECORD_BLOCKING	The symbiont performs single record output, issuing a single output record for each input record.
SMBMSG$V_PAGE_HEADER	The symbiont prints a page header at the top of each page.
SMBMSG$V_PAGINATE	The symbiont inserts a formfeed character when it detects an attempt to print in the bottom margin of the current form.
SMBMSG$V_PASSALL	The symbiont prints the file without formatting and bypasses all formatting normally performed. Furthermore, the symbiont outputs the file without formatting, by causing the output QIO to suppress formatting by the driver.
SMBMSG$V_RECORD_BLOCKING	The symbiont performs record blocking, buffering output to the device.
SMBMSG$V_SEQUENCED	This bit is reserved by DIGITAL.
SMBMSG$V_SHEET_FEED	The symbiont pauses after each page it prints.
SMBMSG$V_TRUNCATE	The symbiont truncates input lines that exceed the right margin of the current form.
SMBMSG$V_WRAP	The symbiont wraps input lines that exceed the right margin, printing the additional characters on a new line.

SMBMSG$K_PRIORITY
This code identifies a longword specifying the priority this job has in the queue in which it is entered. The job controller sends this item to the symbiont.

SMBMSG$K_QUEUE
This code identifies a string specifying the name of the queue in which this job is entered. The job controller sends this item to the symbiont. When generic queues are used, this item specifies the name of the generic queue, and the SMBMSG$K_EXECUTOR item specifies the name of the device queue or the server queue.

SMBMSG$K_RELATIVE_PAGE
This code identifies a signed, longword value specifying the number of pages that the symbiont is to move forward (positive value) or backward (negative value) from the current position in the file. The job controller sends this item to the symbiont.

SMBMSG$K_REQUEST_CONTROL
This code identifies a longword bit vector each bit of which specifies information that the symbiont is to use in processing the request that the

job controller is making. The job controller sends this item to the symbiont. The $SMBDEF macro defines the following symbols for each bit.

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

SMBMSG$K_REQUEST_RESPONSE

This code identifies a longword specifying the type of request for which the symbiont is currently signalling completion. The symbiont sends this item to the job controller. The following symbols define types of requests you can specify in this item:

SMBMSG$K_START_STREAM	SMBMSG$K_STOP_STREAM
SMBMSG$K_START_TASK	SMBMSG$K_PAUSE_TASK
SMBMSG$K_RESUME_TASK	SMBMSG$K_STOP_TASK
SMBMSG$K_RESET_STREAM	SMBMSG$K_COMPLETE_TASK
SMBMSG$K_TASK_STATUS	

SMBMSG$K_RIGHT_MARGIN

This code identifies a longword specifying the number of character positions to be left empty at the end of each line. The job controller sends this item to the symbiont. When the right margin is exceeded, the symbiont truncates the line, wraps the line, or continues processing, depending on the settings of the WRAP and TRUNCATE bits in the SMBMSG$K_PRINT_CONTROL item.

SMBMSG$K_SEARCH_STRING

This code identifies a string containing the value specified in the START /QUEUE/SEARCH command. The job controller sends this item to the symbiont. This string identifies the page at which to restart the current printing task on a paused queue.

SMBMSG$K_SEPARATION_CONTROL

This code identifies a longword bit vector, each bit of which specifies an operation that the symbiont is to perform between jobs or between files within a job. The job controller sends this item to the symbiont. The $SMBDEF macro defines the following symbols for each bit.

Symbol	Description
SMBMSG$V_FILE_BURST	The symbiont is to print a file-burst page.
SMBMSG$V_FILE_FLAG	The symbiont is to print a file-flag page.
SMBMSG$V_FILE_TRAILER	The symbiont is to print a file-trailer page.
SMBMSG$V_FILE_TRAILER_ABORT	The symbiont is to print a file-trailer page when a task completes abnormally.
SMBMSG$V_FIRST_FILE_OF_JOB	The current file is the first file of the job. When specified with SMBMSG$V_LAST_FILE_OF_JOB, the current job contains a single file.
SMBMSG$V_JOB_FLAG	The symbiont is to print a job-flag page.
SMBMSG$V_JOB_BURST	The symbiont is to print a job-burst page.
SMBMSG$V_JOB_RESET	The symbiont is to execute a job-reset sequence when the task completes.
SMBMSG$V_JOB_RESET_ABORT	The symbiont is to execute a job-reset sequence when a task completes abnormally.
SMBMSG$V_JOB_TRAILER	The symbiont is to print a job-trailer page.
SMBMSG$V_JOB_TRAILER_ABORT	The symbiont is to print a job-trailer page when a task completes abnormally.
SMBMSG$V_LAST_FILE_OF_JOB	The current file is the last file of the job. When specified with SMBMSG$V_FIRST_FILE_OF_JOB, the current job contains a single job.

SMBMSG$K_STOP_CONDITION
This code identifies a longword containing a condition specifying the reason the job controller issued a STOP_TASK request. The job controller sends this item to the symbiont.

SMBMSG$K_TIME_QUEUED
This code identifies a quadword specifying the time the file was entered into the queue. The time is expressed as 64-bit, absolute time. The job controller sends this item to the symbiont.

SMBMSG$K_TOP_MARGIN
This code identifies a longword specifying the number of lines that the symbiont is to leave blank at the top of each page. PRTSMB inserts linefeeds into the output stream after every formfeed until the margin is cleared.

SMBMSG$K_UIC
This code identifies a longword specifying the User Identification Code (UIC) of the user who submitted the job.

SMBMSG$K_USER_NAME
This code identifies a string specifying the name of the user who submitted the job.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine completed successfully.
SMB$_NOMOREITEMS	End of item list reached.

This routine also returns any condition code returned by the Run-Time Library string-handling (STR$) routines.

SMB$SEND_TO_JOBCTL Send Message to Job Controller

The SMB$SEND_TO_JOBCTL routine is used by your symbiont to send messages to the job controller. Three types of message can be sent: request-completion messages, task-completion messages, and task-status messages.

FORMAT

SMB$SEND_TO_JOBCTL *stream [,request] [,accounting] [,checkpoint] [,device_status] [,error]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

stream

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

Stream number specifying the stream to which the message refers. The **stream** argument is the address of a longword containing the number of the stream to which the message refers.

request

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Request code identifying the request being completed. The **request** argument is the address of a longword containing the code that identifies the request that has been completed.

The code usually corresponds to the code the job controller passed to the symbiont by means of a call to SMB$READ_MESSAGE. But the symbiont can also initiate task-completion and task-status messages that are not in response to a request. (See the DESCRIPTION section.)

accounting

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

Accounting information about a task. The **accounting** argument is the address of a descriptor pointing to the accounting information about a task. Note that this structure is passed by descriptor and not by reference.

See the description of the SMBMSG$K_ACCOUNTING_DATA item for more information about this accounting information.

The following diagram depicts the contents of the 16-byte structure.

31	0
NUMBER OF PAGES PRINTED FOR THE JOB	
NUMBER OF READS FROM DISK OR TAPE	
NUMBER OF WRITES TO THE PRINTING DEVICE	
UNUSED	

ZK-2012-84

checkpoint

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

Checkpoint data about the currently executing task. The **checkpoint** argument is the address of the descriptor that points to checkpointing information that relates to the status of a task. When the symbiont sends this information to the job controller, the job controller saves it in the queue's data file. When a restart-from-checkpoint request is executed for the queue, the job controller retrieves the checkpointing information from the queue's data file and sends it to the symbiont in the SMBMSG$K_CHECKPOINT_DATA item that accompanies a SMBMSG$K_START_TASK request.

Print symbionts can use the checkpointing information to reposition the input file to the point corresponding to the page being output when the last checkpoint was taken. Other symbionts might use checkpoint information to specify restart information for partially completed tasks.

Note: **Because each checkpoint causes information to be written into the job controller's queue-data file, taking a checkpoint incurs significant overhead. Use caution in regard to the size and frequency of checkpoints. When determining how often to checkpoint, weigh processor and file-system overhead against the convenience of restarting.**

device_status

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

Status of the device served by the symbiont. The **device_status** argument is the address of a longword passed to the job controller, which contains the status of the device to which the symbiont is connected.

This longword contains a longword bit vector, each bit of which specifies device-status information. The $SMBDEF macro defines these device-status bits. The following table describes each bit.

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

error

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

Condition codes returned by the requested task. The **error** argument is the address of a vector of longword condition codes. The first longword contains the number of longwords following it.

If the low bit of the first condition code is clear, the job controller aborts further processing of the job. Output of any remaining files, copies of files, or copies of the job is canceled. In addition, the job controller saves up to three condition values in the queue's data file. The first condition value is included in the job-accounting record that is written to the system's accounting file (SYS$MANAGER:ACCOUNTNG.DAT).

DESCRIPTION

The symbiont uses the SMB$SEND_TO_JOBCTL routine to send messages to the job controller.

Most messages the symbiont sends to the job controller are responses to requests made by the job controller. These responses inform the job controller that the request has been completed, either successfully or with an error. When the symbiont sends the message, it usually indicates that the request has been completed.

In such messages, the **request** argument corresponds to the function code of the request that has been completed. Thus, if the job controller sends a request using the SMBMSG$K_START_TASK code, the symbiont responds by sending a SMB$SEND_TO_JOBCTL message using SMBMSG$K_START_TASK as the **request** argument.

The responses to some requests use additional arguments to send more information in addition to the request code. The following table shows which additional arguments are allowed in response to each different request.

Request	Arguments
SMBMSG$K_START_STREAM	request
	device_status
	error
SMBMSG$K_STOP_STREAM	request
SMBMSG$K_RESET_STREAM	request
SMBMSG$K_START_TASK	request
SMBMSG$K_PAUSE_TASK	request
SMBMSG$K_RESUME_TASK	request
SMBMSG$K_STOP_TASK	request
	error[1]

[1]This is usually the value specified in the SMBMSG$K_STOP_CONDITION item that was sent by the job controller with the SMBMSG$K_STOP_TASK request.

In addition to responding to requests from the job controller, the symbiont can send other messages to the job controller. If the symbiont sends a message that is not a response to a request, it uses either the SMBMSG$K_TASK_COMPLETE or SMBMSG$K_TASK_STATUS code. Following are the additional arguments that you can use with the messages identified by these codes.

Code	Arguments
SMBMSG$K_TASK_COMPLETE	request
	accounting
	error
SMBMSG$K_TASK_STATUS	request
	checkpoint
	device_status

The symbiont uses the SMB$K_TASK_STATUS message to update the job controller on the status of a task during the processing of that task. The checkpoint information passed to the job controller with this message permits the job controller to restart an interrupted task from an appropriate point. The device-status information permits the symbiont to report changes in device's status (device stalled, for example).

The symbiont can use the SMB$K_TASK_STATUS message to request that the job controller send a stop-stream request. It does this by setting the stop-stream bit in the **device-status** argument.

The symbiont can also use the SMB$K_TASK_STATUS message to notify the job controller that the symbiont has paused in processing a task. It does so by setting the pause-task bit in the **device-status** argument.

The symbiont uses the SMB$K_TASK_COMPLETE message to signal the completion of a task. Note that, when the symbiont receives a START_TASK request, it responds by sending a SMB$SEND_TO_JOBCTL message with SMBSMG$K_START_TASK as the **request** argument. This response means that the symbiont has started the task; it does not mean the task has been completed. When the symbiont has completed a task, it sends a SMB$SEND_TO_JOBCTL message with SMBSMG$K_TASK_COMPLETE as the **request** argument.

Optionally, the symbiont can specify accounting information when sending a task-completion message. The accounting statistics accumulate to give a total for the job when the job is completed.

Also, if the symbiont is aborting the task because of a symbiont-detected error, you can specify up to three condition values in the **error** argument. Aborting a task causes the remainder of the job to be aborted.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine completed successfully.

This routine also returns any condition value returned by the $QIO system service and the LIB$GET_VM routine.

12 Sort/Merge (SOR) Routines

12.1 Introduction to SOR Routines

The SOR routines allow you to integrate a sort or merge operation into a program application. Using these callable routines, you can process some records, sort or merge them, and then process them again.

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

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

Note: You can still call SOR$DO_MERGE (from VMS Version 3.0) as the equivalent of SOR$END_SORT; you can still call SOR$INIT_MERGE and SOR$INIT_SORT (from VMS Version 3.0) as the equivalent of SOR$BEGIN_SORT and SOR$BEGIN_MERGE. However, for any new programs that you are creating, you are advised to use SOREND_SORT, SORBEGIN_SORT, and SOR$BEGIN_MERGE.

You can call these SOR routines from any language that supports the VAX Procedure Calling and Condition Handling Standard.

After being called, each of these routines performs its function and returns control to a program. It also returns a 32-bit condition code value indicating success or error, which a program can test to determine success or failure conditions.

12.1.1 Arguments to SOR Routines

For a sort operation, the arguments to the SOR routines provide SORT with file specifications, key information, and instructions about the sorting process. For a merge operation, the arguments to the SOR routines provide MERGE with the number of input files, input and output file specifications, record information, key information, and input routine information.

There are both mandatory and optional arguments. The mandatory arguments appear first in the argument list. You must specify all arguments in the order in which they are positioned in the argument list, separating each with a comma. Pass a zero by value to specify any optional arguments that you are omitting from within the list. You can end the argument list any time after specifying all the mandatory and desired optional arguments.

12.1.2 Interfaces to SOR Routines

You can submit data to the SOR routines as complete files or as single records. When your program submits one or more files to SORT or MERGE, which then creates one sorted or merged output file, you are using the file interface. When your program submits records one at a time and then receives the ordered records one at a time, you are using the record interface.

You can combine the file interface with the record interface by submitting files on input and receiving the ordered records on output, or by releasing records on input and writing the ordered records to a file on output. Combining the two interfaces provides greater flexibility. If you use the record interface on input, you can process the records before they are sorted; if you use the record interface on output, you can process the records after they are sorted.

The SOR routines used and the order in which they are called depend on the type of interface used in a sorting or merging operation. The following sections detail the calling sequence for each of the interfaces. Note, however, that if you use the SOR$STAT routine, it must be called before any other SOR routine.

12.1.2.1 Sort Operation Using File Interface

For a sort operation using the file interface, pass the input and output file specifications to SORT by calling SOR$PASS_FILES. You must call SOR$PASS_FILES for each input file specification. Pass the output file specification in the first call. If no input files are specified before the call to SOR$BEGIN_SORT, the record interface is used for input; if no output file is specified, the record interface is used for output.

Next, call SOR$BEGIN_SORT to pass instructions about keys and sort options. At this point, you must indicate if you want to use your own key comparison routine. SORT automatically generates a key comparison routine that is efficient for key data types; however, you may want to provide your own comparison routine to handle special sorting requirements. (For example, you may want names beginning with "Mc" and "Mac" to be placed together.) If you use your own key comparison routine, you must pass its address with the **user_compare** argument.

Call SOR$SORT_MERGE to execute the sort and direct the sorted records to the output file. Finally, call SOR$END_SORT to end the sort and release resources. The SOR$END_SORT routine may be called at any time to abort a sort, or to merge and release all resources allocated to the sort or merge process.

12.1.2.2 Sort Operation Using Record Interface

For a sort operation using the record interface, first call SOR$BEGIN_SORT. As in the file interface, this routine sets up work areas and passes arguments that define keys and sort options. Note that, if you use the record interface, you must use a record-sorting process (not a tag, address, or index process).

Next, call SOR$RELEASE_REC to release a record to SORT. Call SOR$RELEASE_REC once for each record to be released. After all records have been passed to SORT, call SOR$SORT_MERGE to perform the sorting.

After the sort has been performed, call SOR$RETURN_REC to return a record from the sort operation. Call this routine once for each record to be returned. Finally, call the last routine, SOR$END_SORT, to complete the sort operation and release resources.

12.1.2.3 Merge Operation Using File Interface

For a merge operation using the file interface, pass the input and output file specifications to MERGE by calling SOR$PASS_FILES. You can merge up to 10 input files by calling SOR$PASS_FILES once for each file. Pass the file specification for the merged output file in the first call. If no input files are specified before the call to SOR$BEGIN_MERGE, the record interface is used for input; if no output file is specified, the record interface is used for output.

Next, to execute the merge, call SOR$BEGIN_MERGE to pass key information and merge options. At this point, you must indicate if you want to use your own key comparison routine tailored to your data. Finally, call SOR$END_SORT to release resources.

12.1.2.4 Merge Operation Using Record Interface

For a merge operation using the record interface, first call SOR$BEGIN_MERGE. As in the file interface, this routine passes arguments that define keys and merge options. It also issues the first call to the input routine, which you must create, to begin releasing records to the merge.

