digital

VAX-11
Run-Time Library
Reference Manual

Order No. AA-D036B-TE

VAX11

April 1980

This document contains detailed descriptions of all general purpose procedures in the VAX-11 Common Run-Time Procedure Library. It also contains information about calling library procedures, including programming techniques. The information in this book is not introductory in nature.

VAX-11
Run-Time Library
Reference Manual

Order No. AA-D036B-TE

OPERATING SYSTEM AND VERSION: VAX/VMS V02

SOFTWARE VERSION: VAX/VMS V02

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First printing, August 1978
Revised April 1980

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

Contents

Chapter 3 General Utility Procedures

Chapter 4 Mathematics Procedures

Chapter 5 Process-Wide Resource Allocation Procedures

Chapter 6 Signaling and Condition Handling Procedures

Chapter 7 Syntax Analysis Procedures

Chapter 8 Cross-Reference Procedures

Appendix A Summary of Run-Time Library Entry Points

Appendix B Run-Time Library Error Messages

Appendix C Vax-11 Procedure Calling and Condition Handling Standard

Appendix D Algorithms for Mathematics Procedures

Appendix E Image Initialization and Termination

Appendix F CALLG, CALLS Instructions

Appendix G Sample Programs Using LIB$TPARSE

Figures

Tables

Commercial Engineering Publications typeset this manual using DIGITAL's TMS-11 Text Management System.

036ALL

Preface

Document Objectives

The VAX-11 Run-Time Library comprises two types of procedures: general purpose and language-support. This manual introduces the entire library and describes the callable interface to the general utility procedures. The *VAX-11 Guide to Creating Modular Library Procedures* describes how to write modular procedures.

This manual introduces the library, describes the calling and naming conventions, and presents all procedures of a general nature. Each procedure is documented with a functional description including algorithms and examples, where appropriate, and instructions for access in all VAX-11 supported languages.

Intended Audience

This manual is intended for system and application programmers who are already familiar with VAX/VMS system concepts but require a detailed knowledge of the Run-Time Library. Readers are assumed to be familiar with the VAX/VMS operating system, and proficient in a language supported by VAX/VMS.

Document Structure

The first two chapters of this manual are tutorial, providing an overview of the Run-Time Library.

- Chapter 1 is an introduction to the library, detailing how it can be used and how it is organized.

- Chapter 2 explains how to call library procedures and describes the naming conventions and procedure parameters.

Chapters 3 through 8 contain reference material, providing detailed descriptions of each library procedure:

- Chapter 3 describes the general utility procedures.
- Chapter 4 contains the mathematics procedures.
- Chapter 5 details the resource allocation procedures.
- Chapter 6 presents the signaling and condition handling procedures, and information on how you can control the handling of error conditions and the printing of error messages by writing your own condition handlers.
- Chapter 7 describes syntax analysis procedures.
- Chapter 8 describes cross-reference procedures.

The appendixes provide useful background information:

- Appendix A lists all general purpose entry points in the Run-Time Library, including coding information for the parameters.
- Appendix B lists all error messages and condition symbols returned from or signaled by library procedures.
- Appendix C is the VAX-11 Procedure Calling Standard.
- Appendix D contains algorithms for the mathematics procedures.
- Appendix E explains image initialization and termination and how users can control them.
- Appendix F explains in detail the operation of CALLS and CALLG instructions.
- Appendix G contains detailed MACRO and BLISS examples using the syntax analysis procedures.

Associated Documents

The following document in association with this manual comprise the VAX-11 Run-Time Library Documentation:

- *VAX-11 Guide to Creating Modular Library Procedures*

The following documents are associated with this manual:

- *VAX-11 MACRO User's Guide*
- *VAX-11 MACRO Language Reference Manual*
- *VAX-11 BLISS-32 User's Guide*

- *BLISS Language Guide*
- *VAX-11 BASIC User's Guide*
- *VAX-11 BASIC Language Reference Manual*
- *VAX-11 COBOL-74 User's Guide*
- *VAX-11 COBOL-74 Language Reference Manual*
- *VAX-11 FORTRAN User's Guide*
- *VAX-11 FORTRAN Language Reference Manual*
- *VAX-11 PASCAL User's Guide*
- *VAX-11 PASCAL Language Reference Manual*
- *VAX/VMS System Services Reference Manual*

For a complete list of all VAX-11 documents, including brief descriptions of each, see the *VAX-11 Information Directory*.

Conventions

Unless otherwise noted:

- all numeric values are represented in decimal notation
- all commands terminate with a carriage return

Variable information is indicated by lowercase characters; literal information, which you must enter exactly as shown, is indicated by uppercase characters.

Brackets ([ ]) in procedure descriptions indicate optional arguments. An equal sign after an optional parameter indicates the default value if you omit the parameter.

Ellipses (...) indicate parameters that can be repeated one or more times.

Unless otherwise specified, the term:

- MACRO means VAX-11 MACRO
- BLISS means BLISS-32
- BASIC means VAX-11 BASIC
- COBOL means VAX-11 COBOL-74
- FORTRAN means VAX-11 FORTRAN
- PASCAL means VAX-11 PASCAL
- Run-Time Library means VAX-11 Common Run-Time Procedure Library
- Linker means VAX-11 Linker

Summary of Technical Changes

This manual documents the *VAX-11 Run-Time Library Reference Manual* Version 2.0. This section summarizes the technical changes from Version 1.0.

Languages
Added examples and instructions for calling Run-Time Library procedures from BASIC, COBOL, and PASCAL.

Miscellaneous
Chapters 3,4 and 5 have been rearranged and restructured to accommodate the many new procedures. Appendices A and D have correspondingly been reordered. See the Index for an enumeration of the procedure names.

General Utility Procedures
Added new procedures for performance measurement, I/O control, interlocked queue instructions, formatted I/O conversion, terminal independent screen functions, emulate G_floating, H_floating, and O (octaword) instructions, simulate floating traps on machines which have floating faults, date/time utility procedures, translation tables and routines.

String Facility
Added new STR$ facility with string arithmetic and many additional string manipulation procedures (see Chapters 3 and 5). This facility has:

- CALL entry points, with scalar arguments passed by reference
- JSB entry points, with scalar arguments passed by immediate value
- Support for all string data types specified in the VAX-11 Procedure Calling Standard
- Mechanism for being called directly from higher-level languages

Math Library
Added G_ and H_ floating mathematical functions and D_ and G_complex mathematical routines. Added a new JSB entry point (MTH$SQRT_R3) to improve the accuracy of the single-precision square root. Other JSB entry points (MTH$ACOS_R4, MTH$ASIN_R4, MTH$DACOS_R7, MTH$DASIN_R7, MTH$DEXP_R6) have been improved so they use fewer registers without impacting execution speed.

FLOOR routines were added which return a truncated (towards minus infinity) integer part of a number in a floating-point representation. SGN routines were added which return -1, 0, or 1 depending on the sign of the floating-point input.

Resource Allocation
Added new routines for allocation of dynamic strings, event flags, and logical unit numbers.

Syntax Analysis
Added a new example in BLISS and moved both examples (MACRO and BLISS) to a new Appendix G.

Cross-reference
Added Chapter 8 which contains instructions and examples for using the cross-reference procedures.

Error messages
Added new error messages for LIB$, MTH$ and STR$ and removed the FOR$ messages. The FOR$ error messages are in the *VAX-11 FORTRAN User's Guide.*

VAX-11 Procedure Calling Standard
Appendix C has been updated with many new data types and other features. A complete revision history can be found at the end of the appendix.

Algorithms
Added new algorithms for G__ and H__floating math functions and for D__ and G__complex math procedures.

USEROPEN
The appendix on USEROPEN has been removed. The old Appendix G (detailing CALLS and CALLG) is now Appendix F.

All chapters and appendixes have been revised to bring this manual up to the VAX/VMS V2.0 level.

Chapter 1
Introduction

The VAX-11 Run-Time Library (or simply, the Run-Time Library) contains sets of general purpose and language support procedures. MACRO, BLISS, or high-level language user programs call these procedures in any combination to perform tasks required for program execution. Because all procedures follow the VAX-11 Modular Programming Standard, a common run-time environment is provided for user programs.

The common run-time environment means that any program written in MACRO, BLISS, or a supported high-level language (BASIC, COBOL, FORTRAN, PASCAL) can call any procedure in the Run-Time Library. This environment lets your program contain procedures written in different languages, thus increasing program flexibility.

A procedure is a set of related instructions that performs a particular task. It is an executable program unit, and can be a main program, subroutine, or function. A procedure has an entry point, a parameter (or argument) list, a return point, and, optionally, a function value or completion status.

Run-Time Library procedures are written using the VAX-11 Procedure Calling Standard. They are reentrant and position-independent. In addition, VAX/VMS system services are callable procedures that can be used with Run-Time Library procedures. (See the *VAX/VMS System Services Reference Manual.*)

1.1 Run-Time Library Capabilities

The Run-Time Library provides the following capabilities:

- Language-independent support procedures that perform common language services only once, rather than once for each language.
- Compiler-generated procedures written in any language that can be called from procedures written in any other language. Each procedure can use its

language-specific features fully without affecting other procedures. In certain cases, one language can use some of the features of the other languages.

- File, data type, and procedure-call compatibility between the languages supported by VAX/VMS. File and error handling compatibility between the VAX-11 and the 16-bit PDP-11 is also provided.

- Capability to add new languages.

- File input/output (I/O) that interfaces solely with VAX-11 Record Management Services (RMS).

- For each VAX-11 native-mode language, the ability of the Run-Time Library to produce files compatible with files produced by the corresponding PDP-11 and VAX-11 compatibility-mode language.

- The ability for each VAX-11 native-mode language to process files produced by programs written in other languages.

- Use of all procedures from both the Asynchronous System Trap (AST) and nonAST levels in the same image (two levels maximum). Thus, all procedures are reentrant.

1.2 Linking with the Run-Time Library

Figure 1-1 shows the program development cycle for a user program that calls the library.

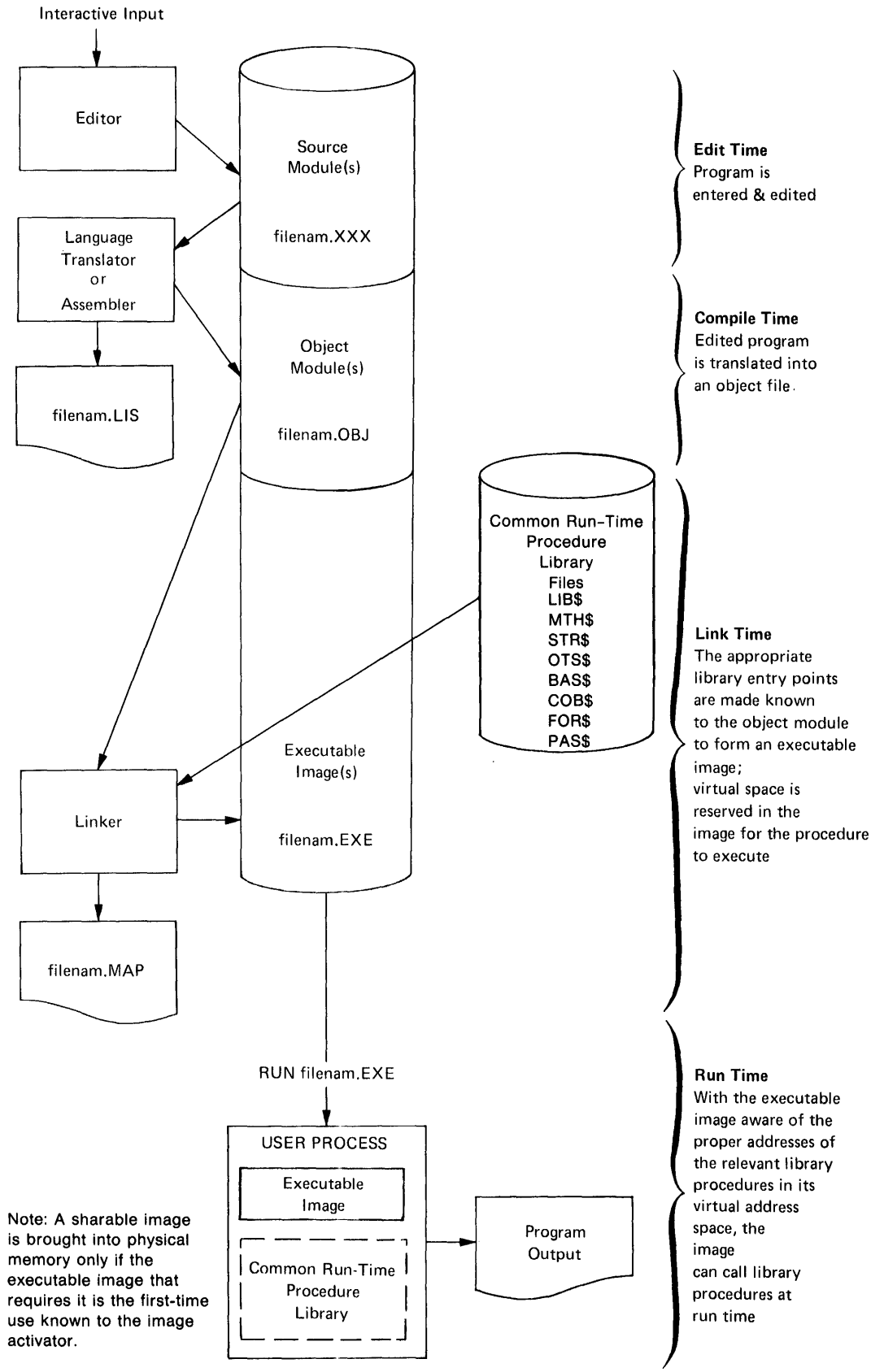
The Run-Time Library is part of the system library automatically searched when user programs are linked. Run-Time Library procedures execute entirely in user mode and work only when called by native-mode user programs.

Normally, the image activator incorporates sections of the Run-Time Library into an executable image at run time when you type the RUN command. You can also link copies of procedures from the library directly to your image by typing the LINK command with the /INCLUDE qualifier.

When a user program calls the Run-Time Library, the program refers to a storage location in the library that points to the starting address of the procedure to be executed. This storage location is called a *transfer vector*.

Transfer vectors permit a single, position-independent copy of the library procedure to be associated with different virtual addresses in the user images sharing the procedures. This is done by allocating a global section to the Run-Time Library to make it a sharable image.

Figure 1-1: Development of a Program that Calls the Run-Time Library



The sharable image is mapped into the address space of an executable image, which is in turn activated by the RUN command. At run time, a call instruction in the user program passes control to a transfer vector that in turn branches to the called library procedure. This mechanism lets many users share the same image: the procedure's code can be in different places in several users' address space simultaneously.

The transfer vectors and the mapping of global sections into a process's address space at run time also permit a new version of the library to be installed without relinking the user images. This is possible because the location of transfer vectors remains the same— only their contents change for each new version.

1.3 Library Calling Conventions

The Run-Time Library conforms to the VAX-11 Procedure Calling Standard (see Appendix C). Therefore, its procedures can be called by all native-mode languages. Chapter 2 describes the explicit calls you can use to any procedure.

Each procedure has a call entry point. Frequently used procedures also have a jump-to-subroutine (JSB) entry point. JSB instructions execute faster than call instructions, but they have some limitations: for example, they do not create a stack frame, and thus execute in the environment of the caller.

Each procedure belongs to a library facility, which is a set of related procedures. Each procedure's facility is indicated by a four-character prefix to the procedure's name. For example, the MTH$SIN procedure belongs to the mathematics facility, as indicated by MTH$. Each facility has its own error messages, parameter passing mechanisms, and specific parameter forms. The facilities currently in the library are:

- LIB$ - General purpose procedures such as utility, resource allocation, signaling and condition handling
- MTH$ - Mathematics procedures
- STR$ - String manipulation procedures
- OTS$ - Language-independent support procedures
- BAS$ - BASIC-specific support procedures
- COB$ - COBOL-specific support procedures
- FOR$ - FORTRAN-specific support procedures
- PAS$ - PASCAL-specific support procedures

To execute properly, each library procedure requires you to supply parameters. Each parameter must be of the data type and form required by the procedure and must be passed in the proper order by the correct mechanism. For many procedures, some of the parameters are optional. You can select your own parameter names, but you must code them as outlined in Chapter 2.

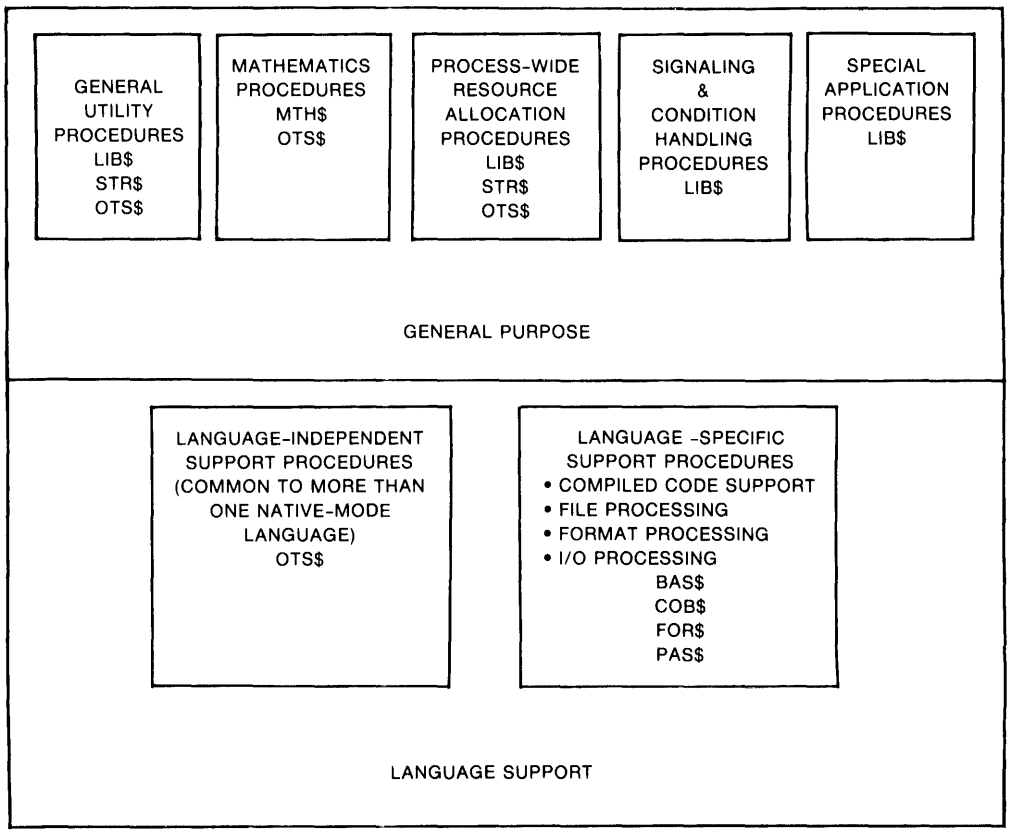
Some procedures return a completion value or a function value. Procedures called from a high-level language receive this as the value of the function. Procedures called in assembly language (MACRO) can access this value in register R0 or R0/R1.

Some procedures allocate image resources (for example virtual memory). Any library procedure that needs such resources automatically calls the necessary library resource allocation procedures. Your programs should also call these procedures when they need image resources.

1.4 Organization of the Run-Time Library

Figure 1-2 illustrates the organization of the Run-Time Library. The library consists of two major parts: general purpose procedures and language support procedures. General purpose procedures are documented in Chapters 3 through 8. Appendix A of this manual summarizes these Run-Time Library entry points.

Figure 1-2: The VAX-11 Run-Time Procedure Library



1.4.1 General Purpose Procedures

The following sections summarize general purpose procedures. Normally, user programs call these procedures using explicit CALL statements or function references (see Chapter 2).

1.4.1.1 General Utility Procedures — General utility procedures include procedures for getting a record from a logical device, string manipulation, input and output conversion, and date/time functions.

Chapter 3 details these procedures.

1.4.1.2 Mathematics Procedures — Mathematics procedures perform common arithmetic, algebraic, and trigonometric functions; for example, taking the sine of an angle. They are written in MACRO to use the speed and accuracy of the VAX-11 floating-point instructions. The frequently used mathematics procedures have both JSB and standard call entry points.

Chapter 4 details these procedures.

1.4.1.3 Resource Allocation Procedures — Resource allocation procedures allocate the following process resources:

- Virtual memory — one procedure to allocate and another to deallocate arbitrary sized blocks of the program region
- VMS event flags — one procedure to allocate and another to deallocate event flags
- BASIC/FORTRAN logical unit numbers — one procedure to allocate and another to deallocate logical unit numbers
- Character strings — procedures to copy and convert both fixed length and dynamic strings; procedures to allocate and deallocate dynamic strings

Chapter 5 details these procedures.

1.4.1.4 Signaling and Condition Handling Procedures — Signaling and condition handling procedures signal exception conditions and support condition handlers so that you can control errors and change system default responses. Specifically, the signaling and condition handling procedures let you:

- Communicate errors between user programs, the Run-Time Library, and VAX/VMS
- Alter the default condition handling mechanisms, including the printing of error messages
- Establish and write special condition handlers to correct, report, and control errors

- Enable and disable hardware traps
- Establish and remove condition handlers associated with a procedure activation

Chapter 6 details these procedures.

1.4.1.5 Syntax Analysis Procedures — Syntax analysis procedures analyze strings. The library includes a table-driven parser called LIB$TPARSE, and a keyword recognizing procedure called LIB$MATCH_KEY.

