VMS

digital

VMS RTL String Manipulation (STR$) Manual

Order Number AA–LA75A–TE

VMS RTL String Manipulation (STR$) Manual

Order Number: AA-LA75A-TE

April 1988

This manual documents the string manipulation routines contained in the STR$ facility of the VMS Run-Time Library.

Revision/Update Information:	This document supersedes the STR$ section of the *VAX/VMS Run-Time Library Routines Reference Manual,* Version 4.4.
Software Version:	VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1988 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

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.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

ZK4613

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

STR$ REFERENCE SECTION

Preface

This manual provides users of the VMS operating system with detailed usage and reference information on string manipulation routines supplied in the STR$ facility of the Run-Time Library.

Run-Time Library routines can only be used in programs written in languages that produce native code for the VAX hardware. At present, these languages include VAX MACRO and the following compiled high-level languages:

VAX Ada
VAX BASIC
VAX BLISS-32
VAX C
VAX COBOL
VAX COBOL-74
VAX CORAL
VAX DIBOL
VAX FORTRAN
VAX Pascal
VAX PL/I
VAX RPG
VAX SCAN

Interpreted languages that can also access Run-Time Library routines include VAX DSM and DATATRIEVE.

Intended Audience

This manual is intended for system and application programmers who want to call Run-Time Library routines.

Document Structure

This manual is organized into two parts as follows:

- Part I provides guidelines and reference material on STR$ routines.

  Chapter 1 provides a brief overview of the STR$ routines and lists the routines and their functions.

  Chapter 2 discusses in detail how to use the Run-Time Library STR$ routines.

- Part II provides detailed reference information on each routine contained in the STR$ facility of the Run-Time Library. This information is presented using the documentation format described in the *Introduction to the VMS Run-Time Library*. Routine descriptions appear in alphabetical order by routine name.

Associated Documents

The Run-Time Library routines are documented in a series of reference manuals. A general overview of the Run-Time Library and a description of how the Run-Time Library routines are accessed are presented in the *Introduction to the VMS Run-Time Library*. Descriptions of the other RTL facilities and their corresponding routines and usages are discussed in the following books:

- The *VMS RTL DECtalk (DTK$) Manual*
- The *VMS RTL Library (LIB$) Manual*
- The *VMS RTL Mathematics (MTH$) Manual*
- The *VMS RTL General Purpose (OTS$) Manual*
- The *VMS RTL Parallel Processing (PPL$) Manual*
- The *VMS RTL Screen Management (SMG$) Manual*

The VAX Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines*, contains useful information for anyone who wants to call Run-Time Library routines.

Application programmers in any language may refer to the *Guide to Creating VMS Modular Procedures* for the Modular Programming Standard and other guidelines.

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

The *Guide to Using VMS Command Procedures* may also be useful.

For a complete list and description of the manuals in the VMS documentation set, see the *Overview of VMS Documentation*.

Conventions

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.

Other conventions used in the documentation of Run-Time Library routines are described in the *Introduction to the VMS Run-Time Library.*

New and Changed Features

The following STR$ routine has been added to the VMS Run-Time Library for Version 5.0:

Table 1 New STR$ Routine for V5.0

Routine	Function
STR$ELEMENT	Extract Delimited Element Substring

1 Overview of the STR$ Facility

The *VMS RTL String Manipulation (STR$) Manual* discusses the Run-Time Library STR$ routines that perform string functions. This chapter gives a brief overview of the STR$ routines and lists the routines and their functions. Chapter 2 explains in detail how the RTL handles strings. The second part of this manual is a reference section describing all the Run-Time Library STR$ routines.

The STR$ facility provides you with routines that perform the following functions:

- Perform mathematical operations on strings
- Compare strings
- Extract and replace substrings
- Append and concatenate strings
- Copy strings
- Search characters and substrings
- Free and allocate dynamic strings
- Perform miscellaneous functions on strings

Mathematical Operation Routines

STRADD, STRDIVIDE, STRMUL, STRRECIP, and STR$ROUND are routines that perform mathematical functions on strings. These routines allow you to add, divide, and multiply two strings. You can also take the reciprocal of a string or round a string.

Compare Routines

STR$CASE_BLIND_COMPARE, STR$COMPARE, STR$COMPARE_EQL, and STR$COMPARE_MULTI compare the contents of two strings and return a value (–1, 0, or 1) that denotes whether the first string is less than, equal to, or greater than the second string.

Extract and Replace Routines

STR$ELEMENT, STR$LEFT, STRLEN_EXTR, STRPOS_EXTR, STR$REPLACE, and STR$RIGHT are routines that extract a substring from a string or replace a substring with another substring.

Append and Concatenate Routines

STR$APPEND and STR$CONCAT allow you to append a string to another string, or to concatenate up to 254 strings into one string.

Copy Routines

STR$COPY_DX and STR$COPY_R allow you to copy a string passed by descriptor or by reference to another string.

Search Routines

STR$FIND_FIRST_IN_SET, STR$FIND_FIRST_NOT_IN_SET, and STR$FIND_FIRST_SUBSTRING are routines that search a string one character at a time, comparing each character to the characters in a specified set of characters.

Allocate and Deallocate Routines

STR$FREE1_DX and STR$GET1_DX deallocate and allocate a dynamic string.

Miscellaneous Routines

STR$ANALYZE_SDESC, STR$DUPL_CHAR, STR$MATCH_WILD, STR$POSITION, STR$TRANSLATE, STR$TRIM, and STR$UPCASE analyze string descriptors, duplicate a character, match wildcard specifications, prefix a string, return a relative position, translate matched characters, trim trailing blanks and tabs, and convert strings to uppercase characters.

The following list contains all of the STR$ routines and their functions.

Table 1–1 STR$ Routines

Routine Name	Function
STR$ADD	Add two decimal strings
STR$ANALYZE_SDESC	Analyze a string descriptor
STR$APPEND	Append a string
STR$CASE_BLIND_COMPARE	Compare strings without regard to case
STR$COMPARE	Compare two strings
STR$COMPARE_EQL	Compare two strings for equality
STR$COMPARE_MULTI	Compare two strings for equality using the DEC Multinational Character Set
STR$CONCAT	Concatenate two or more strings
STR$COPY_DX	Copy a source string passed by descriptor to a destination string
STR$COPY_R	Copy a source string passed by reference to a destination string
STR$DIVIDE	Divide two decimal strings
STR$DUPL_CHAR	Duplicate character *n* times
STR$ELEMENT	Extract delimited element substring
STR$FIND_FIRST_IN_SET	Find the first character in a set of characters
STR$FIND_FIRST_NOT_IN_SET	Find the first character that does not occur in the set
STR$FIND_FIRST_SUBSTRING	Find the first substring in the input string
STR$FREE1_DX	Free one dynamic string
STR$GET1_DX	Allocate one dynamic string
STR$LEFT	Extract a substring of a string
STR$LEN_EXTR	Extract a substring of a string

Table 1–1 (Cont.) STR$ Routines

Routine Name	Function
STR$MATCH_WILD	Match a wildcard specification
STR$MUL	Multiply two decimal strings
STR$POSITION	Return relative position of a substring
STR$POS_EXTR	Extract a substring of a string
STR$PREFIX	Prefix a string
STR$RECIP	Return the reciprocal of a decimal string
STR$REPLACE	Replace a substring
STR$RIGHT	Extract a substring of a string
STR$ROUND	Round or truncate a decimal string
STR$TRANSLATE	Translate matched characters
STR$TRIM	Trim trailing blanks and tabs
STR$UPCASE	Convert string to all uppercase

2 Introduction to String Manipulation (STR$) Routines

This chapter explains in detail the following topics:

- Types of strings recognized by Run-Time Library routines
- Relationship of descriptor classes to string semantics
- Differences in string handling among the LIB$, OTS$, and STR$ facilities of the Run-Time Library
- Conventions for reading and writing string arguments in the Run-Time Library string routines
- Selection of the proper string manipulation routines
- Allocation and deallocation of dynamic string resources

2.1 String Semantics in the Run-Time Library

The *semantics* of a string refers to the conventions that determine how a string is stored, written, and read. The VAX architecture supports three string semantics: fixed length, varying length, and dynamic length.

2.1.1 Fixed-Length Strings

Fixed-length strings have two attributes:

- An address
- A length

The length of a fixed-length string is constant. It is usually initialized when the program is compiled or linked. After initialization, this length is read but never written. When a Run-Time Library routine copies a source string into a longer fixed-length destination string, the routine pads the destination string with trailing blanks.

When you pass a string to a Run-Time Library routine, you pass the string by descriptor. For a fixed-length string, the descriptor must contain this information:

- The descriptor class
- The data type of the string
- The length of the string
- The address of the beginning of the string

In most cases, you will not have to construct an actual descriptor. By default, most VAX languages pass strings by descriptor. For information about how the language you are using handles strings, see your language reference manual. For more information about descriptors used for fixed-length strings, refer to the *Introduction to VMS System Routines.*

Note: In contrast to Run-Time Library routines, system services do not pad output strings. For this reason, when a program calls a system service that returns a fixed-length string, the program should supply an additional argument that indicates how many bytes the system service actually deposited in the fixed-length buffer of the calling program. Some system service routines have corresponding Run-Time Library routines that provide the proper semantics for fixed-length, varying-length, and dynamic output strings.

2.1.2 Varying-Length Strings

Varying-length strings have the following three attributes:

- A current length
- An address
- A maximum length

The current length of a varying-length string is stored in a two-byte field, called CURLEN, preceding the text of the string. The address of the string points to the beginning of this CURLEN field, not to the beginning of the string's text.

The maximum string length is a field in the string's descriptor. The maximum string length field specifies how much space is allocated to the string in a program. The maximum string length is fixed and does not change.

The value in the CURLEN field specifies how many bytes beyond the CURLEN field are occupied by the string's text. The character positions beyond this range are reserved for the growth of the string. Their contents are undefined.

For example, assume a varying string whose CURLEN is 3 and whose maximum length is 6. If a string 'ABCD' is copied into this string, the result is 'ABCD' and the CURLEN is changed to 4. If a string 'XYZ' is now copied into the same varying string, the resulting string is 'XYZ' with a CURLEN of 3. The maximum length is still 6. The bytes beyond the range designated by CURLEN are undefined.

2.1.3 Dynamic-Length Strings

Dynamic-length strings have two attributes:

- A current length
- An address pointing to the beginning of the text

Theoretically, dynamic strings have unbounded length. The length field is an unsigned value occupying two bytes, however, so that its maximum value is 65,535. Thus the length of a dynamic string is limited to 65,535 characters. In most cases, a program allocates only the descriptor for this kind of string.

The actual space for a dynamic-length string is allocated from heap storage by the Run-Time Library. When a Run-Time Library routine copies a character string into a dynamic string, and the currently allocated heap storage is not large enough to contain the string, the currently allocated storage returns to a pool of heap storage maintained by the string routines. Then the string routines obtain a new area of the correct size. As a result of this process of deallocation and reallocation, both the current-length field and the address portion of the string's descriptor may change. Often, dynamic strings are the most convenient type to write.

The Run-Time Library string manipulation routines are the only routines that you should use to alter the length or address of a dynamic string. Do not use LIB$GET_VM for this purpose. For information on allocating dynamic strings, see Section 2.1.3.

2.1.4 Examples

The following examples illustrate what happens when the string 'ABCDEF' (of length 6) is copied into various destination strings.

- Fixed-length string

  If 'ABCDEF' is copied into a fixed-length string, three results are possible:

  1 If the length of the output string is greater than the length of the source string, the string is padded with trailing spaces.

Length of output string	10
Result	*'ABCDEF '*

  2 If the length of the output string is the same as that of the input string, the string is simply copied with no modification.

Length of output string	6
Result	*'ABCDEF'*

  3 If the length of the output string is less than the length of the source string, truncation on the right occurs.

Length of output string	3
Result	*'ABC'*

- Varying-length string

  If the same string ('ABCDEF') is copied into a varying-length string, two results are possible:

  1 If the MAXSTRLEN field of the destination is greater than or equal to the length of the source, the input string is written into the output string without modification, and the CURLEN (current length) field of the output string becomes 6.

  2 If the MAXSTRLEN field of the destination is less than the length of the source string, the source string is truncated on the right and the CURLEN field is rewritten to its current length. For example, if MAXSTRLEN = 4, the resulting string contains 'ABCD' and CURLEN = 4.

- Dynamic-length string

  If a Run-Time Library routine copies the string 'ABCDEF' into a dynamic destination string, three results are possible:

  1. If the length of the destination string is greater than the length of the source string (6), the result is a dynamic string of length 6 containing 'ABCDEF'. No padding takes place. The Run-Time Library may deallocate the string and reallocate a new string closer in length to the length of the source string.
  2. If the length of the destination string is less than the length of the source string, the result is also 'ABCDEF', with a length of 6. The Run-Time Library deallocates the destination string and allocates a new string large enough to hold the 6 characters.
  3. If the destination string and source string are of equal length, a simple copy is done. No allocation, deallocation, or padding takes place, and the destination descriptor is not modified.

2.2 Descriptor Classes and String Semantics

A calling program passes strings to a Run-Time Library STR$ routine by descriptor. That is, the argument list entry for an input or output string is actually the address of a string descriptor. The calling program allocates a descriptor for the input string that indicates the string's address and length, so that the called routine can find the string's text and operate on it. The calling program also allocates a descriptor for the output string. In addition to length and address fields, each descriptor contains a field (DSC$B_CLASS) indicating the descriptor's class. The STR$ routine reads the class field to determine whether to write the output string as fixed length, varying length, or dynamic string.

To determine the address and length of the data in the input string, Run-Time Library routines call LIB$ANALYZE_SDESC or STR$ANALYZE_SDESC. LIB$ routines call LIB$ANALYZE_SDESC; STR$ routines call STR$ANALYZE_SDESC, so that they can signal errors instead of returning a status.

The STR$ routines provide a centralized facility for analyzing string descriptors, so that string-handling routines can function independently of the class of the input string. This means that if the Run-Time Library recognizes new string types, only the analysis routine needs to be changed, not the string routines themselves. If you are writing a routine that recognizes all the string types recognized by the Run-Time Library, your routine should first call LIB$ANALYZE_SDESC or STR$ANALYZE_SDESC to obtain the address and length of the input string.