Next, call SOR$RETURN_REC to return the merged records to your program. You must call this routine once for each record to be returned. SOR$RETURN_REC continues to call the input routine. MERGE, unlike SORT, does not need to hold all the records before it can begin returning them in the desired order. The releasing, merging, and returning of records all take place in this phase of the merge.

Finally, after all the records have been returned, call the last routine, SOR$END_SORT, to clean up and release resources.

12.1.3 Reentrancy

The SOR routines are reentrant; that is, a number of sort or merge operations can be active at the same time. Thus, a program does not need to finish one sort or merge operation before beginning another. For example, reentrancy allows you to perform multiple sorts on a file such as a mailing list and to create several output files, one with the records sorted by name, another sorted by state, another sorted by zip code, and so on.

The context argument, which may optionally be passed with any of the SOR routines, distinguishes among multiple sort or merge operations. When using multiple sort or merge operations, the context argument is required. On the first call, the context longword must be zero. It is then set (by SORT/MERGE) to a value identifying the sort or merge operation. Additional calls to the same sort or merge operation must pass the same context longword. The SOR$END_SORT routine clears the context longword.

12.2 Examples of Using SOR Routines

Example 12–1 is a VAX FORTRAN program demonstrating a merge operation using a record interface.

Example 12–1 Using SOR Routines to Perform a Merge Using Record Interface in a VAX FORTRAN Program

```
       FORTRAN Program
C...
C...   This program demonstrates the FORTRAN calling sequences
C...   for the merge record interface.
C...
C
C      THE INPUT FILES ARE LISTED BELOW.
C
C              INFILE1.DAT
C
C 1 BBBBBBBBBB REST OF DATA IN RECORD................................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD................................END OF RECORD
C
C              INFILE2.DAT
C
C 1 AAAAAAAAAA REST OF DATA IN RECORD................................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD................................END OF RECORD
C
C              INFILE3.DAT
C
C 1 TTTTTTTTTT REST OF DATA IN RECORD................................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD................................END OF RECORD
C
C              FOROUT.DAT
C
C 1 AAAAAAAAAA REST OF DATA IN RECORD................................END OF RECORD
C 1 BBBBBBBBBB REST OF DATA IN RECORD................................END OF RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD................................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD................................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD................................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD................................END OF RECORD
C
C
C................................................................................
C
C
```

Example 12–1 Cont'd. on next page

Example 12–1 (Cont.) Using SOR Routines to Perform a Merge Using Record Interface in a VAX FORTRAN Program

```
        IMPLICIT INTEGER (A-Z)

        CHARACTER*80 REC                        ! A record.

        EXTERNAL READ_REC                       ! Routine to read a record.
        EXTERNAL KOMPAR                         ! Routine to compare records.
        EXTERNAL SS$_ENDOFFILE                  ! System end-of-file value

        INTEGER*4 SOR$BEGIN_MERGE               ! SORT/MERGE function names
        INTEGER*4 SOR$RETURN_REC
        INTEGER*4 SOR$END_SORT
        INTEGER*4 ISTAT                         ! storage for SORT/MERGE function value
        INTEGER*4 LENGTH                        ! length of the returned record
        INTEGER*2 LRL                           ! Longest Record Length (LRL)

        LOGICAL*1 ORDER                         ! #files to merge (merge order)

        DATA ORDER,LRL/3,80/                    ! Order of the merge=3,LRL=80
C...
C...    First open all the input files.
C...
        OPEN (UNIT=10, FILE='INFILE1.DAT',TYPE='OLD',READONLY,
     *  FORM='FORMATTED')
        OPEN (UNIT=11, FILE='INFILE2.DAT',TYPE='OLD',READONLY,
     *  FORM='FORMATTED')
        OPEN (UNIT=12, FILE='INFILE3.DAT',TYPE='OLD',READONLY,
     *  FORM='FORMATTED')
C
C...    Open the output file.
C
        OPEN (UNIT=8, FILE='TEMP.TMP', TYPE='NEW')
C...
C...    Initialize the merge.  Pass the merge order, the largest
C...    record length, the compare routine address, and the
C...    input routine address.
C...
        ISTAT = SOR$BEGIN_MERGE (,LRL,,ORDER,
     *  KOMPAR,,READ_REC)
        IF (.NOT. ISTAT) GOTO 10        ! Check for error.


C...
C...    Now loop getting merged records.  SOR$RETURN_REC will
C...    call READ_REC when it needs input.
C...
5       ISTAT = SOR$RETURN_REC (REC, LENGTH)
        IF (ISTAT .EQ. %LOC(SS$_ENDOFFILE)) GO TO 30    ! Check for end of file.
        IF (.NOT. ISTAT) GO TO 10       ! Check for error.

        WRITE(8,200) REC                        ! Output the record.
200     FORMAT(' ',A)
        GOTO 5                                  ! And loop back.
C...
C...    Now tell SORT that we are all done.
C...

30      ISTAT = SOR$END_SORT()
        IF (.NOT. ISTAT) GOTO 10        ! Check for error.
        CALL EXIT
```

Example 12–1 Cont'd. on next page

Example 12–1 (Cont.) Using SOR Routines to Perform a Merge Using Record Interface in a VAX FORTRAN Program

```
C...
C...    Here if an error occurred.  Write out the error status
C...    and exit.
C...
10      WRITE(8,201)ISTAT
201     FORMAT(' ?ERROR CODE', I20)
        CALL EXIT
        END

        FUNCTION READ_REC (RECX, FILE, SIZE)
C...
C...    This routine reads a record from one of the input files
C...    for merging.  It will be called by SOR$BEGIN_MERGE and by
C...    SOR$RETURN_REC.
C...    Parameters:
C...
C...            RECX.wcp.ds     character buffer to hold the record after
C...                            it is read in.
C...
C...            FILE.rl.r       indicates which file the record is
C...                            to be read from.  1 specifies the
C...                            first file, 2 specifies the second
C...                            etc.
C...
C...            LENGTH.wl.r     is the actual number of bytes in
C...                            the record.  This is set by READ_REC.
C...
        IMPLICIT INTEGER (A-Z)

        PARAMETER MAXFIL=10                     ! Max number of files.

        EXTERNAL SS$_ENDOFFILE                  ! End of file status code.
        EXTERNAL SS$_NORMAL                     ! Success status code.

        LOGICAL*1 FILTAB(MAXFIL)
        CHARACTER*(80) RECX                     ! MAX LRL =80

        DATA FILTAB/10,11,12,13,14,15,16,17,18,19/ ! Table of I/O unit numbers.

        READ_REC = %LOC(SS$_ENDOFFILE)        ! Give end of file return
        IF (FILE .LT. 1 .OR. FILE .GT. MAXFIL) RETURN   !   if illegal call.

        READ (FILTAB(FILE), 100, ERR=75, END=50) RECX   ! Read the record.
100     FORMAT(A)

        READ_REC = %LOC(SS$_NORMAL)             ! Return success code.
        SIZE = LEN (RECX)                       ! Return size of record.
        RETURN

C...    Here if end of file.
50      READ_REC = %LOC(SS$_ENDOFFILE)          ! Return "end of file" code.
        RETURN

C...    Here if error while reading
75      READ_REC = 0
        SIZE = 0
        RETURN
        END
```

Example 12–1 Cont'd. on next page

Example 12–1 (Cont.) Using SOR Routines to Perform a Merge Using Record Interface in a VAX FORTRAN Program

```
        FUNCTION KOMPAR (REC1,REC2)
C...
C...    This routine compares two records.  It returns -1
C...    if the first record is smaller than the second,
C...    0 if the records are equal, and 1 if the first record
C...    is larger than the second.
C...
        PARAMETER KEYSIZ=10

        IMPLICIT INTEGER (A-Z)

        LOGICAL*1 REC1(KEYSIZ),REC2(KEYSIZ)

        DO 20 I=1,KEYSIZ
        KOMPAR = REC1(I) - REC2(I)
        IF (KOMPAR .NE. 0) GOTO 50
20      CONTINUE

        RETURN

50      KOMPAR = ISIGN (1, KOMPAR)
        RETURN
        END
```

Example 12–2 is a VAX FORTRAN program demonstrating a sort operation using a file interface on input and a record interface on output.

Example 12–2 Using SOR Routines to Sort Using Mixed Interface in a VAX FORTRAN Program

```
Program

        PROGRAM CALLSORT
C
C
C       This is a sample FORTRAN program that calls the SOR
C       routines using the file interface for input and the
C       record interface for output.  This program requests
C       a record sort of the file 'R010SQ.DAT'  and  writes
C       the records to SYS$OUTPUT.  The key is  an  80-byte
C       character ascending  key starting in  position 1 of
C       each record.
C
C       A short version of the input and output files follows:
C
C                       Input file R010SQ.DAT
C 1 BBBBBBBBBB REST OF DATA IN RECORD...............................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD...............................END OF RECORD
C 1 AAAAAAAAAA REST OF DATA IN RECORD...............................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD...............................END OF RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD...............................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD...............................END OF RECORD
C 1 QQQQQQQQQQ REST OF DATA IN RECORD...............................END OF RECORD
C 2 AAAAAAAAAA REST OF DATA IN RECORD...............................END OF RECORD
C 1 UUUUUUUUUU REST OF DATA IN RECORD...............................END OF RECORD
C 2 QQQQQQQQQQ REST OF DATA IN RECORD...............................END OF RECORD
C
```

Example 12–2 Cont'd. on next page

Example 12–2 (Cont.) Using SOR Routines to Sort Using Mixed Interface in a VAX FORTRAN Program

```
C                       Output file SYS$OUTPUT
C
C 1 AAAAAAAAAA REST OF DATA IN RECORD..............................END OF RECORD
C 1 BBBBBBBBBB REST OF DATA IN RECORD..............................END OF RECORD
C 1 QQQQQQQQQQ REST OF DATA IN RECORD..............................END OF RECORD
C 1 TTTTTTTTTT REST OF DATA IN RECORD..............................END OF RECORD
C 1 UUUUUUUUUU REST OF DATA IN RECORD..............................END OF RECORD
C 2 AAAAAAAAAA REST OF DATA IN RECORD..............................END OF RECORD
C 2 BBBBBBBBBB REST OF DATA IN RECORD..............................END OF RECORD
C 2 QQQQQQQQQQ REST OF DATA IN RECORD..............................END OF RECORD
C 2 TTTTTTTTTT REST OF DATA IN RECORD..............................END OF RECORD
C 2 UUUUUUUUUU REST OF DATA IN RECORD..............................END OF RECORD
C
C-----------------------------------------------------------------------------
C
C       Define external functions and data.
C
        CHARACTER*80 RECBUF
        CHARACTER*10 INPUTNAME          !Input file name
        INTEGER*2 KEYBUF(5)             !Key definition buffer
        INTEGER*4 SOR$PASS_FILES        !SORT function names
        INTEGER*4 SOR$BEGIN_SORT
        INTEGER*4 SOR$SORT_MERGE
        INTEGER*4 SOR$RETURN_REC
        INTEGER*4 SOR$END_SORT
        INTEGER*4 ISTATUS               !Storage for SORT function value
        EXTERNAL SS$_ENDOFFILE
        EXTERNAL DSC$K_DTYPE_T
        EXTERNAL SOR$GK_RECORD
        INTEGER*4 SRTTYPE

C
C       Initialize data -- first the file names, then the key buffer for
C       one 80-byte character key starting in position 1, 3 work files,
C       and a record sort process.
C
        DATA INPUTNAME/'R010SQ.DAT'/
        KEYBUF(1) = 1
        KEYBUF(2) = %LOC(DSC$K_DTYPE_T)
        KEYBUF(3) = 0
        KEYBUF(4) = 0
        KEYBUF(5) = 80
        SRTTYPE = %LOC(SOR$GK_RECORD)

C
C       Call the SORT -- each call is a function.
C
C
C       Pass SORT the file names.
C
        ISTATUS = SOR$PASS_FILES(INPUTNAME)
        IF (.NOT. ISTATUS) GOTO 10
C
C       Initialize the work areas and keys.
C
        ISTATUS = SOR$BEGIN_SORT(KEYBUF,,,,,SRTTYPE,%REF(3))
        IF (.NOT. ISTATUS) GOTO 10
```

Example 12–2 Cont'd. on next page

Example 12–2 (Cont.) Using SOR Routines to Sort Using Mixed Interface in a VAX FORTRAN Program

```
C
C       Sort the records.
C
        ISTATUS = SOR$SORT_MERGE( )
        IF (.NOT. ISTATUS) GOTO 10
C
C       Now retrieve the individual records and display them.
C
5       ISTATUS = SOR$RETURN_REC(RECBUF)
        IF (.NOT. ISTATUS) GOTO 6
        ISTATUS = LIB$PUT_OUTPUT(RECBUF)
        GOTO 5
6       IF (ISTATUS .EQ. %LOC(SS$_ENDOFFILE)) GOTO 7
        GOTO 10
C
C       Clean up the work areas and files.
C
7       ISTATUS = SOR$END_SORT()
        IF (.NOT. ISTATUS) GOTO 10
        STOP 'SORT SUCCESSFUL'
10      STOP 'SORT UNSUCCESSFUL'
        END
```

Example 12–3 is a VAX Pascal program demonstrating a merge operation using a file interface.

Example 12–3 Using SOR Routines to Merge Three Input Files in a VAX Pascal Program

```
Program

(* This program merges three input files, (IN_FILE.DAT,
 IN_FILE2.DAT IN_FILE3.DAT), and creates one merged output file.  *)

program mergerecs( output, in_file1, in_file2, in_file3, out_file );

CONST
   SS$_NORMAL = 1;
   SS$_ENDOFFILE = %X870;
   SOR$GK_RECORD = 1;
   SOR$M_STABLE = 1;
   SOR$M_SEQ_CHECK = 4;
   SOR$M_SIGNAL = 8;
   DSC$K_DTYPE_T = 14;

TYPE
   $UBYTE = [BYTE] 0..255;
   $UWORD = [WORD] 0..65535;

const
    num_of_keys = 1;
    merge_order = 3;
    lrl         = 131;

    ascending   = 0;
    descending  = 1;
```

Example 12–3 Cont'd. on next page

Example 12–3 (Cont.) Using SOR Routines to Merge Three Input Files in a VAX Pascal Program

```
type
    key_buffer_block=
        packed record
        key_type:       $uword;
        key_order:      $uword;
        key_offset:     $uword;
        key_length:     $uword;
        end;

   key_buffer_type=
        packed record
        key_count:      $uword;
        blocks:         packed array[1..num_of_keys] of key_buffer_block;
        end;

   record_buffer =      packed array[1..lrl] of char;

   record_buffer_descr =
        packed record
        length: $uword;
        dummy:  $uword;
        addr:   ^record_buffer;
        end;

var
   in_file1,
   in_file2,
   in_file3,
   out_file:    text;
   key_buffer:  key_buffer_type;
   rec_buffer:  record_buffer;
   rec_length:  $uword;
   status:      integer;
   i:           integer;

function sor$begin_merge(
        var buffer:     key_buffer_type;
        lrl:            $uword;
        mrg_options:    integer;
        merge_order:    $ubyte;
        %immed cmp_rtn: integer := 0;
        %immed eql_rtn: integer := 0;
        %immed [unbound] function
            read_record(
                var rec:        record_buffer_descr;
                var filenumber: integer;
                var recordsize: $uword): integer
        ): integer; extern;

function sor$return_rec(
        %stdescr rec:   record_buffer;
        var rec_size:   $uword
        ): integer; extern;
```

Example 12–3 Cont'd. on next page

Example 12–3 (Cont.) Using SOR Routines to Merge Three Input Files in a VAX Pascal Program

```
function sor$end_sort: integer; extern;

procedure sys$exit( %immed status : integer ); extern;

function read_record(
        var rec:        record_buffer_descr;
        var filenumber: integer;
        var recordsize: $uword
        ): integer;

procedure readone( var filename: text );
begin
recordsize := 0;
if eof(filename)
then
    read_record := ss$_endoffile
else
    begin
    while not eoln(filename) and (recordsize < rec.length) do
        begin
        recordsize := recordsize + 1;
        read(filename,rec.addr^[recordsize]);
        end;
    readln(filename);
    end;
end;

begin
read_record := ss$_normal;
case filenumber of
    1: readone(in_file1);
    2: readone(in_file2);
    3: readone(in_file3);
    otherwise
        read_record := ss$_endoffile;
    end;
end;

procedure initfiles;
begin
open( in_file1, 'infile1.dat', old );
open( in_file2, 'infile2.dat', old );
open( in_file3, 'infile3.dat', old );
open( out_file, 'temp.tmp' );
reset( in_file1 );
reset( in_file2 );
reset( in_file3 );
rewrite( out_file );
end;

procedure error( status : integer );
begin
writeln( 'merge unsuccessful.  status=%x', status:8 hex );
sys$exit(status);
end;
```

Example 12–3 Cont'd. on next page

Example 12–3 (Cont.) Using SOR Routines to Merge Three Input Files in a VAX Pascal Program

```
begin

with key_buffer do
    begin
    key_count := 1;
    with blocks[1] do
        begin
        key_type := dsc$k_dtype_t;
        key_order := ascending;
        key_offset := 0;
        key_length := 5;
        end;
    end;

initfiles;

status := sor$begin_merge( key_buffer, lrl,
        sor$m_seq_check + sor$m_signal,
        merge_order, 0, 0, read_record );

repeat
    begin
    rec_length := 0;
    status := sor$return_rec( rec_buffer, rec_length );
    if odd(status)
    then
        begin
        for i := 1 to rec_length do write(out_file, rec_buffer[i]);
        writeln(out_file);
        end;
    end
until not odd(status);

if status <> ss$_endoffile then error(status);

status := sor$end_sort;
if not odd(status) then error(status);

writeln( 'merge successful.' );

end.
```

Example 12–4 is a VAX Pascal program demonstrating a sort operation using a record interface.