Chapter 7 details these procedures.

1.4.1.6 Cross-Reference Procedures — The cross-reference procedures are contained in a separate sharable image. They can create a cross-reference analysis of symbols. The procedures accept cross-reference data, summarize it, and format it for output. The interface to the cross-reference procedures is through a set of control blocks and format definition tables.

Chapter 8 details these procedures.

1.4.2 Language Support Procedures

Language support procedures are generally called implicitly by compiler-generated code, as a result of a statement in the higher-level language. The language support procedures consist of:

- Procedures that support a specific language compiler
- Procedures that support more than one native-mode language compiler

1.4.2.1 Language-Specific Procedures — The language-specific procedures support the in-line code generated by the language compilers. Some language-specific procedures are of general utility such as input/output conversion and date/time. For example, to perform a Language A function from a Language B program, you may find it easier to write a short Language A procedure to perform the function, and to call that procedure from your Language B program. Chapter 3 documents language-specific procedures, which generally include:

- File processing support procedures
- Auxiliary input/output procedures
- System procedures
- Compiled-code support procedures
- Compatibility procedures

1.4.2.2 Language-Independent Support Procedures — Language-independent support procedures consist of all procedures used by more than one native-mode language compiler. These include:

- Initialization and termination procedures
- Error and exception condition procedures
- Data type conversion procedures

1.5 Procedure Descriptions

Chapters 3 through 8 describe each library procedure. Sections in these chapters are arranged by major category (for example, Performance Measurement Procedures). Each section presents the procedures in related groups or alphabetically by functional description. In addition, Appendix A summarizes the procedure names and calling sequences.

Each procedure description consists of the following categories, as applicable:

Format

Shows the high-level language format of the procedure, giving the procedure name and parameter order. JSB entry points (if any) are also listed.

Parameters

Describes each parameter. A parameter to the left of the entry point name in the format is written by the procedure; parameters to the right are read and sometimes written by the procedure. For example:

old-setting = LIB$FLT_UNDER (new-setting)

In this call, the procedure writes old-setting and reads new-setting.

In the format, required input parameters occur first, followed by required output parameters (if any). Required input and output parameters are followed by optional input and output parameters.

Function Value

Library procedures return: (1) nothing (a subroutine) (2) a function value or (3) a return status that indicates whether the procedure completed successfully.

In case (1), the format begins with CALL ... . No function value or status code is returned, and the contents of registers R0/R1 are unspecified at completion.

In case (2) or (3), the parameter to the left of the equal sign is either: (a) a descriptive name indicating the nature of the function value returned in R0 or R0/R1, or (b) ret-status indicating a return status in R0.

Function values follow the parameters.

Implicit Inputs (JSB Entry)

Includes any parameters passed in registers for JSB entries.

Implicit Outputs (JSB Entry)

Includes any parameters passed in registers for JSB entries.

Return Status

Lists the possible completion codes that the procedure returns in register R0 or R0/R1. The successful returns are listed first, in alphabetical order, followed by error return status codes, also in alphabetical order. Successful completion (bit 0 = 1) is always shown by "procedure successfully completed." If an error status is returned, the severity field of the condition value is always SEVERE (bits 2:0 = 4) unless ERROR (bits 2:0 = 2) or WARNING (bits 2:0 = 0) is the first word of the explanation.

Messages

Lists the error messages produced when procedures signal error conditions. Unless stated otherwise, all error messages are signaled as SEVERE by calling LIB$STOP.

Notes

Describes any actions taken or side effects performed by the procedure that are not covered under one of the other headings. When an action is identical for all procedures in a given library facility, the action is listed in the chapter introduction only.

Examples

Gives a simple example(s) using the procedure in a short program segment to clarify the passing mechanisms in the various languages.

Chapter 2
Calling Run–Time Library Procedures

User programs call Run–Time Library procedures using the VAX–11 Procedure Calling Standard (see Appendix C). All of the programming languages that generate VAX–11 native-mode code provide mechanisms for coding the procedure calls. Sections 2.2 through 2.6 describe general aspects of calling procedures on VAX/VMS. Sections 2.7 through 2.12 describe how to call library procedures using MACRO, BLISS, BASIC, COBOL, FORTRAN and PASCAL.

When you code instructions to call a library procedure, you must furnish whatever parameters the procedure requires.

When the procedure completes execution, it returns control to the calling program. If the procedure returns a status code, the calling program should analyze the code to determine the success or failure of the procedure so it can, if necessary, change the flow of execution.

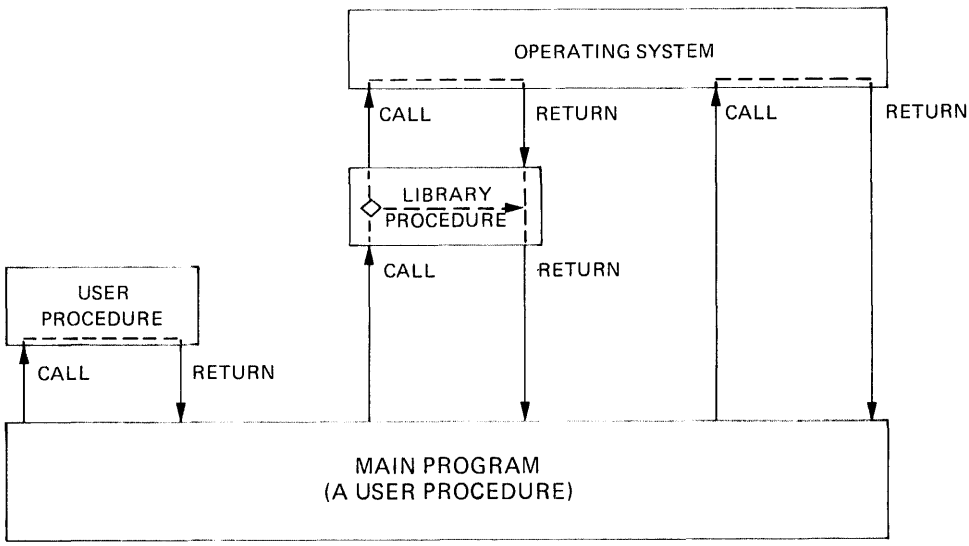
2.1 How to Call Library Procedures

A process is created when you log in and exists until you log out. Each time you run a program, VAX/VMS activates an executable image in your process that contains the program to be executed. The program consists of user procedures, one of which is the main program. The term "main program" or "main procedure" refers to the first user program or procedure called after image initialization. However, before the main program or main procedure is called, VAX/VMS calls a number of initialization procedures. (See Appendix E for more information on initialization procedures.)

Figure 2–1 shows the calling relationship among a main program, user procedures, library procedures, and VAX/VMS. In this figure, "CALL" indicates a request for information or for some action; "RETURN" indicates that the information requested was returned to the caller, or that the action requested was performed.

User procedures can call both other user procedures and library procedures. From the point of view of the library, user procedures are procedures outside the library that can call the library. User procedures can be DIGITAL supplied, such as a compiler or a utility, or they can be customer written. The term "user program" refers to all of one user's procedures, including the main program.

Figure 2-1: Calling the Run-Time Library



Library procedures can call other library procedures or VAX/VMS; however, they cannot call user procedures except in the following instances:

- When initialization is required before the main program gets control (see Appendix E)
- When users establish their own condition handlers (see Chapter 6)
- When a user procedure passes the address of a procedure as a parameter to the library to be called later by the library

2.2 Call Summary

Each procedure requires a specific calling sequence, as shown in the format section of each procedure description in Chapters 3 through 8. A calling sequence takes the general form of:

- Call type
- Library facility prefix
- Procedure name
- Parameter list

- The MACRO calling sequences are:

CALLS	#n,fac$procedure-name
CALLG	parameter-list, fac$procedure-name
JSB	fac$procedure-name

  Section 2.7 provides a complete explanation of how to code calls to library procedures using MACRO. Some examples of MACRO calls are:

```
CALLS       #2,G^LIB$GET_INPUT
CALLG       ARGLST,G^LIB$GET_VM
JSB         MTH$SIN_R4
```

- The FORTRAN calling sequences are:

CALL statement	fac$procedure-name	(parameter-list)
function reference	fac$procedure-name	(parameter-list)

  Section 2.11 provides a complete explanation of how to code calls to library procedures using FORTRAN. Some examples of FORTRAN calls are:

```
CALL LIB$MOVTC (SRC, FILL, TABLE, DEST)
STATUS = LIB$GET_INPUT (STRING, 'NAME:')
```

As these calling sequences and examples show, the call forms vary from language to language. For example, MACRO does not distinguish between functions and subroutines in its CALLS and CALLG instructions, and higher-level languages provide no explicit JSB call form. In addition, some procedures provide both call (CALLS/CALLG) and JSB entry points.

Each procedure is identified by a unique entry point name, consisting of the library facility prefix (LIB$, MTH$, etc.) plus the procedure name, (for example, MTH$SIN). Section 2.3 provides more detailed information on library naming conventions.

Parameters passed to a procedure must be coded in the order shown in the descriptions in Chapters 3 through 8. Each parameter has four characteristics: access type, data type, passing mechanism, and parameter form (see Appendix A).

The access types include:

- Function call (before return)
- JMP (after unwind) access
- Modify (Read and Write) access
- Read-only access
- Write-only access

The data types include:

- Absolute virtual address
- Bit (variable bit field)

- Byte integer (signed)
- Byte logical (unsigned)
- F_floating complex
- D_floating complex
- G_floating complex
- Data type in descriptor
- F_floating
- D_floating
- G_floating
- H_floating
- Longword condition value
- Longword integer (signed)
- Longword logical (unsigned)
- Quadword integer (signed)
- Text (character) string
- Word integer (signed)
- Word logical (unsigned)

The passing mechanisms include:

- By descriptor
- By reference
- By immediate value

The parameter forms include:

- Array reference or descriptor
- Dynamic string descriptor
- Fixed-length string descriptor
- Procedure reference or descriptor
- Scalar
- String form specified in descriptor

The procedure descriptions in Chapters 3 through 8 provide specific information on parameter characteristics, while Section 2.4 provides general information on the same topic. Section 2.5 describes valid combinations of passing mechanisms and data forms.

The caller of a library procedure can omit optional parameters at the end of the parameter list by passing a shortened list. (This differs from a call to VAX/VMS System Services.) Thus, the format for a library procedure with two optional parameters would be:

```
CALL fac$name (parameter1 [,parameter2 [,parameter3]])
```

The following calls could be made to this procedure in FORTRAN:

```
CALL fac$name (A,B,C)
CALL fac$name (A,B)
CALL fac$name (A,B,)
CALL fac$name (A,,C)
CALL fac$name (A)
CALL fac$name (A,)
CALL fac$name (A,,)
```

NOTE

Optional parameters apply only to the CALL entry points. JSB entry points do not have optional parameters; all specified registers are used.

2.3 Library Naming Conventions

This section explains the naming conventions that the Run-Time Library follows for its entry point names, return status codes, and condition value symbols.

2.3.1 Entry Point Names

The Run-Time Library entry point naming conventions follow the VAX-11 global symbol naming conventions. A global symbol takes the general form:

fac$symbol

where:

fac — is a two- or three-character facility name.

symbol — is a one- to eleven-character symbol.

The facility names are maintained in a system-wide, DIGITAL registry. A unique, 12-bit facility number is assigned to each facility name for use in: (1) condition value symbols, and (2) condition values in procedure return status codes, signaled conditions, and messages. All library entry point names begin with a facility prefix. The high order bit of the 12-bit facility number is zero for facilities assigned by DIGITAL and one for those assigned by Computer Special Services (CSS) and customers.

The library facility prefixes are:

Facility Name	Facility Number	Facility
LIB$	21	General utility procedures–for use with all languages including MACRO
MTH$	22	Mathematics procedures
OTS$	23	Language-independent support procedures
FOR$	24	FORTRAN-specific support procedures
COB$	25	COBOL-specific support procedures
BAS$	26	BASIC-specific support procedures
PAS$	33	PASCAL-specific support procedures
STR$	36	String procedures

2.3.2 JSB Entry Point Names

JSB entry point names follow the standard entry point naming conventions except that they include the number of the highest register accessed or modified. This helps ensure agreement between the caller and the called procedure about the number of registers that the called procedure is going to change (see Section 2.7.1.3). For example:

```
JSB MTH$SIN_R4          ; F_floating sine uses R0 to R4
```

NOTE

JSB entry points are available only to MACRO and BLISS programs, not high-level languages.

2.3.3 Library Return Status and Condition Value Symbols

Library return status and condition value symbols have the general form:

fac$__abcmnoxyz

where:

- fac is the three-letter facility symbol (LIB, MTH, STR, OTS, BAS, FOR, PAS).
- abc are the first three letters of the first word of the associated message.
- mno are the first three letters of the next word.
- xyz are the first three letters of the third word, if any.

Note that articles and prepositions are not considered significant words in this format. If a significant word is only two letters long, an underscore character is used to fill out the third space. The VAX/VMS normal or success code is used to indicate successful completion. Some examples follow:

LIB$_INSVIRMEM	Insufficient virtual memory
FOR$_NO_SUCDEV	No such device
SS$_NORMAL	Routine successfully completed
MTH$_FLOOVEMAT	Floating overflow in Math Library procedure
BAS$_SUBOUTRAN	Subscript out of range

2.4 Procedure Parameter Characteristics

The Run-Time Library lets you pass parameters of various types and forms to its procedures. However, some procedures accept certain types of parameters.

Each parameter has the following characteristics:

- Access type (read, write, modify ...)
- Data type (floating, longword ...)
- Passing mechanism (by immediate value, by reference, by descriptor)
- Parameter form (scalar, array, string ...)

The calling program must ensure that parameters passed to a called procedure are of the type and form that the procedure accepts. For your convenience, Appendix A uses an abbreviated notation to indicate these characteristics. The following sections describe the four parameter characteristics.

2.4.1 Parameter Access Types

The following parameter access types are available:

- Read-only access — parameter is read but not written; at the calling program's option, the parameter can be in read-only storage.
- Write-only access — parameter is written without regard to its original value.
- Modify access — parameter can be modified, that is, both read and written.
- Function call — parameter is an address of a function to be (optionally) called before the procedure returns to its caller.
- JMP access — parameter is an address to be (optionally) jumped to after stack is unwound to the frame of the calling program; no data type field is given because the parameter is a sequence of instructions (for example, in FORTRAN, ERR=).

2.4.2 Parameter Data Types

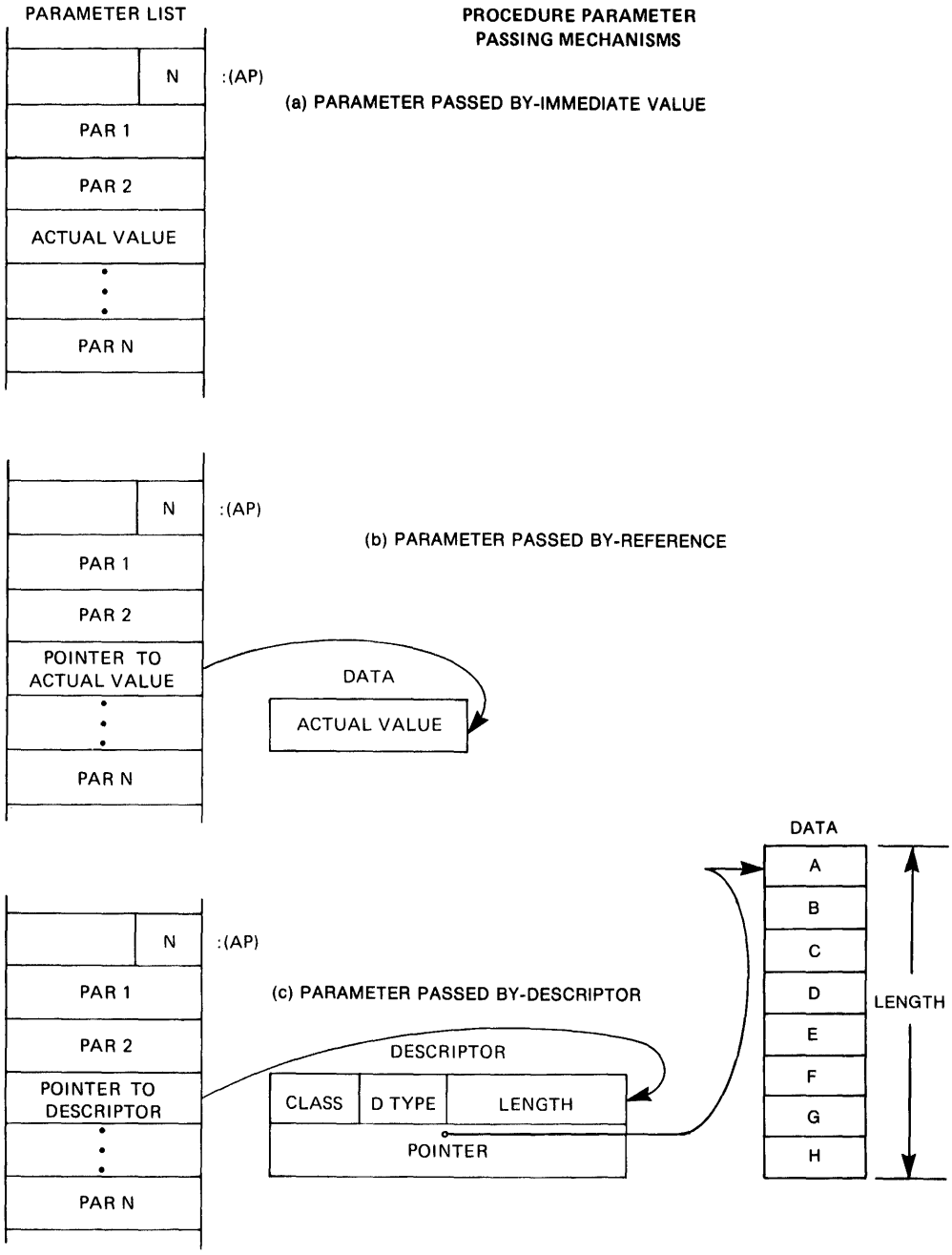
The procedure descriptions in Chapters 3 through 8 indicate the expected data types for each parameter. The following parameter data types are used by the Run-Time Library:

- Byte integer (8-bit signed 2's complement integer)
- Byte logical (8-bit unsigned quantity)
- Word integer (16-bit signed 2's complement integer)
- Word logical (16-bit unsigned quantity)
- Longword integer (32-bit signed 2's complement integer)
- Longword logical (32-bit unsigned quantity)
- Longword condition value
- Absolute 32-bit virtual address
- Quadword integer (64-bit signed 2's complement integer)
- Quadword logical (64-bit unsigned quantity)
- Octaword integer (128-bit signed 2's complement integer)
- Octaword logical (128-bit unsigned quantity)
- F_floating (32-bit F_floating quantity)
- D_floating (64-bit D_floating quantity)
- G_floating (64-bit G_floating quantity)
- H_floating (128-bit H_floating quantity)
- F_floating complex (ordered pair of F_floating quantities representing a single-precision complex number — the lower (first) addressed quantity represents the real part, the higher (second) addressed quantity represents the imaginary part)
- D_floating complex (ordered pair of D_floating quantities representing a double-precision complex number — the lower (first) addressed quantity represents the real part, the higher (second) addressed quantity represents the imaginary part)
- G_floating complex (ordered pair of G_floating quantities representing a double-precision quantities representing a double-precision complex number — the lower (first) addressed quantity represents the real part, the higher (second) addressed quantity represents the imaginary part)
- ASCII text string (a sequence of 8-bit ASCII characters)
- Procedure entry mask

2.4.3 Parameter Passing Mechanisms

Each procedure has a parameter list of 32-bit longwords; each longword specifies a separate parameter. A called procedure interprets each parameter using one of three standard passing mechanisms: by immediate value, by reference, and by descriptor. Figure 2-2 illustrates the three passing mechanisms.

Figure 2-2: Procedure Parameter Passing Mechanisms



Note: PAR 1, PAR 2, PAR N can be passed by-immediate value, by-reference, or by-descriptor in any of the above examples.

2.4.3.1 Passing Parameters by Immediate Value — When parameters are passed using the immediate value mechanism, the parameter list entry contains the actual, uninterpreted 32-bit value of the parameter. Usually, parameters passed by immediate value are constants. For example, to pass 100 by immediate value, the caller puts 100 directly in the parameter list. However, when a variable is passed by immediate value, the variable's value is copied to the parameter list. For example, to pass variable X, the caller must copy the current value of X to the parameter list.

Since higher-level languages normally pass scalar parameters by reference, the %VAL argument list built-in function or equivalent must be used to call procedures that accept parameters by immediate value. For example:

- **BLISS** `LIB$SIGNAL(SS$_INTOVF)`
- **BASIC** `CALL LIB$SIGNAL(SS$_INTOVF BY VALUE)`
- **FORTRAN** `CALL LIB$SIGNAL (%VAL (SS$_INTOVF))`
- **PASCAL** `LIB$SIGNAL(%IMMED(SS$_INTOVF))`

The equivalent **MACRO** code is:

```
PUSHL   #SS$_INTOVF      ; Push longword by immediate value
CALLS   #1,G^LIB$SIGNAL  ; Call LIB$SIGNAL
```

NOTE

The Run-Time Library is intended to be called from higher-level languages, so most library procedures do not use the immediate value mechanism.