You can also use the string descriptor analysis routines to find the length of a returned string. Assume that your called routine calls one of the Run-Time Library string-copying routines to create a new string. You now want the called routine to return the actual length of the new string to the calling program. The called routine calls LIB$ANALYZE_SDESC to compute this length. This sequence of calls allows you to create the new string without knowing its ultimate length at the time it is created.

2.2 Descriptor Classes and String Semantics

The Run-Time Library routines recognize the following classes of string descriptors:

- Z—unspecified
- S—scalar, fixed-length string
- SD—decimal scalar
- VS—varying-length string
- D—dynamic string
- A—array
- NCA—noncontiguous array

For a detailed description of these descriptor classes and their fields, see the VAX Procedure Calling and Condition Handling Standard in *Introduction to VMS System Routines.*

Table 2–1 indicates how the Run-Time Library routines access the fields of the descriptor for input and output string arguments. Given the class of the string and the field of the descriptor, the table shows whether the routine reads, writes, or modifies the field.

Table 2–1 String Passing Techniques Used by the Run-Time Library

	String Descriptor Fields		
String Type	**Class**	**Length**	**Pointer**
Input Argument to Routines			
Input string passed by descriptor	Read	Read	Read
Output Argument from Routines; Called Routine Assumes the Descriptor Class			
Output string passed by descriptor, fixed-length	Ignored	Read	Read
Output string passed by descriptor, dynamic	Ignored	Read, can be modified	Read, can be modified
Output Argument from Routines; Calling Program Specifies the Descriptor Class in the Descriptor			
Output string, fixed-length— DSC$K_CLASS = S, Z, A, NCA, SD	Read	Read	Read
Output string, dynamic— DSC$K_CLASS_D	Read	Read, can be modified	Read, can be modified
Output string, varying-length— DSC$K_CLASS_VS	Read	MAXSTRLEN is read; CURLEN is modified	Read

2.2.1 Conventions for Reading Input String Arguments

When a calling program passes an input string as an argument to a Run-Time Library routine, the argument list entry is the address of a descriptor. The called routine examines the class code field of the descriptor to determine where the routine looks to find the length of the string and the first byte of the string's text. For each descriptor class, Table 2–2 indicates which field of the descriptor the routine uses to locate the text and length of the string. For diagrams of the descriptors, see the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*.

Table 2–2 How Run-Time Library Routines Read Strings

Class	Length	Address of First Byte of Data
Z	DSC$W_LENGTH	DSC$A_POINTER
S	DSC$W_LENGTH	DSC$A_POINTER
D	DSC$W_LENGTH	DSC$A_POINTER
A	DSC$L_ARSIZE	DSC$A_POINTER
SD	DSC$W_LENGTH	DSC$A_POINTER
NCA	DSC$L_ARSIZE	DSC$A_POINTER
VS	Word at DSC$A_POINTER (CURLEN field)	Value of DSC$A_POINTER + 2 (Byte after CURLEN field)

Note:

- If the descriptor class is NCA, it is assumed that the string is actually contiguous.
- If the descriptor class is A or NCA, the element size is assumed to be one byte.
- If the descriptor class is A or NCA, and the array being passed is multidimensional, you should be aware of how your language stores arrays (by column or by row).

2.2.2 Semantics for Writing Output String Arguments

Normally, Run-Time Library routines return the result of an operation in one of two ways:

- The called routine returns the result as a *function value* in R0/R1. If the result is too large to fit in R0/R1, it is returned as a function value in the first position in the argument list, and the other arguments are shifted one position to the right.
- The called routine returns the result as an *output argument*. The calling program passes to the called routine an argument naming a variable in which the routine will write the output string. In each RTL routine, the access field of an output argument contains "write only".

The STR$ routines that produce string results use the first method to pass the results back to the calling program. Because a return string, by definition, does not fit in R0/R1, the function value from a STR$ routine is placed in the first position in the argument list.

On the other hand, the string manipulation routines in the LIB$ and OTS$ facilities use the second method. They return their results as output arguments.

For example, there are three entry points for the string-copying routine, LIB$SCOPY_DXDX, OTS$SCOPY_DXDX, and STR$COPY_DX. These copy the source string (**source-string**) to the destination string (**destination-string**). Their formats are as follows:

```
LIB$SCOPY_DXDX(SOURCE-STRING ,DESTINATION-STRING)
OTS$SCOPY_DXDX(SOURCE-STRING ,DESTINATION-STRING)
STR$COPY_DX(DESTINATION-STRING ,SOURCE-STRING)
```

Because the STR$ entry point places the result string in the first position, you can call STR$COPY_DX using a function reference in languages that support string functions. In FORTRAN, for example, you can use a function reference to invoke STR$COPY_DX in the following two ways:

```
CHARACTER*80 STR$COPY_DX

RETURN-STATUS = STR$COPY_DX(DESTINATION-STRING, SOURCE-STRING)

or

DESTINATION-STRING = STR$COPY_DX(SOURCE-STRING)
```

If you use the second form, you cannot access the return status, which is used to indicate truncation.

If you use a function reference to invoke a string manipulation routine in a language that does not support the concept of a string function (such as MACRO, BLISS, and Pascal), you must place the destination string variable in the argument list. In Pascal, for example, you can use a function reference to invoke STR$COPY_DX as follows:

```
STATUS := STR$COPY_DX(DESTINATION-STRING, SOURCE-STRING);
```

However, the following statement results in an error:

```
DESTINATION-STRING := STR$COPY_DX(SOURCE-STRING)
```

In addition to allocating a variable for the output string, the calling program must allocate the space for and fill in the fields of the output string descriptor at compile, link, or run time. High-level languages do this automatically.

When a Run-Time Library routine returns an output string argument to the calling program, the argument list entry is the address of a descriptor. The routine determines the semantics of the output string (fixed, varying, or dynamic) by examining the class of the descriptor for the destination string. Given the class of the output string's descriptor, Table 2–3 specifies the semantics used by Run-Time Library routines when writing the string.

2.2 Descriptor Classes and String Semantics

Table 2–3 Semantics and Descriptor Classes

Class	Description	Restrictions	Semantics
Z	Unspecified	Treated as class S	Fixed-length string
S	Scalar, string	None	Fixed-length string
D	Dynamic string	String length < 64K bytes (DSC$W_LENGTH < 64K)	Dynamic string
A	Array	Array is one-dimensional (DSC$B_DIMCT = 1) String length < 64K bytes (DSC$L_ARSIZE < 64K) Length of array elements is one byte (DSC$W_LENGTH = 1)	Fixed-length string
SD	Scalar decimal	DSC$B_DIGITS and DSC$B_SCALE are ignored	Fixed-length string
NCA	Noncontiguous array	Array is one-dimensional (DSC$B_DIMCT = 1) String length < 64K bytes (DSC$L_ARSIZE < 64K) Array is contiguous (DSC$L_S1 = DSC$W_LENGTH) Length of array elements is one byte (DSC$W_LENGTH = 1)	Fixed-length string
VS	Varying string	Current length less than maximum length of string (CURLEN <= DSC$W_MAXSTRLEN)	Varying string

When a called routine returns a string whose length cannot be determined by the calling routine, the calling routine should also pass an optional argument to contain the output length. This argument should be an unsigned 16-bit integer. If the output string is a fixed-length string, the length argument would reflect the number of characters written, not counting the fill characters.

The output length argument is useful, for instance, when your program is reading variable-length records. The program can read the input strings into a buffer that is large enough to contain the largest. When you wish to perform the next operation on the contents of the buffer, the length argument indicates exactly how many characters have been read, so that the program does not need to manipulate the whole buffer.

For example, LIB$GET_INPUT has the optional argument **resultant-length**. If LIB$GET_INPUT is called with a fixed-length, five-character string as an argument, and the routine reads a record containing 'ABC', then **resultant-length** will have a value of 3 and the output string will be *'ABC '*. But if the routine reads a record containing the value 'ABCDEFG', **resultant-length** will have a value of 5, and the output string will be 'ABCDE'. In either case, the calling program will know exactly how many characters (not counting fillers) the routine has read.

A routine such as STR$COPY_DX does not need the length argument, because the calling program can determine the length of the output string. If the output string is dynamic, the length is the same as the input string length. If the output string is fixed-length, the length is the shorter of the two input lengths.

2.3 Selecting String Manipulation Routines

To perform a given string manipulation operation, you can often choose one of several routines from the Run-Time Library. The LIB$, OTS$, and STR$ facilities all contain string copying and dynamic string allocation routines. Furthermore, a MACRO or BLISS program can call several of these routines using either a JSB or CALL entry point.

You should consider the factors discussed below when choosing the routine to perform the desired operation.

2.3.1 Efficiency

One of the major considerations in choosing among several routines is the efficiency of the various options.

In general, LIB$ and STR$ routines execute more efficiently than the corresponding OTS$ routines. OTS$ routines usually invoke the LIB$ entry point to perform an operation.

JSB entry points usually execute more efficiently than CALL entry points. However, a high-level language cannot explicitly access a JSB entry point. Further, a JSB entry point does not establish a stack frame and executes entirely in the environment of the calling program. This means, for instance, that the called routine cannot establish its own condition handler, so it cannot regain control if an exception occurs during execution. Also, some of the efficiency gained by using the JSB may be lost because the calling routine must explicitly save all of the registers that the called routine uses.

Some routines perform a specific operation that is a subset of a more general capability. These more specialized routines are usually more efficient. For example, if you want to join two strings together, STR$APPEND and STR$PREFIX are more specific, and more efficient, than STR$CONCAT. Similarly, STR$LEFT and STR$RIGHT are subsets of the capabilities of STR$POS_EXTR.

2.3.2 Argument Passing

The mechanism by which a routine passes or receives arguments may also help you to decide among several routines that perform basically the same function.

Routines in the LIB$ and STR$ facilities pass scalar input arguments by reference to CALL entry points and by immediate value to JSB entry points. OTS$ routines pass scalar input arguments by immediate value to all entry points. For most high-level languages, the default passing mechanism is by reference. Thus if you call a LIB$ or STR$ routine from one of these languages, it will not be necessary to specify the passing mechanism for input scalar arguments.

Some routines require you to set up and pass more arguments than others. For example, some use a single string descriptor, while others require separate arguments for the length and the address of the string. Which routine you choose then depends on the form of the information already available in your program.

2.3.3 Error Handling

Routines from the LIB$, OTS$, and STR$ facilities handle errors in string copying differently:

- LIB$

  The LIB$ string-copying routines return a completion status. When an output string must be truncated and its length depends on input arguments, LIB$ routines consider this to be a partial success; they therefore return LIB$_STRTRU instead of a severe error. This process corresponds to the convention of many higher-level languages, which do not consider truncation to be an error.

- STR$

  The STR$ string-copying routines generally signal errors instead of returning a completion status. In the case of truncation errors, STR$ routines return an error status with a severity of WARNING (STR$_TRU). STR$ routines consider range errors to be qualified success.

- OTS$

  The OTS$ string-copying routines also signal errors that are considered fatal (such as invalid descriptor class). However, they are designed to leave the registers as they would be after a MOVC5 instruction. Thus, the call entry point, like MOVC5, returns in R0 the number of bytes in the source string that were not moved to the destination string. The JSB entry points for OTS$ string-copying routines also leave registers R1 through R5 as they would be after a MOVC5 instruction. See the *VAX Architecture Reference Manual* for a complete description of the MOVC5 instruction.

Table 2–4 indicates the errors that each facility considers severe, and the corresponding message:

Table 2–4 Severe Errors, by Facility

Error	LIB$_	OTS$_	STR$_
Fatal internal error	FATERRLIB	FATINTERR	FATINTERR
Illegal string class	INVSTRDES	INVSTRDES	ILLSTRCLA
Insufficient virtual memory	INSVIRMEM	INSVIRMEM	INSVIRMEM

Some Run-Time Library routines require you to specify the length of a string or the position of a character within a string. The maximum length for a string in VMS is 65,535 characters. When you refer to character positions in a string, the first position is 1. Given a string with length *L*, containing a substring specified by character positions *M* to *N*, the following evaluation rules apply:

- If *M* is less than 1, *M* is considered to equal 1.
- If *M* is greater than *L*, the substring specified is the null string.
- If *N* is greater than *L*, *N* is considered to equal the length of the source string.
- If *M* is greater than *N*, the substring specified is the null string.

When specifying a substring of length *L*, the following applies:

- If *L* is less than zero, the substring specified is the null string. (A null string is a descriptor with zero length. A descriptor with a nonzero length and a zero pointer generates an error and yields unspecified results.)

If any of these evaluation rules applies, the range error status (qualified success) is returned. STR$POSITION represents the exception to this convention. This routine returns a function value giving the character position of a substring within a string. If the function value is zero, the substring was not found.

2.4 Allocating Resources for Dynamic Strings

This section tells how to use the Run-Time Library string resource allocation routines. These routines allocate virtual memory for a dynamic string and place the address of the allocated memory in a descriptor.

Dynamic strings may be the most convenient type to write, since you need not specify constant length, maximum length, or position for them. However, there are some restrictions on dynamic strings.

- They may cause program execution to be slower at run time.
- They require larger address space.
- They are not supported by all VAX languages.

In most cases, when you call a Run-Time Library routine to manipulate dynamic strings, the Run-Time Library routine itself allocates the required memory for the string. Your program needs to allocate only the descriptors.

For example, if you are copying a source string into a dynamic destination string, simply use one of the library's string-copying routines. Copy the input string into a dynamic string whose length and address have been initialized to zero. The string-copying routine will then itself allocate the space that the calling program needs.

However, if your program must explicitly construct or modify a dynamic string descriptor, it must use the Run-Time Library allocation and deallocation routines. This technique may be necessary, for instance, if you are constructing a string out of components that are not themselves in string form. Further, you can use one of the deallocation routines to free the dynamic string after the string resources are no longer needed, in order to optimize the program's use of resources.

The Run-Time Library provides eight entry points for string resource allocation and deallocation, all with slightly different input arguments, calling techniques, or methods of indicating errors. The following lists summarize these routines and their functions.