Example 12–4 Using SOR Routines to Sort Records from Two Input Files in a VAX Pascal Program

```
PASCAL Program

PROGRAM FILETORECORDSORT (OUTPUT,SORTOUT);

(*      This program calls SOR routines to read and sort records from
        two input files, (PASINPUT1.DAT and PASINPUT2.DAT) and to return
        sorted records to this program to be written to the output file,
        (TEMP.TMP).  *)

(*      Declarations for external status codes, and data structures, such as
        the types $UBYTE (an unsigned byte) and $UWORD (an unsigned word). *)

CONST
   SS$_NORMAL = 1;
   SS$_ENDOFFILE = %X870;
   SOR$GK_RECORD = 1;
   SOR$M_STABLE = 1;
   SOR$M_SEQ_CHECK = 4;
   SOR$M_SIGNAL = 8;
   DSC$K_DTYPE_T = 14;

TYPE
   $UBYTE = [BYTE] 0..255;
   $UWORD = [WORD] 0..65535;

CONST
   Numberofkeys = 1 ;    (* Number of keys for this sort *)
   LRL = 131 ;           (* Longest Record Length for output records *)

(* Key orders *)

   Ascending = 0 ;
   Descending = 1 ;


TYPE
   Keybufferblock= packed record
                   Keytype : $UWORD ;
                   Keyorder : $UWORD ;
                   Keyoffset : $UWORD ;
                   Keylength : $UWORD
                   end ;

(* The keybuffer. Note that the field buffer is a one-component array in
   this program. This type definition would allow a multikeyed sort. *)

   Keybuffer= packed record
              Numkeys : $UWORD ;
              Blocks : packed array[1..Numberofkeys] OF Keybufferblock
              end ;

(* The record buffer. This buffer will be used to hold the returned
      records from SORT. *)

   Recordbuffer = packed array[1..LRL] of char ;

(* Name type for input and output files. A necessary fudge for %stdescr
   mechanism.  *)

   nametype= packed array[1..13] of char ;
```

Example 12–4 Cont'd. on next page

Example 12–4 (Cont.) Using SOR Routines to Sort Records from Two Input Files in a VAX Pascal Program

```
VAR
   Sortout : text ;            (* the output file *)
   Buffer : Keybuffer ;        (* the actual keybuffer *)
   Sortoptions : integer ;     (* flag for sorting options *)
   Sorttype : $UBYTE ;         (* sorting process *)
   Numworkfiles : $UBYTE ;     (* number of work files *)
   Status : integer ;          (* function return status code *)
   Rec : Recordbuffer ;        (* a record buffer *)
   Recordlength : $UWORD ;     (* the returned record length *)
   Inputname:  nametype ;      (* input file name *)
   i : integer ;               (* loop control variable *)

(* function and procedure declarations *)

(* Declarations of SORT functions *)
(* Note that the following SORT routine declarations
        do not use all of the possible routine parameters. *)
(* The parameters used MUST have all preceding parameters specified,
        however. *)

FUNCTION SOR$PASS_FILES
   (%STDESCR Inname : nametype )
   : INTEGER ; EXTERN ;

FUNCTION SOR$BEGIN_SORT(
    VAR Buffer : Keybuffer ;
    Lrlen : $UWORD ;
    VAR Sortoptions : INTEGER ;
    %IMMED Filesize : INTEGER ;
    %IMMED Usercompare : INTEGER ;
    %IMMED Userequal : INTEGER ;
    VAR Sorttype : $UBYTE ;
    VAR Numworkfiles : $UBYTE )
    : INTEGER ; EXTERN ;

FUNCTION SOR$SORT_MERGE
   : INTEGER ; EXTERN ;

FUNCTION SOR$RETURN_REC(
   %STDESCR  Rec : Recordbuffer ;
   VAR Recordsize : $UWORD )
   : INTEGER ; EXTERN ;

FUNCTION SOR$END_SORT
   : INTEGER ; EXTERN ;

(* End of the SORT function declarations *)
```

Example 12–4 Cont'd. on next page

Example 12–4 (Cont.) Using SOR Routines to Sort Records from Two Input Files in a VAX Pascal Program

```
(* The CHECKSTATUS routine checks the return status for errors. *)
(* If there is an error, write an error message and exit via sys$exit *)
PROCEDURE CHECKSTATUS( var status : integer ) ;

        procedure sys$exit( status : integer ) ; extern ;

begin           (* begin checkstatus *)
   if odd(status) then
        begin
        writeln( ' SORT unsuccessful. Error status = ', status:8 hex ) ;
        SYS$EXIT( status ) ;
        end ;
end ;           (* end checkstatus *)

(* end function and routine declarations *)


BEGIN   (* begin the main routine *)

(* Initialize data for one 8-byte character key, starting at record
   offset 0, 3 work files, and the record sorting process *)

Inputname := 'PASINPUT1.DAT' ;
WITH Buffer DO
   BEGIN
   Numkeys := 1;
   WITH Blocks[1] DO
      BEGIN
      Keytype := DSC$K_DTYPE_T ;          (* Use VMS descriptor data types to
                                                define SORT data types. *)
      Keyorder := Ascending ;
      Keyoffset := 0 ;
      Keylength := 8 ;
      END;
   END;

Sorttype := SOR$GK_RECORD ;               (* Use the global SORT constant to
                                             define the sort process. *)
Sortoptions := SOR$M_STABLE ;             (* Use the global SORT constant to
                                             define the stable sort option. *)
Numworkfiles := 3 ;

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

(* pass the first filename to SORT *)
Status := SOR$PASS_FILES( Inputname ) ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* pass the second filename to SORT *)
Inputname := 'PASINPUT2.DAT' ;

Status := SOR$PASS_FILES( Inputname ) ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* initialize work areas and keys *)
Status := SOR$BEGIN_SORT( Buffer, 0, Sortoptions, 0, 0, 0,
                                Sorttype, Numworkfiles ) ;
```

Example 12–4 Cont'd. on next page

Example 12–4 (Cont.) Using SOR Routines to Sort Records from Two Input Files in a VAX Pascal Program

```
(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* sort the records *)
Status := SOR$SORT_MERGE ;

(* Check status for error. *)
CHECKSTATUS( Status ) ;

(* Ready output file for writing returned records from SORT. *)
OPEN( SORTOUT, 'TEMP.TMP' ) ;
REWRITE( SORTOUT ) ;

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

   if odd( Status )
   then                 (* if successful, write record to output file. *)
        begin
        for i := 1 to Recordlength do
           write( sortout, Rec[i] ) ;   (* write each character *)
        writeln (sortout) ;                     (* end output line *)
        end;
UNTIL not odd( Status ) ;

(* If there was just no more data to be returned (eof) continue, otherwise
        exit with an error. *)
if Status <> SS$_ENDOFFILE then
   CHECKSTATUS( Status ) ;

(* The sort has been successful to this point. *)

(* Close the output file *)
CLOSE( sortout ) ;

(* clean up work areas and files *)
Status := SOR$END_SORT ;

(* Check status for error. *)
CHECKSTATUS( Status );

WRITELN ('SORT SUCCESSFUL') ;

END.
```

12.3 SOR Routines

The following pages describe the individual SOR routines.

SOR$BEGIN_MERGE Initialize a Merge Operation

The SOR$BEGIN_MERGE routine initializes the merge operation by opening the input and output files and by providing the number of input files, the key specifications, and the merge options.

FORMAT

SOR$BEGIN_MERGE *[key-buffer] [,lrl] [,options] [,merge_order] [,user_compare] [,user_equal] [,user_input] [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

key_buffer

VMS usage:	**vector_word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by reference**

Array of words describing the keys on which you plan to merge. The **key_buffer** argument is the address of an array containing the key descriptions.

The first word of this array contains the number of keys described (up to 255). Following the first word, each key is described (in order of priority) in blocks of four words. The four words specify the key's data type, order, offset, and length, respectively.

The first word of the block specifies the key's data type. The following data types are accepted:

DSC$K_DTYPE_Z	Unspecified (uninfluenced by collating sequence)
DSC$K_DTYPE_B	Byte integer (signed)
DSC$K_DTYPE_BU	Byte (unsigned)
DSC$K_DTYPE_W	Word integer (signed)
DSC$K_DTYPE_WU	Word (unsigned)
DSC$K_DTYPE_L	Longword integer (signed)
DSC$K_DTYPE_LU	Longword (unsigned)
DSC$K_DTYPE_Q	Quadword integer (signed)

DSC$K_DTYPE_QU	Quadword (unsigned)
DSC$K_DTYPE_O	Octaword integer (signed)
DSC$K_DTYPE_OU	Octaword (unsigned)
DSC$K_DTYPE_F	Single-precision floating
DSC$K_DTYPE_D	Double-precision floating
DSC$K_DTYPE_G	G-format floating
DSC$K_DTYPE_H	H-format floating
DSC$K_DTYPE_T	Text (may be influenced by collating sequence)
DSC$K_DTYPE_NU	Numeric string, unsigned
DSC$K_DTYPE_NL	Numeric string, left separate sign
DSC$K_DTYPE_NLO	Numeric string, left overpunched sign
DSC$K_DTYPE_NR	Numeric string, right separate sign
DSC$K_DTYPE_NRO	Numeric string, right overpunched sign
DSC$K_DTYPE_NZ	Numeric string, zoned sign
DSC$K_DTYPE_P	Packed decimal string

The VAX Procedure Calling and Condition Handling Standard, documented in Chapter 2 in the *Introduction to VMS System Routines*, describes each of these data types.

The second word of the block specifies the key order: *0* for ascending order, *1* for descending order. The third word of the block specifies the relative offset of the key in the record. (Note that the first byte in the record is at position *0*.) The fourth word of the block specifies the key length in bytes (in digits for packed decimal—DSC$K_DTYPE_P).

If you do not specify the **key_buffer** argument, you must pass either a key comparison routine or use a specification file to define the key.

lrl

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Length of the longest record that will be released for merging. The **lrl** argument is the address of a word containing the length. If the input file is on a disk, this argument is not required. It is required when you use the record interface. For VFC records, this length must include the length of the fixed-length portion of the record.

options

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

Flags that identify merge options. The **options** argument is the address of a longword bit mask whose settings determine the merge options selected. The following table lists and describes the bit mask values available.

Flag	Description
SOR$M_STABLE	Keeps records with equal keys in the same order as they appeared on input.
SOR$M_EBCDIC	Orders ASCII character keys according to EBCDIC collating sequence. No translation takes place.
SOR$M_MULTI	Orders character keys according to the multinational collating sequence, which collates the international character set.
SOR$M_NOSIGNAL	Returns a status code instead of signaling errors. (This is the default behavior.)
SOR$M_NODUPS	Omits records with duplicate keys. You cannot use this option if you specify your own equal-key routine.
SOR$M_SEQ_CHECK	Requests an "out of order" error return if an input file is not already in sequence. By default, this check is not done. You must request sequence checking if you specify an equal-key routine.

All other bits in the longword are reserved and must be zero.

merge_order

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Number of input streams to be merged. The **merge_order** argument is the address of a byte containing the number of files (1 through 10) to be merged. When you use the record interface on input, this argument is required.

user_compare

VMS usage: **procedure**
type: **procedure entry mask**
access: **function call**
mechanism: **by reference**

Routine that compares records to determine their merge order. The **user_compare** argument is the address of the entry mask for this user-written routine. If you do not specify the **key_buffer** argument or if you define key information in a specification file, this argument is required.

MERGE calls the comparison routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records to be compared, the lengths of these two records, and the context longword.

The comparison routine must return a 32-bit integer value:

- –1 if the first record collates before the second
- 0 if the records collate as equal
- 1 if the first record collates after the second

user_equal

VMS usage: **procedure**
type: **procedure entry mask**
access: **function call**
mechanism: **by reference**

Routine that resolves the merge order when records have duplicate keys. The **user_equal** argument is the address of the entry mask for this user-written routine. If you specify SOR$M_STABLE or SOR$M_NODUPS in the **options** argument, do not use this argument.

MERGE calls the duplicate key routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records that compare equally, the lengths of the two records that compare equally, and the context longword.

The routine must return one of the following 32-bit condition codes.

Code	Description
SOR$_DELETE1	Delete the first record from the merge.
SOR$_DELETE2	Delete the second record from the merge.
SOR$_DELBOTH	Delete both records from the merge.
SS$_NORMAL	Keep both records in the merge.

Any other failure value causes the error to be signaled or returned. Any other success value causes an undefined result.

user_input

VMS usage: **procedure**
type: **procedure entry mask**
access: **function call**
mechanism: **by reference**

Routine that releases records to the merge operation. The **user_input** argument is the address of the entry mask for this user-written routine. SOR$BEGIN_MERGE and SOR$RETURN_REC call this routine until all records have been passed.

This input routine must read (or construct) a record, place it in a record buffer, store its length in an output argument, and then return control to MERGE.

The input routine must accept the following four arguments:

- A descriptor of the buffer where the routine must place the record
- A longword, passed by reference, containing the stream number from which to input a record (the first file is 1, the second 2, and so on)
- A word, passed by reference, where the routine must return the actual length of the record
- The context longword, passed by reference

The input routine must also return one of the following status values:

- SS$_NORMAL or any other success status causes the merge operation to continue.

- SS$_ENDOFFILE indicates that no more records are in the file. The contents of the buffer are ignored.
- Any other error status terminates the merge operation and passes the status value back to the caller of SOR$BEGIN_MERGE or SOR$RETURN_REC.

context

VMS usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value that was supplied by SORT/MERGE.

DESCRIPTION

The SOR$BEGIN_MERGE routine initializes the merge process by passing arguments that provide the number of input streams, the key specifications, and any merge options.

You must define the key by passing either the key buffer address argument or your own comparison routine address. (You can also define the key in a specification file and call the SOR$SPEC_FILE routine.)

The SOR$BEGIN_MERGE routine initializes the merge process in the file, record, and mixed interfaces. For record interface on input, you must also pass the merge order, the input routine address, and the longest record length. For files not on disk, you must pass the longest record length.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, you should use LIB$MATCH_COND if you want to check for a specific status.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_BADDTYPE	Invalid or unsupported CDD datatype.
SOR$_BADLENOFF	Length and offset must be multiples of 8 bits.
SOR$_BADLOGIC	Internal logic error detected.
SOR$_BADOCCURS	Invalid OCCURS clause.
SOR$_BADOVRLAY	Invalid overlay structure.
SOR$_BADPROTCL	Node is an invalid CDD object.
SOR$_BAD_KEY	Invalid key specification.
SOR$_BAD_LRL	Record length *n* greater than specified longest record length.