2.4.3.2 Passing Parameters by Reference — When parameters are passed using the reference mechanism, the parameter list entry contains the address of (that is, points to) the location that contains the value of the parameter. For example, if variable X is allocated to location 1000, which currently contains the value 100, the parameter list entry will contain 1000.

The following high-level language statements pass a parameter to LIB$FLT_UNDER by reference:

- **BLISS** `LIB$FLT_UNDER(%REF(1))`
- **BASIC** `CALL LIB$FLT_UNDER(1%)`
- **FORTRAN** `CALL LIB$FLT_UNDER(1)`
- **PASCAL** `LIB$FLT_UNDER(1)`

The equivalent **MACRO** code is:

```
ONE:    .LONG   1                       ; Address of longword
           .
           .
           .
        PUSHAL  ONE                     ; Push address of longword
        CALLS   #1,G^LIB$FLT_UNDER      ; Call LIB$FLT_UNDER
```

2.4.3.3 Passing Parameters by Descriptor — When parameters are passed using the descriptor mechanism, the parameter list entry contains the address of a VAX-11 descriptor of the parameter. This form is used to pass more complicated data than can be passed using the preceding forms. All descriptors include the following fields to describe data:

DSC$W_LENGTH	data length in bytes
DSC$B_DTYPE	data type
DSC$B_CLASS	descriptor class field
DSC$A_POINTER	address of start of data

Appendix C describes these fields in greater detail.

The following high-level language statements pass a parameter by descriptor:

- **BASIC** `CALL LIB$PUT_OUTPUT('HELLO')`
- **FORTRAN** `CALL LIB$PUT_OUTPUT('HELLO')`
- **PASCAL** `LIB$PUT_OUTPUT(%STDESCR('HELLO')`

The equivalent **MACRO** code is:

```
MSGDSC: .WORD LEN               ; DESCRIPTOR: DSC$W_LENGTH
        .BYTE 14                ; DSC$B_DTYPE
        .BYTE 1                 ; DSC$B_CLASS
        .ADDRESS MSG            ; DSC$A_POINTER

MSG:    .ASCII/Hello/           ; String itself
LEN = .-MSG                     ; Define the length of the string

        PUSHAQ MSGDSC           ; Push address of descriptor
        CALLS #1,G^LIB$PUT_OUTPUT  ; Call procedure
```

2.4.4 Parameter Data Forms

Possible data forms for Run-Time Library parameters are:

- Scalars (numbers) - a numeric representation of a value
- Arrays - a one or more dimensional arrangement of data
- Dynamic strings - a string whose length and address can be changed when the string is written

- Fixed-length strings – a string whose length and address does not change when the string is written
- Procedure references or descriptors – a descriptor or reference to a procedure to be passed as a parameter

2.5 Combinations of Data Forms/Passing Mechanisms

Each library facility uses a subset of parameter qualifiers permitted by the VAX-11 Procedure Calling Standard. Table 2-2 (in Section 2.5.4) summarizes the subset of combinations of data forms and passing mechanisms that each library facility accepts. Section 2.5.1 discusses scalars, Section 2.5.2 discusses arrays, and Section 2.5.3 discusses strings.

2.5.1 Passing Scalars as Parameters

Input scalar parameters are passed by reference to general utility procedures (LIB$) and mathematics procedures (MTH$); these procedures are most likely to be called explicitly from a high-level language program. Input scalar parameters are passed by immediate value to language-support procedures (OTS$, BAS$, COB$, FOR$, and PAS$); these procedures are most likely to be called implicitly from code generated by a language compiler.

Output scalar parameters are always passed by reference to Run-Time Library procedures.

2.5.2 Passing Arrays as Parameters

Arrays are passed by reference or by descriptor to Run-Time Library procedures depending on the facility.

2.5.3 Passing Strings as Parameters

Strings are always passed by descriptor to Run-Time Library procedures. The three classes of strings supported by the Run-Time Library are: unspecified, fixed length, and dynamic. The descriptor format is the same for all three string types, except for the class code field. The descriptor and the class code field (bits 31:24) are one of the following:

String Class	Symbol	Value
Unspecified	DSC$K_CLASS_Z	0
Fixed length	DSC$K_CLASS_S	1
Dynamic	DSC$K_CLASS_D	2

Fixed-length strings are allocated at compile, link, or run time by the calling program. The called procedure cannot change the length or address of the

string. This means that the descriptor for a fixed-length string can be in read-only memory. Fixed-length strings can be more efficient (as long as you avoid excessive space filling), but they require you to specify the length of each string in your program. FORTRAN and PASCAL support fixed-length strings only.

Dynamic strings are allocated at run time using library resource allocation procedures. Therefore, both the length and the address change during execution and no space filling is needed. Dynamic strings are usually more convenient, since you do not need to specify their length in your program. However, the dynamic allocation usually takes more execution time. BASIC supports both fixed-length and dynamic strings.

2.5.3.1 Passing Input Parameter Strings to the Library — The parameter list entry for an input string is the address of a two longword descriptor. The descriptor can be any of the three classes of string descriptor, since their formats are identical, except for the class code field. The called procedure uses the length (DSC$W_LENGTH) and address (DSC$A_POINTER) of the string, as specified in the descriptor. When an input string is compared with another string for each class of descriptor, the shorter string is extended with the ASCII space character (hexadecimal 20) as the fill character.

2.5.3.2 Returning Output Parameter Strings from the Library — Library procedures do not return strings as they do other function values. Instead, the parameter to accept the string function value is passed as the first parameter, and other parameters are shifted to the right by one position. For example:

char-string = LIB$func (a, b, c)

is equivalent to:

CALL LIB$func (char-string, a, b, c)

In addition, the caller must allocate the space for and fill in the fields of the output string descriptor at compile, link, or run time.

In languages that support the concept of a string function (such as BASIC and FORTRAN), the following two examples are equivalent, although the first more clearly illustrates the function concept:

BASIC	FORTRAN
`DECLARE STRING STR`	`CHARACTER*10 STR`
`STR = LIB$func(A,B,C)`	`STR = LIB$func(A,B,C)`
`DECLARE STRING STR`	`CHARACTER*10 STR`
`CALL LIB$func(STR,A,B,C)`	`CALL LIB$func(STR,A,B,C)`

In languages that do not support the concept of a string function (such as MACRO, BLISS and PASCAL), a procedure that returns strings must be called using an explicit CALL statement. In the following example, a descriptor address for each parameter is pushed onto the stack and a CALLS call is

made. Note that the actual descriptors for each parameter would appear elsewhere in the program and would resemble the form shown in the MACRO example in Section 2.4.3.3.

```
PUSHAQ          C_DESCR          ; Push descr address of C
PUSHAQ          B_DESCR          ; Push descr address of B
PUSHAQ          A_DESCR          ; Push descr address of A
PUSHAQ          CHAR_STR_DESCR   ; Push descr address of char-str
CALLS           #4, LIB$func     ; Call LIB$func
```

Procedures can use other parameters to return additional strings passed by descriptor. Run-Time Library procedures return strings using the following methods. The FORTRAN specific (FOR$) procedures assume that the caller passes a fixed length string descriptor, and thus use only the first method. General utility procedures (LIB$) and language independent support procedures (OTS$) examine the class field code of the descriptor (DSC$K_CLASS) passed by the caller and return the string using either of these methods:

1. Returning fixed-length or unspecified strings (DSCK_CLASS_S, DSCK_CLASS_Z). The contents of the parameter list entry is the address of the two-longword descriptor with a class field of zero or one. In the descriptor, the calling program specifies the length (DSC$W_LENGTH) and address (DSC$A_POINTER) of the area where the string is to be written. The called procedure copies the string to the indicated area and, if necessary, trailing ASCII space characters (hexadecimal 20) are used to fill out the string. If insufficient space is available, one of the following events occurs, depending on the procedure:

   a. The string is truncated on the right; there is no error indication (normal BASIC and FORTRAN technique).

   b. The string is truncated on the right and a success or error condition value is returned (STR$ facility).

   c. The string is set to asterisks and an error condition is returned (FORTRAN error technique).

2. Returning dynamic strings (DSC$K_CLASS_D).

   The parameter list entry contains the address of the two longword descriptor. In the descriptor, the caller can optionally specify the address of a previously allocated dynamic string area in the DSC$A_POINTER field. The two bytes immediately preceding the first byte of the string area contain the number of bytes allocated to the area; that is, the number of bytes following the first byte. If the string to be returned fits in the area already allocated (specified by the word preceding the string itself), the new string is copied to the old area and the length field (DSC$W_LENGTH) is changed in the descriptor.

   If the string to be returned does not fit in the space allocated, the space is returned to free storage and a new block is allocated. If the address of the

area (DSC$A_POINTER) is 0, no space is returned and a new block is allocated. Both the length (DSC$W_LENGTH) and address fields (DSC$A_POINTER) are modified in the descriptor, and the string is copied to the newly allocated area.

Note that DSC$A_POINTER is set to the address of the first byte of the string, and the allocated length is stored in the precedingtwo bytes. Thus, a dynamic string appears the same as any other string when passed as an input parameter.

User programs that allocate dynamic strings should always use the string resource allocation procedures provided by the VAX-11 Run-Time Library rather than attempt to control dynamic string area descriptors directly. This is because the arrangement and size of control information that affects a dynamic string is subject to change with new releases of the Run-Time Library.

Dynamic strings are the usual string form in BASIC. Dynamic strings are not generally available to FORTRAN and PASCAL programmers. However, a calling program can pass a dynamic string to a FORTRAN program. The FORTRAN procedure makes a copy of the descriptor setting the class field to DSC$K_CLASS_S. If the string is an input parameter, the results are the same. If the string is an output parameter, the FORTRAN procedure call uses the current length of the string, space filling if necessary. If the string is too long, it is truncated. When a dynamic string is passed as an output parameter, the caller must ensure that the string is of sufficient length before calling any procedure that expects a fixed-length string.

Procedures which return a string as an output parameter where there is no way for the caller to know the length of the returned string should have an optional output length parameter. This parameter should be an unsigned, 16-bit integer to contain the output string length. If the output string is a fixed-length string, the optional length parameter would reflect the number of characters written not counting the fill characters, if any.

For example, LIB$GET_INPUT has the optional parameter, out-len (see Chapter 3). If LIB$GET_INPUT were called with a fixed-length, five character string and a record containing 'ABC' were read, then out-len would have a value of three and the output string would be 'ABC '. But, if the record read contained the value 'ABCDEFG', out-len would have a value of five, and the output string would be 'ABCDE'.

STR$COPY_DX does not need the optional length parameter, because the output string length is known by the caller. 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 minimum of the two lengths before the transfer.

2.5.3.3 Summary of String Passing Techniques — Table 2-1 shows the string passing techniques used by library facilities in the Run-Time Library.

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

String Type	String Descriptor Fields			
	Class	**Length**	**Pointer**	**Facility**
Input Parameter to Procedures				
Input String Passed by Descriptor	Ignored	Read	Read	LIB,OTS STR,lan*
Output Parameter from Procedures (class assumed by called procedure)				
Output String Passed by Descriptor (fixed-length)	Ignored	Read	Read	lan
Output String Passed by Descriptor (dynamic)	Ignored	Always Written	Can be Written	LIB,OTS STR
Output Parameter from Procedures (class specified by calling program)				
Output String (unspecified) (DSC$K_CLASS_Z)	Read	Read	Read	LIB,OTS STR
Output String (fixed-length) (DSC$K_CLASS_S)	Read	Read	Read	LIB,OTS STR
Output String (dynamic) (DSC$K_CLASS_D)	Read	Always Written	Can Be Written	LIB,OTS STR
*where *lan* is a language-specific facility.				

2.5.4 Summary of Parameter Passing Mechanisms

Table 2–2 summarizes parameter passing mechanisms that can be used with each data form for each library facility.

Table 2-2: Valid Run-Time Library Parameter Passing Mechanisms

Data forms	By Immediate Value	By Reference	By Descriptor
Scalars	OTS,lan*	LIB,MTH	-
Input Output	-	OTS,lan,LIB	-
Arrays	-	OTS,lan,LIB	lan
Input	-	OTS,lan,LIB	lan
Output			
Strings	-	-	LIB,lan,OTS
Input			
Output	-	-	LIB,lan,OTS
Fixed length	-	-	LIB,OTS,STR
Dynamic			

*where *lan* is a language-specific facility.

Any deviations from the information in Table 2-2 are documented parenthetically in the parameter descriptions.

2.6 Errors From Run-Time Library Procedures

A procedure can indicate errors to its caller by either returning a condition value as a completion code or signaling the error. When the completion code is returned as a value in R0, the caller can test R0 and choose a recovery path. When the completion code is signaled, the caller must establish a handler to get control before taking action. (See Chapter 6 for a description of signaling and condition handling.)

Each facility has a convention for returning errors to its callers:

LIB — Always communicates errors by a condition value.

MTH
OTS — Indicates errors by signaling.

STR — Returns errors in both forms. Severe errors, those judged to be programming errors, or conditions which prevent the procedure from doing any useful work are signaled. Errors that can be corrected using default values or those judged to be not serious are returned as a status code.

2.7 Calling a Library Procedure in MACRO

This section describes how to code MACRO calls to library procedures using a CALLS, CALLG, or JSB instruction. Procedures have either CALL or JSB entry points or both. Procedure descriptions in Chapters 3 through 8 give the entry points for each procedure. You can use either a CALLS or a CALLG instruction to invoke a procedure with a CALL entry point; you must use a JSB procedure to invoke a procedure with a JSB entry point. All MACRO calls are explicit.

2.7.1 Calling Sequence Examples

CALLS and CALLG are hardware instructions.

Parameters are passed to CALLS and CALLG entry points by a pointer to the parameter list. The only difference between CALLS and CALLG instructions is:

- For CALLS, the caller pushes the parameter list onto the stack (in reverse order) before performing the call. (The list is automatically removed from the stack upon return.)
- For CALLG, the caller specifies the address of the parameter list, which can be anywhere in memory. The list remains in memory upon return.

The effect of either call instruction on the called procedure is identical.

Either a CALLS or CALLG instruction specifies the address of the entry point of the procedure being called. The entry point consists of an entry mask, followed by the instructions to implement the procedure. An entry mask is a 16-bit word whose bits represent the registers to be saved on a procedure call that uses the CALLS or CALLG instructions; these registers are subsequently restored by a corresponding RET (return) instruction.

The called procedure must specify in its entry mask any of the registers, R2 through R11, that are written or modified. This ensures that the contents of R2 through R11 are preserved from the point of view of the calling program. The CALLS, CALLG, and the RET instructions automatically save and restore registers R12 (the argument pointer, AP), R13 (the frame pointer, FP), and R14 (the stack pointer, SP). Registers R0 and R1 are temporary registers, will not be preserved, and should not be specified in the entry mask.

Both CALLS and CALLG instructions also save the state of the caller's trap enables, that is, integer overflow, decimal overflow, and floating underflow. They then set them as indicated by the entry mask, thus isolating the called procedure from the calling program.

Appendix F contains detailed information about the operation of CALLS and CALLG instructions and the VAX-11 procedure stack architecture. This information is particularly pertinent to user control of signaling and condition handling.

2.7.1.1 CALLS Instruction Example — The following example shows how the procedure that allocates virtual memory in the program region (LIB$GET_VM) could be called from a MACRO program. The format of the LIB$GET_VM procedure is described in Section 5.1.5.

A call to LIB$GET_VM using a CALLS instruction in MACRO is:

```
PUSHAL      START              ; Push address of longword to receive
                                 address of block
PUSHAL      LEN                ; Push address of longword containing
                                 number of bytes desired
CALLS       #2, G^LIB$GET_VM   ; Allocate memory
BLBC        R0, error          ; Branch if memory not available
```

Upon return from LIB$GET_VM, the calling program branches to an appropriate error routine if any errors occurred. (Note that because the stack grows toward location 0 (that is, the top of the stack), parameters are pushed onto the stack in reverse order from the order shown in the procedure formats.)

2.7.1.2 CALLG Instruction Example — The following example of a CALLG instruction is equivalent to the preceding CALLS example:

```
ARGLST: .LONG 2                ; Argument list count
        .ADDRESS LEN           ; Address of longword containing
                                 number of bytes desired
        .ADDRESS START         ; Address of longword to receive
                                 address of block
        .
        .
        .
        CALLG ARGLST, G^LIB$GET_VM
```

2.7.1.3 JSB Entry Points — JSB instructions execute faster than CALL instructions. They do not set up a new stack frame, do not change the hardware trap enables, and do not preserve the contents of registers R0 through Rn before modifying them. The value of Rn is always indicated at the end of the procedure's JSB entry point name. Parameters are passed to JSB entry points in registers.

A calling program must use a JSB instruction to call a procedure in the library at its JSB entry point. For example:

```
MOVF        ...., R0           ; Set up input parameter
JSB         MTH$SIN_R4         ; Call F_floating sine procedure
                               ; Return with value in R0
```

In this example, MTH$SIN_R4 changes the contents of registers R0 through R4, as indicated by "R4" in the entry point name (see Section 2.3.2). The routine does not change the contents of or reference registers R5 through R11.

Since the JSB entry point routines do not save the contents of any registers, the calling program is responsible for saving the contents of registers R2 through Rn. This is done by specifying the entry mask bits for at least R2 through Rn in its own entry mask, so a stack unwind correctly restores all registers from the stack. In the following example, the function Y=PROC(A,B) returns the value Y, where Y=SIN(A)*SIN(B). Registers R2 through R5 are saved when procedure PROC is called with a CALLS or CALLG instruction:

```
.ENTRY      PROC, ^M <R2, R3, R4, R5>  ; Save R2:R5
MOVF        @4(AP),R0                  ; R0 = A
JSB         MTH$SIN_R4                 ; R0 = SIN(A)
MOVF        R0, R5                     ; Copy result to register
                                       ; not modified by MTH$SIN
MOVF        @8(AP), R0                 ; R0 = B
JSB         MTH$SIN_R4                 ; R0 = SIN(B)
MULF        R5, R0                     ; R0 = SIN(A)*SIN(B)
RET                                    ; Return
```

If DIGITAL should provide JSB replacement routines that change R0 through Rm, where m is greater than n, both the old and the new routines will be maintained indefinitely with separate entry points. This means that old programs will not need to be relinked when new versions of the Run-Time Library are released (for example, see MTH$SQRT, Chapter 4).

2.7.2 Passing Parameters to Library Procedures

In many cases, you have to tell a library procedure where to find input values and store output values. You must select a data type for each parameter when you code your program. Most procedures accept and return 32-bit parameters.

For input parameters of byte, word or longword values, you can supply either a constant value, a variable name, or an expression in the library procedure call. If you supply a variable name for the parameter, the variable data type must be as large as or larger than the data type required. If, for example, the called procedure expects a byte in the range 0 to 100, you can use a variable data type of a byte, word, or longword with a value between 0 and 100.

For each output parameter, you must declare a variable of exactly the length required to avoid extraneous data. If, for example, the called procedure returns a byte value to a word-length variable, the left-most eight bits of the variable (15:8) are not overwritten on output. Conversely, if a procedure returns a longword value to a word-length variable, it modifies variables in adjacent locations.

2.7.3 Return Status

Some procedures return a 32-bit status code in register R0. A return status code is either a success (bit 0=1) or error condition value (bit 0=0). In an error condition value, the low-order 3 bits specify the severity of the error. Bits 27 through 16 contain the facility number, and bits 15 through 3 indicate the particular condition. The high-order 4 bits are control bits. (See Appendix C.)

To test for errors, check for a 0 in bit 0. This is done with a Branch on Low Bit Set (BLBS) or Branch on Low Bit Clear (BLBC) instruction.

To test for a particular condition value, perform a 32-bit comparison of the return status with the appropriate return status symbol. You do this with a compare long (CMPL) instruction.

There are three ways to define a symbol for a condition value returned by a library procedure:

- By default. The assembler automatically declares the condition value as an external symbol that is defined as a global symbol in the Run-Time Library.
- Using the .EXTRN LIB$__INPSTRTRU instruction. This causes the assembler to generate an external symbol declaration.
- Using the $LIBDEF instruction. This causes the assembler to define all LIB$ condition values using the default macro library.

The following example asks for the user's name. If the name is longer than 30 characters (the space allocated to receive the name), the error LIB$__INPSTRTRU - 'input string truncated' is usually returned. This example checks for that specific error while treating any other error in the usual manner.

```
PROMPT:   .WORD     6,0                     ; Length, class/type
          .ADDRESS  PRO_ADR                 ; Address

PRO_ADR:  .ASCII    /Name: /                ; String descriptor
                                            ; to receive string

STRING:   .WORD     30,0                    ; Length, class/type
          .ADDRESS  STR_ADR                 ; Address

STR_ADR:  .BLKB     30                      ; Area to receive string

          PUSHAQ    PROMPT                  ; Push adr of prompt
                                            ; descriptor
          PUSHAQ    STRING                  ; Push address of string
                                            ; descriptor
          CALLS     #2, G^LIB$GET_INPUT     ; Get input string
          BLBS      R0, 10$                 ; Check for success
          CMPL      R0, #LIB$_INPSTRTRU     ; Error, was it
                                            ; truncated string?
          BEQL      10$

          error                             ; No, more serious error

10$:      success                           ; Success, or name too
                                            ; long
```

2.7.4 Function Return Values

Function values are always 32-bit values returned in register R0, or 64-bit values returned in registers R0/R1.