The following routines allocate a specified number of bytes of dynamic virtual memory to a specified string descriptor.

Routine	JSB Entry Point
LIB$SGET1_DD	LIB$SGET1_DD_R6
OTS$SGET1_DD	OTS$SGET1_DD_R6
STR$GET1_DX	STR$GET1_DX_R4

The following routines return one dynamic string area to free storage, and set DSC$A_POINTER and DSC$W_LENGTH to zero.

Routine	JSB Entry Point
LIB$SFREE1_DD	LIB$SFREE1_DD6
OTS$SFREE1_DD	OTS$SFREE1_DD6
STR$FREE1_DX	STR$FREE1_DX_R4

The following routines return one or more dynamic string areas to free storage, and set DSC$A_POINTER and DSC$W_LENGTH to zero.

Routine	JSB Entry Point
LIB$SFREEN_DD	LIB$SFREEN_DD6
OTS$SFREEN_DD	OTS$SFREEN_DD6

If you find it necessary to call the dynamic string allocation routines, there are several factors to consider.

- When your program calls a string allocation routine, it needs to allocate space only for the string descriptor before making the call. Your program does this using the statement of the particular language, either statically at compile time, or dynamically in local stack storage or heap storage.
- If your routine explicitly allocates dynamic string descriptors in stack storage, it must explicitly free the associated dynamic string areas by calling the LIB$SFREE1_DD, OTS$SFREE1_DD, or STR$FREE1_DX routine. Then your routine must free the storage for the descriptor. After both areas have been freed, your routine can return to the calling program. If the deallocation is not done, the dynamic string area becomes unavailable when the RET instruction removes the descriptors that point to the string area.
- If a routine has explicitly allocated dynamic string areas, and the routine is then unwound by the condition handling facility, the allocated address space cannot be referenced again. For this reason, your program should establish a handler that will free the associated dynamic string areas when the SS$_UNWIND condition is signaled. The handler can free these areas by calling one of the deallocation routines. This technique is especially important if a large amount of address space is involved, or if the routine allocates space within a repeating loop.

You can call the string resource allocation routines only from user mode, at AST or non-AST level. However, be extremely careful if you manipulate dynamic strings at AST level. The string manipulation routines in the Run-Time Library do not prevent the strings that they are manipulating at non-AST level from being modified at AST level.

For example, consider the case in which a string manipulation routine has calculated the lengths and addresses involved in a concatenation operation. This string manipulation routine may be interrupted by an AST. The user, at AST level, may write to the same string, changing its length and address. It is then possible to resume execution of the routine with addresses that are no longer allocated or string lengths that are no longer valid. For this reason, if you use dynamic strings at AST level, you should allocate, use, and deallocate them within the AST code.

The dynamic string manipulation routines are intended for use at user mode only. If you need to manipulate dynamic strings at another access mode, you should allocate and deallocate storage for each string at that access mode to avoid side effects. Link each segment of your program that will run at a different access mode with the /NOSYSSHR qualifier. In this way, you will establish a separate copy of the string database for each access mode.

2.4.1 String Zone

All virtual memory for dynamic strings is allocated from a Run-Time Library zone called the string zone.

The string zone has the following benefits:

- Efficient memory utilization.
- Allocation and deallocation for long strings (more than 136 bytes) is twice as fast.
- Elimination of paging contention with the default zone by isolation of the string virtual memory accesses to a separate zone. A direct side effect of this is that corruptions caused by writing into previously freed strings will no longer affect items allocated in the default zone, directly easing the debugging effort for such problems.

The following table shows attribute values for the string zone.

Attribute	Value
Algorithm	Quick fit
Number of lookaside lists	17 (short strings from 8 to 136 bytes)
Area of initial size	4 pages
Area of extension size	32 pages
Block size	8 bytes
Alignment	Quadword boundary
Smallest block size	16 bytes (includes boundary tags)
Boundary tags	Boundary tags are used for long strings
Page limit	No page limit
Fill on allocate	No fill on allocate
Fill on free	No fill on free

STR$ Reference Section

This section provides detailed descriptions of the routines provided by the VMS RTL String Manipulation (STR$) Facility.

STR$ADD Add Two Decimal Strings

The Add Two Decimal Strings routine adds two decimal strings of digits.

FORMAT

STR$ADD *asign ,aexp ,adigits ,bsign ,bexp ,bdigits ,csign ,cexp ,cdigits*

RETURNS

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

ARGUMENTS

asign

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

Sign of the first operand. The **asign** argument is the address of an unsigned longword containing this sign. Zero is considered positive; 1 is considered negative.

aexp

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

Power of 10 by which **adigits** is multiplied to get the absolute value of the first operand. The **aexp** argument is the address of a signed longword containing this exponent.

adigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

String of unsigned digits representing the absolute value of the first operand before **aexp** is applied. The **adigits** argument is the address of a descriptor pointing to this string. This string must be an unsigned decimal number.

bsign

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

Sign of the second operand. The **bsign** argument is the address of an unsigned longword containing the second operand's sign. Zero is considered positive; 1 is considered negative.

bexp

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

Power of 10 by which **bdigits** is multiplied to get the absolute value of the second operand. The **bexp** argument is the address of a signed longword containing the second operand's exponent.

bdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

String of unsigned digits representing the absolute value of the second operand before **bexp** is applied. The **bdigits** argument is the address of a descriptor pointing to this string. This string must be an unsigned decimal number.

csign

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

Sign of the result. The **csign** argument is the address of an unsigned longword containing the result's sign. Zero is considered positive; 1 is considered negative.

cexp

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

Power of 10 by which **cdigits** is multiplied to get the absolute value of the result. The **cexp** argument is the address of a signed longword containing this exponent.

cdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **write only**
mechanism: **by descriptor**

String of unsigned digits representing the absolute value of the result before **cexp** is applied. The **cdigits** argument is the address of a descriptor pointing to this string. This string is an unsigned decimal number.

DESCRIPTION

STR$ADD adds two strings of decimal numbers (**a** and **b**). Each number to be added is passed to STR$ADD in three arguments:

1. **xdigits**—the string portion of the number
2. **xexp**—the power of ten needed to obtain the absolute value of the number
3. **xsign**—the sign of the number

The value of the number **x** is derived by multiplying **xdigits** by 10^{xexp} and applying **xsign**. Therefore, if **xdigits** is equal to '2' and **xexp** is equal to 3 and **xsign** is equal to 1, then the number represented in the **x** arguments is $2 * 10^3$ plus the sign, or –2000.

The result of the addition (c) is also returned in those three parts.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters.

CONDITION VALUES SIGNALED

LIB$_INVARG	Invalid argument.
STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$ADD could not allocate heap storage for a dynamic or temporary string.
STR$_WRONUMARG	Wrong number of arguments.

EXAMPLE

```
100 !+
    ! This is a sample arithmetic program
    ! showing the use of STR$ADD to add
    ! two decimal strings.
    !-

    ASIGN% = 1%
    AEXP% = 3%
    ADIGITS$ = '1'
    BSIGN% = 0%
    BEXP% = -4%
    BDIGITS$ = '2'
    CSIGN% = 0%
    CEXP% = 0%
    CDIGITS$ = '0'
    PRINT "A = "; ASIGN%; AEXP%; ADIGITS$
    PRINT "B = "; BSIGN%; BEXP%; BDIGITS$
    CALL STR$ADD        (ASIGN%, AEXP%, ADIGITS$, &
                         BSIGN%, BEXP%, BDIGITS$,  &
                         CSIGN%, CEXP%, CDIGITS$)
    PRINT "C = "; CSIGN%; CEXP%; CDIGITS$
999 END
```

This BASIC example uses STR$ADD to add two decimal strings, where the following values apply:

A = –1000 (ASIGN = 1, AEXP = 3, ADIGITS = '1')
B = .0002 (BSIGN = 0, BEXP = –4, BDIGITS = '2')

The output generated by this program is listed below; note that the decimal value of C equals –999.9998 (CSIGN = 1, CEXP = –4, CDIGITS = '9999998').

```
A = 1  3 1
B = 0 -4 2
C = 1 -4 9999998
```

STR$ANALYZE_SDESC Analyze String Descriptor

The Analyze String Descriptor routine extracts the length and starting address of the data for a variety of string descriptor classes.

FORMAT

STR$ANALYZE_SDESC *input-descriptor ,word-integer-length ,data-address*

corresponding jsb entry point

STR$ANALYZE_SDESC_R1

RETURNS

None.

ARGUMENTS

input-descriptor

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

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

word-integer-length

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Length of the data; this length is extracted from the descriptor by STR$ANALYZE_SDESC. The **word-integer-length** argument is the address of an unsigned word integer into which STR$ANALYZE_SDESC writes the data length.

data-address

VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Address of the data; this address is extracted from the descriptor by STR$ANALYZE_SDESC. The **data-address** argument is an unsigned longword into which STR$ANALYZE_SDESC writes the address of the data.

STR$ANALYZE_SDESC

DESCRIPTION

STR$ANALYZE_SDESC takes as input a descriptor argument and extracts from the descriptor the length of the data and the address at which the data starts for a variety of string descriptor classes. See LIB$ANALYZE_SDESC for a list of classes.

STR$ANALYZE_SDESC returns the length of the data in the **word-integer-length** argument and the starting address of the data in the **data-address** argument.

STR$ANALYZE_SDESC signals an error if an invalid descriptor class is found.

CONDITION VALUES SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

STR$APPEND Append String

The Append String routine appends a source string to the end of a destination string. The destination string must be a dynamic or varying-length string.

FORMAT

STR$APPEND *destination-string ,source-string*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string to which STR$APPEND appends the source string. The **destination-string** argument is the address of a descriptor pointing to the destination string. This destination string must be dynamic or varying-length.

source-string

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

Source string that STR$APPEND appends to the end of the destination string. The **source-string** argument is the address of a descriptor pointing to this source string.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.

STR$APPEND

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$APPEND could not allocate heap storage for a dynamic or temporary string.
STR$_STRTOOLON	The combined lengths of the source and destination strings exceeded 65,535.

EXAMPLE

```
10 !+
   !  This example program uses
   !  STR$APPEND to append a source
   !  string to a destination string.
   !-

   DST$ = 'VAX/'
   SRC$ = 'VMS'
   CALL STR$APPEND (DST$, SRC$)
   PRINT "DST$ = ";DST$
   END
```

This BASIC example uses STR$APPEND to append a source string 'VMS', to a destination string 'VAX/'.

The output generated by this program is as follows:

```
DST$ = VAX/VMS
```

STR$CASE_BLIND_COMPARE Compare Strings Without Regard to Case

The Compare Strings Without Regard to Case routine compares two input strings of any supported class and data type without regard to whether the alphabetic characters are uppercase or lowercase.

FORMAT

STR$CASE_BLIND_COMPARE *first-source-string ,second-source-string*

RETURNS

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by value**

The values returned by STR$CASE_BLIND_COMPARE and the conditions to which they translate are as follows:

Returned Value	Condition
−1	**First-source-string** is less than **second-source-string**
0	Both are the same (with blank fill for shorter string)
1	**First-source-string** is greater than **second-source-string**

ARGUMENTS

first-source-string

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

First string. The **first-source-string** argument is the address of a descriptor pointing to the first string.

second-source-string

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

Second string. The **second-source-string** argument is the address of a descriptor pointing to the second string.

DESCRIPTION

STR$CASE_BLIND_COMPARE does not distinguish between uppercase and lowercase characters. The contents of both strings are converted to uppercase before the strings are compared, but the source strings themselves are not changed. STR$CASE_BLIND_COMPARE uses the DEC Multinational Character Set.

STR$CASE_BLIND_COMPARE

CONDITION VALUE SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
PROGRAM CASE_BLIND(INPUT, OUTPUT);

{+}
{  This program demonstrates the use of
{  STR$CASE_BLIND_COMPARE.
{
{  First, declare the external function.
{-}

FUNCTION STR$CASE_BLIND_COMPARE(STR1 : VARYING
         [A] OF CHAR; STR2 : VARYING [B] OF
         CHAR) : INTEGER; EXTERN;

{+}
{  Declare the variables to be used in the
{  main program.
{-}

VAR
  STRING1       : VARYING [256] OF CHAR;
  STRING2       : VARYING [256] OF CHAR;
  RET_STATUS    : INTEGER;

{+}
{  Begin the main program.  Read values for
{  the strings to be compared.  Call
{  STR$CASE_BLIND_COMPARE.  Print the
{  result.
{-}

BEGIN
  WRITELN('ENTER THE FIRST STRING: ');
  READLN(STRING1);
  WRITELN('ENTER THE SECOND STRING: ');
  READLN(STRING2);
  RET_STATUS := STR$CASE_BLIND_COMPARE(STRING1, STRING2);
  WRITELN(RET_STATUS);
END.
```

This Pascal example shows how to call STR$CASE_BLIND_COMPARE to determine whether two strings are equal regardless of case. One example of the output of this program is as follows:

```
$ RUN CASE_BLIND
ENTER THE FIRST STRING:  KITTEN
ENTER THE SECOND STRING:  kItTeN
    0
```

STR$COMPARE Compare Two Strings

The Compare Two Strings routine compares the contents of two strings.

FORMAT

STR$COMPARE *first-source-string*
,second-source-string

RETURNS

VMS usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by value**

The values returned by STR$COMPARE and the conditions to which they translate are as follows:

Returned Value	Condition
−1	**First-source-string** is less than **second-source-string**
0	**First-source-string** is equal to **second-source-string**
1	**First-source-string** is greater than **second-source-string**

ARGUMENTS

first-source-string

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

First string. The **first-source-string** argument is the address of a descriptor pointing to the first string.

second-source-string

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

Second string. The **second-source-string** argument is the address of a descriptor pointing to the second string.

DESCRIPTION

STR$COMPARE compares two strings for the same contents. If the strings are unequal in length, the shorter string is considered to be filled with blanks to the length of the longer string before the comparison is made. This routine distinguishes between uppercase and lowercase alphabetic characters.

STR$COMPARE

CONDITION VALUE SIGNALED		
	STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
100 EXTERNAL INTEGER FUNCTION STR$COMPARE
    SRC1$ = 'ABC'
    SRC2$ = 'BCD     '

    !+
    !  Note that STR$COMPARE will treat SRC1$ as if it were the same
    !  length as SRC2$ for the purpose of the comparison.  Thus, it
    !  will treat the contents of SRC1$ as 'ABC     '.  However, it
    !  will only 'treat' the contents as longer; the contents of
    !  the source string are not actually changed.
    !-

    I% = STR$COMPARE(SRC1$, SRC2$)
    IF I% = 1 THEN RESULT$ = ' IS GREATER THAN '
    IF I% = 0 THEN RESULT$ = ' IS EQUAL TO '
    IF I% = -1 THEN RESULT$ = ' IS LESS THAN '
    PRINT SRC1$; RESULT$; SRC2$
999 END
```

This BASIC program uses STR$COMPARE to compare two strings. The output generated by this program is as follows:

```
ABC IS LESS THAN BCD
```

STR$COMPARE_EQL Compare Two Strings for Equality

The Compare Two Strings for Equality routine compares two strings to see if they have the same length and contents. Uppercase and lowercase characters are not considered equal.