SOR$_BAD_MERGE	Number of work files must be between 0 and 10.
SOR$_BAD_ORDER	Merge input is out of order.
SOR$_BAD_SRL	Record length *n* is too short to contain keys.
SOR$_BAD_TYPE	Invalid sort process specified.
SOR$_CDDERROR	CDD error at node 'name'.
SOR$_CLOSEIN	Error closing 'file' as input.
SOR$_CLOSEOUT	Error closing 'file' as output.
SOR$_COL_CHAR	Invalid character definition.
SOR$_COL_CMPLX	Collating sequence is too complex.
SOR$_COL_PAD	Invalid pad character.
SOR$_COL_THREE	Cannot define 3-byte collating values.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_ILLBASE	Nondecimal base is invalid.
SOR$_ILLLITERL	Record containing symbolic literals is unsupported.
SOR$_ILLSCALE	Nonzero scale invalid for floating-point data item.
SOR$_INCDIGITS	Number of digits is not consistent with the type or length of item.
SOR$_INCNODATA	Include specification references no data, at line *n*.
SOR$_INCNOKEY	Include specification references no keys, at line *n*.
SOR$_IND_OVR	Indexed output file must already exist.
SOR$_KEYAMBINC	Key specification is ambiguous or inconsistent.
SOR$_KEYED	Mismatch between SORT/MERGE keys and primary file key.
SOR$_KEY_LEN	Invalid key length, key number *n*, length *n*.
SOR$_LRL_MISS	Longest record length must be specified.
SOR$_MISLENOFF	Length and offset required.
SOR$_MISS_PARAM	A required subroutine argument is missing.
SOR$_MULTIDIM	Invalid multidimensional OCCURS.
SOR$_NODUPEXC	Equal-key routine and no-duplicates option cannot both be specified.
SOR$_NOTRECORD	Node 'name' is a name, not a record definition.
SOR$_NUM_KEY	Too many keys specified.
SOR$_OPENIN	Error opening 'file' as input.
SOR$_OPENOUT	Error opening 'file' as output.
SOR$_READERR	Error reading 'file'.
SOR$_RTNERROR	Unexpected error status from user-written routine.
SOR$_SIGNCOMPQ	Absolute Date and Time data type represented in one-second units.
SOR$_SORT_ON	Sort or merge routines called in incorrect order.
SOR$_SPCIVC	Invalid collating sequence specification at line *n*.
SOR$_SPCIVD	Invalid data type at line *n*.

SOR$_SPCIVF	Invalid field specification at line *n*.
SOR$_SPCIVI	Invalid include or omit specification at line *n*.
SOR$_SPCIVK	Invalid key or data specification at line *n*.
SOR$_SPCIVP	Invalid sort process at line *n*.
SOR$_SPCIVS	Invalid specification at line *n*.
SOR$_SPCIVX	Invalid condition specification at line *n*.
SOR$_SPCMIS	Invalid merge specification at line *n*.
SOR$_SPCOVR	Overridden specification at line *n*.
SOR$_SPCSIS	Invalid sort specification at line *n*.
SOR$_SRTIWA	Insufficient space. The specification file is too complex.
SOR$_STABLEEX	Equal-key routine and stable option cannot both be specified.
SOR$_SYSERROR	System service error.
SOR$_UNDOPTION	Undefined option flag was set.
SOR$_UNSUPLEVL	Unsupported core level for record 'name'.
SOR$_WRITEERR	Error writing 'file'.

SOR$BEGIN_SORT Begin a Sort Operation

The SOR$BEGIN_SORT routine initializes a sort operation by opening input and output files and by passing the key information and any sort options.

FORMAT

SOR$BEGIN_SORT *[key_buffer] [,lrl] [,options] [,file_alloc] [,user_compare] [,user_equal] [,sort_process] [,work_files] [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

key_buffer

VMS usage: **vector_word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Array of words describing the keys on which you plan to sort. The **key_buffer** argument is the address of an array containing the key descriptions.

The first word of this array contains the number of keys described (up to 255). Following the first word, each key is described (in order of priority) in blocks of four words. The four words specify the key's data type, order, offset, and length, respectively.

The first word of the block specifies the data type of the key. The following data types are accepted:

DSC$K_DTYPE_Z	Unspecified (uninfluenced by collating sequence)
DSC$K_DTYPE_B	Byte integer (signed)
DSC$K_DTYPE_BU	Byte (unsigned)
DSC$K_DTYPE_W	Word integer (signed)
DSC$K_DTYPE_WU	Word (unsigned)
DSC$K_DTYPE_L	Longword integer (signed)
DSC$K_DTYPE_LU	Longword (unsigned)
DSC$K_DTYPE_Q	Quadword integer (signed)
DSC$K_DTYPE_QU	Quadword (unsigned)
DSC$K_DTYPE_O	Octaword integer (signed)

DSC$K_DTYPE_OU	Octaword (unsigned)
DSC$K_DTYPE_F	Single-precision floating
DSC$K_DTYPE_D	Double-precision floating
DSC$K_DTYPE_G	G-format floating
DSC$K_DTYPE_H	H-format floating
DSC$K_DTYPE_T	Text (may be influenced by collating sequence)
DSC$K_DTYPE_NU	Numeric string, unsigned
DSC$K_DTYPE_NL	Numeric string, left separate sign
DSC$K_DTYPE_NLO	Numeric string, left overpunched sign
DSC$K_DTYPE_NR	Numeric string, right separate sign
DSC$K_DTYPE_NRO	Numeric string, right overpunched sign
DSC$K_DTYPE_NZ	Numeric string, zoned sign
DSC$K_DTYPE_P	Packed decimal string

The VAX Procedure Calling and Condition Handling Standard, documented in Chapter 2 in the *Introduction to VMS System Routines*, describes each of these data types.

The second word of the block specifies the key order: *0* for ascending order, *1* for descending order. The third word of the block specifies the relative offset of the key in the record. Note that the first byte in the record is at position *0*. The fourth word of the block specifies the key length in bytes (in digits for packed decimal—DSC$K_DTYPE_P).

The **key_buffer** argument specifies the address of the key buffer in the data area. If you do not specify this argument, you must either pass a key comparison routine or use a specification file to define the key.

lrl

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Length of the longest record that will be released for sorting. The **lrl** argument is the address of a word containing the length. This argument is not required if the input file(s) is on disk but is required when you use the record interface. For VFC records, this length must include the length of the fixed-length portion of the record.

options

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

Flags that identify sort options. The **options** argument is the address of a longword bit mask whose settings determine the merge options selected. The following table lists and describes the bit mask values available.

Flags	Description
SOR$M_STABLE	Keeps records with equal keys in the same order in which they appeared on input. With multiple input files that have records that collate as equal, records from the first input file are placed before the records from the second input file, and so on.
SOR$M_EBCDIC	Orders ASCII character keys according to EBCDIC collating sequence. No translation takes place.
SOR$M_MULTI	Orders character keys according to the multinational collating sequence, which collates the international character set.
SOR$M_NOSIGNAL	Returns the condition code instead of signaling an error. The default is SOR$M_NOSIGNAL.
SOR$M_NODUPS	Omits records with duplicate keys. You cannot use this option if you specify your own equal-key routine.

All other bits in the longword are reserved and must be zero.

file_alloc

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

Input file size in blocks. The **file_alloc** argument is the address of a longword containing the size. This argument is never required because, by default, SORT uses the allocation of the input files. If you are using the record interface or if the input files are not on disk, the default is 1000 blocks.

However, you can use this optional argument to improve the efficiency of the sort by adjusting the amount of resources the sort process allocates.

user_compare

VMS usage: **procedure**
type: **procedure entry mask**
access: **function call**
mechanism: **by reference**

User-written routine that compares records to determine their sort order. The **user_compare** argument is the address of the entry mask for this user-written routine. If you do not specify the **key_buffer** argument or if you define key information in a specification file, this argument is required.

SORT/MERGE calls the comparison routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records to be compared, the lengths of these two records, and the context longword.

The comparison routine must return a 32-bit integer value:

- –1 if the first record collates before the second
- 0 if the records collate as equal
- 1 if the first record collates after the second

user_equal

VMS usage: **procedure**
type: **procedure entry mask**
access: **function call**
mechanism: **by reference**

User-written routine that resolves the sort order when records have duplicate keys. The **user_equal** argument is the address of the entry mask for this user-written routine. If you specify SOR$M_STABLE or SOR$M_NODUPS in the **options** argument, do not use this argument.

SORT/MERGE calls the duplicate key routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two records that compare equally, the lengths of the two records that compare equally, and the context longword.

The routine must return one of the following 32-bit integer condition codes.

Code	Description
SOR$_DELETE1	Delete the first record from the sort.
SOR$_DELETE2	Delete the second record from the sort.
SOR$_DELBOTH	Delete both records from the sort.
SS$_NORMAL	Keep both records in the sort.

Any other failure value causes the error to be signaled or returned. Any other success value causes an undefined result.

sort_process

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Code indicating the type of sort process. The **sort_process** argument is the address of a byte whose value indicates whether the sort type is record, tag, index, or address. The default is record. If you select the record interface on input, you can use only a record sort process.

To specify a byte containing the value for the type of sort process you want, enter one of the following:

- SOR$GK_RECORD (record sort)
- SOR$GK_TAG (tag sort)
- SOR$GK_ADDRESS (address sort)
- SOR$GK_INDEX (index sort)

work_files

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Number of work files to be used in the sorting process. The **work_files** argument is the address of a byte containing the number of work files; permissible values range from *0* through *10*.

By default, SORT creates two temporary work files when it needs them and determines their size from the size of your input files. By increasing the number of work files, you can reduce their individual size so that each fits into less disk space. You can also assign each of them to different disk-structured devices.

context

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

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

The SOR$BEGIN_SORT routine initializes the sort process by setting up sort work areas and provides key specification and sort options.

Specify the key information with the **key_buffer** argument, with the **user_compare** argument, or in a specification file. If no key information is specified, the default (character for the entire record) is used.

You must use the SOR$BEGIN_SORT routine to initialize the sort process for the file, record, and mixed interfaces. For record interface on input, you must use the **lrl** (longest record length) argument.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_BADLOGIC	Internal logic error detected.
SOR$_BAD_KEY	Invalid key specification.
SOR$_BAD_LRL	Record length *n* greater than specified longest record length.
SOR$_BAD_MERGE	Number of work files must be between 0 and 10.
SOR$_BAD_TYPE	Invalid sort process specified.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_INSVIRMEM	Insufficient virtual memory.
SOR$_KEYAMBINC	Key specification is ambiguous or inconsistent.
SOR$_KEY_LEN	Invalid key length, key number *n*, length *n*.
SOR$_LRL_MISS	Longest record length must be specified.
SOR$_NODUPEXC	Equal-key routine and no-duplicates option cannot both be specified.
SOR$_NUM_KEY	Too many keys specified.
SOR$_RTNERROR	Unexpected error status from user-written routine.
SOR$_SORT_ON	Sort or merge routine called in incorrect order.
SOR$_STABLEEXC	Equal-key routine and stable option cannot both be specified.
SOR$_SYSERROR	System service error.
SOR$_UNDOPTION	Undefined option flag was set.

SOR$DTYPE Define Data Type

The SOR$DTYPE routine defines a key data type that is not normally supported by SORT/MERGE. This routine returns a key data type code that can be used in the **key_buffer** argument to SOR$BEGIN_SORT or SOR$BEGIN_MERGE to describe special key data types (such as extended data types and NCS collating sequences).

FORMAT

SOR$DTYPE *[context], dtype_code, usage, p1*

RETURNS

VMS usage: **Cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

context

VMS usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the context longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

dtype_code

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

Returned key data type code. The **dtype_code** argument is the address of a word into which SORT/MERGE writes the key data type code that can be used in the **key_buffer** argument to SOR$BEGIN_SORT or SOR$BEGIN_MERGE.

usage

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

Address of a longword containing a code that indicates the interpretation of the **p1** argument. The following table lists and describes the valid usage codes.

Flag	Description
SOR$K_ROUTINE	The **p1** argument should be interpreted as the address of the entry mask of a routine that SORT /MERGE will call to compare keys described by the **dtype_code** returned by the call to SOR$DTYPE.
SOR$K_NCS_TABLE	The **p1** argument should be interpreted as the address of a collating sequence identification returned by a call to NCS$GET_CS. SORT/MERGE will use this collating sequence to compare keys described by the **dtype_code** returned by the call to SOR$DTYPE.

If SOR$K_ROUTINE is returned, SORT/MERGE will call this routine with five reference arguments—ADRS1, ADRS2, LENG1, LENG2, CNTX—corresponding to the addresses of the two keys to be compared, the lengths of the two keys, and the context longword.

The comparison routine must return a 32-bit integer value:

- –1 if the first key collates before the second
- 0 if the keys collate as equal
- +1 if the first key collates after the second

p1

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

Address of an entry mask of a routine or the address of a collating sequence identification, depending on the **usage** argument.

DESCRIPTION

Call SOR$DTYPE to define a key data type not normally supported by SORT/MERGE.

If your SORT/MERGE application needs to compare dates (for example) that are stored in text form *and* that is the only key in the records, then use the **user_compare** argument to SOR$BEGIN_SORT or SOR$BEGIN_MERGE. However, if the records contain several keys besides the dates in text form, it may be easier to call SOR$DTYPE to allocate a key data type code that can then be used in the the **key_buffer** argument to SOR$BEGIN_SORT or SOR$BEGIN_MERGE.

If your SORT/MERGE application has a string key that should be collated by a collating sequence defined by the VMS National Character Set (NCS) Utility, the NCS$GET_CS routine can be used to fetch the collating sequence definition, and SOR$DTYPE can be called to allocate a key data type code for the collating sequence. This key data type code can then be used to describe keys that should be compared by this collating sequence.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_SORT_ON	Sort or merge routine called in incorrect order.

SOR$END_SORT End a Sort Operation

The SOR$END_SORT routine does cleanup functions, such as closing files and releasing memory.

FORMAT

SOR$END_SORT *[context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

context
VMS usage: **context**
type: **longword**
access: **write only**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

The SOR$END_SORT routine ends a sort or merge operation, either at the end of a successful process or between calls because of an error. If an error status is returned, you must call SOR$END_SORT to release all allocated resources. In addition, this routine may be called at any time to close files and release memory.

The value of the optional context argument is cleared when the SOR$END_SORT routine completes its operation.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_CLOSEIN	Error closing 'file' as input.
SOR$_CLOSEOUT	Error closing 'file' as output.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_END_SORT	SORT/MERGE terminated, context = 'context'.
SOR$_SYSERROR	System service error.

SOR$PASS_FILES Pass File Names

The SOR$PASS_FILES routine passes the names of input and output files and output file characteristics to SORT or MERGE.

FORMAT

SOR$PASS_FILES *[inp_desc] [,out_desc] [,org] [,rfm] [,bks] [,bls] [,mrs] [,alq] [,fop] [,fsz] [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

inp_desc

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

Input file specification. The **inp_desc** argument is the address of a descriptor pointing to the file specification. In the file interface, you must call SOR$PASS_FILES to pass SORT the input file specifications. For multiple input files, call SOR$PASS_FILES once for each input file, passing one input file specification descriptor each time.

In the mixed interface, if you are using the record interface on input, pass only the output file specification; do not pass any input file specifications. If you are using the record interface on output, pass only the input file specifications; do not pass an output file specification or any of the optional output file arguments.

out_desc

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

Output file specification. The **out_desc** argument is the address of a descriptor pointing to the file specification. In the file interface, when you call SOR$PASS_FILES, you must pass the output file specification. Specify the output file specification and characteristics only once, as part of the first call, as in the following:

```
Call SOR$PASS_FILES(Input1,Output)
Call SOR$PASS_FILES(Input2)
Call SOR$PASS_FILES(Input3)
```

In the mixed interface, if you are using the record interface on input, pass only the output file specification; do not pass any input file specifications. If you are using the record interface on output, pass only the input file specifications; do not pass an output file specification or any of the optional output file arguments.

org

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

File organization of the output file, if different from the input file. The **org** argument is the address of a byte whose value specifies the organization of the output file; permissible values include the following:

```
FAB$C_SEQ
FAB$C_REL
FAB$C_IDX
```

For the record interface on input, the default value is sequential. For the file interface, the default value is the file organization of the first input file for record or tag sort and sequential for address and index sort.

For more information about the FAB fields, see the *VMS Record Management Services Manual.*

rfm

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Record format of the output file, if different from the input file. The **rfm** argument is the address of a byte whose value specifies the record format of the output file; permissible values include the following:

```
FAB$C_FIX
FAB$C_VAR
FAB$C_VFC
```

For the record interface on input, the default value is variable. For the file interface, the default value is the record format of the first input file for record or tag sort and fixed format for address or index sort. For the mixed interface with record interface on input, the default value is variable format.

bks

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Bucket size of the output file, if different from the first input file. The **bks** argument is the address of a byte containing this size. Use this argument with relative and indexed-sequential files only. If the bucket size of the output file is to differ from that of the first input file, specify a byte to indicate the bucket size. Acceptable values are from *1* to *32*. If you do not pass this argument—and the output file organization is the same as that of the first input file—the bucket size defaults to the value of the first input file. If the file organizations differ or if the record interface is used on input, the default value is *1* block.

bls

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Block size of a magnetic tape output file. The **bls** argument is the address of a word containing this size. Use this argument with magnetic tapes only. Permissible values range from *20* to *65,532*. However, to ensure compatibility with non-DIGITAL systems, ANSI standards require that the block size be less than or equal to *2048*.