2.8 Calling a Library Procedure in BLISS

This section describes how to code BLISS calls to library procedures. A called procedure can return one of the following:

- No value.
- A function value (typically, an integer or floating-point number). For example, MTH$SIN returns an F_floating value.
- A return status (typically, a 32-bit condition value) indicating that the procedure has either successfully executed or failed. For example, LIB$GET_INPUT returns a return status.

2.8.1 Calling Sequence Example

The following example shows how to call the procedure that outputs a record to the user's terminal (LIB$PUT_OUTPUT) from a BLISS program.

```
MODULE SHOWTIME(IDENT='1-1' %TITLE'Print time', MAIN=TIMEOUT)=
BEGIN
LIBRARY 'SYS$LIBRARY:STARLET';  ! Defines System Services,etc.

MACRO
    DESC[]=%CHARCOUNT(%REMAINING), ! VAX-11 String Descriptor
        UPLIT BYTE(%REMAINING) %;
! definition
OWN
    TIMEBUF: VECTOR[2],           ! 64-bit system time
    MSGBUF: VECTOR[80,BYTE],      ! Output message buffer
    MSGDESC: VECTOR[2] INITIAL( 80,MSGBUF );
BIND
    FMTDESC=UPLIT( DESC('At the tone, the time will be',
            %CHAR(7), '!%T' ));
EXTERNAL ROUTINE
    LIB$PUT_OUTPUT: ADDRESSING_MODE(GENERAL);

ROUTINE TIMEOUT=
    BEGIN
    LOCAL
            RSLT: WORD;           ! Resultant string length

    $GETTIM( TIMADR=TIMEBUF );    ! Get time as 64-bit integer

    $FAOL( CTRSTR=FMTDESC,        ! Format Descriptor
            OUTLEN=RSLT,          ! Output length (only a word!)
            OUTBUF=MSGDESC,       ! Output buffer descriptor
            PRMLST= %REF(TIMEBUF));! Adr of 64-bit time block
    MSGDESC[0] = ,RSLT;           ! Modify output descriptor
    LIB$PUT_OUTPUT( MSGDESC )     ! Return status
    END;
END
ELUDOM
```

2.8.2 Passing Parameters to Library Procedures

Generally, Run-Time Library parameters are passed by reference. Thus, when passing a variable, it appears "un-dotted" in the procedure-call parameter list. A constant value can be easily passed using the %REF built-in function.

For example to pass the address of a text buffer (MYBUF) and its length (80 characters):

```
OWN
      MYBUF: VECTOR[80,BYTE];
      .
      .
      .
      LIB$... (MYBUF, %REF(80))
```

2.8.3 Return Status

The return status can be treated as any other BLISS value.

2.8.4 Function Return Values

Function values are always 32-bit values returned in register R0, or 64-bit values returned in registers R0/R1.

2.8.5 Calling JSB Entry Points from BLISS

Many of the Math Library routines have JSB linkage entry points. These routines can be efficiently invoked directly from BLISS using LINKAGE and EXTERNAL ROUTINE declarations.

For example:

```
LINKAGE
      MATH_R4 = JSB(REGISTER=0, ... ):NOPRESERVE(0,1,2,3,4)
      .
      .
      .
EXTERNAL ROUTINE
      MTH$SIN_R4 : MATH_R4;
      .
      .
      .
      IF MTH$SIN( ... ) EQL %E'0.0' THEN
```

2.9 Calling a Library Procedure in BASIC

This section describes how to code BASIC calls to library procedures using either a CALL statement or function reference. CALL statements invoke subroutines that do not return meaningful values. Function references, on the other hand, return one of the following:

- A function value (typically, an integer or floating point number). For example, MTH$COS returns an F_floating value.
- A return status (typically, a 32-bit condition value) indicating that the procedure has either successfully executed or failed. For example, LIB$GET_INPUT returns a return status.

You can invoke a subroutine as if it were a function; this normally returns a meaningless value. You can also invoke a function as if it were a subroutine if

you are not interested in the function value or return status. However, it is good programming practice to always check a return status for success or failure.

2.9.1 Calling Sequence Examples

The following example shows how to call the procedure that inserts a variable bit field (LIB$INSV) from a BASIC program. The format of the LIB$INSV procedure is explained in Section 3.4.1.

To set the low order three bits of RET_STATUS to four, you would code the following:

```
DECLARE INTEGER RET_STATUS
CALL LIB$INSV (4%, 0%, 3%, RET_STATUS)
```

The following example shows how to call the procedure that enables and disables detection of floating underflow (LIB$FLT_UNDER) from a BASIC program. The format of the LIB$FLT_UNDER procedure is explained in Section 6.5.2.

This procedure could be called in a BASIC program to set floating underflow as follows:

```
EXTERNAL INTEGER FUNCTION LIB$FLT_UNDER
DECLARE INTEGER OLD_SET
OLD_SET = LIB$FLT_UNDER (1%)
```

If the old setting is of no interest, you can ignore it by treating the function LIB$FLT_UNDER as a subroutine:

```
CALL LIB$FLT_UNDER (1%)
```

The following example shows how to call the procedure that finds the first clear bit in a given bit field (LIB$FFC). This procedure returns a 32-bit condition value, represented in the example as COND_VALUE:

```
EXTERNAL INTEGER FUNCTION LIB$FFC
DECLARE INTEGER COND_VALUE, BITS, POS
COND_VALUE = LIB$FFC (0%, 32%, BITS, POS)
IF (COND_VALUE AND 1%) = 0% THEN GO TO error
```

You can also test the success or failure of a function returning a return status directly by using an IF statement:

```
DECLARE INTEGER BITS, POS
IF (LIB$FFC (0%, 32%, BITS, POS) AND 1%) = 0% THEN GO TO error
```

2.9.2 Passing Parameters to Library Procedures

By default, BASIC uses the call by reference or call by descriptor mechanism for passing parameters, depending on the argument's data type. In some

cases, however (a function reference or call to a non-BASIC procedure, for example), a library procedure can require you to supply arguments in a different form. Therefore, BASIC provides three modifiers for passing parameters when you cannot use the BASIC default mechanism. These modifiers are:

- BY VALUE
- BY REF
- BY DESC

They can appear only in actual argument lists.

The following sections describe the use of these modifiers. Note that they are never used to call a procedure written in BASIC.

2.9.2.1 BY VALUE — This modifier forces the argument list entry to use the call by immediate value mechanism. It has the form:

```
arg BY VALUE
```

The argument list entry (arg) is the value of the entry. Because argument list entries are longwords, the argument value must be a constant (integer, or F_floating), a variable, an array element, or an expression.

2.9.2.2 BY REF — The modifier forces the argument list entry to use the call by reference mechanism. It has the form:

```
arg BY REF
```

The argument list entry (arg) is the address of the value. The argument value can be a numeric or string expression, an array, an array element, or a function name. BY REF is the default BASIC method for passing all numeric values except entire arrays.

2.9.2.3 BY DESC — This modifier forces the argument list entry to use the call by descriptor mechanism. It has the form:

```
arg BY DESC
```

The argument list entry (arg) is the address of a descriptor of the value. The argument value must be an entire array or any string expression. BY DESC is the default BASIC mechanism for passing strings and entire arrays.

For more information, consult the *VAX-11 BASIC User's Guide*.

2.9.3 Return Status

You should always check the return status (when there is one) to make sure that the procedure executed correctly. The return status indicates either success or failure. To test for errors, use an IF statement (see Section 2.9.1).

To test for a particular return condition, perform a 32-bit comparison of the return status with the appropriate return status symbol listed in the procedure descriptions.

For BASIC programs, condition value symbols are available as EXTERNAL CONSTANTS. The user simply declares the appropriate symbolic value and the VAX-11 linker resolves the value.

The following example shows how to call the procedure that accepts input typed by the user from SYS$INPUT. The format of the LIB$GET_INPUT procedure is in Chapter 3.

Note that whenever a procedure description specifies a string descriptor parameter, the parameter being passed should always be a string constant, variable or expression. The BASIC compiler automatically produces descriptors for these parameters.

The following is a BASIC example that asks for the user's name using LIB$GET_INPUT:

```
EXTERNAL INTEGER CONSTANT LIB$_INPSTRTRU
COM STRING USER_LINE = 30
DECLARE INTEGER COND_VALUE
EXTERNAL INTEGER FUNCTION LIB$GET_INPUT

COND_VALUE = LIB$GET_INPUT (USER_LINE, 'Type Your Name: ')
IF COND_VALUE = LIB$_INPSTRTRU THEN
     (user name too long)
ELSE IF (COND_VALUE AND 1%) = 0% THEN
     (more serious error)
```

LIB$GET_INPUT sets the variable USER_LINE to the 30-character string input by the user. The INTEGER condition value (COND_VALUE) indicates success or failure. In BASIC, an even condition value indicates an error and an odd condition value indicates success. The first IF statement tests for the return status that indicates the input string was too long and was truncated. The second IF statement tests for any other errors.

The library procedure LIB$MATCH_COND (see Section 6.10.1) is useful for matching a return status or error condition value with a condition value symbol or any list of condition value symbols.

2.9.4 Function Return Values

The method of returning function procedure values depends on the data type of the value, as summarized in Table 2-3.

Table 2-3: Function Return Values

Data Type	Return Method
Integer F_Floating	General Register R0
D_Floating G_Floating	R0 = High-order part of result R1 = Low-order part of result
String	An extra entry is added as the first entry of the argument list. This new first argument entry points to a character string descriptor. At run time, storage is allocated to contain the value of the result, and the proper address is stored in the descriptor.

2.10 Calling a Library Procedure in COBOL

This section describes how to code COBOL calls to library procedures using either a CALL statement or function reference. CALL statements invoke subroutines that do not return meaningful values. Function references, on the other hand, return one of the following:

- A function value (typically, an integer or floating point number). For example LIB$INDEX returns an integer value.

- A return status which is a 32-bit condition value indicating that the procedure has either successfully executed or failed. For example, LIB$GET_INPUT returns a return status.

You can invoke a subroutine as if it were a function; this normally returns a meaningless value. You can also invoke a function as if it were a subroutine if you are not interested in the function value or return status. However, it is good programming practice to always check a return status for success or failure.

2.10.1 Calling Sequence Examples

The following example shows how to call the procedure that inserts a variable bit field (LIB$INSV) from a COBOL program. The format of the LIB$INSV procedure is explained in Section 3.4.1.

To set the low order three bits of RET-STATUS to four, you would code the following:

```
WORKING-STORAGE SECTION.
01   SRC              PIC S9(9) USAGE IS COMP.
01   POS              PIC S9(9) USAGE IS COMP.
01   SIZ              PIC S9(9) USAGE IS COMP.
01   RET-STATUS       PIC S9(9) USAGE IS COMP.
.
.
.
PROCEDURE DIVISION.
.
.
.
P0.
     MOVE 4 TO SRC.
     MOVE 0 TO POS.
     MOVE 3 TO SIZ.

     CALL "LIB$INSV" USING SRC, POS, SIZ, RET-STATUS.
```

The following example shows how to call the procedure that enables and disables detection of floating underflow (LIB$FLT_UNDER) from a COBOL program. The format of the LIB$FLT_UNDER procedure is explained in Section 6.5.2.

This procedure could be called in a COBOL program to enable floating underflow as follows:

```
WORKING-STORAGE SECTION.
01   NEW-SET          PIC S9(9) USAGE IS COMP.
01   OLD-SET          PIC S9(9) USAGE IS COMP.
.
.
.
PROCEDURE DIVISION.
.
.
.
P0.
     MOVE 1 TO NEW-SET.
     CALL "LIB$FLT_UNDER" USING NEW-SET GIVING OLD-SET.
```

The following example shows how to call the procedure that finds the first clear bit in a given bit field (LIB$FFC). This procedure returns a 32-bit condition value, represented in the example as COND-VALUE:

```
WORKING-STORAGE SECTION.
01   START-POS       PIC S9(9) USAGE IS COMP.
01   SIZ             PIC S9(9) USAGE IS COMP.
01   BITS            PIC S9(9) USAGE IS COMP.
01   POS             PIC S9(9) USAGE IS COMP.
01   COND-VALUE-VAR  PIC S9(9) USAGE IS COMP.
     88 COND-VALUE   VALUE IS 1.
.
.
.
PROCEDURE DIVISION.
P0.
.
.
.
     MOVE 0 to START-POS.
     MOVE 32 TO SIZ.
     CALL "LIB$FFC USING START-POS,
                         SIZ,
                         BITS,
                         POS
                         GIVING COND-VALUE-VAR.

IF COND-VALUE
THEN
     GO TO error-proc.
```

2.10.2 Passing Parameters to Library Procedures

By default, COBOL uses the call by reference mechanism for passing parameters. In some cases, however, a function reference or call to a non-COBOL procedure (for example, a library procedure) can require you to supply arguments in a different form. Therefore, COBOL provides three qualifiers for passing parameters when you cannot use the COBOL default mechanism. They are:

- BY VALUE
- BY REFERENCE
- BY DESCRIPTOR

They can appear only in actual argument lists.

The following sections describe the use of these qualifiers. Note that they are never used to call a procedure written in COBOL.

2.10.2.1 BY VALUE — This qualifier forces the argument list entry to use the call by immediate value mechanism. It has the form:

```
BY VALUE arg
```

The value of arg is passed to the calling program. If arg is a data-name, its description in the Data Division can be:

- COMP usage with no scaling positions. The picture clause can specify no more than nine digits.
- COMP-1 usage. This is the standard VAX-11 F_Floating value.

2.10.2.2 BY REFERENCE — This qualifier forces the argument list entry to use the call by reference mechanism. It has the form:

```
BY REFERENCE arg
```

The address of (pointer to) arg is passed to the called program. This is the COBOL default mechanism.

2.10.2.3 BY DESCRIPTOR — This qualifier forces the argument list entry to use the call by descriptor mechanism. It has the form:

```
BY DESCRIPTOR arg
```

The address of (pointer to) the data item's descriptor is passed to the called program.

For more information, see the *VAX-11 COBOL-74 User's Guide*.

2.10.3 Return Status

You should always check the return status (when there is one) to make sure that the procedure executed correctly. The return status indicates success or failure. To test for errors, use an IF statement (see Section 2.10.1).

The following is a COBOL example that asks for the user's name using LIB$GET_INPUT:

```
WORKING-STORAGE SECTION.
01 USER-LINE PIC X(30).
01 PROMPT-STR PIC X(16) VALUE IS "Type Your Name: ".
01 OUT-LEN PIC S9(4) USAGE IS COMP.
01 COND-VALUE PIC S9(9) USAGE IS COMP VALUE IS 0.
   88 SS-NORMAL VALUE IS 1.
   88 LIB-INPSTRTRU VALUE IS 1409564.
.
.
.
PROCEDURE DIVISION.
P0.
   CALL "LIB$GET_INPUT" USING BY DESCRIPTOR USER-LINE
             BY DESCRIPTOR PROMPT-STR
             BY REFERENCE OUT-LEN
        GIVING COND-VALUE.
   IF LIB-INPSTRTRU
        DISPLAY "User name too long"
   ELSE
        IF NOT SS-NORMAL
             DISPLAY "More serious error"
        ELSE
             GO TO P0.
```

LIB$GET_INPUT sets the variable USER-LINE to the 30-character string input by the user. The return status is returned to the variable COND-VALUE. The first IF statement tests for the error condition that indicates the input string was too long and was truncated. The second IF statement tests for any other errors.

Note that in the preceding example, USER-LINE and PROMPT-STR are passed by descriptor, while OUT-LEN is passed by reference.

2.11 Calling a Library Procedure in FORTRAN

This section describes how to code FORTRAN calls to library procedures using either a CALL statement or function reference. CALL statements invoke subroutines that do not return meaningful values. Function references, on the other hand, return one of the following:

- A function value (typically, an integer or floating point number). For example, LIB$INDEX returns an integer value.
- A return status which is a 32-bit condition value indicating that the procedure has either successfully executed or failed. For example, LIB$GET_INPUT returns a return status.

You can invoke a subroutine as if it were a function; this normally returns a meaningless value. You can also invoke a function as if it were a subroutine if you are not interested in the function value or return status. However, it is good programming practice always to check a return status for success or failure.

2.11.1 Calling Sequence Examples

The following example shows how to call the procedure that inserts a variable bit field (LIB$INSV) from a FORTRAN program. The format of the LIB$INSV procedure is explained in Section 3.4.1. To set the low order three bits of RET_STATUS to four, you would code the following:

```
INTEGER*4 RET_STATUS
CALL LIB$INSV (4, 0, 3, RET_STATUS)
```

The following example shows how to call the procedure that enables and disables detection of floating-point underflow (LIB$FLT_UNDER) from a FORTRAN program. The format of the LIB$FLT_UNDER procedure is explained in Section 6.5.2. This procedure could be called in a FORTRAN program to enable floating underflow as follows:

```
INTEGER*4 OLD_SET
OLD_SET = LIB$FLT_UNDER (1)
```

If the old setting is of no interest, you can ignore it by treating the function LIB$FLT_UNDER as a subroutine:

```
CALL LIB$FLT_UNDER (1)
```

The following example shows how to call the procedure that finds the first clear bit in a given bit field (LIB$FFC). This procedure returns a 32-bit condition value, represented in the example as COND_VALUE:

```
INTEGER*4 COND_VALUE, BITS, POS
COND_VALUE = LIB$FFC (0, 32, BITS, POS)
IF (COND_VALUE) GO TO error
```

You can also test the success or failure of a function returning a return status directly by using an IF statement:

```
INTEGER*4 BITS, POS
IF (LIB$FFC (0,32,BITS,POS)) GO TO error
```

The following example passes a prompt string (by descriptor) as an input parameter and receives a terminal line as an output string (by descriptor) along with an output length (by reference).

```
 CHARACTER*80 TERM_LINE INTEGER*2 LEN
 IF (LIB$GET_INPUT(TERM_LINE, 'Name: ',LEN))
1THEN GO TO error
 ... = TERM_LINE(1:LEN)
```

2.11.2 Passing Parameters to Library Procedures

By default, FORTRAN uses the call by reference or call by descriptor mechanism for passing parameters, depending on the argument's data type. In some

cases, however, a function reference or call to a non-FORTRAN procedure, for example, a library procedure, can require you to supply arguments in a different form. Therefore, FORTRAN provides three compile-time functions for passing parameters when you cannot use the FORTRAN default mechanism. These compile-time functions are:

- %VAL
- %REF
- %DESCR

They can appear only in actual argument lists.

The following sections describe the use of these functions. Note that they are never used to call a procedure written in FORTRAN.

2.11.2.1 %VAL — This function forces the argument list entry to use the call by immediate value mechanism. It has the form:

```
%VAL(arg)
```

The argument list entry (arg) is the value of the entry. Because argument list entries are longwords, the argument value must be a constant (integer, logical, or F_floating), a variable, an array element, or an expression.

2.11.2.2 %REF — This function forces the argument list entry to use the call by reference mechanism. It has the form:

```
%REF(arg)
```

The argument list entry (arg) is the address of the value. The argument value can be a numeric or character expression, array, array element, or procedure name. %REF is the default FORTRAN method for passing all numeric values.

2.11.2.3 %DESCR — This function forces the argument list entry to use the call by descriptor mechanism. It has the form:

```
%DESCR(arg)
```

The argument list entry (arg) is the address of a descriptor of the value. The argument value can be any type of FORTRAN expression. %DESCR is the default FORTRAN mechanism for passing character arguments.

For more information, see the *VAX-11 FORTRAN User's Guide.*

2.11.3 Return Status

You should always check the return status (when there is one) to make sure that the procedure executed correctly. The return status indicates success or failure. To test for errors, use an IF statement (see Section 2.11.1).

To test for a particular return condition, perform a 32-bit comparison of the return status with the appropriate return status symbol listed in the procedure descriptions.

For FORTRAN programs, condition value symbols are available: (1) as parameter definition files using the INCLUDE statement and (2) as global symbols defined by the library.

SYS$LIBRARY contains the following condition value files:

- FORTRAN condition values - FORDEF.FOR
- General library condition values - LIBDEF.FOR
- Mathematics condition values - MTHDEF.FOR
- Signaling condition values - SIGDEF.FOR

The following example shows how to call the procedure that accepts input typed by the user from SYS$INPUT. The format of the LIB$GET_INPUT procedure is in Chapter 3.

Note that whenever a procedure description specifies a string descriptor parameter, the parameter being passed should always be a CHARACTER constant, variable, or expression. The FORTRAN compiler automatically produces descriptors for these parameters. The following FORTRAN example asks the user to type his or her name using LIB$GET_INPUT.

```
INCLUDE 'SYS$LIBRARY:LIBDEF'          ! Define LIB$_... condition
CHARACTER*30 USER_LINE                ! value symbols
INTEGER*4 COND_VALUE

COND_VALUE = LIB$GET_INPUT (USER_LINE, 'Type Your Name: ')
IF (COND_VALUE .EQ. LIB$_INPSTRTRU) THEN
        (user name too long)
ELSE IF (.NOT. COND_VALUE) THEN
        (more serious error)
ENDIF
```

LIB$GET_INPUT sets the variable USER_LINE to the 30-character string input by the user. The INTEGER*4 condition value (COND_VALUE) indicates success or failure. In FORTRAN, a .FALSE. condition value indicates an error and a .TRUE. condition value indicates success. The first IF statement tests for the return status that indicates that the input string was too long and was truncated. The second IF statement tests for any other errors.

The library procedure LIB$MATCH_COND (see Section 6.10.1) is useful for matching a return status or error condition value with a condition value symbol or any list of condition value symbols.

2.11.4 Function Return Values

The method of returning function procedure values depends on the data type of the value, as summarized in Table 2-4.