FORMAT

STR$COMPARE_EQL *first-source-string*
,second-source-string

RETURNS

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

The values returned by STR$COMPARE and the conditions to which they translate are as follows:

Returned Value	Condition
0	The length and the contents of **first-source-string** are equal to the length and contents of **second-source-string**.
1	Either the length of **first-source-string** is not equal to the length of **second-source-string**, or the contents of **first-source-string** are not equal to the contents of **second-source-string**, or both.

ARGUMENTS

first-source-string

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

First source string. The **first-source-string** argument is the address of a descriptor pointing to the first source string.

second-source-string

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

Second source string. The **second-source-string** argument is the address of a descriptor pointing to the second source string.

STR$COMPARE_EQL

CONDITION VALUES SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
PROGRAM COMPARE_EQL(INPUT, OUTPUT);

{+}
{  This program demonstrates the use of
{  STR$COMPARE_EQL to compare two strings.
{  Strings are considered equal only if they
{  have the same contents and the same length.
{
{  First, declare the external function.
{-}

FUNCTION STR$COMPARE_EQL(SRC1STR : VARYING
         [A] OF CHAR; SRC2STR : VARYING [B]
         OF CHAR) : INTEGER; EXTERN;

{+}
{  Declare the variables used in the main program.
{-}

VAR
  STRING1       : VARYING [256] OF CHAR;
  STRING2       : VARYING [256] OF CHAR;
  RET_STATUS    : INTEGER;

{+}
{  Begin the main program.  Read the strings
{  to be compared.  Call STR$COMARE_EQL to compare
{  the strings.  Print the result.
{-}

BEGIN
  WRITELN('ENTER THE FIRST STRING: ');
  READLN(STRING1);
  WRITELN('ENTER THE SECOND STRING: ');
  READLN(STRING2);
  RET_STATUS := STR$COMPARE_EQL(STRING1, STRING2);
  WRITELN(RET_STATUS);
END.
```

This Pascal example demonstrates the use of STR$COMPARE_EQL. A sample of the output generated by this program is as follows:

```
$ RUN COMPARE_EQL
ENTER THE FIRST STRING:  frog
ENTER THE SECOND STRING:  Frogs
     1
```

STR$COMPARE_MULTI Compare Two Strings for Equality Using Multinational Character Set

The Compare Two Strings for Equality Using Multinational Character Set routine compares two character strings for equality using the DEC Multinational Character Set.

FORMAT

STR$COMPARE_MULTI *first-source-string*
,second-source-string
[,flags-value]
[,foreign-language]

RETURNS

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by value**

The values returned by STR$COMPARE_MULTI and the conditions to which they translate are as follows:

Returned Value	Condition
-1	**First-source-string** is less than **second-source-string**.
0	Both strings are the same; the shorter string is blank filled.
1	**First-source-string** is greater than **second-source-string**.

ARGUMENTS

first-source-string

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

First string in the comparison. The **first-source-string** argument is the address of a descriptor pointing to the first string.

second-source-string

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

Second string in the comparison. The **second-source-string** argument is the address of a descriptor pointing to the second string.

flags-value

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

A single flag bit. The **flags-value** argument is a signed longword integer that contains this flag bit. The default value of **flags-value** is zero; in other words, **flags-value** is case sensitive.

Symbol	Meaning
CASEBLIND	If bit is set, uppercase and lowercase characters are equivalent.

foreign-language

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

Indicator that determines the foreign language table to be used. The **foreign-language** argument is an unsigned longword that contains this foreign language table indicator. The default value of **foreign-language** is 1.

Value	Language
1	Multinational table
2	Danish table
3	Finnish/Swedish table
4	German table
5	Norwegian table
6	Spanish table

DESCRIPTION

STR$COMPARE_MULTI compares two character strings to see if they have the same contents. Two strings are "equal" if they contain the same characters in the same sequence, even if one of them is blank filled to a longer length than the other. The DEC Multinational Character Set, or foreign language variations of the DEC Multinational Character Set, are used in the comparison.

See the *VMS I/O User's Reference Volume* for more information about the DEC Multinational Character Set.

CONDITION VALUES SIGNALED

STR$_ILLSTRCLA	Illegal string class. Severe error. The descriptor of **first-source-string** and/or **second-source-string** contains a class code that is not supported by the VAX Procedure Calling and Condition Handling Standard.
LIB$_INVARG	Invalid argument. Severe error.

STR$CONCAT Concatenate Two or More Strings

The Concatenate Two or More Strings routine concatenates all specified source strings into a single destination string.

FORMAT

STR$CONCAT *destination-string ,source-string [,source-string...]*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$CONCAT concatenates all specified source strings. The **destination-string** argument is the address of a descriptor pointing to this destination string.

source-string

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

First source string; STR$CONCAT requires at least one source string. The **source-string** argument is the address of a descriptor pointing to the first source string. The maximum number of source strings that STR$CONCAT allows is 254.

source-string

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

Additional source strings; STR$CONCAT requires at least one source string. The **source-string** argument is the address of a descriptor pointing to the additional source string. The maximum number of source strings that STR$CONCAT allows is 254.

DESCRIPTION

STR$CONCAT concatenates all specified source strings into a single destination string. The strings can be of any class and data type, provided that the length fields of the descriptors indicate the lengths of the strings in bytes. You must specify at least one source string, and you can specify up to 254 source strings. The maximum length of the concatenated string is 65,535 bytes.

A warning status is returned if one or more input characters were not copied to the destination string.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion. All characters in the input strings were copied into the destination string.
STR$_TRU	String truncation warning. One or more input characters were not copied into the destination string. This can happen when the destination is a fixed-length string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$CONCAT could not allocate heap storage for a dynamic or temporary string.
STR$_STRTOOLON	String length exceeds 65,535 bytes.
STR$_WRONUMARG	Wrong number of arguments. You tried to pass fewer than two or more than 255 arguments to STR$CONCAT.

EXAMPLES

1

```
10 !+
   !  This example program uses STR$CONCAT
   !  to concatenate four source strings into a
   !  single destination string.
   !-

   EXTERNAL INTEGER FUNCTION STR$CONCAT
   STATUS% = STR$CONCAT (X$, 'A', 'B', 'C', 'D')
   PRINT "X$ = ";X$
   END
```

The output generated by this BASIC program is as follows:

```
X$ = ABCD
```

2
```
MSG1:               .ASCII /VMS Run-/          ; first string
MSG1_LEN = .-MSG1                              ; its length
MSG2:               .ASCII /Time Library/      ; second string
MSG2_LEN = .-MSG2                              ; its length
RESULT:             .BLKB MSG1_LEN + MSG2_LEN  ; string to hold concatenation

MSG1_DSC:           .WORD MSG1_LEN             ; DSC$W_LENGTH
                    .BYTE 14                   ; DSC$B_DTYPE
                    .BYTE 1                    ; DSC$B_CLASS
                    .ADDRESS MSG1              ; DSC$A_POINTER

MSG2_DSC:           .WORD MSG2_LEN             ; DSC$W_LENGTH
                    .BYTE 14                   ; DSC$B_DTYPE
                    .BYTE 1                    ; DSC$B_CLASS
                    .ADDRESS MSG2              ; DSC$A_POINTER

RESULT_DSC:         .WORD MSG1_LEN + MSG2_LEN  ; DSC$W_LENGTH
                    .BYTE 14                   ; DSC$B_DTYPE
                    .BYTE 1                    ; DSC$B_CLASS
                    .ADDRESS RESULT            ; DSC$A_POINTER

        .ENTRY  EXAM1, ^M<>     ; entry point
        PUSHAQ  MSG2_DSC        ; push the descriptors
        PUSHAQ  MSG1_DSC        ; in reverse order
        PUSHAQ  RESULT_DSC      ; .
        CALLS   #3, G^STR$CONCAT        ; concatenate strings

        PUSHAQ  RESULT_DSC              ; descr. of string to display
        CALLS   #1, G^LIB$PUT_OUTPUT    ; display it
        RET                             ; return to calling routine
        .END    EXAM1
```

The output generated by this MACRO program is as follows:

```
VMS Run-Time Library
```

STR$COPY_DX Copy a Source String Passed by Descriptor to a Destination String

The Copy a Source String Passed by Descriptor to a Destination String routine copies a source string to a destination string. Both strings are passed by descriptor.

FORMAT

STR$COPY_DX *destination-string ,source-string*

corresponding jsb entry point

STR$COPY_DX_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$COPY_DX writes the source string; depending on the class of the destination string, the following actions occur:

Class Field	Action
DSC$K_CLASS_S,Z,SD,A,NCA	Copy the source string. If needed, space is filled or truncated on the right.
DSC$K_CLASS_D	If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation (if any) and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.
DSC$K_CLASS_VS	Copy the source string to the destination string up to the limit of DSC$W_MAXSTRLEN with no padding. Adjust current length field to actual number of bytes copied.

The **destination-string** argument is the address of a descriptor pointing to the destination string.

STR$COPY_DX

source-string

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

Source string that STR$COPY_DX copies into the destination string; the descriptor class of the source string can be unspecified, fixed length, dynamic, scalar decimal, array, noncontiguous array, or varying length. The **source-string** argument is the address of a descriptor pointing to this source string. (See the description of LIB$ANALYZE_SDESC for possible restrictions.)

DESCRIPTION

STR$COPY_DX copies a source string to a destination string, where both strings are passed by descriptor. All conditions except success and truncation are signaled; truncation is returned as a warning condition value.

STR$COPY_DX passes the source string by descriptor. In addition, an equivalent JSB entry point is provided, with R0 being the first argument (the descriptor of the destination string), and R1 the second (the descriptor of the source string).

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion. All characters in the input string were copied to the destination string.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters copied from the source string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$COPY_DX could not allocate heap storage for a dynamic or temporary string.

STR$COPY_R Copy a Source String Passed by Reference to a Destination String

The Copy a Source String Passed by Reference to a Destination String routine copies a source string passed by reference to a destination string.

FORMAT

STR$COPY_R *destination-string*
,word-integer-source-length
,source-string-address

corresponding jsb entry point

STR$COPY_R_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$COPY_R copies the source string. The **destination-string** argument is the address of a descriptor pointing to the destination string.

The class field determines the appropriate action.

word-integer-source-length

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

Length of the source string. The **word-integer-source-length** argument is the address of an unsigned word containing the length of the source string.

source-string-address

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

Source string that STR$COPY_R copies into the destination string. The **source-string-address** argument is the address of the source string.

See the description of LIB$ANALYZE_SDESC for possible restrictions.

DESCRIPTION

STR$COPY_R copies a source string passed by reference to a destination string. All conditions except success and truncation are signaled; truncation is returned as a warning condition value.

A JSB entry point is provided, with R0 being the first argument, R1 the second, and R2 the third. The length argument is passed in bits 15:0 of R1.

Depending on the class of the destination string, the following actions occur:

Class Field	Action
DSC$K_CLASS_S,Z,SD,A,NCA	Copy the source string. If needed, space is filled or truncated on the right.
DSC$K_CLASS_D	If the area specified by the destination descriptor is large enough to contain the source string, copy the source string and set the new length in the destination descriptor. If the area specified is not large enough, return the previous space allocation (if any) and then dynamically allocate the amount of space needed. Copy the source string and set the new length and address in the destination descriptor.
DSC$K_CLASS_VS	Copy the source string to the destination string up to the limit of DSC$W_MAXSTRLEN with no padding. Adjust current length field to actual number of bytes copied.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion. All characters in the input string were copied to the destination string.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters copied from the source string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$COPY_R could not allocate heap storage for a dynamic or temporary string.

STR$DIVIDE Divide Two Decimal Strings

The Divide Two Decimal Strings routine divides two decimal strings.

FORMAT

STR$DIVIDE *asign ,aexp ,adigits ,bsign ,bexp ,bdigits ,total-digits ,round-truncate-indicator ,csign ,cexp ,cdigits*

RETURNS

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

ARGUMENTS

asign

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

Sign of the first operand. The **asign** argument is the address of an unsigned longword containing the first operand's sign. Zero is considered positive; 1 is considered negative.

aexp

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

Power of 10 by which **adigits** is multiplied to get the absolute value of the first operand. The **aexp** argument is the address of the first operand's exponent.

adigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

First operand's numeric string. The **adigits** argument is the address of a descriptor pointing to the first operand's numeric string. The string must be an unsigned decimal number.

bsign

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

Sign of the second operand. The **bsign** argument is the address of an unsigned longword containing the second operand's string. Zero is considered positive; 1 is considered negative.

bexp

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

Power of 10 by which **bdigits** is multiplied to get the absolute value of the second operand. The **bexp** argument is the address of the second operand's exponent.

bdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

Second operand's numeric string. The **bdigits** argument is the address of a descriptor pointing to the second operand's number string. The string must be an unsigned decimal number.

total-digits

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

Number of digits to the right of the decimal point. The **total-digits** argument is the address of a signed longword containing the number of total digits. STR$DIVIDE uses this number to carry out the division.

round-truncate-indicator

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

Indicator of whether STR$DIVIDE is to round or truncate the result; zero means truncate, 1 means round. The **round-truncate-indicator** argument is the address of a longword bit mask containing this indicator.

csign

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

Sign of the result. The **csign** argument is the address of an unsigned longword containing the sign of the result. Zero is considered positive; 1 is considered negative.

cexp

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

Power of 10 by which **cdigits** is multiplied to get the absolute value of the result. The **cexp** argument is the address of a signed longword containing the exponent.

cdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **write only**
mechanism: **by descriptor**

Result's numeric string. The **cdigits** argument is the address of a descriptor pointing to the numeric string of the result. This string is an unsigned decimal number.

DESCRIPTION

STR$DIVIDE divides two decimal strings. The divisor and dividend are passed to STR$DIVIDE in three parts: (1) the sign of the decimal number, (2) the power of 10 needed to obtain the absolute value, and (3) the numeric string. The result of the division is also returned in those three parts.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

CONDITION VALUES SIGNALED

LIB$_INVARG	Invalid argument.
STR$_DIVBY_ZER	Division by zero.
STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$DIVIDE could not allocate heap storage for a dynamic or temporary string.
STR$_WRONUMARG	Wrong number of arguments.

EXAMPLE