The block size defaults to the block size of the input file magnetic tape. If the input file is not on magnetic tape, the output file block size defaults to the size used when the magnetic tape was mounted.

mrs

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

Maximum record size for the output file. The **mrs** argument is the address of a word specifying this size. Following are acceptable values for each type of file.

File Organization	Acceptable Value
Sequential	0 to 32,767
Relative	0 to 16,383
Indexed sequential	0 to 16,362

If you omit this argument or if you specify a value of *0*, SORT does not check maximum record size.

If you do not specify this argument, the default is based on the output file organization and format, unless the organization is relative or the format is fixed. The longest output record length is based on the longest calculated input record length, the type of sort, and the record format.

alq

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

Number of preallocated output file blocks. The **alq** argument is the address of a longword specifying the number of blocks you want to preallocate to the output file. Acceptable values range from *1* to *4,294,967,295*.

Pass this argument if you know your output file allocation will be larger or smaller than that of your input files. The default value is the total allocation of all the input files. If the allocation cannot be obtained for any of the input files or if record interface is used on input, the file allocation defaults to 1000 blocks.

fop

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

File-handling options. The **fop** argument is the address of a longword whose bit settings determine the options selected. For a list of valid options, see the description of the FAB$L_FOP field in the *VMS Record Management Services Manual*. By default, only the DFW (deferred write) option is set. If your output file is indexed, you should set the CIF (create if) option.

fsz

VMS usage: **byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference**

Size of the fixed portion of VFC records. The **fsz** argument is the address of a byte containing this size. If you do not pass this argument, the default is the size of the fixed portion of the first input file. If you specify the VFC size as *0*, RMS defaults the value to *2* bytes.

context

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

Value that distinguishes between multiple concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

The SOR$PASS_FILES routine passes input and output file specifications to SORT. The SOR$PASS_FILES routine must be repeated for multiple input files. The output file name string and characteristics should be specified only in the first call to SOR$PASS_FILES.

This routine also accepts optional arguments that specify characteristics for the output file. By default, the output file characteristics are the same as the first input file; specified output file characteristics are used to change these defaults.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_DUP_OUTPUT	Output file has already been specified.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_INP_FILES	Too many input files specified.
SOR$_SORT_ON	Sort or merge routine called in incorrect order.
SOR$_SYSERROR	System service error.

SOR$RELEASE_REC Pass One Record to Sort

The SOR$RELEASE_REC routine is used with the record interface to pass one input record to SORT or MERGE.

FORMAT

SOR$RELEASE_REC *desc [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

desc

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor**

Input record buffer. The **desc** argument is the address of a descriptor pointing to the buffer containing the record to be sorted. If you use the record interface, this argument is required.

context

VMS usage: **context**
type: **longword**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

Call SOR$RELEASE_REC to pass records to SORT or MERGE with the record interface. SOR$RELEASE_REC must be called once for each record to be sorted.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_BADLOGIC	Internal logic error detected.
SOR$_BAD_LRL	Record length *n* greater than longest specified record length.
SOR$_BAD_SRL	Record length *n* too short to contain keys.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_EXTEND	Unable to extend work file for needed space.
SOR$_MISS_PARAM	The **desc** argument is missing.
SOR$_NO_WRK	Work files required, cannot do sort in memory as requested.
SOR$_OPENOUT	Error opening 'file' as output.
SOR$_OPERFAIL	Error requesting operator service.
SOR$_OPREPLY	Operator reply is 'reply'.
SOR$_READERR	Error reading 'file'.
SOR$_REQ_ALT	Specify alternate 'name' file (or nothing to try again).
SOR$_RTNERROR	Unexpected error status from user-written routine.
SOR$_SORT_ON	Sort or merge routines called in incorrect order.
SOR$_SYSERROR	System service error.
SOR$_USE_ALT	Using alternate file 'name'.
SOR$_WORK_DEV	Work file 'name' must be on random access local device.

SOR$RETURN_REC Return One Sorted Record

The SOR$RETURN_REC routine is used with the record interface to return one sorted or merged record to a program.

FORMAT

SOR$RETURN_REC *desc [,length] [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

desc

VMS usage: **char_string**
type: **character-coded text string**
access: **write only**
mechanism: **by descriptor**

Output record buffer. The **desc** argument is the address of a descriptor pointing to the buffer that receives the sorted or merged record.

length

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

Length of the output record. The **length** argument is the address of a word receiving the length of the record returned from SORT/MERGE.

context

VMS usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

Call the SOR$RETURN_REC routine to release the sorted or merged records to a program. Call this routine once for each record to be returned.

SOR$RETURN_REC places the record into a record buffer that you set up in the program's data area. After SORT has successfully returned all the records to the program, it returns the status code SS$_ENDOFFILE, which indicates that there are no more records to return.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_BADLOGIC	Internal logic error detected.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_EXTEND	Unable to extend work file for needed space.
SOR$_MISS_PARAM	A required subroutine argument is missing.
SOR$_OPERFAIL	Error requesting operator service.
SOR$_OPREPLY	Operator reply is 'reply'.
SOR$_READERR	Error reading 'file'.
SOR$_REQ_ALT	Specify alternate 'name' file (or nothing to simply try again).
SOR$_RTNERROR	Unexpected error status from user-written routine.
SOR$_SORT_ON	Sort or merge routines called in incorrect order.
SOR$_SYSERROR	System service error.
SOR$_USE_ALT	Using alternate file 'name'.
SOR$_WORK_DEV	Work file 'name' must be on random access local device.

SOR$SORT_MERGE Sort

The SOR$SORT_MERGE routine sorts the input records.

FORMAT

SOR$SORT_MERGE *[context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

context
VMS usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

After you have passed either the file names or the records to SORT, call the SOR$SORT_MERGE routine to sort the records. For file interface on input, the input files are opened and the records are released to the sort. For the record interface on input, the record must have already been released (by calls to SOR$RELEASE_REC). For file interface on output, the output records are reformatted and directed to the output file. For the record interface on output, SOR$RELEASE_REC must be called to get the sorted records.

Some of the return values are used with different severities depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SS$_NORMAL	Success.
SOR$_BADDTYPE	Invalid or unsupported CDD data type.
SOR$_BADLENOFF	Length and offset must be multiples of 8 bits.
SOR$_BADLOGIC	Internal logic error detected.

SOR$_BADOCCURS	Invalid OCCURS clause.
SOR$_BADOVRLAY	Invalid overlay structure.
SOR$_BADPROTCL	Node is an invalid CDD object.
SOR$_BAD_LRL	Record length *n* greater than longest specified record length.
SOR$_BAD_TYPE	Invalid sort process specified.
SOR$_CDDERROR	CDD error at node 'name'.
SOR$_CLOSEIN	Error closing 'file' as input.
SOR$_CLOSEOUT	Error closing 'file' as output.
SOR$_COL_CHAR	Invalid character definition.
SOR$_COL_CMPLX	Collating sequence is too complex.
SOR$_COL_PAD	Invalid pad character.
SOR$_COL_THREE	Cannot define 3-byte collating values.
SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_EXTEND	Unable to extend work file for needed space.
SOR$_ILLBASE	Nondecimal base is invalid.
SOR$_ILLLITERL	Record containing symbolic literals is unsupported.
SOR$_ILLSCALE	Nonzero scale invalid for floating-point data item.
SOR$_INCDIGITS	Number of digits is inconsistent with the type or length of item.
SOR$_INCNODATA	Include specification references no 'data' keyword, at line *n*.
SOR$_INCNOKEY	Include specification references no 'keys' keyword, at line *n*.
SOR$_IND_OVR	Indexed output file must already exist.
SOR$_KEYED	Mismatch between SORT/MERGE keys and primary file key.
SOR$_LRL_MISS	Longest record length must be specified.
SOR$_MISLENOFF	Length and offset required.
SOR$_MULTIDIM	Invalid multidimensional OCCURS.
SOR$_NOTRECORD	Node 'name' is a name, not a record definition.
SOR$_NO_WRK	Work files required, cannot do sort in memory as requested.
SOR$_OPENIN	Error opening 'file' as input.
SOR$_OPENOUT	Error opening 'file' as output.
SOR$_OPERFAIL	Error requesting operator service.
SOR$_OPREPLY	Operator reply is 'reply'.
SOR$_READERR	Error reading 'file'.
SOR$_REQ_ALT	Specify alternate 'name' file (or nothing to try again).
SOR$_RTNERROR	Unexpected error status from user-written routine.
SOR$_SIGNCOMPQ	Absolute Date and Time data type represented in one-second units.

SOR$_SORT_ON	Sort or merge routines called in incorrect order.
SOR$_SPCIVC	Invalid collating sequence specification, at line *n*.
SOR$_SPCIVD	Invalid data type, at line *n*.
SOR$_SPCIVF	Invalid field specification, at line *n*.
SOR$_SPCIVI	Invalid include or omit specification, at line *n*.
SOR$_SPCIVK	Invalid key or data specification, at line *n*.
SOR$_SPCIVP	Invalid sort process, at line *n*.
SOR$_SPCIVS	Invalid specification, at line *n*.
SOR$_SPCIVX	Invalid condition specification, at line *n*.
SOR$_SPCMIS	Invalid merge specification, at line *n*.
SOR$_SPCOVR	Overridden specification, at line *n*.
SOR$_SPCSIS	Invalid sort specification, at line *n*.
SOR$_SRTIWA	Insufficient space. Specification file is too complex.
SOR$_SYSERROR	System service error.
SOR$_UNSUPLEVL	Unsupported core level for record 'name'.
SOR$_USE_ALT	Using alternate file 'name'.
SOR$_WORK_DEV	Work file 'name' must be on random access local device.
SOR$_WRITEERR	Error writing 'file'.

SOR$SPEC_FILE Pass a Specification File Name

The SOR$SPEC_FILE routine is used to pass a specification file or specification text to a sort or merge operation.

FORMAT

SOR$SPEC_FILE *[spec_file] [,spec_buffer] [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

spec_file

VMS usage: **char_string**
type: **character-coded text string**
access: **read-only**
mechanism: **by descriptor**

Specification file name. The **spec_file** argument is the address of a descriptor pointing to the name of a file that contains the text of the options requested for the sort or merge. The specification file name string and the specification file buffer arguments are mutually exclusive.

spec_buffer

VMS usage: **char_string**
type: **character-coded text string**
access: **read-only**
mechanism: **by descriptor**

Specification text buffer. The **spec_buffer** argument is the address of a descriptor pointing to a buffer containing specification text. This text has the same format as the text within the specification file. The specification file name string and the specification file buffer arguments are mutually exclusive.

context

VMS usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

Call SOR$SPEC_FILE to pass a specification file name or a buffer with specification text to a sort or merge operation. Through the use of a specification file, you may selectively omit or include particular records from the sort or merge operation and specify the reformatting of the output records. (See the Sort Utility in the *VMS Sort/Merge Utility Manual* for a complete description of specification files.)

If you call the SOR$SPEC_FILE routine, you must do so before you call any other routines. You must pass either the **spec_file** or **spec_buffer** argument, but not both.

Some of the return condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_SORT_ON	Sort or merge routine called in incorrect order.
SOR$_SYSERROR	System service error.

SOR$STAT Obtain a Statistic

The SOR$STAT routine returns one statistic about the sort or merge operation to the user program.

FORMAT

SOR$STAT *code ,result [,context]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

code

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by reference**

SORT/MERGE statistic code. The **code** argument is the address of a word containing the code that identifies the statistic you want returned in the **result** argument. The following values are accepted.

Code	Description
SOR$K_IDENT	Address of ASCII string for version number
SOR$K_REC_INP	Number of records input
SOR$K_REC_SOR	Records sorted
SOR$K_REC_OUT	Records output
SOR$K_LRL_INP	Longest Record Length (LRL) for input
SOR$K_LRL_INT	Internal LRL
SOR$K_LRL_OUT	LRL for output
SOR$K_NODES	Nodes in sort tree
SOR$K_INI_RUNS	Initial dispersion runs
SOR$K_MRG_ORDER	Maximum merge order
SOR$K_MRG_PASSES	Number of merge passes
SOR$K_WRK_ALQ	Work file allocation
SOR$K_MBC_INP	Multiblock count for input

Code	Description
SOR$K_MBC_OUT	Multiblock count for output
SOR$K_MBF_INP	Multibuffer count for input
SOR$K_MBF_OUT	Multibuffer count for output

Note that performance statistics (such as direct I/O, buffered I/O, and elapsed and CPU times) are not available because user-written routines may affect those values. However, they are available if you call LIB$GETJPI.

result

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

SORT/MERGE statistic value. The **result** argument is the address of a longword into which SORT/MERGE writes the value of the statistic identified by the **code** argument.

context

VMS usage: **context**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Value that distinguishes between multiple, concurrent SORT/MERGE operations. The **context** argument is the address of a longword containing the context value. When your program makes its first call to a SORT/MERGE routine for a particular sort or merge operation, the **context** longword must equal zero. SORT/MERGE then stores a value in the longword to identify the operation just initiated. When you make subsequent routine calls for the same operation, you must pass the context value supplied by SORT/MERGE.

DESCRIPTION

The SOR$STAT routine returns one statistic about the sort or merge operation to your program. You can call the SOR$STAT routine at any time while the sort or merge is active.

Some of the following condition values are used with different severities, depending on whether SORT/MERGE can recover. Thus, if you want to check for a specific status, you should use LIB$MATCH_COND.

CONDITION VALUES RETURNED

SOR$_ENDDIAGS	Completed with diagnostics.
SOR$_MISS_PARAM	A required subroutine argument is missing.
SOR$_NYI	Functionality is not yet implemented.
SOR$_SYSERROR	System service error.

13 VAX Text Processing Utility (VAXTPU) Routines

13.1 Introduction to VAXTPU Routines

This chapter describes callable VAX Text Processing (VAXTPU) Utility Routines. It describes the purpose of the VAXTPU callable routines, the parameters for the routine call, and the primary status returns. The parameter in the call syntax represents the object that you pass to a VAXTPU routine. Each parameter description lists the data type and the passing mechanism for the object. The data types are standard VMS data types. The passing mechanism indicates how the parameter list is interpreted.

Callable VAXTPU Routines make VAXTPU accessible from within other VAX languages and applications. VAXTPU can be called from a program written in any VAX language that generates calls using the VAX Procedure Calling and Condition Handling Standard. You can also call VAXTPU from VMS utilities, for example, MAIL. Callable VAXTPU allows you to perform text processing functions within your program.

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

This chapter is written for system programmers who are assumed to be familiar with the following:

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

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

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

13.1.1 Two Interfaces to Callable VAXTPU

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

Simplified Callable Interface

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

- Initialize the editor.
- Provide the editor with the parameters necessary for its operation.
- Control the editing session.
- Perform error handling.

When using the simplified callable interface, you can use the TPU$TPU routine to specify a VMS command line for VAXTPU, or you can call the TPU$EDIT routine to specify an input file and an output file. TPU$EDIT builds a command string that is then passed to the TPU$TPU routine. These two routines are described in detail in Section 13.2.

Full Callable Interface

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

- Initialize the editor (TPU$INTIALIZE).
- Execute VAXTPU procedures (TPU$EXECUTE_INIFILE and TPU$EXECUTE_COMMAND).
- Give control to the editor (TPU$CONTROL).
- Terminate the editing session (TPU$CLEANUP).

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

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

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

- Section 13.3 begins by briefly describing the interface. However, most of this section describes the main callable VAXTPU routines (TPU$INITIALIZE, TPU$EXECUTE_INIFILE, TPU$CONTROL, TPU$EXECUTE_COMMAND, and TPU$CLEANUP).
- Section 13.3.2 discusses additional routines that VAXTPU provides for use with the full callable interface.
- Section 13.3.3 defines the requirements for routines that you can write for use with the full callable interface.

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

13.1.2 Shareable Image

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

- TPU$VERSION—the version of the shareable image
- TPU$UPDATE—the update number of the shareable image

For more information about how to link to the shareable image TPUSHR.EXE, refer to the *VMS System Services Reference Manual.*

13.1.3 Passing Parameters to Callable VAXTPU Routines

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

A bound procedure value is a two-longword entity in which the first longword contains the address of a procedure entry mask, and the second longword is the environment value (see Figure 13–1). The environment value is determined in a language-specific manner when the original bound procedure value is generated. When the bound procedure is called, the calling program loads the second longword into R1.