Table 2-4: Function Return Values

Data Type	Return Method
Logical Integer F_floating	General Register R0
D_floating G_floating	R0= High-order part of result R1= Low-order part of result
F_complex	R0= Real Part R1= Imaginary Part
H_floating	An extra entry is added as the first entry of the argument list. This new first argument entry points to the area where the result is to be stored.
Character	An extra entry is added as the first entry of the argument list. This new first argument entry points to a character string descriptor. At run time, storage is allocated to contain the value of the result, and the proper address is stored in the descriptor.

2.12 Calling a Library Procedure in PASCAL

You can invoke a Run-Time Library routine from a PASCAL program by defining it as an external function and including the appropriate function reference.

2.12.1 Calling Sequence Example

The following example shows how to invoke the procedure that returns a pseudorandom number, MTH$RANDOM.

```
VAR SEED_VAL : INTEGER;
    RAND_RSLT : REAL;
.
.
.
FUNCTION MTH$RANDOM(VAR SEED : INTEGER) : REAL; EXTERN;
.
.
.
RAND_RSLT = MTH$RANDOM(SEED_VAL);
```

When defining a function for a Run-Time Library routine, you should note the following:

- The mechanism by which each parameter is passed (by immediate value, by reference, or by descriptor)
- The data types appropriate for the parameters and the result

In the pseudorandom number generator, the seed parameter is passed by reference and the result is a real number.

2.12.2 Passing Parameters to a Library Procedure

By default, PASCAL uses the by reference mechanism for passing parameters. In some cases, however, a function reference or call to a non-PASCAL procedure (for example, a library procedure) can require you to supply arguments in a different form. Therefore, PASCAL provides four specifiers for passing parameters when you cannot use the PASCAL default mechanism. They are:

- %IMMED
- VAR
- %STDESCR
- %DESCR

The following sections describe the use of these specifiers. Note that they are never used to call a procedure written in PASCAL.

2.12.2.1 %IMMED — This specifier forces the argument list entry to use the call by immediate value mechanism. It has the form:

```
%IMMED arg : type
```

The value of arg is passed to the calling program. Variables that require more than 32 bits of storage, including all file variables, cannot be passed as immediate value.

2.12.2.2 VAR — This specifier forces the argument list entry to use the call by reference mechanism. It has the form:

```
VAR arg : type;
```

The address of arg is passed to the calling program. The actual parameter must be a variable or a component of an unpacked structural variable; constants, expressions, procedure names, and function names are not allowed.

2.12.2.3 %STDESCR — This specifier forces the argument list entry to use the call by descriptor mechanism. It has the form:

```
%STDESCR arg : type;
```

The address of a string descriptor is passed to the calling program. Only string constants, packed character arrays with subscripts from 1 to n, and packed dynamic character arrays with subscripts of an integer or integer subscript type can be passed by string descriptor.

2.12.2.4 %DESCR — This specifier forces the argument list entry to use the call by descriptor mechanism. It has the form:

```
%DESCR arg : type;
```

The argument list entry contains the address of the descriptor of an array or scalar variable. The type can be any predefined scalar type or an unpacked array (fixed or dynamic) of a predefined scalar type.

2.12.2.5 Function and Procedure Names as Parameters — You can pass procedure and function names by the immediate mechanism to routines written in another language, using these formats:

%IMMED PROCEDURE procedure-name-list

%IMMED FUNCTION function-name-list : type

The procedure name list specifies the name of one or more formal procedure parameters. The function name list specifies the name of one or more formal function parameters of the indicated type. The corresponding actual parameter lists specify the names of the actual procedures and functions to be passed as parameters.

For example:

```
PROCEDURE FORCALLER (%IMMED PROCEDURE UTILITY);
FORTRAN;
```

NOTE

The %IMMED mechanism for passing procedures and functions is valid only for the formal parameter list of procedures not written in PASCAL.

The FORTRAN subroutine FORCALLER calls a PASCAL procedure and requires that the name of the procedure as a parameter. A call to the FORTRAN procedure might be:

```
FORCALLER (PRINTER);
```

Any subprogram passed with %IMMED, should access only its own variables and those declared at program level.

2.12.3 Return Status

You should always check the return status (when there is one) to make sure that the procedure executed correctly. The return status indicates either success or failure. You can also check for a particular return status, such as lack of privileges, by comparing the return status to one of the status codes defined by the system.

To test for a particular return condition, perform a 32-bit comparison of the return status with the appropriate return status symbol listed in the procedure descriptions.

VAX/VMS provides three files containing condition symbol definitions. When you declare a Run-Time Library procedure, you should specify the appropriate file in the CONST section to define the condition values in your PASCAL program. Use the %INCLUDE directive to specify the file name, as described in the *VAX-11 PASCAL Language Reference Manual*. The three files are:

- General library condition values - LIBDEF.PAS
- Mathematics condition values - MTHDEF.PAS
- Signaling condition values - SIGDEF.PAS

2.12.4 Function Return Value

A function returns a value to the calling program by assigning that value to the function's name. The value must be of a scalar or subrange type; structured types are not allowed. The method by which a value is returned depends on its type, as pictured in Table 2-5.

Table 2-5: Function Return Values in PASCAL

Type	Return Method
Integer, Real, Single, Character, Boolean, Pointer, User-defined scalar	General Register R0
D_floating	R0: Low-order part of result R1: High-order part of result

Chapter 3
General Utility Procedures

General utility procedures include common I/O control procedures, terminal independent screen procedures, string manipulation procedures, data type conversion procedures, variable bit field manipulation procedures, performance measurement procedures, date/time utility procedures, and interlocked queue procedures.

All general utility procedures can be called explicitly from MACRO, BLISS or any VAX native mode higher-level language. Procedures with a LIB$ or STR$ prefix are designed to be called explicitly from programs written in higher-level languages; therefore the input parameters are passed by-reference. This is also true for FOR$ procedures documented in this manual. Those with an OTS$ or SCR$ prefix are usually called implicitly from programs written in higher-level languages or explicitly from MACRO or BLISS; the input scalar parameters are usually passed by immediate value.

Table 3–1 lists general utility procedures. The sections that follow this table describe the procedures in detail.

Table 3–1: General Utility Procedures

Section	Entry Point Name	Title
Common Input/Output Control Procedures		
3.1.1	LIB$ASN_WTH_MBX	Assign Channel with Mailbox
3.1.2	LIB$RUN_PROGRAM	Chain to Program
3.1.3	LIB$DO_COMMAND	Execute Command
3.1.4	LIB$GET_COMMAND	Get Line from SYS$COMMAND
3.1.4	LIB$GET_INPUT	Get Line from SYS$INPUT
3.1.5	LIB$GET_FOREIGN	Get Line from "FOREIGN" Command Line

(continued on next page)

Table 3–1: General Utility Procedures (Cont.)

3.1.6	LIB$GET_COMMON	Get String from Common
3.1.7	LIB$SYS_GETMSG	Get System Message
3.1.8	LIB$CURRENCY	Get Currency Symbol
3.1.8	LIB$DIGIT_SEP	Get Digit Group Separator Symbol
3.1.8	LIB$LP_LINES	Listing Control
3.1.8	LIB$RADIX_POINT	Get Radix Point Symbol
3.1.9	LIB$PUT_OUTPUT	Put Line to SYS$OUTPUT
3.1.10	LIB$PUT_COMMON	Put String to Common
3.1.11	LIB$SYS_TRNLOG	Translate Logical Name
	Terminal Independent Screen Procedures	
3.2.3	LIB$ERASE_LINE	Erase Line
3.2.4	LIB$ERASE_PAGE	Erase Page
3.2.5	LIB$SCREEN_INFO	Get Screen Information
3.2.6	LIB$GET_SCREEN	Get Text from Screen
3.2.7	LIB$DOWN_SCROLL	Move Cursor Up One Line
3.2.8	LIB$PUT_BUFFER	Put Current Buffer to Screen
3.2.9	LIB$PUT_SCREEN	Put Text to Screen
3.2.10	LIB$SET_BUFFER	Set/Clear Buffer Mode
3.2.11	LIB$SET_CURSOR	Set Cursor to Character Position
	String Manipulation Procedures	
3.3.2.1	STR$COMPARE	Compare Two Strings
3.3.2.2	STR$COMPARE_EQL	Compare Two Strings for Equal
3.3.2.3	LIB$LOCC	Locate Character
3.3.2.4	LIB$LEN	Return Length of String
3.3.2.5	LIB$INDEX	Return Relative Position of Substring
3.3.2.5	LIB$MATCHC	Return Relative Position of Substring
3.3.2.5	STR$POSITION	Return Relative Position of Substring
3.3.2.6	LIB$SCANC	Scan Characters
3.3.2.7	LIB$SKPC	Skip Characters
3.3.2.8	LIB$SPANC	Span Characters
3.3.2.9	LIB$CHAR	Transform Byte to a 1-Byte String
3.3.2.10	LIB$ICHAR	Transform First Character of String

(continued on next page)

Table 3-1: General Utility Procedures (Cont.)

3.3.3.1	STR$ADD	Add Two Decimal Strings
3.3.3.2	STR$MUL	Multiply Two Decimal Strings
3.3.3.3	STR$RECIP	Reciprocal of a Decimal String
3.3.3.4	STR$ROUND	Round or Truncate a Decimal String
3.3.4.1	STR$APPEND	Append a String
3.3.4.2	STR$CONCAT	Concatenate Two or more Strings
3.3.4.3	LIB$SCOPY_DXDX	Copy String Passed by Descriptor
3.3.4.3	OTS$SCOPY_DXDX	Copy String Passed by Descriptor
3.3.4.3	STR$COPY_DX	Copy String Passed by Descriptor
3.3.4.3	LIB$SCOPY_R_DX	Copy String Passed by Reference
3.3.4.3	OTS$SCOPY_R_DX	Copy String Passed by Reference
3.3.4.3	STR$COPY_R	Copy String Passed by Reference
3.3.4.4	STR$LEN_EXTR	Extract Substring by Length
3.3.4.4	STR$POS_EXTR	Extract Substring from Position
3.3.4.4	STR$LEFT	Extract Leftmost Substring
3.3.4.4	STR$RIGHT	Extract Rightmost Substring
3.3.4.5	STR$DUPL_CHAR	Generate a String
3.3.4.6	STR$PREFIX	Prefix a String
3.3.4.7	STR$REPLACE	Replace a Substring
3.3.4.8	STR$TRIM	Trim Trailing Blanks and Tabs
3.3.5.1	LIB$MOVTC	Move Translated Characters
3.3.5.2	LIB$MOVTUC	Move Translated until Character
3.3.5.3	LIB$TRA_ASC_EBC	Translate ASCII to EBCDIC
3.3.5.4	LIB$TRA_EBC_ASC	Translate EBCDIC to ASCII
3.3.5.5	STR$TRANSLATE	Translate Matched Characters
3.3.5.6	STR$UPCASE	Uppercase Conversion
	Formatted Input/Output Conversion Procedures	
3.4.1.1	OTS$CVT_T_D	Convert Text to D_Floating
3.4.1.1	OTS$CVT_T_G	Convert Text to G_Floating
3.4.1.1	OTS$CVT_T_H	Convert Text to H_Floating
3.4.1.2	OTS$CVT_TI_L	Convert Text (integer) to Longword
3.4.1.3	OTS$CVT_TL_L	Convert Text (logical) to Longword
3.4.1.4	OTS$CVT_TO_L	Convert Text (octal) to Longword
3.4.1.5	OTS$CVT_TZ_L	Convert Text (hexadecimal) to Longword

(continued on next page)

Table 3-1: General Utility Procedures (Cont.)

3.4.1.6	LIB$CVT_DTB	Decimal to Binary Conversion
3.4.1.6	LIB$CVT_OTB	Octal to Binary Conversion
3.4.1.6	LIB$CVT_HTB	Hexadecimal to Binary Conversion
3.4.2.1	OTS$CVT_L_TI	Convert Longword to Text (integer)
3.4.2.2	OTS$CVT_L_TL	Convert Longword to Text (logical)
3.4.2.3	OTS$CVT_L_TO	Convert Longword to Text (octal)
3.4.2.4	OTS$CVT_L_TZ	Convert Longword to Text (hexadecimal)
3.4.2.5	FOR$CVT_D_TD,E,F,G	Convert D_floating to text
3.4.2.5	FOR$CVT_G_TD,E,F,G	Convert G_floating to text
3.4.2.5	FOR$CVT_H_TD,E,F,G	Convert H_floating to text
3.4.3.1	LIB$SYS_FAO	Formatted ASCII Output
3.4.3.2	LIB$SYS_FAOL	Formatted ASCII Output with LIST
	Variable Bit Field Instruction Procedures	
3.5.1	LIB$INSV	Insert a Variable Bit Field
3.5.2	LIB$EXTV	Extract and Sign-extend a Bit Field
3.5.3	LIB$EXTZV	Extract a Zero-extended Bit Field
3.5.4	LIB$FFC	Find First Clear Bit
3.5.5	LIB$FFS	Find First Set Bit
	Performance Measurement Procedures	
3.6.1	LIB$FREE_TIMER	Free Timer Storage
3.6.2	LIB$INIT_TIMER	Initialize Times/Counts
3.6.3	LIB$STAT_TIMER	Return Accumulated Times/Counts
3.6.4	LIB$SHOW_TIMER	Show Accumulated Times/Counts
	Date/Time Utility Procedures	
3.7.1	LIB$SYS_ASCTIM	Convert Binary Date/Time to ASCII String
3.7.2	FOR$IDATE	Return Month, Day, Year as a Word Integer
3.7.3	FOR$JDATE	Return Month, Day, Year as a Longword Integer
3.7.4	FOR$DATE	Return System Date as 9-Byte String
3.7.4	FOR$DATE_T_DS	Return System Date as Fixed-Length String

(continued on next page)

Table 3-1: General Utility Procedures (Cont.)

3.7.5	FOR$SECNDS	Return System Time in Seconds
3.7.6	FOR$TIME_T_DS	Return System Time to Fixed-Length String
3.7.6	FOR$TIME	Return System Time as 8-Byte String
3.7.7	LIB$DAY	Return Day Number as a Longword Integer
3.7.8	LIB$DATE_TIME	Return System Date/Time
	Miscellaneous Procedures	
3.8.1	LIB$AST_IN_PROG	AST in Progress
3.8.2	LIB$CRC	Calculate Cyclic Redundancy Check
3.8.3	LIB$CRC_TABLE	Construct Cyclic Redundancy Check Table
3.8.4	LIB$EMULATE	Emulate VAX-11 Instructions
3.8.5	LIB$ADDX	Multiple Precision Binary Add
3.8.5	LIB$SUBX	Multiple Precision Binary Subtract
3.8.6	LIB$SIM_TRAP	Simulate Floating Trap
3.8.7	LIB$EMODD	Extended Multiply D_Floating
3.8.7	LIB$EMODF	Extended Multiply F_floating
3.8.7	LIB$EMODG	Extended Multiply G_Floating
3.8.7	LIB$EMODH	Extended Multiply H_Floating
3.8.8	LIB$POLYD	Evaluate Polynomial D_Floating
3.8.8	LIB$POLYF	Evaluate Polynomial F_floating
3.8.8	LIB$POLYG	Evaluate Polynomial G_Floating
3.8.8	LIB$POLYH	Evaluate Polynomial H_Floating
3.8.9.1	LIB$INSQHI	Queue Entry Inserted at Head
3.8.9.2	LIB$INSQTI	Queue Entry Inserted at Tail
3.8.9.3	LIB$REMQHI	Queue Entry Removed at Head
3.8.9.4	LIB$REMQTI	Queue Entry Removed at Tail

3.1 Common Input and Output Control Procedures

When you log in to VAX/VMS, process-permanent files identified with the logical names SYS$INPUT, SYS$COMMAND, and SYS$OUTPUT are created as default I/O control streams for your process. These files are the interface between your interactive input (or batch control) and the VAX/VMS software. You can use the library procedures LIB$GET_INPUT,

LIB$GET_COMMAND and LIB$PUT_OUTPUT to read a record from SYS$INPUT, SYS$COMMAND, or write a record to SYS$OUTPUT using the VAX-11 Record Management Services (RMS).

You can change SYS$INPUT to obtain control information from any file using a DCL command. Similarly, you can change SYS$OUTPUT so that control information is output to any file. SYS$INPUT and SYS$COMMAND are usually identical. However, the input and the command streams can be different (such as during the execution of an indirect command file from an interactive terminal). In this case, SYS$COMMAND refers to input from the terminal and SYS$INPUT refers to input from the file. LIB$GET_COMMAND is used only when input is to come from the terminal rather than an indirect command file. For example, when a program asks a question that the user could not provide an answer for in an indirect command file.

The following software gets controlling input from SYS$INPUT and directs controlling output to SYS$OUTPUT:

- Command interpreter
- Utilities
- Run-Time Library
- All other user-mode software

Typically, a record corresponds to a line for an interactive device. However, no ASCII carriage-return (CR) and/or line-feed (LF) are part of the data in the record. Formatting is handled entirely by RMS when the data is input or output.

Because VAX/VMS creates SYS$INPUT and SYS$OUTPUT as process permanent files, each procedure can perform its own OPEN, GET, CLOSE, and PUT operations. Therefore, LIBGET_INPUT, LIBGET_COMMAND and LIB$PUT_OUTPUT are not image resource allocation procedures.

For the LIB$ procedures in this section that have strings as parameters, the following severe errors can be returned as a completion status:

LIB$_FATERRLIB	fatal internal error
LIB$_INSVIRMEM	insufficient virtual memory
LIB$_INVSTRDES	invalid string descriptor

To save space the preceding errors are listed by name only in each procedure description. Other errors, more specific to a particular procedure are listed and explained under each procedure description.

3.1.1 Assign Channel with Mailbox

LIB$ASN_WTH_MBX assigns a channel to a specified device and associates a mailbox with the device. It returns both the device channel and the mailbox channel.

Normally, when a mailbox is created, the corresponding logical name is placed in the GROUP logical name table. This implies that any process running in the same group and using the same logical name uses the same mailbox. There are times when this is not desirable. For example, when a nontransparent network connect is done, a mailbox is used to obtain the connect confirm data and asynchronous messages from the other task. Multiple processes running under the same group and sharing a common mailbox for their network links do not work correctly. These processes read each other's mailbox messages. LIB$ASN_WTH_MBX avoids the problem by associating the physical mailbox name with the channel assigned to the device.

Format

ret-status = LIB$ASN_WTH_MBX (dev-na-, max-msg, buf-quo, dev-chn, mbx-chn)

dev-nam
: Address of the device name descriptor. This string is input to the $ASSIGN service.

max-msg
: A longword integer representing the maximum size of messages that can be sent to the mailbox. This parameter is input to the $CREMBX service.

buf-quo
: A longword integer representing the number of bytes of system dynamic memory that can be used to buffer messages sent to the mailbox. This parameter is input to the $CREMBX service.

dev-chn
: Address of a word to receive the device channel. This value is output from the $ASSIGN service.

mbx-chn
: Address of a word to receive the mailbox channel. This value is output from the $CREMBX service.

Return Status

SS$_NORMAL
: Routine successfully completed.

SS$_xyz
: Any return status from a called system service. $ASSIGN, $CREMBX, $GETCHN, and $FAO services are used.

3.1.2 Chain to Program

LIB$RUN_PROGRAM causes the current program to stop running and begins execution of another program. If successful, control does not return to the calling program. Instead, the $EXIT system service is called, the new program image replaces the old image in the user process, and control is given to the new image by the command interpreter. If unsuccessful, control returns to the command interpreter.

Format

ret-status = LIB$RUN_PROGRAM (pgm-name)

pgm-name
: Address of the descriptor of a character string containing the file name of the program to be run in place of the current program. The maximum length of the file name is 256 characters. The default file type is .EXE.

Return Status

LIB$_INVARG
: Invalid argument.

3.1.3 Execute Command

LIB$DO_COMMAND causes the current program to stop running and then executes the new command. If successful, control does not return to the calling program. Instead, the $EXIT system service is called, and the new command is passed to the command interpreter. Note that the command can execute an indirect file using the at-sign (@) feature of DCL.

Format

ret-status = LIB$DO_COMMAND (cmd-text)

cmd-text
: Address of the descriptor of a character string containing the text of the command to be executed. The maximum length of the command is 256 characters.

Return Status

LIB$_INVARG
: Invalid argument.

3.1.4 Get Line from SYS$INPUT

LIB$GET_INPUT gets one record of ASCII text from the current controlling input-device, specified by SYS$INPUT. LIB$GET_INPUT uses the VAX-11 RMS $GET service.

LIB$GET_INPUT opens file SYS$INPUT on the first call. The VAX-11 RMS internal stream identifier (ISI) is stored in the procedure's static storage for subsequent calls.

If prompt-str is provided and the SYS$INPUT device is a terminal, LIB$GET_INPUT outputs the prompt message. If the device is not a terminal, the prompt-str is ignored.

LIB$GET_COMMAND is identical to LIB$GET_INPUT, except that input comes from SYS$COMMAND.

Format

ret-status = LIB$GET_INPUT (get-str [,prompt-str [,out-len]])

ret-status = LIB$GET_COMMAND (get-str [,prompt-str [,out-len]])

get-str
Address of string descriptor to receive the string (fixed-length or dynamic).

prompt-str
Address of a string descriptor specifying an optional prompt message that is output to the controlling terminal. If no other conventions are established, prompts are English words followed by a colon(:), one space, and no CRLF (carriage-return/line-feed).

out-len
Optional address of a word to receive the number of bytes written into get-str, not counting padding in the case of a fixed string. If the input string is truncated to the size specified in the get-str descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of get-str.