```
100 !+
    !  This BASIC example program uses STR$DIVIDE
    !  to divide two decimal strings and truncates
    !  the result.
    !-

    ASIGN% = 1%
    AEXP% = 3%
    ADIGITS$ = '1'
    BSIGN% = 0%
    BEXP% = -4%
    BDIGITS$ = '2'
    CSIGN% = 0%
    CEXP% = 0%
    CDIGITS$ = '0'
    PRINT "A = "; ASIGN%; AEXP%; ADIGITS$
    PRINT "B = "; BSIGN%; BEXP%; BDIGITS$
    CALL STR$DIVIDE     (ASIGN%, AEXP%, ADIGITS$, &
                         BSIGN%, BEXP%, BDIGITS$,  &
                         3%, 0%, CSIGN%, CEXP%, CDIGITS$)
     PRINT "C = "; CSIGN%; CEXP%; CDIGITS$
1500 END
```

This BASIC program uses STR$DIVIDE to divide two decimal strings, A divided by B, where the following values apply:

A = –1000 (ASIGN = 1, AEXP = 3, ADIGITS = '1')
B = .0002 (BSIGN = 0, BEXP = –4, BDIGITS = '2')

The output generated by this program is as follows:

```
A = 1  3 1
B = 0 -4 2
C = 1 -3 5000000000
```

Thus, the decimal value of C equals –5000000 (CSIGN = 1, CEXP = –3, CDIGITS = 5000000000).

STR$DUPL_CHAR Duplicate Character *n* Times

The Duplicate Character *n* Times routine generates a string containing *n* duplicates of the input character. If the destination string is an "empty" dynamic string descriptor, STR$DUPL_CHAR allocates and initializes the string.

FORMAT

STR$DUPL_CHAR *destination-string [,repetition-count] [,ASCII-character]*

corresponding jsb entry point

STR$DUPL_CHAR_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$DUPL_CHAR writes **repetition-count** copies of the input character. The **destination-string** argument is the address of a descriptor pointing to the destination string.

repetition-count

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

Number of times **ASCII-character** is duplicated; this is an optional argument (if omitted, the default is 1). The **repetition-count** argument is the address of a signed longword containing the number.

ASCII-character

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

ASCII character that STR$DUPL_CHAR writes **repetition-count** times into the destination string. The **ASCII-character** argument is the address of a character string containing this character. This is an optional argument; if omitted, the default is a space.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_NEGSTRLEN	Alternate success. The length argument contained a negative value; zero was used.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$DUPL_CHAR could not allocate heap storage for a dynamic or temporary string.
STR$_STRTOOLON	String length exceeds 65,535 bytes.

EXAMPLE

```
10  !+
    !  This example uses STR$DUPL_CHAR to
    !  duplicate the character 'A' four times.
    !-

    EXTERNAL INTEGER FUNCTION STR$DUPL_CHAR
    STATUS% = STR$DUPL_CHAR (X$, 4%, 'A' BY REF)
    PRINT X$
    END
```

These BASIC statements set X$ equal to 'AAAA'.

The output generated by this program is as follows:

```
AAAA
```

STR$ELEMENT Extract Delimited Element Substring

The Extract Delimited Element Substring routine extracts an element from a string in which the elements are separated by a specified delimiter.

FORMAT

STR$ELEMENT *destination-string ,element-number ,delimiter-string ,source-string*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$ELEMENT copies the selected substring. The **destination-string** argument is the address of a descriptor pointing to the destination string.

element-number

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

Element number of the delimited element substring to be returned. The **element-number** argument is the address of a signed longword containing the desired element number. Zero is used to represent the first delimited element substring, one is used to represent the second, and so forth.

delimiter-string

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

Delimiter string used to separate element substrings. The **delimiter-string** argument is the address of a descriptor pointing to the delimiter string. **Delimiter-string** must be exactly one character long.

source-string

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

Source string from which STR$ELEMENT extracts the requested delimited substring. The **source-string** argument is the address of a descriptor pointing to the source string.

DESCRIPTION

STR$ELEMENT extracts an element from a string in which the elements are separated by a specified delimiter.

For example, if **source-string** is MON^TUE^WED^THU^FRI^SAT^SUN, **delimiter-string** is ^, and **element-number** is 2, then STR$ELEMENT returns the string WED.

Once the specified element is located, all the characters in that delimited element are returned. That is, all characters between the **element-number** and the **element-number** + 1 delimiters are written to **destination-string**. At least **element-number** delimiters must be found. If exactly **element-number** delimiters are found, then all values from the **element-number** delimiter to the end of the string are returned. If **element-number** equals 0 and no delimiters are found, the entire input string is returned.

STR$ELEMENT duplicates the functions of the DCL lexical function F$ELEMENT.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_INVDELIM	Delimiter string is not exactly one character long (warning).
STR$_NOELEM	Not enough delimited characters found to satisfy requested element number (warning).
STR$_TRU	String truncation. The fixed-length destination string could not contain all the characters in the delimited substring (warning).

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$ELEMENT could not allocate heap storage for a dynamic or temporary string.

STR$FIND_FIRST_IN_SET Find First Character in a Set of Characters

The Find First Character in a Set of Characters routine searches a string one character at a time, from left to right, comparing each character in the string to every character in a specified set of characters for which it is searching.

FORMAT

STR$FIND_FIRST_IN_SET *source-string*
,set-of-characters

RETURNS

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by value**

Position in **source-string** where the first match is found; zero if no match is found.

ARGUMENTS

source-string

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

String that STR$FIND_FIRST_IN_SET compares to the set of characters, looking for the first match. The **source-string** argument is the address of a descriptor pointing to the character string.

set-of-characters

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

Set of characters that STR$FIND_FIRST_IN_SET is searching for in the string. The **source-string** argument is the address of a descriptor pointing to the set of characters.

DESCRIPTION

STR$FIND_FIRST_IN_SET compares each character in the string to every character in the specified set of characters. As soon as the first match is found, STR$FIND_FIRST_IN_SET returns the position in the string where the matching character was found. If no match is found, 0 is returned. If either **source-string** or **set-of-characters** is of zero length, 0 is returned.

CONDITION VALUE SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
PROGRAM FIND_FIRST(INPUT, OUTPUT);

{+}
{  This example uses STR$FIND_FIRST_IN_SET
{  to find the first character in the source
{  string (STRING1) that matches a character
{  in the set of characters being searched for
{  (CHARS).
{
{  First, declare the external function.
{-}

FUNCTION STR$FIND_FIRST_IN_SET(STRING :
          VARYING [A] OF CHAR; SETOFCHARS :
          VARYING [B] OF CHAR) : INTEGER;
          EXTERN;

{+}
{  Declare the variables used in the main program.
{-}

VAR
  STRING1        : VARYING [256] OF CHAR;
  CHARS          : VARYING [256] OF CHAR;
  RET_STATUS     : INTEGER;

{+}
{  Begin the main program.  Read the source string
{  and the set of characters being searched for.  Call
{  STR$FIND_FIRST_IN_SET to find the first match.
{  Print the result.
{-}

BEGIN
  WRITELN('ENTER THE STRING: ');
  READLN(STRING1);
  WRITELN('ENTER THE SET OF CHARACTERS: ');
  READLN(CHARS);
  RET_STATUS := STR$FIND_FIRST_IN_SET(STRING1, CHARS);
  WRITELN(RET_STATUS);
END.
```

This Pascal program demonstrates the use of STR$FIND_FIRST_IN_SET. If you run this program and set STRING1 equal to ABCDEFGHIJK and CHARS equal to XYZA, the value of RET_STATUS will be 1. The output generated by this program is as follows:

```
ENTER THE STRING:
ABCDEFGHIJK
ENTER THE SET OF CHARACTERS:
XYZA
       1
```

STR$FIND_FIRST_NOT_IN_SET Find First Character That Does Not Occur in Set

The Find First Character That Does Not Occur in Set routine searches a string, comparing each character to the characters in a specified set of characters. The string is searched character by character, from left to right. STR$FIND_FIRST_NOT_IN_SET returns the position of the first character in the string that does not match any of the characters in the selected set of characters.

FORMAT

STR$FIND_FIRST_NOT_IN_SET *source-string ,set-of-characters*

RETURNS

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by value**

Position in **source-string** where a nonmatch was found.

Returned value	Condition
0	Either all characters in **source-string** match some character in **set-of-characters**, or there were no characters in **set-of-characters**.
1	Either the first nonmatching character in **source-string** was found in position 1, or there were no characters in **source-string**.
N	The first nonmatching character was found in position N within **source-string**.

ARGUMENTS

source-string

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

String that STR$FIND_FIRST_NOT_IN_SET searches. The **source-string** argument is the address of a descriptor pointing to the string.

set-of-characters

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

The set of characters that STR$FIND_FIRST_NOT_IN_SET compares to the string, looking for a nonmatch. The **set-of-characters** argument is the address of a descriptor pointing to this set of characters.

DESCRIPTION

STR$FIND_FIRST_NOT_IN_SET searches a string, comparing each character to the characters in a specified set of characters. The string is searched character by character, from left to right. When STR$FIND_FIRST_NOT_IN_SET finds a character from the string that is not in **set-of-characters**, it stops searching and returns, as the value of STR$FIND_FIRST_NOT_IN_SET, the position in **source-string** where it found the nonmatching character. If all characters in the string match some character in the set of characters, STR$FIND_FIRST_NOT_IN_SET returns 0. If the string is of zero length, the position returned is 1 since none of the elements in the set of characters (particularly the first element) will be found in the string. If there are no characters in the set of characters, zero is returned since "nothing" can always be found.

CONDITION VALUE SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
PROGRAM NOT_IN_SET(INPUT, OUTPUT);

{+}
{  This example uses STR$FIND_FIRST_NOT_IN_SET
{  to find the position of the first nonmatching
{  character from a set of characters (CHARS)
{  in a source string (STRING1).
{
{  First, declare the external function.
{-}

FUNCTION STR$FIND_FIRST_NOT_IN_SET(STRING :
          VARYING [A] OF CHAR; SETOFCHARS :
          VARYING [B] OF CHAR) : INTEGER;
          EXTERN;
{+}
{  Declare the variables used in the main program.
{-}

VAR
  STRING1       : VARYING [256] OF CHAR;
  CHARS         : VARYING [256] OF CHAR;
  RET_STATUS    : INTEGER;

{+}
{  Begin the main program.  Read the source string
{  and set of characters.  Call STR$FIND_FIRST_NOT_IN_SET.
{  Print the result.
{-}

BEGIN
  WRITELN('ENTER THE STRING: ');
  READLN(STRING1);
  WRITELN('ENTER THE SET OF CHARACTERS: ');
  READLN(CHARS);
  RET_STATUS := STR$FIND_FIRST_NOT_IN_SET(STRING1, CHARS);
  WRITELN(RET_STATUS);
END.
```

STR$FIND_FIRST_NOT_IN_SET

This Pascal program demonstrates the use of STR$FIND_FIRST_NOT_IN_SET. If you run this program and set STRING1 equal to FORTUNATE and CHARS equal to FORT, the value of RET_STATUS will be 5.

The output generated by this program is as follows:

```
ENTER THE STRING:
FORTUNATE
ENTER THE SET OF CHARACTERS:
FORT
       5
```

STR$FIND_FIRST_SUBSTRING Find First Substring in Input String

The Find First Substring in Input String routine finds the first substring (in a provided list of substrings) occurring in a given string.

FORMAT

STR$FIND_FIRST_SUBSTRING *source-string ,index ,substring-index ,substring [,substring...]*

RETURNS

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

The values returned by STR$FIND_FIRST_SUBSTRING and the conditions to which they translate are as follows:

Returned Value	Condition
0	**Source-string** did not contain any of the specified substrings.
1	STR$FIND_FIRST_SUBSTRING found at least one of the specified substrings in the string.

ARGUMENTS

source-string

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

String that STR$FIND_FIRST_SUBSTRING searches. The **source-string** argument is the address of a descriptor pointing to the string.

index

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

Earliest position within **source-string** at which STR$FIND_FIRST_SUBSTRING found a matching substring; zero if no matching substring was found. The **index** argument is the address of a signed longword containing this position.

substring-index

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

Ordinal number of the **substring** that matched (1 for the first, 2 for the second, and so on), or zero if STR$FIND_FIRST_SUBSTRING found no substrings that matched. The **substring-index** argument is the address of a signed longword containing this ordinal number.

substring

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

Specified substring for which STR$FIND_FIRST_SUBSTRING searches in **source-string**. The **substring** argument is the address of a descriptor pointing to the first substring. You can specify multiple substrings to be searched for.

substring

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

Additional specified substrings for which STR$FIND_FIRST_SUBSTRING searches in **source-string**. The **substring** argument is the address of a descriptor pointing to the substring. You can specify multiple substrings to be searched for.

DESCRIPTION

STR$FIND_FIRST_SUBSTRING takes as input a string to be searched and an unspecified number of substrings for which to search. It searches the specified string and returns the position of the substring that is found earliest in the string. This is not necessarily the position of the first substring specified. That is, STR$FIND_FIRST_SUBSTRING returns the position of the leftmost matching substring. The order in which the substrings are searched for is irrelevant.

Unlike many of the compare and search routines, STR$FIND_FIRST_SUBSTRING does not return the position in a return value. The position of the substring which is found earliest in the string is returned in the **index** argument. If none of the specified substrings is found in the string, the value of **index** is zero.

Zero-length strings, or null arguments, produce unexpected results. Any time the routine is called with a null substring as an argument, STR$FIND_FIRST_SUBSTRING always returns the position of the null substring as the first substring found. All other substrings are interpreted as appearing in the string after the null string.

CONDITION VALUES SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_WRONUMARG	Wrong number of arguments. You must supply at least one substring.

EXAMPLE