Figure 13–1 Bound Procedure Value

NAME OF YOUR ROUTINE
ENVIRONMENT

ZK-4046-85

13.1.4 Error Handling

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

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

The default condition handler, TPU$HANDLER, is described in the routine description section of this chapter. Information about writing your own condition handler can be found in the *Introduction to VMS System Routines.*

13.1.5 Return Values

All VAXTPU condition codes are declared as universal symbols. Therefore, you automatically have access to these symbols when you link your program to the shareable image. The condition code values are returned in R0. Return codes for VAXTPU can be found in the *VAX Text Processing Utility Manual.* VAXTPU return codes and their messages are included in the *VMS System Messages and Recovery Procedures Reference Volume.*

Additional information about condition codes is provided in the descriptions of callable VAXTPU routines found in subsequent sections. This information is provided under the heading CONDITION VALUES RETURNED and indicates the values that are returned when the default condition handler is established.

13.2 The Simplified Callable Interface

The VAXTPU simplified callable interface consists of two routines: TPU$TPU and TPU$EDIT. These entry points to VAXTPU are useful for the following kinds of application:

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

13.2.1 Example of the Simplified Interface

The following example shows how the simplified interface might be used to call VAXTPU.

```
/* Sample C program that calls VAXTPU. This program uses TPU$EDIT
to provide the names of the input and output files, and TPU$TPU
to pass a complete command line to the editor. */
#include descrip
int return_status;
char command_line [100];
static $DESCRIPTOR(input_file,"infile.dat");
static $DESCRIPTOR(output_file,"outfile.dat");
static $DESCRIPTOR(command_desc,command_line);
static $DESCRIPTOR(first_part_desc,"EDIT/TPU/NOJOURNAL/NOCOMMAND/OUTPUT=");
static $DESCRIPTOR(space_desc," ");
main (argc,argv)
    int argc;
    char *argv[];
    {
/* Call the routine that accepts the name of the input and output file. */
/* This passes the name of the input file and output file to VAXTPU. */
/* These values are made available to VAXTPU procedures for processing. */

    return_status = TPU$EDIT(&input_file,&output_file);
    if (! return_status)
    exit(return_status);

/* Now we build a command line and pass it to VAXTPU.                 */
/* Note that in this case we want to do more than just specify the     */
/* file names.  Our command also includes the /NOJOURNAL and /NOCOMMAND  */
/* qualifiers.

/* Concatenate all the command information into one string              */

    return_status = STR$CONCAT(&command_desc,&first_part_desc,&output_file,
    &space_desc,&input_file);
    if (! return_status)
    exit(return_status);
/* Now call VAXTPU                                                     */
    return_status = TPU$TPU(&command_desc);
    exit(return_status);
    }
```

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

13.3 The Full Callable Interface

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

- Specify initialization parameters.
- Control file input/output.
- Specify commands to be executed by the editor.
- Control how conditions are handled.

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

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

13.3.1 Main Callable VAXTPU Utility Routines

The following callable VAXTPU routines are described in this chapter:

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

Note: **Before calling any of these routines, you must establish TPU$HANDLER or provide your own condition handler. See the routine description of TPU$HANDLER at the end of this chapter and the "VAX Condition Handling Standard" in the *Introduction to VMS System Routines* for information about establishing a condition handler.**

13.3.2 Other VAXTPU Utility Routines

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

- TPU$CLIPARSE—parses a command line and builds the item list for TPU$INITIALIZE
- TPU$PARSEINFO—parses a command and builds an item list for TPU$INITIALIZE
- TPU$FILEIO—the default file I/O routine
- TPU$MESSAGE—writes error messages and strings using the built-in procedure MESSAGE
- TPU$HANDLER—the default condition handler
- TPU$CLOSE_TERMINAL—closes VAXTPU's channel to the terminal (and its associated mailbox) for the duration of a CALL_USER routine

13.3.3 User-Written Routines

This section defines the requirements for user-written routines. When these routines are passed to VAXTPU, they must be passed as bound procedure values. (See Section 13.1.3 for a description of bound procedure values.) Depending on your application, you may have to write one or all of the following routines:

- Routine for initialization callback—This is a routine that TPU$INITIALIZE calls to obtain values for initialization parameters. If you choose to use TPU$INITIALIZE as a way of interfacing with VAXTPU, then you must write your own callback routine. This callback routine must call a parsing routine. The parsing routine can be TPU$CLIPARSE, or it can be a user-written routine.
- Routine for file I/O—This is a routine that handles file operations. Instead of writing your own file I/O routine, you can use the TPU$FILEIO utility routine. VAXTPU does not use this routine for journal file operations or for operations performed by the built-in procedure SAVE.
- Routine for condition handling—This is a routine that handles error conditions. Instead of writing your own condition handler, you can use the default condition handler, TPU$HANDLER.
- Routine for the built-in procedure CALL_USER—This is a routine that is called by the built-in procedure CALL_USER. You can use this mechanism to cause your program to get control during an editing session.

13.4 Examples of Using VAXTPU Routines

Examples 13–1 through 13–4 use callable VAXTPU. The examples are included here for illustrative purposes only; DIGITAL does not assume responsibility for supporting these examples.

Example 13–1 Sample VAX BLISS Template for Callable VAXTPU

```
MODULE file_io_example (MAIN = top_level) =

BEGIN

FORWARD ROUTINE
    top_level,                  ! Main routine of this example
    tpu_init,                   ! Initialize TPU
    tpu_io;                     ! File I/O routine for TPU
!
! Declare the stream data structure passed to the file I/O routine
!
MACRO
    stream_file_id =  0, 0, 32, 0 % ,     ! File ID
    stream_rat =      6, 0,  8, 0 % ,     ! Record attributes
    stream_rfm =      7, 0,  8, 0 % ,     ! Record format
    stream_file_nm =  8, 0,  0, 0 % ;     ! File name descriptor
```

Example 13–1 Cont'd. on next page

Example 13–1 (Cont.) Sample VAX BLISS Template for Callable VAXTPU

```
!
! Declare the routines that would actually do the I/O.  These must be supplied
! in another module
!
EXTERNAL ROUTINE
    my_io_open,                 ! Routine to open a file
    my_io_close,                ! Routine to close a file
    my_io_get_record,           ! Routine to read a record
    my_io_put_record;           ! Routine to write a record

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

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

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

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

    tpu$m_reset_terminal,       ! Masks for cleanup bitvector
    tpu$m_kill_process,
    tpu$m_delete_exith,
    tpu$m_last_time,
```

Example 13–1 Cont'd. on next page

Example 13–1 (Cont.) Sample VAX BLISS Template for Callable VAXTPU

```
    tpu$_nofileaccess,           ! VAXTPU status codes
    tpu$_openin,
    tpu$_inviocode,
    tpu$_failure,
    tpu$_closein,
    tpu$_closeout,
    tpu$_readerr,
    tpu$_writeerr,
    tpu$_success;

ROUTINE top_level =

    BEGIN
!++
! Main entry point of your program
!--
! Your_initialization_routine must be declared as a BPV

    LOCAL
        initialize_bpv: VECTOR [2],
        status,
        cleanup_flags;
    !
    ! First establish the condition handler
    !
    ENABLE
        tpu$handler ();
    !
    ! Initialize the editing session, passing TPU$INITIALIZE the address of
    ! the bound procedure value which defines the routine which VAXTPU is
    ! to call to return the initization item list
    !
    initialize_bpv [0] = tpu_init;
    initialize_bpv [1] = 0;
    tpu$initialize (initialize_bpv);
    !
    ! Call VAXTPU to execute the contents of the command file, the debug file
    ! or the TPU$INIT_PROCEDURE from the section file.
    !
    tpu$execute_inifile();
    !
    ! Let VAXTPU take over.
    !
    tpu$control();
    !
    ! Have VAXTPU cleanup after itself
    !
    cleanup_flags = tpu$m_reset_terminal OR      ! Reset the terminal
                    tpu$m_kill_process OR        ! Delete Subprocesses
                    tpu$m_delete_exith OR        ! Delete the exit handler
                    tpu$m_last_time;             ! Last time calling the editor

    tpu$cleanup (cleanup_flags);

    RETURN tpu$_success;

    END;

ROUTINE tpu_init =

    BEGIN
```

Example 13–1 Cont'd. on next page

Example 13–1 (Cont.) Sample VAX BLISS Template for Callable VAXTPU

```
!
! Allocate the storage block needed to pass the file I/O routine as a
! bound procedure variable as well as the bitvector for the initialization
! options
!
OWN
    file_io_bpv: VECTOR [2, LONG]
                 INITIAL (TPU_IO, 0),
    options;
!
! These macros define the file names passed to VAXTPU
!
MACRO
    out_file = 'OUTPUT.TPU' % ,
    com_file = 'TPU$COMMAND.TPU' % ,
    sec_file = 'TPU$SECTION.TPU$SECTION' % ,
    inp_file = 'FILE.TPU' % ;

!
! Create the item list to pass to VAXTPU.  Each item list entry consists of
! two words which specify the size of the item and its code, the address of
! the buffer containing the data, and a longword to receive a result (always
! zero, since VAXTPU does not return any result values in the item list)
!
!               +--------------------------------+
!               | Item Code       | Item Length   |
!               +----------------+---------------+
!               |        Buffer Address           |
!               +--------------------------------+
!               |    Return Address (always 0)    |
!               +--------------------------------+
!
! Remember that the item list is always terminated with a longword containing
! a zero
!
BIND
    item_list = UPLIT BYTE (
        WORD (4),                       ! Options bitvector
        WORD (tpu$_options),
        LONG (options),
        LONG (0),

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

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

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

Example 13–1 Cont'd. on next page

Example 13–1 (Cont.) Sample VAX BLISS Template for Callable VAXTPU

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

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

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

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

    END;                                   ! End of routine tpu_init

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

    LOCAL
        status;
!
! Is this one of ours, or do we pass it to TPU's file I/O routines?
!
    IF (..p_opcode NEQ tpu$k_open) AND (.stream [stream_file_id] GTR 511)
    THEN
        RETURN tpu$fileio (..p_opcode, .stream, .data);

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

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

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

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

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

Example 13–1 Cont'd. on next page

Example 13–1 (Cont.) Sample VAX BLISS Template for Callable VAXTPU

```
        [OTHERWISE]:
            status = tpu$_failure;

 TES;

    RETURN .status;

    END;                                      ! End of routine TPU_IO

END                                           ! End Module file_io_example

ELUDOM
```

Example 13–2 Normal VAXTPU Setup in VAX FORTRAN

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

        INTEGER*4       CLEAN_OPT       !options for clean up routine
        INTEGER*4       STATUS          !return status from VAXTPU routines
        INTEGER*4       BPV_PARSE(2)    !set up a Bound Procedure Value
        INTEGER*4       LOC_PARSE       !a local function call
C       declare the VAXTPU functions

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

        INTEGER*4       RESET_TERMINAL
        INTEGER*4       DELETE_JOURNAL
        INTEGER*4       DELETE_BUFFERS,DELETE_WINDOWS
        INTEGER*4       DELETE_EXITH,EXECUTE_PROC
        INTEGER*4       PRUNE_CACHE,KILL_PROCESSES
        INTEGER*4       CLOSE_SECTION
C       declare the VAXTPU functions used as external

        EXTERNAL        TPU$HANDLER
        EXTERNAL        TPU$CLIPARSE

        EXTERNAL        TPU$_SUCCESS    !external error message

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

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

100     CALL LIB$ESTABLISH ( TPU$HANDLER )      !establish the condition handler
C       set up the Bound Procedure Value for the call to TPU$INITIALIZE

        BPV_PARSE( 1 ) = %LOC( LOC_PARSE )
        BPV_PARSE( 2 ) = 0
```

Example 13–2 Cont'd. on next page

Example 13–2 (Cont.) Normal VAXTPU Setup in VAX FORTRAN

```
C       call the VAXTPU initialization routine to do some set up work

        STATUS = TPU$INITIALIZE ( BPV_PARSE )

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

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

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

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

        STATUS = TPU$EXECUTE_INIFILE ( )

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

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

        ENDIF
C       invoke the editor as it normally would appear

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

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

                CALL LIB$SIGNAL( %VAL( STATUS ) )
C               GOTO 9999
        ENDIF
C       Get the value of the option from the external literals.  In FORTRAN you
C       cannot use external literals directly so you must first get the value
C       of the literal from its external location.  Here we are getting the
C       values of the options that we want to use in the call to TPU$CLEANUP.

        DELETE_JOURNAL  = %LOC ( TPU$M_DELETE_JOURNAL )
        DELETE_EXITH    = %LOC ( TPU$M_DELETE_EXITH )
        DELETE_BUFFERS  = %LOC ( TPU$M_DELETE_BUFFERS )
        DELETE_WINDOWS  = %LOC ( TPU$M_DELETE_WINDOWS )
        EXECUTE_PROC    = %LOC ( TPU$M_EXECUTE_PROC )
        RESET_TERMINAL  = %LOC ( TPU$M_RESET_TERMINAL )
        KILL_PROCESSES  = %LOC ( TPU$M_KILL_PROCESSES )
        CLOSE_SECTION   = %LOC ( TPU$M_CLOSE_SECTION )
C       Now that we have the local copies of the variables we can do the
C       logical OR to set the multiple options that we need.

        CLEAN_OPT = DELETE_JOURNAL .OR. DELETE_EXITH .OR.
        1       DELETE_BUFFERS .OR. DELETE_WINDOWS .OR. EXECUTE_PROC
        1       .OR. RESET_TERMINAL .OR. KILL_PROCESSES .OR. CLOSE_SECTION

C       do the necessary clean up
C       TPU$CLEANUP wants the address of the flags as the parameter so
C       pass the %LOC of CLEAN_OPT which is the address of the variable

        STATUS = TPU$CLEANUP ( %LOC ( CLEAN_OPT ) )

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

                CALL LIB$SIGNAL( %VAL(STATUS) )

        ENDIF
```

Example 13–2 Cont'd. on next page

Example 13–2 (Cont.) Normal VAXTPU Setup in VAX FORTRAN

```
9999    CALL LIB$REVERT          !go back to normal processing -- handlers

        STOP
        END
C
C
        INTEGER*4  FUNCTION LOC_PARSE

        INTEGER*4       BPV(2)           !A local Bound Procedure Value

        CHARACTER*12    EDIT_COMM        !A command line to send to TPU$CLIPARSE
C       Declare the VAXTPU functions used

        INTEGER*4       TPU$FILEIO
        INTEGER*4       TPU$CLIPARSE
C       Declare this routine as external because it is never called directly and
C       we need to tell FORTRAN that it is a function and not a variable

        EXTERNAL        TPU$FILEIO

        BPV(1) = %LOC(TPU$FILEIO)       !set up the BOUND PROCEDURE VALUE
        BPV(2) = 0

        EDIT_COMM(1:12) = 'TPU TEST.TXT'
C       parse the command line and build the item list for TPU$INITIALIZE
9999     LOC_PARSE = TPU$CLIPARSE (EDIT_COMM, BPV , 0)

        RETURN
        END
```

Example 13–3 Building a Callback Item List with VAX FORTRAN

```
       PROGRAM  TEST_TPU
C
       IMPLICIT NONE
C
C       Define the expected VAXTPU return statuses
C
       EXTERNAL        TPU$_SUCCESS
       EXTERNAL        TPU$_QUITTING
C
C       Declare the VAXTPU routines and symbols used
C
       EXTERNAL        TPU$M_DELETE_CONTEXT
       EXTERNAL        TPU$HANDLER
       INTEGER*4       TPU$M_DELETE_CONTEXT
       INTEGER*4       TPU$INITIALIZE
       INTEGER*4       TPU$EXECUTE_INIFILE
       INTEGER*4       TPU$CONTROL
       INTEGER*4       TPU$CLEANUP
C
C       Declare the external callback routine
C
       EXTERNAL        TPU_STARTUP       ! the VAXTPU set-up function
       INTEGER*4       TPU_STARTUP

       INTEGER*4       BPV(2)            ! Set up a bound procedure value
```