Return Status

SS$_NORMAL
Routine successfully completed. VAX-11 RMS completion status.

LIB$_FATERRLIB
An internal consistency check on Run-Time Library data structures has failed. This may indicate a programming error in the Run-Time Library or that the user has overwritten those data structures.

LIB$__INPSTRTRU
The input string is truncated to the size specified in the get-str descriptor (fixed-length or unspecified string types only). Out-len is also set to this size. This is an error (as opposed to LIB$__STRTRU which is a success) because the truncation is not under program control.

LIB$__INSVIRMEM
Insufficient virtual memory to allocate dynamic string.

LIB$__INVARG
Invalid arguments. Descriptor class field is not a recognized code or zero.

LIB$__STRIS__INT
String is interlocked. The parameter get-str was being accessed at non-AST level or in a previous AST. Writing into the parameter at this time could invalidate that previous access.

RMS$__xyz
Any VAX-11 RMS error code indicates a VAX-11 RMS error.

Examples

The following FORTRAN code fragment asks at the terminal for the user's name and age.

```
      CHARACTER NAME*30, AGE*2
      INTEGER IAGE
      IF (.NOT. LIB$GET_INPUT (NAME, 'Last Name: ')) GO TO 999
50    IF (.NOT. LIB$GET_INPUT (AGE, 'Age: '))  GO TO 999
      READ (AGE,150,ERR=50) IAGE
150   FORMAT (BNI2)
```

If any error occurs during the input of the name or age, control goes to statement 999. Otherwise, the 2-character AGE string is converted to an integer. If a formatting error occurs, the user is asked for age again.

The following is an example of what the user might see at the terminal (lowercase characters indicate what the user typed):

```
LAST NAME: Jones
AGE: 3o
AGE: 30
```

Age was asked again because the letter o was typed instead of the number 0.

The following FORTRAN example asks for last name, first name separately and concatenates them without any of the trailing blanks.

```
INTEGER*2, LLEN, FLEN
CHARACTER NAME*62, LNAME*30, FNAME*30
IF(.NOT. LIB$GET_INPUT(LNAME, 'LAST NAME: ',LLEN)) GOTO 999
IF(.NOT. LIB$GET_INPUT(FNAME, 'FIRST NAME: ',FLEN)) GOTO 999
NAME = LNAME(1:LLEN)//' '//FNAME(1:FLEN)
```

3.1.5 Get Line from FOREIGN Command Line

LIB$GET_FOREIGN gets the command line from the "foreign command" line that activated the current image. A foreign command is used to run a user program as if it were a native command. A program run by a foreign command can request the remainder of the command line (after the command name) and can parse it for whatever options needed.

To define a foreign command, use the following DCL command:

```
$ command_name  :== $filespec
```

where:

command_name is the name of the foreign command you want to define and filespec is the fully qualified file specification of the executable image to be run when command_name is invoked.

For example:

```
$ VULCAN :== $DB0:[SPOCK]VULCAN.EXE
```

The "$" prefix is required and must immediately precede the file specification.

Assuming that the command VULCAN was defined, the command line:

```
$VULCAN/OUTPUT=GANYMEDE TITAN.DAT
```

would start running the image DB0:[SPOCK]VULCAN.EXE. If that program then calls LIB$GET_FOREIGN, it can obtain the remainder of the command line:

```
/OUTPUT=GANYMEDE TITAN.DAT
```

The user program can analyze this returned string in any manner it desires (see Chapter 7). No interpretation is done by the command interpreter.

If the image resides in the SYS$SYSTEM: directory, the image could be invoked by the MCR command and the command line text following the image name would be returned. If the image were not invoked by a foreign command or MCR, or if there were no information remaining on the command line, and the user-supplied prompt were present, LIB$GET_INPUT would be called to prompt for a command line. Otherwise, a zero length string would be returned, subject to the appropriate semantics of the destination string class.

Format

ret-status = LIB$GET_FOREIGN (get-str [,prompt-str [,out-len]])

get-str
Address of string descriptor to receive the command line (fixed-length or dynamic).

prompt-str
Address of a string descriptor specifying an optional prompt message that is output to the controlling input device, if it is a terminal. The prompt message is sent to the terminal when there is no information in the command line. If prompt-str is omitted, no prompting is performed.

out-len
Optional address of a word to receive the number of bytes written into get-str, not counting padding in the case of a fixed string. If the input string is truncated to the size specified in the get-str descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of get-string.

Return status

SS$_NORMAL
Procedure successfully completed.

LIB$_INPSTRTRU
The string from SYS$INPUT was truncated to the size specified in the get-string descriptor (static or unspecified types only).

LIB$_INVARG
Invalid arguments. Descriptor class is not a recognized class or zero.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

Example

The following BASIC code fragment checks an input line for data or switches:

```
100  DECLARE STRING INPUT_LINE
110  DECLARE INTEGER RET_STATUS, INPUT_LEN
120  EXTERNAL INTEGER FUNCTION LIB$GET_FOREIGN

200  RET_STATUS = LIB$GET_FOREIGN(INPUT_LINE,          &
                  "VULCAN> ",INPUT_LEN)

300  IF (RET_STATUS AND 1%) <> 0% THEN                 &
          IF SEG$(INPUT_LINE,1%,1%) = "/" THEN         &
               PRINT "SWITCHES"                        &
          ELSE IF INPUT_LEN <> 0% THEN                 &
               PRINT "DATA, NO SWITCHES"               &
               ELSE PRINT "NO SWITCHES OR DATA"        &
     ELSE CALL LIB$STOP(RET_STATUS BY VALUE)
```

3.1.6 Get String from Common

LIB$GET_COMMON copies the string in the common area to the destination string. The string length is taken from the first longword of the common area. If the string is too long for the destination, the string is truncated. The number of characters copied is returned by the optional parameter, chars-copied (if given).

Format

ret-status = LIB$GET_COMMON (dst-str [,chars-copied])

dst-str
: Address of the destination string descriptor (fixed-length or dynamic).

chars-copied
: Optional address of a word to receive the number of characters written into dst-str, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the dst-str descriptor, chars-copied is set to this size. Therefore, chars-copied can always be used by the calling program to access a valid substring of dst-str.

Return Status

SS$_NORMAL
: Routine successfully completed.

LIB$_STRTRU
: Successfully completed, but the string was longer than the buffer and was truncated.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

3.1.7 Get System Message

LIB$SYS_GETMSG calls the System Service GETMSG with the caller's input string. The resultant string is returned using the semantics of the caller's string. Parameters msg-id and flags are presented to this routine by reference and are promoted to immediate value for presentation to GETMSG.

Format

ret-status = LIB$SYS_GETMSG (msg-id, [msg-len], dst-str [,flags [,out-arr]])

msg-id
: Address of a longword containing the identification of the message to be retrieved.

msg-len
Optional address of a word to receive the number of characters written into dst-str, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the dst-str descriptor, msg-len is set to this size. Therefore, msg-len can always be used by the calling program to access a valid substring of dst-str.

dst-str
Address of the destination string descriptor (fixed-length or dynamic).

flags
Address of a longword containing the flag bits for message content. This is an optional parameter; the default value is all 1.

Bit	Value	Meaning
0	1	Include text
	0	Do not include text
1	1	Include identifier
	0	Do not include identifier
2	1	Include severity
	0	Do not include severity
3	1	Include component
	0	Do not include component

out-arr
Address of a 4-byte array to receive message specific information. This is an optional parameter.

Byte	Contents
0	Reserved
1	Count of FAO arguments
2	User value
3	Reserved

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_STRTRU
Successfully completed, but source string was truncated.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

SS$_BUFFEROVF
Successfully completed, but the resultant string overflowed the buffer provided and has been truncated.

SS$_MSGNOTFND
Successfully completed, but the message code does not have an associated message in file.

Example

The following BASIC code fragment gets and prints the system error message for the return status when the value returned is not 1:

```
100  EXTERNAL INTEGER FUNCTION LIB$PROC, LIB$SYS_GETMSG
110  DECLARE INTEGER RET_STATUS
200  RET_STATUS = LIB$PROC(A,B,C)
300  IF (RET_STATUS AND 1%) <> 0% THEN                          &
                .                                               &
                normal path                                     &
                .
     ELSE IF (LIB$SYS_GETMSG(RET_STATUS,,                       &
                OUT_STRINGS,1%) <> 0% THEN                      &
                PRINT OUT_STRING$
          ELSE PRINT "DOUBLE ERROR - HALT"
```

3.1.8 Listing Control

These procedures provide the user with the capability of customizing the printer output with respect to the currency symbol, the digit separator, the radix point and the number of lines on each page.

3.1.8.1 Currency Symbol — LIB$CURRENCY returns the system's currency symbol. This symbol should be used before a number to indicate that the number represents money in the local country.

This routine works by attempting to translate the logical name SYS$CURRENCY as a process, group, or system logical name. If the translation fails, the routine returns "$", the United States money symbol. If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS$CURRENCY as a system-wide logical name to provide a default for all users, and an individual user with a special need can define SYS$CURRENCY as a process logical name to override the default.

Format

ret-status = LIB$CURRENCY (currency-str [,out-len])

currency-str
Address of the currency string descriptor (fixed-length or dynamic).

out-len
Optional address of a word to receive the number of characters written into currency-str, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the currency-str descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of currency-str.

Implicit Inputs

Logical name SYS$CURRENCY.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_STRTRU
Successfully completed, but the currency string was truncated.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

3.1.8.2 Digit Separator Symbol — LIB$DIGIT_SEP returns the system's digit separator symbol. This symbol should be used to separate groups of three digits in the integer part of a number, for readability, using the customary symbol.

This routine attempts to translate the logical name SYS$DIGIT_SEP as a process, group, or system logical name. If, the translation fails, the routine returns " , ", the United States digit separator. If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS$DIGIT_SEP as a system-wide logical name to provide a default for all users, and an individual user with a special need can define SYS$DIGIT_SEP as a process logical name to override the default symbol.

Format

ret-status = LIB$DIGIT_SEP (digit-sep-str [,out-len])

digit-sep-str
Address of the digit separator string descriptor (fixed-length or dynamic).

out-len
Optional address of a word to receive the number of characters written into digit-sep-str, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the digit-sep-str descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of digit-sep-str.

Implicit Inputs

Logical name SYS$DIGIT_SEP.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_STRTRU
Successfully completed, but the digit separator string was truncated.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

3.1.8.3 Number of Lines per Line Printer Page — LIB$LP_LINES computes the default number of lines on a line printer page. This procedure can be used by native-mode VAX/VMS utilities that produce "listing" files and do pagination.

United States standard paper stock permits 66 lines on each physical page. From this value, the utility should deduct:

1. Three lines for top margin
2. Three lines for bottom margin
3. Three lines for listing heading information, consisting of:
   a. Language-processor identification line
   b. Source-program identification line
   c. One blank line

The algorithm used by LIB$LP_LINES is:

1. Translate the logical name SYS$LP_LINES.
2. Convert the ASCII value obtained to a binary integer.
3. Verify that the resulting value is in the range [30:99].
4. If any of the prior steps fail, return the default U.S. paper size of 66 lines.

Format

page-len = LIB$LP_LINES ( )

page-len
A longword to receive the default number of lines on a physical line printer page. If the logical name translation or conversion to binary fails, a default value of 66 is returned.

Implicit Inputs

Logical name SYS$LP_LINES.

3.1.8.4 Radix Point Symbol — LIB$RADIX_POINT returns the system's radix point symbol. This symbol should be used inside a digit string to separate the integer part from the fraction part. This routine works by attempting to translate the logical name SYS$RADIX_POINT as a process, group, or system logical name.

If the translation fails, this routine returns ".", the United States radix point symbol. If the translation succeeds, the text produced is returned. Thus, a system manager can define SYS$RADIX_POINT as a system-wide logical name to provide a default for all users, and an individual user with a special need can define SYS$RADIX_POINT as a process logical name to override the default.

Format

ret-status = LIB$RADIX_POINT (radix-point-str [,out-len])

radix-point-str
: Address of the radix point string descriptor (fixed-length or dynamic).

out-len
: Optional address of a word to receive the number of characters written into radix-point-str, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the radix-point-str descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of radix-point-str.

Implicit Inputs

Logical name SYS$RADIX_POINT.

Return Status

SS$_NORMAL
: Procedure completed successfully.

LIB$_STRTRU
: Successfully completed, but the radix point string was truncated.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

3.1.9 Put Line to SYS$OUTPUT

LIB$PUT_OUTPUT outputs a record (line) to the current controlling output device, specified by SYS$OUTPUT, using the VAX-11 RMS $PUT service. LIB$PUT_OUTPUT opens and positions at end-of-file (or creates if not existent) SYS$OUTPUT on the first call in case it is not a process-permanent file. The VAX-11 RMS internal stream identifier (ISI) is stored in the procedure's storage space for all subsequent calls.

Format

ret-status = LIB$PUT_OUTPUT (msg-str)

msg-str
: Address of a string descriptor specifying the message. VAX-11 RMS handles all formatting, so that the message does not need to include such ASCII formatting instructions as carriage return (CR).

Return Status

SS$_NORMAL
: Routine successfully completed.

RMS$_abc
: VAX-11 RMS error code indicates an RMS error.

Example

The following FORTRAN code fragment outputs a string:

```
CALL LIB$PUT_OUTPUT ('Hello There')
```

3.1.10 Put String to Common

LIB$PUT_COMMON copies the contents of a string specified by the caller into the common area. Optionally, it returns the actual number of characters copied.

Format

ret-status = LIB$PUT_COMMON (src-str [,chars-copied])

src-str
Address of the source string descriptor.

chars-copied
Optional address of a word to receive the number of characters copied.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_STRTRU
Successfully completed, but the source string was truncated.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

3.1.11 Translate Logical Name

LIB$SYS_TRNLOG uses the system service TRNLOG to translate a logical name, returning the resultant string using the semantics of the caller's string. Parameter dsb-msk is presented to this routine by reference and is promoted to by immediate value for presentation to TRNLOG.

See the TRNLOG system service description in the *VAX/VMS System Services Reference Manual.*

Format

ret-status = LIB$SYS_TRNLOG (logical-name, [dst-len], dst-str [,table [,acc-mode [,dsb-msk]]])

logical-name
Address of the logical name string descriptor.

dst-len
Optional address of a word to receive the number of characters written into dst-str, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the dst-str descriptor, dst-len is set to this size. Therefore, dst-len can always be used by the calling program to access a valid substring of dst-str.

dst-str
Address of the destination string descriptor (fixed-length or dynamic).

table
Address of a byte to receive the logical name table number. (This is an optional parameter.)

acc-mode
Address of a byte to receive the access mode of entry (process table only). (This is an optional parameter.)

dsb-msk
Address of a byte containing the table search disable mask. (This is an optional parameter.)

Bit Set	Meaning
0	Do not search system logical name table
1	Do not search group logical name table
2	Do not search process logical name table

Return Status

SS$_NORMAL
Procedure successfully completed.

SS$_NOTRAN
Successfully completed, but input logical name string was placed in destination string buffer because no equivalence name was found.

LIB$_STRTRU
Successfully completed, but source string truncated on copy.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

SS$_ACCVIO
The logical name string or string descriptor cannot be read, or the output length, output buffer, table or access mode field cannot be written, by the caller.

SS$_INVLOGNAM
The specified logical name string has a length of zero or has more than 63 characters.

SS$_RESULTOVF
The destination string buffer has a length of zero, or it is smaller than the resultant string.

Example

The following BASIC code fragment translates the logical name ORION by searching only the system table:

```
100  EXTERNAL INTEGER FUNCTION LIB$SYS110
DECLARE INTEGER RET_STATUS
200  RET_STATUS = LIB$SYS_TRNLOG("ORION",,OUTSTRING$,,,3%)
210  PRINT "TRANSLATED:"; OUT_STRING$
```

3.2 Terminal Independent Screen Procedures

The terminal independent screen procedures provide a high-level language interface to DIGITAL video terminals. An assembly language interface (SCR$) is also provided where input parameters are passed by immediate value.

NOTE

If the terminal type is a VT52 or VT100, as specified by the DCL command SET TERMINAL, an escape sequence is output during the first access to the terminal by these procedures to ensure that the terminal is in the correct mode. To operate a VT100 in VT52 mode, you should first type a SET TERMINAL/VT52 command.

3.2.1 Cursor Positioning on a Screen

Several procedures let the user control the cursor position. The top line of a screen is line number one. The leftmost column of a screen is column number one. When the line and column parameters are optional, both must be specified or neither.

For the erase page procedures, line n of the screen is logically contiguous with line n+1.

No checks are made in these procedures to return an error status for cursor position specifications which exceed the maximum number of lines or columns for the terminal. No attempt is made by these procedures to create multiple line output and, thereby, cause line wrap or prevent the loss of text characters.

3.2.2 Screen Functions in Buffer Mode

Buffer mode has the advantage of letting the user format an entire screen of information and present this data on the screen with one call to the queue I/O service. This is particularly more efficient when a communications network is involved.

Buffer sizes can be difficult to determine accurately when a variable amount of data composes a screen of data. Therefore, when a buffer overflow condition is detected, the buffer is put to the screen via a queue I/O service function, the buffer data size is set to zero and the current buffering mode continues.

In a modular programming environment, screen buffering can occur at several levels. That is, a procedure can establish buffer mode then call another procedure which also establishes buffer mode and so on.

Although each procedure which establishes buffer mode must have buffer storage available, only one buffer is active at any point in time. As further levels of buffering occur, the contents on one buffer are copied into the active buffer and the previous buffer is set to indicate an empty buffer. Pointers to the previous level buffer are made available to the user program so it can copy (by calling LIB$PUT_BUFFER) the current buffer back to the calling program's buffer before returning to the calling program.

The copy process indicates that the contents of the buffer are cumulative from the time buffer mode is established. This also indicates that (if automatic QIOs triggered by a buffer overflow condition are to be avoided) the buffer sizes stated for called procedures must take into account the size of the data that has been created by all the calling procedures and the size of the data being created by all the called procedures. Similarly, the calling procedure must allow the size of the buffer in the calling procedure to account for the size of the data being buffered at all called procedures below the calling procedure in addition to the size of the data being buffered in the calling procedure.

3.2.3 Erase Line

LIB$ERASE_LINE and SCR$ERASE_LINE erase all of the character positions on the screen from the specified cursor position to the end of the line.

Format

ret-status = LIB$ERASE_LINE ([line-no, col-no])

ret-status = SCR$ERASE_LINE ([line-no, col-no])

line-no
Optional address of a signed word integer containing the line number where the erase begins. The default is the current line number. For SCR$ERASE_LINE, the line number is passed by immediate value.

col-no
Optional address of a signed word integer containing the column number where the erase begins. The default is the current column number. For SCR$ERASE_LINE, the column number is passed by immediate value.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid argument. The number of parameters specified must be none or two.

LIB$_INVSCRPOS
Invalid screen position values. Line-no or col-no was zero.

Example

The following FORTRAN code fragment would erase the screen from column 41 of line 12 to the end of line 12:

```
ICOL = 41
ILINE = 12
ISTAT = LIB$ERASE_LINE (ILINE,ICOL)
```

3.2.4 Erase Page

LIB$ERASE_PAGE and SCR$ERASE_PAGE erase all of the character positions on the screen from the specified cursor position to the end of the screen.

Format

ret-status = LIB$ERASE_PAGE ([line-no, col-no])

ret-status = SCR$ERASE_PAGE ([line-no, col-no])

line-no
Optional address of a signed word integer containing the line number where the erase begins. The default is the current line number. For SCR$ERASE_PAGE, the line number is passed by immediate value.

col-no
Optional address of a signed word integer containing the column number where the erase begins. The default is the current column number. For SCR$ERASE_PAGE, the column number is passed by immediate value.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid argument. The number of parameters specified must be none or two.

LIB$_INVSCRPOS
Invalid screen position values. Line-no or col-no was zero.

Example

The following FORTRAN code fragment would clear the entire screen:

```
ICOL = 1
ILINE = 1
ISTAT = LIB$ERASE_PAGE (ILINE, ICOL)
```

3.2.5 Get Screen Information

LIB$SCREEN_INFO and SCR$SCREEN_INFO move terminal specifications to user specified area(s). These terminal specifications include miscellaneous flags, device type, screen width and number of lines per screen.

Format

ret-status = LIB$SCREEN_INFO (flags [,dev-type [,line-width [,lines-per-page]]])

ret-status = SCR$SCREEN_INFO (control-block)

flags
Address of a longword to contain a bit map representing special terminal characteristics. Currently, these values are used:

Bit	Value	Meaning
0	1	DIGITAL video terminal
	0	Hard copy or unknown type terminal
1:31	0	Unused at present

dev-type
Optional address of a byte to contain an integer representing the terminal type. The terminal types are defined in the $DCDEF macro. Some of the terminal types are:

0 Unknown type or nongraphic
64 VT52
96 VT100

line-width
Optional address of a word to contain an integer representing the width in columns for which the terminal is configured. This corresponds to the value supplied by the DCL command, SET TERMINAL/WIDTH = n.

lines-per-page
Optional address of a word to contain an integer representing the lines per screen for which the terminal is configured. This corresponds to the value supplied by the DCL command, SET TERMINAL/PAGE = n.

control-block
Address of an area to contain nine bytes which correspond in order to the flags, line-width, lines-per-page and dev-type parameters specified for LIB$SCREEN_INFO.