```
1   !+
    ! This is a BASIC program demonstrating the use of
    ! STR$FIND_FIRST_SUBSTRING.  This program takes as input
    ! four strings that are listed in a data statement
    ! at the end of the program.  STR$FIND_FIRST_SUBSTRING
    ! is called four times (once for each string)
    ! to find the first substring occurring in the given
    ! string.
    !-

    OPTION TYPE = EXPLICIT

    DECLARE STRING      MATCH_STRING
    DECLARE LONG        RET_STATUS, &
                        INDEX, &
                        I, &
                        SUB_STRING_NUM
     EXTERNAL LONG FUNCTION STR$FIND_FIRST_SUBSTRING

     FOR I = 1 TO 4
        READ MATCH_STRING
        RET_STATUS = STR$FIND_FIRST_SUBSTRING( MATCH_STRING, &
            INDEX, SUB_STRING_NUM, 'ING', 'CK', 'TH')
        IF RET_STATUS = 0% THEN
            PRINT MATCH_STRING;" did not contain any of the substrings"
        ELSE
            SELECT SUB_STRING_NUM
                CASE 1
                    PRINT MATCH_STRING;" contains ING at position";INDEX
                CASE 2
                    PRINT MATCH_STRING;" contains CK at position";INDEX
                CASE 3
                    PRINT MATCH_STRING;" contains TH at position";INDEX
            END SELECT
        END IF
     NEXT I

2    DATA CHUCKLE, RAINING, FOURTH, THICK

3    END
```

This BASIC program demonstrates the use of STR$FIND_FIRST_SUBSTRING. The output generated by this program is as follows:

```
$ BASIC FINDSUB
$ LINK FINDSUB
$ RUN FINDSUB
CHUCKLE contains CK at position 4
RAINING contains ING at position 5
FOURTH contains TH at position 5
THICK contains TH at position 1
```

Note that "THICK" contains both the substrings "TH" and "CK". STR$FIND_FIRST_SUBSTRING locates the "CK" substring in "THICK", and then locates the "TH" substring. However, since the "TH" substring is the earliest, or leftmost matching substring, its ordinal number is returned in **substring-index**, and the point at which "TH" occurs is returned in **index**.

STR$FREE1_DX Free One Dynamic String

The Free One Dynamic String routine deallocates one dynamic string.

FORMAT

STR$FREE1_DX *string-descriptor*

corresponding jsb entry point

STR$FREE1_DX_R4

RETURNS

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

ARGUMENT

string-descriptor

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

Dynamic string descriptor of the dynamic string that STR$FREE1_DX deallocates. The **string-descriptor** argument is the address of a descriptor pointing to the string to be deallocated. The class field (DSC$B_CLASS) is checked.

DESCRIPTION

STR$FREE1_DX deallocates the described string space and flags the descriptor as describing no string at all (DSC$A_POINTER = 0, DSC$W_LENGTH = 0).

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

STR$GET1_DX Allocate One Dynamic String

The Allocate One Dynamic String routine allocates a specified number of bytes of dynamic virtual memory to a specified dynamic string descriptor.

FORMAT

STR$GET1_DX *word-integer-length ,character-string*

corresponding jsb entry point

STR$GET1_DX_R4

RETURNS

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

ARGUMENTS

word-integer-length

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

Number of bytes that STR$GET1_DX allocates. The **word-integer-length** argument is the address of an unsigned word containing this number.

character-string

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

Dynamic string descriptor to which STR$GET1_DX allocates the area. The **character-string** argument is the address of an unsigned quadword containing the string descriptor.

The class field (DSC$B_CLASS) is checked.

DESCRIPTION

STR$GET1_DX allocates a specified number of bytes of dynamic virtual memory to a specified string descriptor. The descriptor must be dynamic.

If the string descriptor already has dynamic memory allocated to it, but the amount allocated is less than **word-integer-length**, STR$GET1_DX deallocates that space before it allocates new space.

STR$GET1_DX is the only recommended method for allocating a dynamic descriptor. Simply filling in the length and pointer fields of a dynamic string descriptor can cause serious and unexpected problems with string management.

To deallocate dynamic strings, call STR$FREE1_DX.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$GET1_DX could not allocate heap storage for a dynamic or temporary string.

STR$LEFT Extract a Substring of a String

The Extract a Substring of a String routine copies a substring of a source string into a destination string. The relative starting position in the source string is 1.

FORMAT

STR$LEFT *destination-string ,source-string ,end-position*

corresponding jsb entry point

STR$LEFT_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$LEFT copies the substring. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string from which STR$LEFT extracts the substring that it copies into the destination string. The **source-string** argument is the address of a descriptor pointing to the source string.

end-position

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

Relative position in the source string at which the substring ends. The **end-position** argument is the address of a signed longword containing the ending position.

STR$LEFT copies all characters in the source string from position 1 (the leftmost position) to the position number specified in this **end-position** argument.

DESCRIPTION

STR$LEFT extracts a substring from a source string and copies that substring into a destination string. STR$LEFT defines the substring by specifying the relative ending position in the source string. The relative starting position in the source string is 1. The source string is unchanged, unless it is also the destination string.

This is a variation of STR$POS_EXTR. Other routines that may be used to extract and copy a substring are STR$RIGHT and STR$LEN_EXTR.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_ILLSTRPOS	Alternate success. An argument referenced a character position outside the specified string. A default value was used.
STR$_ILLSTRSPE	Alternate success. The length of the substring was too long for the specified destination string. Default values were used.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters copied from the source string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$LEFT could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
PROGRAM LEFT(INPUT, OUTPUT);

{+}
{ This Pascal program demonstrates the use of
{ STR$LEFT.  This program reads in a source string
{ and the ending position of a substring.
{ It returns a substring consisting of all
{ characters from the beginning (left) of the
{ source string to the ending position entered.
{-}

{+}
{ Declare the external procedure, STR$LEFT.
{-}
```

STR$LEFT

```
PROCEDURE STR$LEFT(%DESCR DSTSTR: VARYING
          [A] OF CHAR; SRCSTR  :
          VARYING [B] OF CHAR; ENDPOS :
          INTEGER); EXTERN;

{+}
{ Declare the variables used by this program.
{-}

VAR
  SRC_STR : VARYING [256] OF CHAR;
  DST_STR : VARYING [256] OF CHAR;
  END_POS : INTEGER;

{+}
{ Begin the main program.  Read the source string
{ and ending position.  Call STR$LEFT.  Print the
{ results.
{-}

BEGIN
  WRITELN('ENTER THE SOURCE STRING: ');
  READLN(SRC_STR);
  WRITELN('ENTER THE ENDING POSITION');
  WRITELN('OF THE SUBSTRING: ');
  READLN(END_POS);
  STR$LEFT(DST_STR, SRC_STR, END_POS);
  WRITELN;
  WRITELN('THE SUBSTRING IS: ',DST_STR);
END.
```

This Pascal example shows the use of STR$LEFT. The following is one sample of the output of this program:

```
$ PASCAL LEFT
$ LINK LEFT
$ RUN LEFT
ENTER THE SOURCE STRING:  MAGIC CARPET
ENTER THE ENDING POSITION OF
THE SUBSTRING:  9

THE SUBSTRING IS:  MAGIC CAR
```

STR$LEN_EXTR Extract a Substring of a String

The Extract a Substring of a String routine copies a substring of a source string into a destination string.

FORMAT

STR$LEN_EXTR *destination-string ,source-string ,start-position ,longword-integer-length*

corresponding jsb entry point

STR$LEN_EXTR_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$LEN_EXTR copies the substring. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string from which STR$LEN_EXTR extracts the substring that it copies into the destination string. The **source-string** argument is the address of a descriptor pointing to the source string.

start-position

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

Relative position in the source string at which STR$LEN_EXTR begins copying the substring. The **start-position** argument is the address of a signed longword containing the starting position.

longword-integer-length

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

Number of characters in the substring that STR$LEN_EXTR copies to the destination string. The **longword-integer-length** argument is the address of a signed longword containing the length of the substring.

DESCRIPTION

STR$LEN_EXTR extracts a substring from a source string and copies that substring into a destination string.

STR$LEN_EXTR defines the substring by specifying the relative starting position in the source string and the number of characters to be copied. The source string is unchanged, unless it is also the destination string.

If the starting position is less than 1, 1 is used. If the starting position is greater than the length of the source string, the null string is returned. If the length is less than 1, the null string is also returned.

Other routines that may be used to extract and copy a substring are STR$RIGHT, STR$LEFT and STR$POS_EXTR.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_ILLSTRPOS	STR$LEN_EXTR completed successfully, except that an argument referenced a character position outside the specified string. A default value was used.
STR$_ILLSTRSPE	STR$LEN_EXTR completed successfully, except that the length was too long for the specified string. Default values were used.
STR$_NEGSTRLEN	STR$LEN_EXTR completed successfully, except that **longword-integer-length** contained a negative value. Zero was used.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters copied from the source string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$LEN_EXTR could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
        CHARACTER*131    IN_STRING
        CHARACTER*1      FRONT_CHAR
        CHARACTER*1      TAIL_CHAR
        INTEGER STR$LEN_EXTR, STR$REPLACE, STR$TRIM
        INTEGER FRONT_POSITION, TAIL_POSITION
10      WRITE (6, 800)
800     FORMAT (' Enter a string, 131 characters or less:',$)
        READ (5, 900, END=200) IN_STRING
900     FORMAT (A)
        ISTATUS = STR$TRIM (IN_STRING, IN_STRING, LENGTH)

        DO 100 I = 1, LENGTH/2
        FRONT_POSITION = I
        TAIL_POSITION = LENGTH + 1 - I
        ISTATUS = STR$LEN_EXTR ( FRONT_CHAR, IN_STRING, FRONT_POSITION,
    A                            %REF(1))

        ISTATUS = STR$LEN_EXTR ( TAIL_CHAR, IN_STRING, TAIL_POSITION,
    A                            %REF(1))

        ISTATUS = STR$REPLACE ( IN_STRING, IN_STRING, FRONT_POSITION,
    A                           FRONT_POSITION, TAIL_CHAR)

        ISTATUS = STR$REPLACE ( IN_STRING, IN_STRING, TAIL_POSITION,
    A                           TAIL_POSITION, FRONT_CHAR)
100     CONTINUE
        WRITE (6, 901) IN_STRING
901     FORMAT (' Reversed string is : ',/,1X,A)
        GOTO 10
200     CONTINUE
        END
```

This FORTRAN program accepts a string as input and writes the string in reverse order as output. This program continues to prompt for input until CTRL/Z is pressed. One sample of the output generated by this program is as follows:

```
$ FORTRAN REVERSE
$ LINK REVERSE
$ RUN REVERSE
Enter a string, 131 characters or less:  Elephants often have
flat feet
 Reversed string is :
 .teef talf evah netfo stnahpelE
Enter a string, 131 characters or less: CTRL/Z
$
```

STR$MATCH_WILD Match Wildcard Specification

The Match Wildcard Specification routine is used to compare a pattern string that includes wildcard characters with a candidate string. It returns a condition value of STR$_MATCH if the strings match and STR$_NOMATCH if they do not match.

FORMAT

STR$MATCH_WILD *candidate-string ,pattern-string*

RETURNS

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

ARGUMENTS

candidate-string

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

String that is compared to the pattern string. The **candidate-string** argument is the address of a descriptor pointing to the candidate string.

pattern-string

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

String containing wildcard characters. The **pattern-string** argument is the address of a descriptor pointing to the pattern string. The wildcards in the pattern string are translated when STR$MATCH_WILD searches the candidate string to determine if it matches the pattern string.

DESCRIPTION

STR$MATCH_WILD translates wildcard characters and searches the candidate string to determine if it matches the pattern string. The pattern string may contain either one or both of the two wildcard characters, asterisk (*) and percent (%). The asterisk character is mapped to zero or more characters. The percent character is mapped to only one character.

The two wildcard characters that may be used in the pattern string may be used only as wildcards. If the candidate string contains an asterisk or percent character, the condition STR$_NOMATCH is returned, because the wildcard characters are never translated literally.

STR$MATCH_WILD

CONDITION VALUES RETURNED

STR$_MATCH	The candidate string and the pattern string match.
STR$_NOMATCH	The candidate string and the pattern string do not match.

CONDITION VALUE SIGNALED

STR$_ILLSTRCLA	Illegal string class. Severe error. The descriptor of **candidate-string** and/or **pattern-string** contains a class code that is not supported by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
/*
 *  Example program using STR$MATCH_WILD.
 *
 *  The following program reads in a master pattern string and then
 *  compares that to input strings until it reaches the end of the
 *  input file.  For each string comparison done, it prints
 *  either 'Matches pattern string' or 'Doesn't match pattern string'.
 */

declare str$match_wild
    external entry (character(*) varying, character(*) varying)
    returns (bit(1));

example: routine options(main);

    dcl pattern_string character(80) varying;
    dcl test_string character(80) varying;

    on endfile(sysin) stop;

    put skip;

    get list(pattern_string) options(prompt('Pattern string> '));

    do while( '1'b );
        get skip list(test_string) options(prompt('Test string> '));
        if str$match_wild(test_string,pattern_string)
            then put skip list('Matches pattern string');
            else put skip list('Doesn''t match pattern string');
        end;

    end;
```

This PL/I program demonstrates the use of STR$MATCH_WILD. The output generated by this program is as follows:

```
$ PLI MATCH
$ LINK MATCH
$ RUN MATCH
Pattern string>  'Must match me exactly.'
Test string>  'Will this work? Must match me exactly.'
Doesn't match pattern string
Test string> 'must match me exactly'
Doesn't match pattern string
Test string>  'must match me exactly.'
Doesn't match pattern string
Test string>  'Must match me exactly'
Doesn't match pattern string
Test String>  'Must match me exactly.'
Matches pattern string
```

STR$MUL Multiply Two Decimal Strings

The Multiply Two Decimal Strings routine multiplies two decimal strings.