Example 13–3 Cont'd. on next page

Example 13–3 (Cont.) Building a Callback Item List with VAX FORTRAN

```
C
C       Declare the functions used for working with the condition handler
C
        INTEGER*4       LIB$ESTABLISH
        INTEGER*4       LIB$REVERT
C
C       Local Flags and Indices
C
        INTEGER*4       CLEANUP_FLAG        ! flag(s) for VAXTPU cleanup
        INTEGER*4       RET_STATUS
C
C       Initializations
C
        RET_STATUS       = 0
        CLEANUP_FLAG     = %LOC(TPU$M_DELETE_CONTEXT)
C
C       Establish the default VAXTPU condition handler
C
        CALL LIB$ESTABLISH(%REF(TPU$HANDLER))
C
C       Set up the bound procedure value for the initialization callback
C
        BPV(1)  =  %LOC (TPU_STARTUP)
        BPV(2)  =  0
C
C       Call the VAXTPU procedure for initialization
C
        RET_STATUS = TPU$INITIALIZE(BPV)

        IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
        CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIF
C
C       Execute the VAXTPU initialization file
C
        RET_STATUS = TPU$EXECUTE_INIFILE()

        IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
        CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIF
C
C       Pass control to VAXTPU
C
        RET_STATUS = TPU$CONTROL()

        IF (RET_STATUS .NE. %LOC(TPU$_QUITTING)
        1       .OR. %LOC(TPU$_QUITTING)) THEN
                CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIF
C
C       Clean up after processing
C
        RET_STATUS = TPU$CLEANUP(%REF(CLEANUP_FLAG))

        IF (RET_STATUS .NE. %LOC(TPU$_SUCCESS)) THEN
        CALL LIB$SIGNAL (%VAL(RET_STATUS))
        ENDIF
```

Example 13–3 Cont'd. on next page

Example 13–3 (Cont.) Building a Callback Item List with VAX FORTRAN

```
C
C       Set the condition handler back to the default
C
        RET_STATUS = LIB$REVERT()

        END

        INTEGER*4 FUNCTION TPU_STARTUP

        IMPLICIT NONE

        INTEGER*4       OPTION_MASK       ! temporary variable for VAXTPU
        CHARACTER*44       SECTION_NAME   ! temporary variable for VAXTPU
C
C       External VAXTPU routines and symbols
C
        EXTERNAL        TPU$K_OPTIONS
        EXTERNAL        TPU$M_READ
        EXTERNAL        TPU$M_SECTION
        EXTERNAL        TPU$M_DISPLAY
        EXTERNAL        TPU$K_SECTIONFILE
        EXTERNAL        TPU$K_FILEIO
        EXTERNAL        TPU$FILEIO
        INTEGER*4       TPU$FILEIO
C
C       The bound procedure value used for setting up the file I/O routine
C
        INTEGER*4       BPV(2)

C
C        Define the structure of the item list defined for the callback
C
        STRUCTURE /CALLBACK/
        INTEGER*2       BUFFER_LENGTH
        INTEGER*2       ITEM_CODE
        INTEGER*4       BUFFER_ADDRESS
        INTEGER*4       RETURN_ADDRESS
        END STRUCTURE
C
C       There are a total of four items in the item list
C
        RECORD /CALLBACK/ CALLBACK (4)
C
C       Make sure it is not optimized!
C
        VOLATILE /CALLBACK/
C
C        Define the options we want to use in the VAXTPU session
C
        OPTION_MASK = %LOC(TPU$M_SECTION) .OR. %LOC(TPU$M_READ)
        1       .OR. %LOC(TPU$M_DISPLAY)
C
C       Define the name of the initialization section file
C
        SECTION_NAME = 'device:[user]TPUSECTION.TPU$SECTION'
```

Example 13–3 Cont'd. on next page

Example 13–3 (Cont.) Building a Callback Item List with VAX FORTRAN

```
C
C       Set up the required I/O routine.  Use the VAXTPU default.
C
        BPV(1) = %LOC(TPU$FILEIO)
        BPV(2) = 0
C
C       Build the callback item list
C
C       Set up the edit session options
C
        CALLBACK(1).ITEM_CODE = %LOC(TPU$K_OPTIONS)
        CALLBACK(1).BUFFER_ADDRESS = %LOC(OPTION_MASK)
        CALLBACK(1).BUFFER_LENGTH =
        CALLBACK(1).RETURN_ADDRESS = 0
C
C       Identify the section file to be used
C
        CALLBACK(2).ITEM_CODE = %LOC(TPU$K_SECTIONFILE)
        CALLBACK(2).BUFFER_ADDRESS = %LOC(SECTION_NAME)
        CALLBACK(2).BUFFER_LENGTH = LEN(SECTION_NAME)
        CALLBACK(2).RETURN_ADDRESS = 0
C
C       Set up the I/O handler
C
        CALLBACK(3).ITEM_CODE = %LOC(TPU$K_FILEIO)
        CALLBACK(3).BUFFER_ADDRESS = %LOC(BPV)
        CALLBACK(3).BUFFER_LENGTH = 4
        CALLBACK(3).RETURN_ADDRESS = 0
C
C       End the item list with zeros to indicate we are finished
C
        CALLBACK(4).ITEM_CODE = 0
        CALLBACK(4).BUFFER_ADDRESS = 0
        CALLBACK(4).BUFFER_LENGTH = 0
        CALLBACK(4).RETURN_ADDRESS = 0
C
C       Return the address of the item list
C
        TPU_STARTUP = %LOC(CALLBACK)

        RETURN
        END
```

Example 13–4 Specifying a User-Written File I/O Routine in VAX C

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

/* data structures needed */

struct bpv_arg                  /* bound procedure value */
    {
    int *routine_add ;          /* pointer to routine */
    int env ;                   /* environment pointer */
    } ;

struct item_list_entry          /* item list data structure */
    {
    short int buffer_length;    /* buffer length */
    short int item_code;        /* item code */
    int *buffer_add;            /* buffer address */
    int *return_len_add;        /* return address */
    } ;

struct stream_type
    {
    int ident;                  /* stream id */
    short int alloc;            /* file size */
    short int flags;            /* file record attributes/format */
    short int length;           /* resultant file name length */
    short int stuff;            /* file name descriptor class & type */
    int nam_add;                /* file name descriptor text pointer */
    } ;

globalvalue tpu$_success;       /* TPU Success code */
globalvalue tpu$_quitting;      /* Exit code defined by TPU */

globalvalue                     /* Cleanup codes defined by TPU */
    tpu$m_delete_journal, tpu$m_delete_exith,
    tpu$m_delete_buffers, tpu$m_delete_windows, tpu$m_delete_cache,
    tpu$m_prune_cache, tpu$m_execute_file, tpu$m_execute_proc,
    tpu$m_delete_context, tpu$m_reset_terminal, tpu$m_kill_processes,
    tpu$m_close_section, tpu$m_delete_others, tpu$m_last_time;
globalvalue                     /* Item codes for item list entries */
    tpu$k_fileio, tpu$k_options, tpu$k_sectionfile,
    tpu$k_commandfile ;
globalvalue                     /* Option codes for option item */
    tpu$m_display, tpu$m_section, tpu$m_command, tpu$m_create ;

globalvalue                     /* Possible item codes in item list */
    tpu$k_access, tpu$k_filename, tpu$k_defaultfile,
    tpu$k_relatedfile, tpu$k_record_attr, tpu$k_maximize_ver,
    tpu$k_flush, tpu$k_filesize;

globalvalue                     /* Possible access types for tpu$k_access */
    tpu$k_io, tpu$k_input, tpu$k_output;

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

Example 13–4 Cont'd. on next page

Example 13–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

```
globalvalue                     /* FILEIO routine functions */
    tpu$k_open, tpu$k_close, tpu$k_close_delete,
    tpu$k_get, tpu$k_put;
int lib$establish ();           /* RTL routine to establish an event handler */
int tpu$cleanup ();             /* TPU routine to free resources used */
int tpu$control ();             /* TPU routine to invoke the editor */
int tpu$execute_inifile ();     /* TPU routine to execute initialization code */
int tpu$handler ();             /* TPU signal handling routine */
int tpu$initialize ();          /* TPU routine to initialize the editor */

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

int *data;
struct stream_type *stream;

{
    struct item_list_entry *item;
    char *access;               /* File access type */
    char filename[256];         /* Max file specification size */

    FILE *fopen();

    /* Process the item list */

    item = data;
    while (item->item_code != 0 && item->buffer_length != 0)
        {
        if (item->item_code == tpu$k_access)
            {
            if (item->buffer_add == tpu$k_io) access = "r+";
            else if (item->buffer_add == tpu$k_input) access = "r";
            else if (item->buffer_add == tpu$k_output) access = "w";
            }
        else if (item->item_code == tpu$k_filename)
            {
            strncpy (filename, item->buffer_add, item->buffer_length);
            filename [item->buffer_length] = 0;
            lib$scopy_r_dx (&item->buffer_length, item->buffer_add,
                                                  &stream->length);
            }
        else if (item->item_code == tpu$k_defaultfile)
            {                           /* Add code to handle default file  */
            }                           /* spec here                        */
        else if (item->item_code == tpu$k_relatedfile)
            {                           /* Add code to handle related       */
            }                           /* file spec here                   */
        else if (item->item_code == tpu$k_record_attr)
            {                           /* Add code to handle record        */
            }                           /* attributes for creating files    */
        else if (item->item_code == tpu$k_maximize_ver)
            {                           /* Add code to maximize version     */
            }                           /* number with existing file here   */
```

Example 13–4 Cont'd. on next page

Example 13–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

```
        else if (item->item_code == tpu$k_flush)
            {                               /* Add code to cause each record    */
            }                               /* to be flushed to disk as written */
        else if (item->item_code == tpu$k_filesize)
            {                               /* Add code to handle specification */
            }                               /* of initial file allocation here  */
        ++item;         /* get next item */
        }
    stream->ident = fopen(filename,access);
    if (stream->ident != 0)
        return tpu$_success;
    else
        return rms$_fnf;
}
/*
  This procedure closes a file
 */
close_file (data,stream)
struct stream_type *stream;

{
    close(stream->ident);
    return tpu$_success;
}
/*
  This procedure reads a line from a file
 */
read_line(data,stream)
struct dsc$descriptor *data;
struct stream_type *stream;

{
    char textline[984];                 /* max line size for TPU records */
    int len;

    globalvalue rms$_eof;               /* RMS End-Of-File code */

    if (fgets(textline,984,stream->ident) == NULL)
        return rms$_eof;
    else
        {
        len = strlen(textline);
        if (len > 0)
            len = len - 1;
        return lib$scopy_r_dx (&len, textline, data);
        }
}
/*
  This procedure writes a line to a file
 */
write_line(data,stream)
struct dsc$descriptor *data;
struct stream_type *stream;

{
    char textline[984];                 /* max line size for TPU records */
```

Example 13–4 Cont'd. on next page

Example 13–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

```
    strncpy (textline, data->dsc$a_pointer, data->dsc$w_length);
    textline [data->dsc$w_length] = 0;
    fputs(textline,stream->ident);
    fputs("\n",stream->ident);
    return tpu$_success;
}
/*
   This procedure will handle I/O for TPU
 */
fileio(code,stream,data)
int *code;
int *stream;
int *data;
{
    int status;
/* Dispatch based on code type.  Note that a full implementation of the      */
/* file I/O routines would have to handle the close and delete code properly */
/* instead of simply closing the file                                        */
    if (*code == tpu$k_open)                    /* Initial access to file */
        status = open_file (data,stream);
    else if (*code == tpu$k_close)              /* End access to file */
        status = close_file (data,stream);
    else if (*code == tpu$k_close_delete)       /* Treat same as close */
        status = close_file (data,stream);
    else if (*code == tpu$k_get)                /* Read a record from a file */
        status = read_line (data,stream);
    else if (*code == tpu$k_put)                /* Write a record to a file */
        status = write_line (data,stream);
    else
        {                                       /* Who knows what we got? */
        status = tpu$_success;
        printf ("Bad FILEIO I/O function requested");
        }
    return status;
}
/*
   This procedure formats the initialization item list and returns it as
   is return value.
 */
callrout()
{
    static struct bpv_arg add_block =
        { fileio, 0 } ;         /* BPV for fileio routine */
    int options ;
    char *section_name = "TPUSECINI";
    static struct item_list_entry arg[] =
        {/* length code             buffer add return add */
               { 4,tpu$k_fileio,     0,         0 },
               { 4,tpu$k_options,    0,         0 },
               { 0,tpu$k_sectionfile,0,         0 },
               { 0,0,                0,         0 }
        };
```

Example 13–4 Cont'd. on next page

Example 13–4 (Cont.) Specifying a User-Written File I/O Routine in VAX C

```
    /* Setup file I/O routine item entry */
    arg[0].buffer_add = &add_block;

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

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

    return arg;
}
/*
   Main program.  Initializes TPU, then passes control to it.
 */
main()
{
    int return_status ;
    int cleanup_options;
    struct bpv_arg add_block;

/* Establish as condition handler the normal VAXTPU handler */

    lib$establish(tpu$handler);

/* Setup a BPV to point to the callback routine */

    add_block.routine_add = callrout ;
    add_block.env = 0;

/* Do the initialize of VAXTPU */

    return_status = tpu$initialize(&add_block);
    if (!return_status)
        exit(return_status);

/* Have TPU execute the procedure TPU$INIT_PROCEDURE from the section file */
/* and then compile and execute the code from the command file */

    return_status = tpu$execute_inifile();
    if (!return_status)
        exit (return_status);

/* Turn control over to VAXTPU */

    return_status = tpu$control ();
    if (!return_status)
        exit(return_status);

/* Now clean up. */

    cleanup_options = tpu$m_last_time | tpu$m_delete_context;
    return_status = tpu$cleanup (&cleanup_options);
    exit (return_status);

    printf("Experiment complete");
}
```

13.5 VAXTPU Routines

The following pages describe the individual VAXTPU routines.

TPU$CLEANUP Free System Resources Used During VAXTPU Session

The TPU$CLEANUP routine cleans up internal data structures, frees memory, and restores terminals to their initial state.

This is the final routine called in each interaction with VAXTPU.

FORMAT

TPU$CLEANUP *flags*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

ARGUMENT

flags

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

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

Symbol[1]	Function
TPU$M_DELETE_JOURNAL	Closes and deletes the journal file if it is open.
TPU$M_DELETE_EXITH	Deletes VAXTPU's exit handler.
TPU$M_DELETE_BUFFERS	Deletes all text buffers. If this is not the last time you are calling VAXTPU, then all variables referring to these data structures are reset, as if by the built-in procedure DELETE. If a buffer is deleted, then all ranges and markers within that buffer, and any subprocesses using that buffer, are also deleted.

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

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

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

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

Symbol[1]	Function
TPU$M_DELETE_OTHERS	Deletes all miscellaneous preallocated data structures. Memory for these data structures is reallocated the next time TPU$INITIALIZE is called.
TPU$M_LAST_TIME	This bit should be set only when you are calling VAXTPU for the last time. Note that if you set this bit and then recall VAXTPU, the results are unpredictable.

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

DESCRIPTION

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

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

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

Notes

1. When you use the simplified interface, VAXTPU automatically sets the following flags:
   - TPU$V_RESET_TERMINAL
   - TPU$V_DELETE_BUFFERS
   - TPU$V_DELETE_JOURNAL
   - TPU$V_DELETE_WINDOWS
   - TPU$V_DELETE_EXITH
   - TPU$V_EXECUTE_PROC
   - TPU$V_EXECUTE_FILE
   - TPU$V_PRUNE_CACHE
   - TPU$V_KILL_PROCESSES
2. If this routine does not return a success status, no other calls to the editor should be made.

CONDITION VALUE RETURNED

TPU$_SUCCESS	Normal successful completion.

TPU$CLIPARSE Parse a Command Line

The TPU$CLIPARSE routine parses a command line and builds the item list for TPU$INITIALIZE.

It calls CLI$DCL_PARSE to establish a command table and a command to parse. It then calls TPU$PARSEINFO to build an item list for TPU$INITIALIZE.

FORMAT

TPU$CLIPARSE *string, fileio, call_user*

RETURNS

VMS usage: **item_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

This routine returns the address of an item list.

ARGUMENTS

string

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

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

fileio

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **read only**
mechanism: **by descriptor**

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

call_user

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **read only**
mechanism: **by descriptor**

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

TPU$CLOSE_TERMINAL Close Channel to Terminal

The TPU$CLOSE_TERMINAL routine closes VAXTPU's channel to the terminal.

FORMAT

TPU$CLOSE_TERMINAL

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. The condition value that this routine can return is listed under CONDITION VALUE RETURNED.

ARGUMENTS

None.

DESCRIPTION

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

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

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

CONDITION VALUE RETURNED

TPU$_SUCCESS	Normal successful completion.

TPU$CONTROL Pass Control to VAXTPU

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

FORMAT

TPU$CONTROL

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

None.

DESCRIPTION

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

Note: **Control is returned to your program only if an error occurs or after you enter either the built-in procedure QUIT or the built-in procedure EXIT.**

CONDITION VALUES RETURNED

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

TPU$EDIT Edit a File

The TPU$EDIT routine builds a command string from its parameters and passes it to the TPU$TPU routine.

TPU$EDIT is another entry point to VAXTPU's simplified callable interface.

FORMAT

TPU$EDIT *input, output*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

input

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

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

output

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

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

DESCRIPTION

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

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

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

CONDITION VALUES RETURNED

This routine returns any value returned by TPU$TPU.