Example

The following FORTRAN code fragment would display the screen information on the first four lines of a cleared screen:

```
      INTEGER*2 FLAGS, DEVTYPE, LINEWIDTH, LINESPP
      ILINE = 1
C     ICOL = 1
C
C     GET SCREEN_INFO AND DISPLAY IT ON FIRST FOUR LINES OF
C     A CLEARED SCREEN

      ISTAT = LIB$ERASE_PAGE (ILINE, ICOL)
200   ISTAT = LIB$SCREEN_INFO (FLAGS, DEVTYPE, LINEWIDTH, LINESPP)
      FORMAT (15H FLAGS       = ,I6,/,
     a15H DEVICE TYPE = ,I6,/,
     b15H LINE WIDTH = ,I6,/,
     c15H LINES/PAGE = ,I6)
      WRITE (6, 200) FLAGS, DEVTYPE, LINEWIDTH, LINESPP
```


3.2.6 Get Text from Screen

LIB$GET_SCREEN and SCR$GET_SCREEN copy text (input by the terminal user) from the screen into a specified destination.

Format

ret-status = LIB$GET_SCREEN (input-text [,prompt-str [,out-len]])

ret-status = SCR$GET_SCREEN (input-text [,prompt-str [,out-len]])

input-text
Address of a descriptor of a string to receive the text copied from the screen (fixed-length or dynamic).

prompt-str
Optional address of a descriptor of a string that is displayed on the screen starting at the current cursor position prior to accepting input from the user terminal.

out-len
Optional address of a word to receive the number of characters written into input-text, not counting padding in the case of a fixed-length string. If the input string is truncated to the size specified in the input-text descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of input-text.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$__INPSTRTRU
The input string is truncated to the size specified in the input-text descriptor.

LIB$__INVARG
Invalid argument. Descriptor class field is not a recognized code or zero.

RMS$__xyz
Any VAX-11 RMS error code.

Example

The following FORTRAN code fragment would prompt the user with "ENTER NAME: ," accept up to 30 characters and put them in INPUT, and set LENGTH equal to the number of characters input:

```
CHARACTER PROMPT*12, INPUT*30
DATA PROMPT/'ENTER NAME: '/
INTEGER*2 LENGTH
ICOL = 1
ILINE = 24
ISTAT = LIB$SET_CURSOR (ILINE, ICOL)
ISTAT = LIB$GET_SCREEN (INPUT, PROMPT, LENGTH)
```

NOTE

This procedure is identical to LIB$GET__INPUT, and is provided for symmetry.

LIB$DOWN__SCROLL

3.2.7 Move Cursor Up One Line, Scroll Down if at Top

LIB$DOWN__SCROLL and SCR$DOWN__SCROLL move the cursor up one line on the screen. If the cursor was already at the top line on the screen, all lines are moved down one line, the top line is replaced with a blank line and the data that was on the bottom line is lost.

Format

ret-status = LIB$DOWN__SCROLL ()

ret-status = SCR$DOWN__SCROLL ()

Return Status

SS$__NORMAL
Routine successfully completed.

Example

The following FORTRAN code fragment would cause the screen to be scrolled down one line:

```
CALL LIB$SET_CURSOR (1, 1)
CALL LIB$DOWN_SCROLL ()
```

3.2.8 Put Current Buffer to Screen or Previous Buffer

LIB$PUT_BUFFER and SCR$PUT_BUFFER procedures terminate the current buffering mode and revert to the previous mode as specified by the parameter. If the parameter is zero or omitted, buffering is terminated and the contents of the current screen buffer are output to the screen. If the parameter is not zero, buffering is terminated at the current level, the parameter is the address of a previous screen buffer to which the data from the current buffer is copied, the current buffer is set to zero length and the previous buffer becomes the active buffer.

Format

ret-status = LIB$PUT_BUFFER ([old-buffer])

ret-status = SCR$PUT_BUFFER ([old-buffer])

old-buffer
Optional address of a longword containing zero or the address of an area previously used as a screen buffer. If old-buffer is omitted or contains zero, the contents of the current screen buffer are output to the screen, the data length of the buffer is set to zero and buffer mode is terminated. If old-buffer is not zero, it is assumed to be the address of an area previously used as a screen buffer where the contents of the current active buffer are to be copied and then this area becomes the new active buffer.

Return Status

SS$_NORMAL
Routine successfully completed.

Example

The following FORTRAN example demonstrates the general pattern used to produce modular programs with the buffer mode of the terminal independent screen procedures. Each modular program should use LIB$SET_BUFFER and LIB$PUT_BUFFER in pairs. (See Section 3.2.10).

LIB$SET_BUFFER establishes the current buffering mode and saves the address of the previous buffer (if any). LIB$PUT_BUFFER reverts from the current buffering mode to the previous mode through the use of the previous buffer address, made available by the corresponding LIB$SET_BUFFER procedure call from the current modular program.

The previous buffering mode can imply: (1) buffering was in effect at the time of the call to LIB$SET_BUFFER in this modular program or (2) no buffering was in effect prior to this modular program. In the first case, the

contents of the current buffer are copied to the previous buffer and the previous buffer is reestablished as the active buffer. In the second case, buffer mode is terminated and the contents of the buffer are output to the terminal.

```
C     BUFFER USED FOR THIS MODULAR PROGRAM
C
      CHARACTER BUF*2000
C
C     LONGWORD TO SAVE ADDRESS OF PREVIOUS ACTIVE BUFFER
C
      INTEGER*4 OLDBUF
      .
      .
      .
C
C     ESTABLISH BUFFER MODE FOR THIS MODULAR PROGRAM
C     AND SAVE PREVIOUS BUFFER ADDRESS
C
      ISTAT = LIB$SET_BUFFER (BUF, OLDBUF)
      .
      .
      .
C
C     REVERT TO PREVIOUS BUFFER MODE - EITHER REVERT TO
C     OLD BUFFER OR OUTPUT CONTENTS OF BUFFER TO SCREEN
C
      ISTAT = LIB$PUT_BUFFER (OLDBUF)
      .
      .
      .
```

The following FORTRAN example demonstrates the use of the buffer mode for the terminal independent screen procedure calls in a modular manner. Both the main program and the subroutine initialize buffer mode. The subroutine could also be called by a main program that did not initialize buffer mode and the LIB$PUT_SCREEN procedure calls in the subroutine would be buffered during the execution of the subroutine and then output to the screen when the LIB$PUT_BUFFER procedure is called in the subroutine.

In addition the main program uses the second parameter on the LIB$SET_BUFFER procedure call as a good modular programming practice. In general, the LIB$SET_BUFFER and LIB$PUT_BUFFER procedures should be used in pairs to preserve a predictable buffer mode at any point in the modular programming environment.

LIB$SET_BUFFER and LIB$PUT_BUFFER procedures should not be called with the first parameter set to zero unless an error situation occurs which will prevent a modular program from returning to its caller. These procedure calls unconditionally force buffer mode to stop and the buffer to be displayed on the screen.

```
C
C     SUBPROGRAM
C
      SUBROUTINE BUFBUF ()
      INTEGER*4 IOLD                  ! Longword to save address of
                                      ! buffer previously in effect
      CHARACTER BUF2*2000             ! Buffer to be used during
                                      ! this subroutine for screen
                                      ! functions
      CHARACTER SUBTEXT*15
      DATA SUBTEXT/'SUBROUTINE TEXT'/
      ISTAT = LIB$SET_BUFFER(NUF2, IOLD) ! Initialize buffering
                                      ! and save caller's buffer
                                      ! address and copy caller's
                                      ! buffer to new buffer
C
C     Put 6 lines in buffer
C
      DO 500 I = 5,10
           J = I - 4
           ISTAT = LIB$PUT_SCREEN (SUBTEXT, I, J)
500   CONTINUE
C
C     Revert to previous buffer mode
C
      ISTAT = LIB$PUT_BUFFER (IOLD)
      RETURN
      END
C
C     MAIN PROGRAM
C
      PROGRAM BUF
      CHARACTER BUF1*3000             ! Buffer to be used by the main
                                      ! program for screen functions
      CHARACTER MAINTEXT*9
      INTEGER*4 OLDBUF                ! Longword to save the address
                                      !of the previous buffer used for
      DATA MAINTEXT/'MAIN TEXT'/      ! Screen functions (if any)
      ILINE=1
      ICOL=1
      ISTAT = LIB$ERASE_PAGE (ILINE,ICOL)  ! Clear the screen
      ISTAT = LIB$SET_BUFFER (BUF1, OLDBUF)  ! Initialize buffering
                                      ! \\ In this case, the main program
                                      ! is the first to
                                      ! initialize buffering \\
C
C     Put four lines in buffer
C
      DO 1000 I = 1,4
           ISTAT = LIB$PUT_SCREEN (MAINTEXT, I, I)
1000  CONTINUE
CALL  BUFBUF()                        ! Call a modular subroutine
                                      ! which also uses buffer mode
C
C     Put four more lines in buffer
C
      DO 2000 I = 11,14
           J = I - 10
           ISTAT = LIB$PUT_SCREEN (MAINTEXT, I, J)
2000  CONTINUE
      ISTAT = LIB$PUT_BUFFER (OLDBUF)  ! Revert to previous buffer
                                      ! mode \\ for the main program
                                      ! the previous buffer mode
                                      ! was a non-buffered mode.
                                      ! Therefore, the contents
                                      ! of the buffer are forced
                                      ! to the screen. \\
END
```

NOTE

The comments enclosed in backslashes are specific to this main program/subroutine configuration and should not be construed as an indication of the lack of modularity of the main program.

3.2.9 Put Text to Screen

LIB$PUT_SCREEN and SCR$PUT_SCREEN output the specified text on the screen beginning at a specified line and column. No carriage return or line feed control characters are inserted.

Format

ret-status = LIB$PUT_SCREEN (text [,line-no, col-no])

ret-status = SCR$PUT_SCREEN (text [,line-no, col-no])

text
Address of a descriptor of a character string that is output to the screen.

line-no
Optional address of a signed word integer containing the line number where the text begins. The default is the current line number. For SCR$PUT_SCREEN, the line number is passed by immediate value.

col-no
Optional address of a signed word integer containing the column number where the text begins. The default is the current column number. For SCR$PUT_SCREEN, the column number is passed by immediate value.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid argument. The number of parameters specified must be one or three.

LIB$_INVSCRPOS
Invalid screen position values. Line-no or col-no was zero.

Example

The following FORTRAN code fragment would put "LINE OF TEXT" in columns 1–12 of line 24:

```
CHARACTER TEXT*12
DATA TEXT/'LINE OF TEXT'/
ICOL = 1
ILINE = 24
ISTAT = LIB$PUT_SCREEN (TEXT, ILINE, ICOL)
```

3.2.10 Set/Clear Buffer Mode

LIB$SET_BUFFER and SCR$SET_BUFFER provide a means of reducing the number of queue I/O service calls (and possible network transfers), thereby, improving efficiency of the screen functions. These procedures set (or clear) buffer mode for the other terminal-independent screen procedures. While in buffer mode, the other screen procedures do not alter the appearance of the screen. Instead, a user-supplied buffer is maintained which represents the sequence of the other screen output functions that have occurred since buffer mode was last initialized. Clearing buffer mode causes the other screen output functions to have an immediate effect on the appearance of the terminal screen.

Format

ret-status = LIB$SET_BUFFER (buffer [,old-buffer])

ret-status = SCR$SET_BUFFER (buffer [,old-buffer])

buffer
Address of a descriptor of a modifiable fixed-length string which is used as the buffer for storage of the characters which would normally be sent to the terminal without buffering by the other screen output procedures until the next LIB$SET_BUFFER or LIB$PUT_BUFFER procedure call occurs. If buffer is omitted (or the argument list entry contains a zero), buffer mode is terminated and the buffer retains the buffered characters.

old-buffer
Optional address of a longword to contain the address of the previous buffer (if any). Old-buffer is most useful for subsequent use as an input parameter to LIB$PUT_BUFFER.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_SCRBUFOVF
Screen buffer overflow. The buffer is less than 12 bytes in length.

LIB$_INVARG
Invalid argument. Zero or more than two parameters were specified.

Example

It is a good programming practice to always use LIB$SET_BUFFER in conjunction with LIB$PUT_BUFFER. Please see the example in the LIB$PUT_BUFFER section which uses both of these procedures (Section 3.2.8).

3.2.11 Set Cursor to Character Position on Screen

LIB$SET_CURSOR and SCR$SET_CURSOR position the cursor to the specified line and column on the screen.

Format

ret-status = LIB$SET_CURSOR (line-no, col-no)

ret-status = SCR$SET_CURSOR (line-no, col-no)

line-no
: Address of a signed word integer containing the line number of the specified position. For SCR$SET_CURSOR, the line number is passed by immediate value.

col-no
: Address of a signed word integer containing the column number of the specified position. For SCR$SET_CURSOR, the column number is passed by immediate value.

Return Status

SS$_NORMAL
: Routine successfully completed.

LIB$_INVARG
: Invalid argument. The number of parameters specified must be two.

LIB$_INVSCRPOS
: Invalid screen position values. Line-no or col-no was zero.

Example

The following FORTRAN code fragment would move the cursor to column 7 of line 5:

```
ISTAT = LIB$SET_CURSOR (5, 7)
```

3.3 String Manipulation Procedures

This section describes string manipulation procedures, including character-oriented, string arithmetic, string-oriented, and translate string routines. Character-oriented routines include compare, locate, scan, skip, span, and transform functions. String-oriented routines include concatenate, copy, extract, match, replace and trim functions.

Some of the LIB$ procedures are named after the VAX-11 hardware instructions whose service they provide. The order of parameters is the same as the order in the corresponding hardware instruction.

LIB$ procedures indicate all errors using return status, whereas STR$ and OTS$ signal errors that are difficult to recover from and return truncated string errors using a return status.

See Section 2.5.3 for more details about string handling conventions for LIB$, OTS$, and STR$ procedures. Chapter 5 contains procedures for allocating dynamic strings. Chapter 7 contains procedures for syntactically analyzing strings.

3.3.1 String Conventions for LIB$, OTS$ and STR$ Facilities

Scalars are normally signed longwords passed by immediate value in registers to JSB entry points and passed by reference to CALL entry points. The signed longword allows negative values and access to all character positions.

Output string length parameters are normally unsigned words passed by immediate value in registers from JSB entry points and passed by reference from CALL entry points.

Strings are passed by descriptor. The LIB$, OTS$, and STR$ procedures accept string descriptors for parameters specified as strings. The routines write strings according to the semantics of the descriptor for all classes defined by the VAX-11 Procedure Calling Standard. The routines can only read strings that look like fixed-length string descriptors. That is, the length field is a word containing the length of the string in bytes and the pointer field is a pointer to the first character of the string. Routines that read and write a string must have an input parameter and an output parameter. These parameters can reference the same string. The only modify access permitted on strings is for STR$APPEND and STR$PREFIX, both specialized cases of STR$CONCAT.

OTS$ and STR$ procedures signal errors that are programming errors or prevent the routine from doing any useful work. LIB$ procedures return severe errors as a completion status. These errors are:

LIB$	OTS$	STR$	
FATERRLIB	FATINTERR	FATINTERR	fatal internal error
INVSTRDES	INVSTRDES	ILLSTRCLA	illegal string class
INSVIRMEM	INSVIRMEM	INSVIRMEM	insufficient virtual memory
STRIS_INT	STRIS_INT	STRIS_INT	string is interlocked

To save space the preceding errors are listed by name only in each procedure description. Other errors, more specific to a particular procedure, are listed and explained under each procedure description.

All errors are returned as a completion status by LIB$ procedures. Consequently, when an output string must be truncated and its length depends solely on input parameters (hence under control of the calling program), LIB$ procedures return a qualified success (LIB$__STRTRU) instead of an error. This corresponds to the semantics of many higher level languages that do not consider truncation as an error. However, when the length of an output string is not completely under program control, such as for LIB$GET__INPUT, a particular error status is returned.

Since most errors are signaled by STR$ procedures, truncation is returned as an error status with warning severity (STR$__TRU). Range errors are returned as qualified success.

In two routines, the function value is not a status. STR$COMPARE returns a logical value and STR$POSITION returns a character position. If STR$APPEND and STR$PREFIX return, they always return success.

The longest string possible is 65,535 characters. When referring to character positions in a string, character positions start at 1. When specifying substrings by character positions M to N, the following evaluation rules apply.

1. If M <1, M is considered to equal 1.
2. If M > the length of the source string, the substring specified is the null string.
3. If N > the length of the source string, N is considered to equal the length of the source string.
4. If M > N, the substring specified is the null string.

When specifying substrings by length L, if L < 0, the substring specified is the null string. If any of these evaluation rules apply, the range error - qualified success status is returned (with the exception noted for STR$POSITION).

A null string is a descriptor with zero length (DSC$W__LENGTH = 0). A descriptor with a nonzero length and a zero pointer is an error and yields unspecified results.

3.3.2 Character Oriented Procedures

The following procedures return a single character or function value or have a parameter that represents a single character, byte or ASCII code.

STR$COMPARE

3.3.2.1 Compare Two Strings — STR$COMPARE compares two strings for the same contents. If the strings are unequal in length, the shorter string is considered as if it is blank filled to the length of the longer string before the comparison is made. The return function value is -1 if string1 is less than string2, 0 if string1 equals string2 and 1 if string1 is greater than string2.

Format

match = STR$COMPARE (src1-str, src2-str)

src1-str
Address of string1 string descriptor.

src2-str
Address of string2 string descriptor.

match
A signed longword to contain the return function value:

-1 string1 < string2
 0 string1 = string2
 1 string1 > string2

Example

If the following BASIC code fragment were executed, the function values would be; I% = -1, J% = 0, K% = 1, L% = 0:

```
EXTERNAL INTEGER FUNCTION STR$COMPARE
I% = STR$COMPARE('ABC', 'XYZ')
J% = STR$COMPARE('MNO', 'MNO')
K% = STR$COMPARE('XYZ', 'ABC')
L% = STR$COMPARE('MNO', 'MNO  ')
```

STR$COMPARE_EQL

3.3.2.2 Compare Two Strings for Equal — STR$COMPARE_EQL compares two strings for the same length and contents. The return function value is 0 if the two strings are identical, and 1 if they are not.

Format

match = STR$COMPARE_EQL (src1-str, src2-str)

src1-str
Address of string1 string descriptor.

src2-str
Address of string2 string descriptor.

match
A longword containing the return function value:

0 length of string1 = length of string2 and
contents of string1 = contents of string2

1 length of string1 <> length of string2 or
contents of string1 <> contents of string2

3.3.2.3 Locate a Character — LIB$LOCC locates a character in a string by comparing successive bytes in the string with the character specified. The string is specified by the string descriptor. The string continues to be searched until the character is found or the string has no more characters. The relative position of the first equal character, or zero, is returned as an index. If the string has a length of zero, then a zero is returned indicating that the character was not found.

Format

index = LIB$LOCC (char-str, src-str)

char-str
Address of string descriptor of character to be found.

src-str
Address of string descriptor of string to be searched.

index
Unsigned longword containing the relative position of the first equal character or zero if no match is found.

NOTE

Only the first character of char is used, and its length is not checked.

Examples

In FORTRAN, I is set to 3, and J to 0:

```
I = LIB$LOCC ('C', 'ABCDE')
J = LIB$LOCC ('Z', 'ABDCE')
```

The following FORTRAN function returns the number of spaces in string:

```
INTEGER*4 FUNCTION COUNT_SPACES (STRING)
INTEGER*4 REL_POS, END_POS
CHARACTER *(*) STRING
COUNT_SPACES = 0            ! Assume no spaces
BEG_POS = 1
END_POS = LEN(STRING)
DO WHILE (BEG_POS .LE. END_POS)
     REL_POS = LIB$LOCC(' ', STRING (BEG_POS:END_POS))
     IF (REL_POS.EQ.0) RETURN
     COUNT_SPACES = COUNT_SPACES + 1
     BEG_POS = BEG_POS + REL_POS
ENDDO
RETURN
END
```

3.3.2.4 Return Length of String as Longword Value — LIB$LEN returns the length of the string parameter as a longword value. The maximum length of a VAX/VMS string is 65,535 characters.

Format

str-len = LIB$LEN (src-str)

src-str
Address of the source string descriptor.

str-len
Length of the source string. The 16-bit length field in the source string descriptor is copied and zero-extended to 32-bits.

Notes

The BASIC and FORTRAN intrinsic function LEN generates equivalent in-line code at run time.

Example

Although LIB$LEN could be called in MACRO, the following code sequence is equivalent to a call to LIB$LEN for dynamic, fixed-length and unspecified class strings:

```
$DSCDEF                     ; define descriptor symbols (DSC$...)
MOVZWL STRING+DSC$W_LENGTH, R0    ; R0 = length of string
```

where:

STRING	is the address of the string descriptor
DSC$W_LENGTH	is the offset of the word within the descriptor (0) containing the length.

3.3.2.5 Return Relative Position of Substring — STR$POSITION returns an index, which is the relative position of the first occurrence of a substring in the source string. The value returned is an unsigned integer longword. The relative character positions are numbered 1, 2, ..., n. Thus, zero is a unique number meaning that the substring was not found.

If the substring has a zero length, one is returned by LIB$INDEX and LIB$MATCHC indicating a found substring whether or not the source string has a zero length, while the minimum of start-pos and the length of src-str 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.