FORMAT

STR$MUL *asign ,aexp ,adigits ,bsign ,bexp ,bdigits ,csign ,cexp ,cdigits*

RETURNS

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

ARGUMENTS

asign

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

Sign of the first operand. The **asign** argument is the address of an unsigned longword containing the first operand's sign. Zero is considered positive; 1 is considered negative.

aexp

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

Power of 10 by which **adigits** is multiplied to get the absolute value of the first operand. The **aexp** argument is the address of a signed longword containing this exponent.

adigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

First operand's numeric string. The **adigits** argument is the address of a descriptor pointing to the numeric string of the first operand. The string must be an unsigned decimal number.

bsign

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

Sign of the second operand. The **bsign** argument is the address of an unsigned longword containing the sign of the second operand. Zero is considered positive; 1 is considered negative.

bexp

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

Power of 10 by which **bdigits** is multiplied to get the absolute value of the second operand. The **bexp** argument is the address of a signed longword containing this exponent.

bdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

Second operand's numeric string. The **bdigits** argument is the address of a descriptor pointing to the second operand's numeric string. The string must be an unsigned decimal number.

csign

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

Sign of the result. The **csign** argument is the address of an unsigned longword containing the sign of the result. Zero is considered positive; 1 is considered negative.

cexp

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

Power of 10 by which **cdigits** is multiplied to get the absolute value of the result. The **cexp** argument is the address of a signed longword containing this exponent.

cdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **write only**
mechanism: **by descriptor**

Result's numeric string. The **cdigits** argument is the address of a descriptor pointing to the numeric string of the result. The string is an unsigned decimal number.

DESCRIPTION

STR$MUL multiplies two decimal strings. The numbers to be multiplied are passed to STR$MUL in three parts: (1) the sign of the decimal number, (2) the power of 10 needed to obtain the absolute value, and (3) the numeric string. The result of the multiplication is also returned in those three parts.

STR$MUL

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters.

CONDITION VALUES SIGNALED

LIB$_INVARG	Invalid argument.
STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$MUL could not allocate heap storage for a dynamic or temporary string.
STR$_WRONUMARG	Wrong number of arguments.

EXAMPLE

```
100 !+
    !  This example program uses
    !  STR$MUL to multiply two decimal
    !  strings (A and B) and place the
    !  results in a third decimal string,
    !  (C)
    !-

    ASIGN% = 1%
    AEXP% = 3%
    ADIGITS$ = '1'
    BSIGN% = 0%
    BEXP% = -4%
    BDIGITS$ = '2'
    CSIGN% = 0%
    CEXP% = 0%
    CDIGITS$ = '0'
    PRINT "A = "; ASIGN%; AEXP%; ADIGITS$
    PRINT "B = "; BSIGN%; BEXP%; BDIGITS$
    CALL STR$MUL        (ASIGN%, AEXP%, ADIGITS$, &
                         BSIGN%, BEXP%, BDIGITS$,  &
                         CSIGN%, CEXP%, CDIGITS$)
    PRINT "C = "; CSIGN%; CEXP%; CDIGITS$
999 END
```

This BASIC example uses STR$MUL to multiply two decimal strings, where the following values apply:

A = –1000 (ASIGN = 1, AEXP = 3, ADIGITS = '1')
B = .0002 (BSIGN = 0, BEXP = –4, BDIGITS = '2')

Listed below is the output generated by this program; note that the decimal value C equals –.2 (CSIGN = 1, CEXP = –1, CDIGITS = 2).

```
A = 1  3 1
B = 0 -4 2
C = 1 -1 2
```

STR$POSITION Return Relative Position of Substring

The Return Relative Position of Substring routine searches for the first occurrence of a single substring within a source string. If STR$POSITION finds the substring, it returns the relative position of that substring. If the substring is not found, STR$POSITION returns a zero.

FORMAT

STR$POSITION *source-string ,substring [,start-position]*

corresponding jsb entry point

STR$POSITION_R6

RETURNS

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

Relative position of the first character of the substring. Zero is the value returned if STR$POSITION did not find the substring.

ARGUMENTS

source-string

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

Source string within which STR$POSITION searches for the substring. The **source-string** argument is the address of a descriptor pointing to the source string.

substring

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

Substring for which STR$POSITION searches. The **substring** argument is the address of a descriptor pointing to the substring.

start-position

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

Relative position in the source string at which STR$POSITION begins the search. The **start-position** argument is the address of a signed longword

containing the starting position. Although this is an optional argument, it is required if you are using the JSB entry point.

If **start-position** is not supplied, STR$POSITION starts the search at the first character position of **source-string**.

DESCRIPTION

STR$POSITION returns the relative position of the first occurrence of a substring in the source string. The value returned is an unsigned longword. The relative character positions are numbered 1, 2, 3, and so on. Zero indicates that the substring was not found.

If the substring has a zero length, the minimum value of **start-position** (and the length of **source-string** plus one) is returned by STR$POSITION.

If the source string has a zero length and the substring has a nonzero length, zero is returned, indicating that the substring was not found.

CONDITION VALUES SIGNALED

STR$_ILLSTRCLA	Illegal string class. The class code found in the string class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.

EXAMPLE

```
PROGRAM POSITION(INPUT,OUTPUT);

{+}
{  This example uses STR$POSITION to determine
{  the position of the first occurrence of
{  a substring (SUBSTRING) within a source
{  string (STRING1) after the starting
{  position (START).
{
{  First, declare the external function.
{-}

FUNCTION STR$POSITION(SRCSTR : VARYING [A]
         OF CHAR; SUBSTR : VARYING [B] OF CHAR;
         STARTPOS : INTEGER) : INTEGER; EXTERN;
{+}
{  Declare the variables used in the main program.
{-}

VAR
  STRING1       : VARYING [256] OF CHAR;
  SUBSTRING     : VARYING [256] OF CHAR;
  START         : INTEGER;
  RET_STATUS    : INTEGER;

{+}
{  Begin the main program. Read the string and substring.
{  Set START equal to 1 to begin looking for the substring
{  at the beginning of the source string.  Call STR$POSITION
{  and print the result.
{-}
```

```
BEGIN
  WRITELN('ENTER THE STRING: ');
  READLN(STRING1);
  WRITELN('ENTER THE SUBSTRING: ');
  READLN(SUBSTRING);
  START := 1;
  RET_STATUS := STR$POSITION(STRING1, SUBSTRING, START);
  WRITELN(RET_STATUS);
END.
```

This Pascal program demonstrates the use of STR$POSITION. If you run this program and set STRING1 equal to KITTEN and substring equal to TEN, the value of RET_STATUS is 4.

The output generated by this program is as follows:

```
ENTER THE STRING:
KITTEN
ENTER THE SUBSTRING:
TEN
         4
```

STR$POS_EXTR Extract a Substring of a String

The Extract a Substring of a String routine copies a substring of a source string into a destination string.

FORMAT

STR$POS_EXTR *destination-string ,source-string ,start-position ,end-position*

corresponding jsb entry point

STR$POS_EXTR_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$POS_EXTR copies the substring. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string from which STR$POS_EXTR extracts the substring that it copies into the destination string. The **source-string** argument is the address of a descriptor pointing to the source string.

start-position

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Relative position in the source string at which STR$POS_EXTR begins copying the substring. The **start-position** argument is the address of a signed longword containing the starting position.

end-position

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Relative position in the source string at which STR$POS_EXTR stops copying the substring. The **end-position** argument is the address of a signed longword containing the ending position.

DESCRIPTION

STR$POS_EXTR extracts a substring from a source string and copies the substring into a destination string. STR$POS_EXTR defines the substring by specifying the relative starting and ending positions in the source string. The source string is unchanged, unless it is also the destination string.

If the starting position is less than 1 then 1 is used. If the starting position is greater than the length of the source string, the null string is returned. If the ending position is greater than the length of the source string, the length of the source string is used.

Other routines that may be used to extract and copy a substring are STR$LEFT, STR$RIGHT and STR$LEN_EXTR.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_ILLSTRPOS	Alternate success. An argument referenced a character position outside the specified string. A default value was used.
STR$_ILLSTRSPE	Alternate success. **End-position** was less than **start-position**. Default values were used.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters copied from the source string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$POS_EXTR could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
   0    |    1    |    2    |    3    |    4    |    5    |    6    |    7    |
12345678901234567890123456789012345678901234567890123456789012345678901234567890

     FTTY     D   F      80            TTY
     C* Initialize source string and position
     C                     MOVE '7 SW Ave'SOURCE  8
     C                     Z-ADD3         BEGPOS  90
     C                     Z-ADD4         ENDPOS  90
     C           POS_EXTR  EXTRN'STR$POS_EXTR'
     C* Extract the 2 character string beginning at position 3
     C                     CALL POS_EXTR
     C                     PARMD          DEST    2
     C                     PARMD          SOURCE
     C                     PARM           BEGPOS     RL
     C                     PARM           ENDPOS     RL
     C* Display on the terminal the extracted string
     C           DEST      DSPLYTTY
     C                     SETON                     LR
```

The RPG II program above displays the string 'SW' on the terminal.

STR$PREFIX Prefix a String

The Prefix a String routine inserts a source string at the beginning of a destination string. The destination string must be dynamic or varying length.

FORMAT

STR$PREFIX *destination-string ,source-string*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string (dynamic or varying length); STR$PREFIX copies the source string into the beginning of this destination string. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string that STR$PREFIX copies into the beginning of the destination string. The **source-string** argument is the address of a descriptor pointing to the source string.

DESCRIPTION

STR$PREFIX inserts the source string at the beginning of the destination string. The destination string must be dynamic or varying length.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$PREFIX could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
10 !+
   !  This example uses STR$PREFIX to
   !  prefix a destination string (D$)
   !  with a source string ('ABCD').
   !-

   EXTERNAL INTEGER FUNCTION STR$PREFIX
   D$ = 'EFG'
   STATUS% = STR$PREFIX (D$, 'ABCD')
   PRINT D$
   END
```

These BASIC statements set D$ equal to 'ABCDEFG'.

STR$RECIP Reciprocal of a Decimal String

The Reciprocal of a Decimal String routine takes the reciprocal of the first decimal string to the precision limit specified by the second decimal string and returns the result as a decimal string.

FORMAT

STR$RECIP *asign ,aexp ,adigits ,bsign ,bexp ,bdigits ,csign ,cexp ,cdigits*

RETURNS

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

ARGUMENTS

asign

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

Sign of the first operand. The **asign** argument is the address of an unsigned longword containing the first operand's sign. Zero is considered positive; 1 is considered negative.

aexp

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

Power of 10 by which **adigits** is multiplied to get the absolute value of the first operand. The **aexp** argument is the address of a signed longword containing this exponent.

adigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

First operand's numeric string. The **adigits** argument is the address of a descriptor pointing to the first operand's numeric string. The string must be an unsigned decimal number.

bsign

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

Sign of the second operand. The **bsign** argument is the address of an unsigned longword containing the sign of the second operand. Zero is considered positive; 1 is considered negative.

bexp

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

Power of 10 by which **bdigits** is multiplied to get the absolute value of the second operand. The **bexp** argument is the address of a signed longword containing this exponent.

bdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

Second operand's numeric string. The **bdigits** argument is the address of a descriptor pointing to the second operand's numeric string. The string must be an unsigned decimal number.

csign

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

Sign of the result. The **csign** argument is the address of an unsigned longword containing the result's sign. Zero is considered positive; 1 is considered negative.

cexp

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

Power of 10 by which **cdigits** is multiplied to get the absolute value of the result. The **cexp** argument is the address of a signed longword containing this exponent.

cdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **write only**
mechanism: **by descriptor**

Result's numeric string. The **cdigits** argument is the address of a descriptor pointing to the result's numeric string. The string is an unsigned decimal number.

DESCRIPTION

STR$RECIP takes the reciprocal of the first decimal string to the precision limit specified by the second decimal string and returns the result as a decimal string.

STR$RECIP

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

CONDITION VALUES SIGNALED

STR$_DIVBY_ZER	Division by zero.
LIB$_INVARG	Invalid argument.
STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$RECIP could not allocate heap storage for a dynamic or temporary string.
STR$_WRONUMARG	Wrong number of arguments.

EXAMPLE

```
100 !+
    !  This example program uses
    !  STR$RECIP to find the reciprocal of
    !  the first decimal string (A) to the
    !  precision specified in the second
    !  decimal string (B), and place the
    !  result in a third decimal string (C).
    !-

    ASIGN% = 1%
    AEXP% = 3%
    ADIGITS$ = '1'
    BSIGN% = 0%
    BEXP% = -4%
    BDIGITS$ = '2'
    CSIGN% = 0%
    CEXP% = 0%
    CDIGITS$ = '0'

    PRINT "A = "; ASIGN%; AEXP%; ADIGITS$
    PRINT "B = "; BSIGN%; BEXP%; BDIGITS$
    CALL STR$RECIP      (ASIGN%, AEXP%, ADIGITS$, &
                         BSIGN%, BEXP%, BDIGITS$,  &
                         CSIGN%, CEXP%, CDIGITS$)
    PRINT "C = "; CSIGN%; CEXP%; CDIGITS$

999 END
```

This BASIC example uses STR$RECIP to find the reciprocal of A to the precision level specified in B.

The following values apply:

A = –1000 (ASIGN = 1, AEXP = 3, ADIGITS = '1')
B = .0002 (BSIGN = 0, BEXP = –4, BDIGITS = '2')

The output generated by this program is as follows, yielding a decimal value of C equal to –.001.

```
A = 1  3 1
B = 0 -4 2
C = 1 -3 1
```

STR$REPLACE Replace a Substring

The Replace a Substring routine copies a source string to a destination string, replacing part of the string with another string. The substring to be replaced is specified by its starting and ending positions.