TPU$EXECUTE_COMMAND Execute One or More VAXTPU Statements

The TPU$EXECUTE_COMMAND routine allows your program to execute VAXTPU statements.

FORMAT

TPU$EXECUTE_COMMAND *string*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

string

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by value**

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

DESCRIPTION

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

CONDITION VALUES RETURNED

TPU$_SUCCESS	Normal successful completion.
TPU$_EXITING	EXIT built-in procedure was invoked.
TPU$_QUITTING	QUIT built-in procedure was invoked.
TPU$_EXECUTEFAIL	Execution aborted. This could be because of execution errors or compilation errors.

TPU$EXECUTE_INIFILE Execute Initialization Files

The TPU$EXECUTE_INIFILE routine allows you to execute a user-written initialization file.

This routine must be executed after the editor is initialized, but before any other commands are processed.

FORMAT

TPU$EXECUTE_INIFILE

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

None.

DESCRIPTION

Calling the TPU$EXECUTE_INIFILE routine causes VAXTPU to perform the following steps:

1. The command file is read into a buffer. The default is TPU$COMMAND.TPU. If you specified a file on the command line that cannot be found, an error message is displayed and the routine is aborted.
2. If you specified the /DEBUG qualifier on the command line, the DEBUG file is read into a buffer. The default is SYS$SHARE:TPU$DEBUG.TPU.
3. The DEBUG file is compiled and executed (if available).
4. TPU$INIT_PROCEDURE is executed (if available).
5. The Command buffer is compiled and executed (if available).
6. TPU$INIT_POSTPROCEDURE is executed (if available).

Note: **If you call this routine after calling TPU$CLEANUP, you must set the flags TPU$_EXECUTEPROCEDURE and TPU$_EXECUTEFILE. Otherwise, the initialization file does not execute.**

CONDITION VALUES RETURNED

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

TPU$FILEIO Perform File Operations

The TPU$FILEIO routine handles all VAXTPU file operations. Your own file I/O routine can call this routine to perform some operations for it. However, the routine that opens the file must perform *all* operations for that file. For example, if TPU$FILEIO opens the file, it must also close it.

FORMAT

TPU$FILEIO *code, stream, data*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

code

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

Item code specifying a VAXTPU function. The **code** argument is the address of a longword containing an item code from VAXTPU specifying a function to perform. Following are the item codes that you can specify in the file I/O routine:

- TPU$K_OPEN—This item code specifies that the data parameter is the address of an item list. This item list contains the information necessary to open the file. The stream parameter should be filled in with a unique identifying value to be used for all future references to this file. The resultant file name should also be copied with a dynamic string descriptor.
- TPU$K_CLOSE—The file specified by the **stream** argument is to be closed. All memory being used by its structures can be released.
- TPU$K_CLOSE_DELETE—The file specified by the **stream** argument is to be closed and deleted. All memory being used by its structures can be released.
- TPU$K_GET—The data parameter is the address of a dynamic string descriptor to be filled with the next record from the file specified by the **stream** argument. The routine should use the routines provided by the VMS Run-Time Library to copy text into this descriptor. VAXTPU frees the memory allocated for the data read when the file I/O routine indicates that the end of the file has been reached.

- TPU$K_PUT—The data parameter is the address of a descriptor for the data to be written to the file specified by the **stream** argument.

stream

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

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

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

Figure 13–2 shows the stream data structure.

Figure 13–2 Stream Data Structure

FILE IDENTIFIER		
RFM	RAT	ALLOCATION
CLASS	TYPE	LENGTH
ADDRESS OF NAME		

ZK-4045-85

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

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

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

data

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

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

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

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

- TPU$K_ACCESS item code allows you to specify one of three item codes in the buffer address field, as follows:
  - TPU$K_IO
  - TPU$K_INPUT
  - TPU$K_OUTPUT
- TPU$K_FILENAME item code is used for specifying the address of a string to use as the name of the file you are opening. The length field contains the length of this string, and the address field contains the address.
- TPU$K_DEFAULTFILE item code is used for assigning a default file name to the file being opened. The buffer length field contains the length, and the buffer address field contains the address of the default file name.
- TPU$K_RELATEDFILE item code is used for specifying a related file name for the file being opened. The buffer length field contains the length, and the buffer address field contains the address of a string to use as the related file name.
- TPU$K_RECORD_ATTR item code specifies that the buffer address field contains the value for the record attribute byte in the FAB (FAB$B_RAT) used for file creation.
- TPU$K_RECORD_FORM item code specifies that the buffer address field contains the value for the record format byte in the FAB (FAB$B_RFM) used for file creation.
- TPU$K_MAXIMIZE_VER item code specifies that the version number of the output file should be one higher than the highest existing version number.
- TPU$K_FLUSH item code specifies that the file should have every record flushed after it is written.
- TPU$K_FILESIZE item code is used for specifying a value to be used as the allocation quantity when creating the file. The value is specified in the buffer address field.

DESCRIPTION

By default, TPU$FILEIO creates variable-length files with carriage-return record attributes (fab$b_rfm = var, fab$b_rat = cr). If you pass to it the TPU$K_RECORD_ATTR or TPU$K_RECORD_FORM item, that item is used instead. The following combinations of formats and attributes are acceptable:

Format	Attributes
STM,STMLF,STMCR	0,BLK,CR,BLK+CR
VAR	0,BLK,FTN,CR,BLK+FTN,BLK+CR

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

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

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

CONDITION VALUES RETURNED

The TPU$FILEIO routine returns an RMS status code to VAXTPU. The file I/O routine is responsible for signaling all errors if any messages are desired.

TPU$HANDLER VAXTPU Condition Handler

The TPU$HANDLER routine is VAXTPU's condition handler.

The VAXTPU condition handler invokes the Put Message (SYS$PUTMSG) system service, passing it the address of TPU$MESSAGE.

FORMAT

TPU$HANDLER *signal_vector, mechanism_vector*

RETURNS

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

Longword condition value.

ARGUMENTS

signal_vector

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

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

mechanism_vector

VMS usage: **arg_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

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

DESCRIPTION

The TPU$MESSAGE routine performs the actual output of the message. The Put Message (SYS$PUTMSG) system service only formats the message. It gets the settings for the message flags and facility name from the variables described in Section 13.1.2. Those values can be modified only by the VAXTPU built-in procedure SET.

If the condition value received by the handler has a FATAL status or does not have VAXTPU's facility code, the condition is resignaled.

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

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

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

See the *Introduction to VMS System Routines* for information about the VAX Condition Handling Standard.

TPU$INITIALIZE Initialize VAXTPU for Editing

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

FORMAT

TPU$INITIALIZE *callback [,user_arg]*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

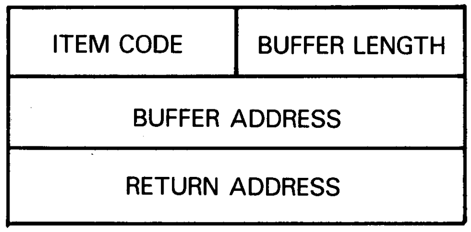
callback

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **read only**
mechanism: **by descriptor**

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

Callable VAXTPU defines thirteen item codes that can be used for specifying initialization parameters. You do not have to arrange the item codes in any particular order in the list. Figure 13–3 shows the general format of an item descriptor. For information about how to build an item list, refer to the VMS programmer's manual associated with the language you are using.

Figure 13–3 Format of an Item Descriptor



ZK-4044-85

The return address in an item descriptor is usually *0*.

The following item codes are available.

Item Code	Description
TPU$_OPTIONS	Enables the command qualifiers. Ten bits in the buffer address field correspond to the various TPU command qualifiers. The remaining 22 bits in the buffer address field are reserved.
TPU$_JOURNALFILE	Passes the string specified with the /JOURNAL qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is available with GET_INFO (COMMAND_LINE,"JOURNAL_FILE"). This string may be a null string.
TPU$_SECTIONFILE	Passes the string that is the name of the binary initialization file (section file) to be mapped in. The buffer length field is the length of the string and the buffer address field is the address of the string. The VAXTPU CLD file has a default value for this string. If the TPU$V_SECTION bit is set, this item code must be specified.
TPU$_OUTPUTFILE	Passes the string specified with the /OUTPUT qualifier. The buffer length field is the length of the string, and the buffer address field specifies the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, "OUTPUT_FILE"). The string may be a null string.
TPU$_DISPLAYFILE	Passes the string specified with the /DISPLAY qualifier. The buffer length field is the length of the string, and the buffer address field specifies the address of the string.

Item Code	Description
TPU$_COMMANDFILE	Passes the string specified with the /COMMAND qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, "COMMAND_FILE"). The string may be a null string.
TPU$_FILENAME	Passes the string that is the name of the input file specified in the command line. The buffer length field specifies the length of this string, and the buffer address field specifies its address. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE, "FILE_NAME"). This file name may be a null string.
TPU$_FILEIO	Passes the bound procedure value of a routine to be used for handling file operations. You may provide your own file I/O routine, or you can call TPU$FILEIO, the utility routine provided by VAXTPU for handling file operations. The buffer address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that TPU loads into R1 before calling the routine.
TPU$_CALLUSER	Passes the bound procedure value of the user-written routine that the built-in procedure CALL_USER is to call. The buffer address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that TPU loads into R1 before calling the routine.
TPU$_INIT_FILE	Passes the string specified with the /INITIALIZATION qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string. This string is returned by the built-in procedure GET_INFO (COMMAND_LINE,"INIT_FILE").
TPU$_START_LINE	Passes the starting line number for the edit. The buffer address field contains the first of the two integer values you specified as part of the /START_POSITION command qualifier. The value is available using the built-in procedure GET_INFO (COMMAND_LINE,"LINE"). Usually an initialization procedure uses this information to set the starting position in the main editing buffer. The first line in the buffer is line 1.

Item Code	Description
TPU$_START_CHAR	Passes the starting column position for the edit. The buffer address field contains the second of the two integer values you specified as part of the /START_POSITION command qualifier. The value is available using the built-in procedure GET_INFO (COMMAND_LINE, "CHARACTER"). Usually an initialization procedure uses this information to set the starting position in the main editing buffer. The first column on a line corresponds to character 1.
TPU$_CONTROLC	Passes the bound procedure value of a routine to be used for handling CTRL/C ASTs. VAXTPU calls the routine when a CTRL/C AST occurs. If the routine returns a FALSE value, VAXTPU assumes that the CTRL/C has been handled. If the routine returns a TRUE value, VAXTPU aborts any currently executing VAXTPU procedure. The buffer address field specifies the address of a two-longword vector. The first longword of the vector contains the address of the routine. The second longword specifies the environment value that TPU loads into R1 before calling the routine.
TPU$_DEBUGFILE	Passes the string specified with the /DEBUG command qualifier. The buffer length field is the length of the string, and the buffer address field is the address of the string.

The following table shows the bits and corresponding masks enabled by the item code TPU$K_OPTIONS.

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

[1]TPU$M . . . indicates a mask.

[2]TPU$V . . . indicates a bit item.

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

user_arg

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism:

User argument. The **user_arg** argument can be used as desired. One common use is to pass the address of the item list used by TPU$INITIALIZE.

DESCRIPTION

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

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

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

CONDITION VALUES RETURNED

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

TPU$MESSAGE Write Message String

The TPU$MESSAGE routine writes error messages and strings using the built-in procedure, MESSAGE.

You can call this routine to have messages written and handled in a manner consistent with VAXTPU. This routine should be used only after TPU$EXECUTE_INIFILE.

FORMAT

TPU$MESSAGE *string*

RETURNS

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

Longword condition value.

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

ARGUMENT

string

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

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

TPU$PARSEINFO Parse Command Line and Build Item List

The TPU$PARSEINFO routine parses a command and builds the item list for TPU$INITIALIZE.

FORMAT

TPU$PARSEINFO *fileio, call_user*

RETURNS

VMS usage: **item_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The routine returns the address of an item list.

ARGUMENTS

fileio

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **read only**
mechanism: **by descriptor**

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

call_user

VMS usage: **vector_longword_unsigned**
type: **bound procedure value**
access: **read only**
mechanism: **by descriptor**

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

DESCRIPTION

The TPU$PARSEINFO routine parses a command and builds the item list for TPU$INITIALIZE.

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

TPU$TPU Invoke VAXTPU

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

FORMAT

TPU$TPU *command*

RETURNS

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

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENT

command
VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Command string. Note that the verb is TPU instead of EDIT/TPU. The **command** argument is the address of a descriptor of a command line.

DESCRIPTION

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

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

If your application parses information that is not related to the operation of VAXTPU, make sure the application obtains and uses all non-VAXTPU parse information before the application calls PU$EDIT. The reason is that TPU$EDIT destroys all parse information obtained and stored before TPU$EDIT was called.

CONDITION VALUES RETURNED

This routine returns any condition value returned by TPU$INITIALIZE, TPU$EXECUTE_INFILE, TPU$CONTROL, and TPU$CLEANUP.

FILEIO User-Written Routine to Perform File Operations

The user-written FILEIO routine is used to handle VAXTPU file operations. The name of this routine can be either your own file I/O routine or the name of the VAXTPU file I/O routine (TPU$FILEIO).

FORMAT

FILEIO *code, stream, data*

RETURNS

VMS usage: **cond_value**
type: **longword (usigned)**
access: **write only**
mechanism: **by reference**

Longword condition value. Most utility routines return a condition value in R0. Condition values that this routine can return are listed under CONDITION VALUES RETURNED.

ARGUMENTS

code

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

Item code specifying a VAXTPU function. The **code** argument is the address of a longword containing an item code from VAXTPU, which specifies a function to perform.

stream

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

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

data

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

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

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

DESCRIPTION

The bound procedure value of this routine is specified in the item list built by the callback routine. This routine is called to perform file operations. Instead of using your own file I/O routine, you can call TPU$FILEIO and pass it the parameters for any file operation that you do not want to handle. Note, however, that TPU$FILEIO must handle all I/O requests for any file it opens. Also, if it does not open the file, it cannot handle any I/O requests for the file. In other words, you cannot intermix the file operations between your own file I/O routine and the one supplied by VAXTPU.

CONDITION VALUES RETURNED

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

HANDLER User-Written Condition Handling Routine

The user-written HANDLER routine performs condition handling.

FORMAT

HANDLER *signal_vector, mechanism_vector*

RETURNS

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

Longword condition value.

ARGUMENTS

signal_vector

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

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

mechanism_vector

VMS usage: **arg_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

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

DESCRIPTION

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

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

INITIALIZE User-Written Initialization Routine

The user-written initialization callback routine is passed to TPU$INITIALIZE as a bound procedure value, and called to supply information needed to initialize VAXTPU.

FORMAT

INITIALIZE

RETURNS

VMS usage: **item_list**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

This routine returns the address of an item list.

ARGUMENTS

None.

DESCRIPTION

The user written initialization call-back routine is passed to TPU$INITIALIZE as a bound procedure value, and called to supply information needed to initialize VAXTPU. The initialization call-back routine is called with no parameters. It is expected to return the address of an item list containing initialization parameters. Because the item list is used outside the scope of the initialization call-back routine, it should be allocated in static memory.

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

USER User-Written Routine Called from a VAXTPU Editing Session

The user-written USER routine allows your program to get control during a VAXTPU editing session (for example, to leave the editor temporarily and perform a calculation).

This user-written routine is invoked by the VAXTPU built-in procedure CALL_USER. The built-in procedure CALL_USER passes three parameters to this routine. These parameters are then passed to the appropriate part of your application to be used as specified. (For example, they may be used as operands in a calculation within a FORTRAN program.) Using the string routines provided by the VMS Run-Time Library, your application fills in the **stringout** parameter in the call-user routine, which returns the **stringout** value to the built-in procedure CALL_USER.

FORMAT

USER *integer, stringin, stringout*

RETURNS

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

Longword condition value.

ARGUMENTS

integer

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

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

stringin

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

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

stringout

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

Return value for the built-in procedure CALL_USER. Your program should fill in this descriptor with a dynamic string allocated by the string routines provided by the VMS Run-Time Library. The VAXTPU editor frees this string when necessary.

DESCRIPTION

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

EXAMPLE

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

     RS = STR$COPY_DX(Z,local_copy(1:lclen))
     TPU$CALLUSER = RS
     END
```

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

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

Index

A

B

C

D

E

F

G

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

Reader's Comments

VMS
Utility Routines Manual
AA–LA67A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page Description

_____ ______________________________

_____ ______________________________

_____ ______________________________

_____ ______________________________

_____ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** _____ of the software this manual describes.

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________________ Phone __________

— Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here

Cut Along Dotted Line