FORMAT

STR$REPLACE *destination-string ,source-string ,start-position ,end-position ,replacement-string*

corresponding jsb entry point

STR$REPLACE_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$REPLACE writes the new string created when it replaces the substring. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string. The **source-string** argument is the address of a descriptor pointing to the source string.

start-position

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Position in the source string at which the substring that STR$REPLACE replaces begins. The **start-position** argument is the address of a signed longword containing the starting position. The position is relative to the start of the source string.

end-position

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Position in the source string at which the substring that STR$REPLACE replaces ends. The **end-position** argument is the address of a signed longword containing the ending position. The position is relative to the start of the source string.

replacement-string

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

Replacement string with which STR$REPLACE replaces the substring. The **replacement-string** argument is the address of a descriptor pointing to this replacement string. The value of **replacement-string** must be equal to **end-position** minus **start-position**.

DESCRIPTION

STR$REPLACE copies a source string to a destination string, replacing part of the string with another string. The substring to be replaced is specified by its starting and ending positions.

If the starting position is less than 1, 1 is used. If the ending position is greater than the length of the source string, the length of the source string is used. If the starting position is greater than the ending position, the overlapping portion of the source string will be copied twice.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_ILLSTRPOS	Alternate success. An argument referenced a character position outside the specified string. A default value was used.
STR$_ILLSTRSPE	Alternate success. **End-position** was less than **start-position** or the length of the substring was too long for the specified string. Default values were used.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

STR$REPLACE

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$REPLACE could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
10 !+
   !  This example uses STR$REPLACE to
   !  replace all characters from the starting
   !  position (2%) to the ending position (3%)
   !  with characters from the replacement string
   !  ('XYZ').
   !-

   EXTERNAL INTEGER FUNCTION STR$REPLACE
   D$ = 'ABCD'
   STATUS% = STR$REPLACE (D$, D$, 2%, 3%, 'XYZ')
   PRINT D$
   END
```

These BASIC statements set D$ equal to 'AXYZD'.

STR$RIGHT Extract a Substring of a String

The Extract a Substring of a String routine copies a substring of a source string into a destination string.

FORMAT

STR$RIGHT *destination-string ,source-string ,start-position*

corresponding jsb entry point

STR$RIGHT_R8

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$RIGHT copies the substring. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string from which STR$RIGHT extracts the substring that it copies into the destination string. The **source-string** argument is the address of a descriptor pointing to the source string.

start-position

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference for CALL entry point, by value for JSB entry point**

Relative position in the source string at which the substring that STR$RIGHT copies starts. The **start-position** argument is the address of a signed longword containing the starting position.

STR$RIGHT

DESCRIPTION

STR$RIGHT extracts a substring from a source string and copies that substring into a destination string. STR$RIGHT defines the substring by specifying the relative starting position. The relative ending position is equal to the length of the source string. The source string is unchanged, unless it is also the destination string.

If the starting position is less than 2, the entire source string is copied. If the starting position is greater than the length of the source string, a null string is copied.

This is a variation of STR$POS_EXTR. Other routines that may be used to extract and copy a substring are STR$LEFT and STR$LEN_EXTR.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_ILLSTRPOS	Alternate success. An argument referenced a character position outside the specified string. A default value was used.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters copied from the source string.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$RIGHT could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
PROGRAM RIGHT(INPUT, OUTPUT);

{+}
{  This example uses STR$RIGHT to extract a substring
{  from a specified starting position (START_POS) to
{  the end (right side) of a source string (SRC_STR)
{  and write the result in a destination string (DST_STR).
{
{  First, declare the external procedure.
{-}

PROCEDURE STR$RIGHT(%DESCR DSTSTR: VARYING
          [A] OF CHAR; SRCSTR  : VARYING [B] OF CHAR;
          STARTPOS : INTEGER); EXTERN;
{+}
{  Declare the variables used in the main program.
{-}
```

```
VAR
  SRC_STR       : VARYING [256] OF CHAR;
  DST_STR       : VARYING [256] OF CHAR;
  START_POS     : INTEGER;

{+}
{  Begin the main program.  Read the source string
{  and starting position. Call STR$RIGHT to extract
{  the substring.  Print the result.
{-}

BEGIN
  WRITELN('ENTER THE SOURCE STRING: ');
  READLN(SRC_STR);
  WRITELN('ENTER THE STARTING POSITION');
  WRITELN('OF THE SUBSTRING: ');
  READLN(START_POS);
  STR$RIGHT(DST_STR, SRC_STR, START_POS);
  WRITELN;
  WRITELN('THE SUBSTRING IS: ',DST_STR);
END.
```

This Pascal program uses STR$RIGHT to extract a substring from a specified starting position (START_POS) to the end of the source string. One sample of the output is as follows:

```
$ RUN RIGHT
ENTER THE SOURCE STRING: BLUE PLANETS ALWAYS HAVE PURPLE PLANTS
ENTER THE STARTING POSITION
OF THE SUBSTRING: 27
THE SUBSTRING IS: URPLE PLANTS
```

STR$ROUND Round or Truncate a Decimal String

The Round or Truncate a Decimal String routine rounds or truncates a decimal string to a specified number of significant digits and places the result in another decimal string.

FORMAT

STR$ROUND *places ,flags ,asign ,aexp ,adigits ,csign ,cexp ,cdigits*

RETURNS

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

ARGUMENTS

places

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

Maximum number of decimal digits that STR$ROUND retains in the result. The **places** argument is the address of a signed longword containing the number of decimal digits.

flags

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

Function flag. Zero indicates that the decimal string is rounded; 1 indicates that it is truncated. The **flags** argument is the address of an unsigned longword containing this function flag.

asign

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

Sign of the first operand. The **asign** argument is the address of an unsigned longword string containing this sign. A value of zero indicates that the number is positive, while a value of 1 indicates that the number is negative.

aexp

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

Power of 10 by which **adigits** is multiplied to get the absolute value of the first operand. The **aexp** argument is the address of a signed longword containing this exponent.

adigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **read only**
mechanism: **by descriptor**

First operand's numeric string. The **adigits** argument is the address of a descriptor pointing to this numeric string. The string must be an unsigned decimal number.

csign

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

Sign of the result. The **csign** argument is the address of an unsigned longword containing the result's sign. A value of zero indicates that the number is positive, while a value of 1 indicates that the number is negative.

cexp

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

Power of 10 by which **cdigits** is multiplied to get the absolute value of the result. The **cexp** argument is the address of a signed longword containing this exponent.

cdigits

VMS usage: **char_string**
type: **numeric string, unsigned**
access: **write only**
mechanism: **by descriptor**

Result's numeric string. The **cdigits** argument is the address of a descriptor pointing to this numeric string. The string is an unsigned decimal number.

DESCRIPTION

The Round or Truncate a Decimal String routine rounds or truncates a decimal string to a specified number of significant digits and places the result in another decimal string.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

STR$ROUND

CONDITION VALUES SIGNALED

LIB$_INVARG	Invalid argument.
STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$ROUND could not allocate heap storage for a dynamic or temporary string.
STR$_WRONUMARG	Wrong number of arguments.

EXAMPLE

```
100 !+
    !  This example shows the difference between
    !  the values obtained when rounding or truncating
    !  a decimal string.
    !-

    ASIGN% = 0%
    AEXP% = -4%
    ADIGITS$ = '9999998'
    CSIGN% = 0%
    CEXP% = 0%
    CDIGITS$ = '0'
    PRINT "A = "; ASIGN%; AEXP%; ADIGITS$

    !+
    !  First, call STR$ROUND to round the value of A.
    !-

    CALL STR$ROUND      (3%, 0%, ASIGN%, AEXP%, ADIGITS$, &
                         CSIGN%, CEXP%, CDIGITS$)
    PRINT "ROUNDED:  C = "; CSIGN%; CEXP%; CDIGITS$

    !+
    !  Now, call STR$ROUND to truncate the value of A.
    !-

    CALL STR$ROUND     (3%, 1%, ASIGN%, AEXP%, ADIGITS$, &
                        CSIGN%, CEXP%, CDIGITS$)
    PRINT "TRUNCATED:  C = "; CSIGN%; CEXP%; CDIGITS$
999 END
```

This BASIC example uses STR$ROUND to first round and then truncate the value of A to the number of decimal places specified by **places**. The following values apply:

A = 999.9998 (ASIGN = 1, AEXP = –4, ADIGITS = '9999998')

Listed below is the output generated by this program; note that the decimal value of C equals 1000 when rounded, and 999 when truncated.

```
A = 1  -4 9999998
ROUNDED: C = 0 1 100
TRUNCATED: C = 0 0 999
```

STR$TRANSLATE Translate Matched Characters

The Translate Matched Characters routine successively compares each character in a source string to all characters in a match string. If a source character has a match, the destination character is taken from the translate string. Otherwise, STR$TRANSLATE moves the source character to the destination string.

FORMAT

STR$TRANSLATE *destination-string ,source-string ,translation-string ,match-string*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string. The **source-string** argument is the address of a descriptor pointing to the source string.

translation-string

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

Translate string. The **translation-string** argument is the address of a descriptor pointing to the translate string.

match-string

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

Match string. The **match-string** argument is the address of a descriptor pointing to the match string.

DESCRIPTION

STR$TRANSLATE successively compares each character in a source string to all characters in a match string. If a source character matches any of the characters in the match string, STR$TRANSLATE moves a character from the translate string to the destination string. Otherwise, STR$TRANSLATE moves the character from the source string to the destination string.

The character taken from the translate string has the same relative position as the matching character had in the match string. When a character appears more than once in the match string, the position of the leftmost occurrence of the multiply-defined character is used to select the translate string character. If the translate string is shorter than the match string and the matched character position is greater than the translate string length, the destination character is a space.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all of the characters.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$TRANSLATE could not allocate heap storage for a dynamic or temporary string.

EXAMPLE

```
10      !+
        !  This example program uses STR$TRANSLATE to
        !  translate all characters of a source string
        !  from uppercase to lowercase characters.
        !-

        EXTERNAL INTEGER FUNCTION STR$TRANSLATE(STRING,STRING,STRING,STRING)
        TO$='abcdefghijklmnopqrstuvwxyz'
        FROM$='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        X% = STR$TRANSLATE(OUT$,'TEST',TO$,FROM$)
        PRINT 'Status = ';x%
        PRINT 'Resulting string = ';out$
32767   END
```

This BASIC example translates uppercase letters to lowercase letters, thus performing the same function as STR$UPCASE.

The output generated by this example is as follows:

```
$ RUN TRANSLATE
Status =  1
Resulting string = test
```

A more practical although more complicated use for STR$TRANSLATE would be to encrypt data by translating the characters to obscure combinations of numbers and alphabetic characters.

STR$TRIM Trim Trailing Blanks and Tabs

The Trim Trailing Blanks and Tabs routine copies a source string to a destination string and deletes the trailing blank and tab characters.

FORMAT

STR$TRIM *destination-string ,source-string [,resultant-length]*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$TRIM copies the trimmed string. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string which STR$TRIM trims and then copies into the destination string. The **source-string** argument is the address of a descriptor pointing to the source string.

resultant-length

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

Number of bytes that STR$TRIM writes into **destination-string**, not counting padding in the case of a fixed-length string. The **resultant-length** argument is the address of an unsigned word into which STR$TRIM writes the length of the output string. If the input string is truncated to the size specified in the **destination-string** description, **resultant-length** is set to this size. Therefore, **resultant-length** can always be used by the calling program to access a valid substring of **destination-string**.

DESCRIPTION

STR$TRIM copies a source string to a destination string and deletes the trailing blank and tab characters.

STR$TRIM

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters.

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$TRIM could not allocate heap storage for a dynamic or temporary string.

STR$UPCASE Convert String to All Uppercase Characters

The Convert String to All Uppercase Characters routine converts a source string to uppercase.

FORMAT

STR$UPCASE *destination-string ,source-string*

RETURNS

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

ARGUMENTS

destination-string

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

Destination string into which STR$UPCASE writes the string it has converted to uppercase. The **destination-string** argument is the address of a descriptor pointing to the destination string.

source-string

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

Source string that STR$UPCASE converts to uppercase. The **source-string** argument is the address of a descriptor pointing to the source string.

DESCRIPTION

STR$UPCASE converts successive characters in a source string to uppercase and writes the converted character into the destination string. The routine converts all characters in the DEC Multinational Character Set.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
STR$_TRU	String truncation warning. The fixed-length destination string could not contain all the characters.

STR$UPCASE

CONDITION VALUES SIGNALED

STR$_FATINTERR	Fatal internal error. An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL in a Software Performance Report (SPR).
STR$_ILLSTRCLA	Illegal string class. The class code found in the class field of a descriptor is not a string class code allowed by the VAX Procedure Calling and Condition Handling Standard.
STR$_INSVIRMEM	Insufficient virtual memory. STR$UPCASE could not allocate heap storage for a dynamic or temporary string.

EXAMPLES

1

```
30  !+
    !  This example uses STR$UPCASE
    !  to convert all characters in
    !  the source string (SRC$) to
    !  uppercase and write the result
    !  in the destination string (DST$).
    !-

    SRC$ = 'abcd'
    PRINT "SRC$ =";SRC$
    CALL STR$UPCASE (DST$, SRC$)
    PRINT "DST$ =";DST$
    END
```

This BASIC program generates the following output:

```
SCR$ =abcd
DST$ =ABCD
```

2

```
     0    |    1    |    2    |    3    |    4    |    5    |    6    |    7    |
12345678901234567890123456789012345678901234567890123456789012345678901234567890
     FTTY     D   F      80            TTY
     C* Initialize string to be converted to uppercase
     C                     MOVE 'rep head'HEAD    8
     C           UPCASE    EXTRN'STR$UPCASE'
     C* Convert the string to uppercase
     C                     CALL UPCASE
     C                     PARMD          RESULT  8
     C                     PARMD          HEAD
     C* Display on the terminal the string in uppercase
     C           RESULT    DSPLYTTY
     C                     SETON                     LR
```

The RPG II program above displays the string 'REP HEAD' on the terminal.

Index

V

Reader's Comments

VMS RTL String Manipulation (STR$) Manual
AA–LA75A–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 ----------