The order of parameters for LIB$INDEX corresponds to the practice in higher level languages, while that of LIB$MATCHC parallels the VAX-11 MATCHC instruction.

Format

index = LIB$INDEX (src-str, sub-str)

index = LIB$MATCHC (sub-str, src-str)

index = STR$POSITION (src-str, sub-str [,start-pos])
JSB entry point: STR$POSITION_R6

src-str
Address of source string descriptor to be searched.

sub-str
Address of substring descriptor to be found.

start-pos
Optional address of a longword containing the relative starting position in the source string to begin the search.

index
Unsigned longword indicating relative position of the first character of the substring if found, or zero if not found.

Implicit Inputs (for STR$POSITION_R6 only)

R0
Address of source string descriptor.

R1
Address of substring descriptor.

R2
A longword containing the relative starting position in the source string to begin the search. Note this is required for the JSB entry point.

Notes

The FORTRAN compiler generates the call to LIB$INDEX for the INDEX built-in function.

Examples

The following FORTRAN function returns the number of occurrences of SUB_STR IN STRING.

```
      FUNCTION COUNT_SUB(STRING, SUB_STR)
      CHARACTER *(*) STRING, SUB_STR
      INTEGER*4 COUNT_SUB, REL_POS, BEG_POS, END_POS
      COUNT_SUB = 0
      BEG_POS = 1
      END_POS = LEN(STRING)
10    REL_POS = STR$POSITION (STRING(BEG_POS:END_POS), SUB_STR)
      IF (REL_POS .GT. 0) THEN
          COUNT_SUB = COUNT_SUB + 1
          BEG_POS = BEG_POS + REL_POS
          GO TO 10
      ENDIF
      RETURN
      END
```

In FORTRAN, I is assigned value 1, J = 3, and K = 0:

```
I = LIB$MATCHC ('ABC', 'ABCDEF')
J = LIB$MATCHC ('CDE', 'ABCDEF')
K = LIB$MATCHC ('XYZ', 'ABCDEF')
```

3.3.2.6 Scan Characters — LIB$SCANC is used to find a specified set of characters in the source string. It uses successive bytes of the string specified by the source descriptor to index into a table. The byte selected from the table is ANDed with the mask byte. The operation continues until the result of the AND is a nonzero value. The relative position of the character in the source string that terminated the operation is returned if such a character is found. Otherwise, zero is returned. If the source string has a zero length, then a zero is returned.

Format

index = LIB$SCANC (src-str, table-arr, mask)

src-str
Address of source string descriptor.

table-arr
Address of unsigned byte array.

mask
Address of the byte containing the mask.

index
Unsigned longword containing the relative position of the character in the source string that terminated the operation or zero.

Example

The following FORTRAN example uses LIB$SCANC to scan a table. In this example, J=1, K=0, L=3, M=3:

```
BYTE TABLE(0:255)
DATA TABLE /48*0, 3*1, 2, 6*1, 198*2/
J=LIB$SCANC('572A614',TABLE,5)
K=LIB$SCANC('ABCD',TABLE,5)
L=LIB$SCANC('**12',TABLE,1)
M=LIB$SCANC('12A3',TABLE,2)
```

3.3.2.7 Skip Characters — LIB$SKPC compares a given string with a given character and returns the relative position of the first nonequal character as an index. The character is compared with successive characters of the specified string until an inequality is found or the string is exhausted. The relative position of the unequal character or zero is returned. If the source string has a zero length, then a zero is returned.

Format

index = LIB$SKPC (char-str, src-str)

char-str
Address of string descriptor of the character to be found.

src-str
Address of string descriptor of the string to be searched.

index
Unsigned longword returned specifying the relative position of the first unequal character, or zero if one was not found.

Notes

Only the first character of char-str is used, and the length is not checked.

Example

In FORTRAN, I would be set to 2 and J to 0:

```
I = LIB$SKPC (' ', 'ABC')
J = LIB$SKPC ('A', 'AAA')
TYPE*,I,J
```

3.3.2.8 Span Characters — LIB$SPANC is used to skip a specified set of characters in the source string. It uses successive bytes of the string specified by the source descriptor to index into a table. The byte selected from the table is ANDed with mask byte. The operation continues until the result of the AND is zero. The relative position of the character in the source string that terminated the operation is returned if such a character is found. Otherwise, zero is returned. If the source string has a zero length, then a zero is returned.

Format

index = LIB$SPANC (src-str, table-arr, mask)

src-str
Address of source string descriptor.

table-arr
Address of unsigned byte array.

mask
Address of the byte containing the mask.

index
Unsigned longword containing the relative position of the character in the source string that terminated the operation or 0.

Example

The following FORTRAN example uses LIB$SPANC to index a table. In this example, J=1, K=0, L=1, M=1:

```
BYTE TABLE(0:255)
DATA TABLE /48*0, 3*1, 2, 6*1, 198*2/
J=LIB$SPANC ('572A614',TABLE,5)
K=LIB$SPANC ('2048',TABLE,5)
L=LIB$SPANC ('A135',TABLE,1)
M=LIB$SPANC ('12A3',TABLE,2)
```

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

Format

ret-status = LIB$CHAR (one-char-str, ascii-code)

one-char-str
: Address of the string descriptor (fixed-length or dynamic) to receive one character result. (This is an output parameter.)

ascii-code
: Address of the unsigned byte integer ASCII character code to be transformed to an ASCII string.

Return Status

SS$_NORMAL
: Procedure successfully completed.

LIB$_STRTRU
: Procedure successfully completed; string truncated. Fixed-length destination string descriptor could not contain all of the characters.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

Notes

LIB$CHAR is the inverse of LIB$ICHAR.

LIB$CHAR is not binary to ASCII conversion. It merely interprets ASCII–code as an ASCII character code and converts it to a string.

Since the output string is the first argument, this procedure can be called as either a subroutine of two arguments or a string function of one argument. The FORTRAN compiler generates equivalent code in-line for the CHAR built-in function rather than calling LIB$CHAR.

Examples

The following FORTRAN code fragment prints out the number of occurrences of each ASCII code indicated by character count in the INTEGER*2 vector CHAR_COUNT.

```
      CHARACTER*1 LIB$CHAR, INPUT*80
      INTEGER*2 CHAR_COUNT (0:255)
      TYPE *, 'TYPE STRING TO BE ANALYZED:'
      ACCEPT 50, INPUT
50    FORMAT (A)
      DO 2 I = 0, 255
2     CHAR_COUNT(I) = 0
      DO 5 I = 1, LEN (INPUT)
          J = ICHAR (INPUT (I:I))
5         CHAR_COUNT (J) = CHAR_COUNT (J) + 1
      DO 10 I = 0, 255
          IF (CHAR_COUNT (I).GT.0) THEN
              WRITE (6,100) CHAR_COUNT (I), LIB$CHAR (I)
100           FORMAT ('THERE WERE', I5, ' ', A1, 'S')
          END IF
10    CONTINUE
      END
```

LIB$CHAR could be called in MACRO as follows:

```
PUSHAB   ASCII_CODE      ; push address of byte
                         ; containing ASCII code as
                         ; second parameter.
PUSHAQ   ONE_CHAR_STR    ; push address of output string
                         ; descriptor (1st parameter)
CALLS    #2, LIB$CHAR
```

However, the following code sequence is equivalent for fixed-length strings:

```
    $DSCDEF                  ; define descr symbols (DSC$ ...)
    MOVAQ   ONE_CHAR_STR, R0
; R0 = adr of string desc
    MOVC5   #1, ASCII_CODE, #A' ', DSC$_LENGTH(R0), -
            @DSC$A_POINTER(R0)
```

3.3.2.10 Transform First Character of String to Longword Value —

LIB$ICHAR transforms the first character of a string to an 8-bit ASCII integer value extended to a longword value.

Format

first-char-value = LIB$ICHAR (src-str)

src-str
Address of the string descriptor.

first-char-value
First character of the string returned as an 8-bit ASCII value extended to a longword value.

Notes

The FORTRAN intrinsic function ICHAR generates equivalent code inline. If the string has zero length, a zero is returned. Zero-length strings are not permitted in FORTRAN.

Examples

The following FORTRAN subroutine adds 1 to the corresponding entry in the INTEGER*2 vector CHAR-COUNT for each ASCII character occurring in the character string STRING, passed as a parameter.

```
      SUBROUTINE  FLAG_CHAR (STRING)
      CHARACTER *(*)  STRING
      INTEGER*2 CHAR_COUNT(0:255)
      DO 10 I=1, LEN(STRING)
          J = LIB$ICHAR(STRING(I:I))
          CHAR_COUNT(J) = CHAR_COUNT(J) + 1
10    CONTINUE
      RETURN
      END
```

Although LIB$ICHAR can be called from MACRO, the following code sequence is equivalent to a call to LIB$ICHAR.

```
$DSCDEF                     ; define desc symbols (DSC$ ...)
MOVAQ   STRDSC, R0          ; R0 = adr of string desc
MOVZBL  @DSC$A_POINTER(R0),R0 ; R0 = 1st char in string
```

3.3.3 String Arithmetic Procedures

The following procedures perform string arithmetic on arbitrary length numbers represented as three separate parameters:

- A sign bit (passed by reference)
- A signed longword power of 10 (passed by reference)
- A text string consisting solely of ASCII digits (passed by descriptor)

The maximum length of the text string is 65,535 bytes. The mathematical functions provided are add, multiply, reciprocal and truncate and round.

STR$ADD

3.3.3.1 Add Two Decimal Strings — STR$ADD adds two decimal strings (A,B) and places the sum in the result string (C).

Format

```
ret-status = STR$ADD (a-sign-adr, a-exp-adr, a-digits,
                      b-sign-adr, b-exp-adr, b-digits,
                      c-sign-adr, c-exp-adr, c-digits)
```

a-sign-adr
Address of a bit containing the sign of operand a (0 is positive).

a-exp-adr
Address of a signed longword containing the power of 10 by which the a-digits have to be multiplied to get the absolute value of operand a.

a-digits
Address of the a-digits string descriptor. The string must be an unsigned decimal number.

b-sign-adr
Address of a bit containing the sign of operand b (0 is positive).

b-exp-adr
Address of a signed longword containing the power of 10 by which the b-digits have to be multiplied to get the absolute value of operand b.

b-digits
Address of the b-digits string descriptor. The string must be an unsigned decimal number.

c-sign-adr
Address of a bit to contain the sign of result c (0 is positive).

c-exp-adr
Address of a signed longword to contain the power of 10 by which the c-digits have to be multiplied to get the absolute value of result c.

c-digits
Address of the c-digits string descriptor (fixed-length or dynamic). The string will be an unsigned decimal number.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

STR$_WRONUMARG
Wrong number of arguments.

Example

See Section 3.3.3.4.

STR$MUL

3.3.3.2 Multiply Two Decimal Strings — STR$MUL multiplies two decimal strings (A,B) and places the product in the result string (C).

Format

```
ret-status = STR$MUL (a-sign-adr, a-exp-adr, a-digits,
                      b-sign-adr, b-exp-adr, b-digits,
                      c-sign-adr, c-exp-adr, c-digits)
```

See Section 3.3.3.1 for parameter descriptions.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$__FATINTERR
STR$__ILLSTRCLA
STR$__INSVIRMEM
STR$__STRIS__INT

STR$__WRONUMARG
Wrong number of arguments.

Example

See Section 3.3.3.4.

STR$RECIP

3.3.3.3 Reciprocal of a Decimal String — STR$RECIP takes the reciprocal of decimal string (A) to the precision limit specified by decimal string (B) and places the result in decimal string (C).

Format

```
ret-status = STR$RECIP (a-sign-adr, a-exp-adr, a-digits,
                        b-sign-adr, b-exp-adr, b-digits,
                        c-sign-adr, c-exp-adr, c-digits)
```

See Section 3.3.3.1 for parameter descriptions.

Return Status

SS$__NORMAL
Routine successfully completed.

STR$__TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$__DIVBY__ZER
Division by zero.

STR$__FATINTERR
STR$__ILLSTRCLA
STR$__INSVIRMEM
STR$__STRIS__INT

STR$__WRONUMARG
Wrong number of arguments.

Example

See Section 3.3.3.4.

3.3.3.4 Round or Truncate a Decimal String — STR$ROUND rounds or truncates a decimal string (A) to a specified number of significant digits and places the result in decimal string (C).

Format

```
ret-status = STR$ROUND (places, trunc-flg,
                        a-sign-adr, a-exp-adr, a-digits,
                        c-sign-adr, c-exp-adr, c-digits)
```

places
Address of a longword containing the maximum number of decimal digits to retain in the result.

trunc-flg
Address of a bit containing the function flag; 0 means round, 1 means truncate.

See Section 3.3.3.1 for additional parameter descriptions.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

STR$_WRONUMARG

Wrong number of arguments.

Example

Suppose A = -1000; that is ASIGN = 1, AEXP = 3 and ADIGITS = '1'.
Suppose also B = .0002; that is BSIGN = 0, BEXP = -4 and BDIGITS = '2'.

Then, applying the string arithmetic functions, you would get the following results:

	CSIGN	CEXP	CDIGITS	value of C
A + B	1	-4	'9999998'	-999.9998
rounded 2,0	1	2	'10'	-1000.
A * B	1	-1	'2'	-.2
rounded 2,0	1	-1	'2'	-.2
reciprocal of A to precision B	1	-3	'1'	-.001
rounded 2,0	1	-3	'1'	-.001

A BASIC program to produce the C-elements in the preceding chart is:

```
100     REM STR$ ARITHMETIC SAMPLE PROGRAM
200     ASIGN% = 1%
300     AEXP% = 3%
400     ADIGITS$ = '1'
500     BSIGN% = 0%
600     BEXP% = -4%
700     BDIGITS$ = '2'
800     CSIGN% = 0%
900     CEXP% = 0%
1000    CDIGITS$ = '0'
1010    PRINT "A = "; ASIGN%; AEXP%; ADIGITS$
1020    PRINT "B = "; BSIGN%; BEXP%; BDIGITS$
1100    CALL STR$ADD   (ASIGN%, AEXP%, ADIGITS$, &
                        BSIGN%, BEXP%, BDIGITS$, &
                        CSIGN%, CEXP%, CDIGITS$)
1200    PRINT "STR$ADD; C = "; CSIGN%; CEXP%; CDIGITS$
1210    CALL STR$ROUND (2%, 0%, CSIGN%, CEXP%, CDIGITS$, &
                                CSIGN%, CEXP%, CDIGITS$)
1220    PRINT "STR$ROUND (2,0); C = "; CSIGN%; CEXP%; CDIGITS$
1300    CALL STR$MUL   (ASIGN%, AEXP%, ADIGITS$, &
                        BSIGN%, BEXP%, BDIGITS$, &
                        CSIGN%, CEXP%, CDIGITS$)
1400    PRINT "STR$MUL; C = "; CSIGN%; CEXP%; CDIGITS$
1410    CALL STR$ROUND (2%, 0%, CSIGN%, CEXP%, CDIGITS$, &
                                CSIGN%, CEXP%, CDIGITS$)
1420    PRINT "STR$ROUND (2,0); C = "; CSIGN%; CEXP%; CDIGITS$
1500    CALL STR$RECIP (ASIGN%, AEXP%, ADIGITS$, &
                        BSIGN%, BEXP%, BDIGITS$, &
                        CSIGN%, CEXP%, CDIGITS$)
1600    PRINT "STR$RECIP; C = "; CSIGN%; CEXP%; CDIGITS$
1610    CALL STR$ROUND (2%, 0%, CSIGN%, CEXP%, CDIGITS$, &
                                CSIGN%, CEXP%, CDIGITS$)
1620    PRINT "STR$ROUND (2,0); C = "; CSIGN%; CEXP%; CDIGITS$
1900    END
```

3.3.4 String Oriented Procedures

The following procedures return a string or substring that is a function of one or more input strings. See Section 3.3.3 for string arithmetic procedures.

3.3.4.1 Append a String — STR$APPEND appends a source string to the end of the destination string. The destination string must be dynamic.

Format

ret-status = STR$APPEND (dst-str, src-str)

dst-str
Address of the destination string descriptor (dynamic).

src-str
Address of the source string descriptor.

Return Status

SS$_NORMAL
Routine successfully completed.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

STR$CONCAT

3.3.4.2 Concatenate Two or More Strings — STR$CONCAT takes up to 254 input strings and concatenates them into a result string. The strings can be of any class and data type, providing that the length field of the descriptor indicates the length of the string in bytes. A warning status is returned if one or more input characters was not copied to the result string. The maximum length of a string is 65,535 bytes.

Format

ret-status = STR$CONCAT (dst-str, src1-str, src2-str [,src3-str ..., srcn-str])

dst-str
Address of the destination string descriptor (fixed-length or dynamic).

srcn-str
Address of source string n descriptor.

Return Status

SS$_NORMAL
Routine successfully completed. All characters in the input strings were copied into the destination string.

STR$_TRU
Warning. String truncated. One or more input characters were not copied into the destination string. This can happen when the destination is a fixed-length string.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

STR$_STRTOOLON
String length exceeds 65,535 bytes.

STR$_WRONUMARG
Wrong number of arguments.

Example

The following BASIC statements (when executed) would yield X$ = 'ABCD':

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

STR$COPY_DX

3.3.4.3 Copy a Source String to a Destination String — Three sets of copy routines are provided for copying a source string to a destination string. These are useful for writing procedures that return strings according to the semantics (fixed-length or dynamic) indicated by the calling program in the destination descriptor. The three sets follow the conventions for LIB$, OTS$, and STR$ facilities:

- LIB. All conditions are returned as a status in R0 (no signals); truncation is a qualified success condition value (bit 0 = 1). Input scalars are passed by reference.
- OTS. All conditions except truncation are signaled; R0:R5 contain results of MOVC5 instruction. Input scalars are passed by immediate value.
- STR. All conditions except truncation are signaled; truncation is returned as a warning condition value (bit 0 = 0) in R0. Input scalars are passed by reference.

Within each set there is an entry point that passes the source string by descriptor and a second one that passes the source string by reference preceded by a length parameter. In addition equivalent JSB entry points are provided, with R0 being the first parameter, R1 the second, and R2 the third, if any. The length parameter is passed in bits 15:0 of the appropriate register.

For LIB$ and OTS$, the destination parameter is last; for STR$, the destination parameter is first so it can be called as a string function (ignoring truncation status) or as a status value returning function when the calling program wishes to detect string truncation. Depending on the class of the destination string, these actions occur:

Class Field	Action
DSC$K_CLASS_S,Z (fixed length,unspecified)	Copy the source string. If needed, space fill or truncate on the right.
DSC$K_CLASS_D (dynamic)	If the area specified by the destination descriptor is large enough (but not too large) 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 or is too large, return the previous destination descriptor space allocation (if any) and then allocate the amount of space dynamically needed. Copy the source string and set the new length and address in one destination descriptor.

Formats

Source by descriptor:

ret-status = LIB$SCOPY_DXDX (src-str, dst-str)
JSB entry point: LIB$SCOPY_DXDX6

unmoved-src = OTS$SCOPY_DXDX (src-str, dst-str)
JSB entry point: OTS$SCOPY_DXDX6

ret-status = STR$COPY_DX (dst-str, src-str)
JSB entry point: STR$COPY_DX_R8

Source by reference:

ret-status = LIB$SCOPY_R_DX (src-len-adr, src-adr, dst-str)
JSB entry point: LIB$SCOPY_R_DX6

unmoved-src = OTS$SCOPY_R_DX (src-len, src-adr, dst-str)
JSB entry point: OTS$SCOPY_R_DX6

ret-status = STR$COPY_R (dst-str, src-len-adr, src-adr)
JSB entry point: STR$COPY_R_R8

dst-str
: Address of the destination string descriptor. The class field determines the appropriate action. The length field (DSC$W_LENGTH) or both the address (DSC$A_POINTER) and length fields can be modified if the string is dynamic. (This is an output parameter.)

src-str
Address of the string descriptor specifying the length and address of the source string. The descriptor class can be unspecified, fixed-length, or dynamic. The data type field can be any data type for which the length field is in units of bytes.

unmoved-src
Number of unmoved source string bytes, if the source string length is greater than the destination string length; otherwise zero.

src-len-adr
Address of an unsigned word containing the length of the source string.

src-len
An unsigned word containing the length of the source string (passed by immediate value).

Implicit Inputs (JSB entry):

	src-str	src-len-adr	src-str-adr	dst-str
LIB$SCOPY_DXDX6	R0			R1
OTS$SCOPY_DXDX6	R0			R1
STR$COPY_DX_R8	R1			R0
LIB$SCOPY_R_DX6		R0<15:0>	R1	R2
OTS$SCOPY_R_DX6		R0<15:0>	R1	R2
STR$COPY_R_R8		R1<15:0>	R2	R0

Return Status

SS$_NORMAL
Procedure successfully completed. All characters in the input string were copied to the destination string.

LIB$_STRTRU
Procedure successfully completed. String truncated. Fixed-length destination string descriptor could not contain all of the characters copied from the source string.

STR$_TRU
Warning. String truncated. Fixed-length destination string descriptor could not contain all of the characters copied from the source string.

Messages

LIB$_INSVIRMEM, LIB$_INVSTRDES, LIB$_STRIS_INT, LIB$_FATERRLIB
OTS$_INSVIRMEM, OTS$_INVSTRDES, OTS$_STRIS_INT, OTS$_FATINTERR
STR$_INSVIRMEM, STR$_ILLSTRCLA, STR$_STRIS_INT, STR$_FATINTERR

Examples

The following FORTRAN subroutine returns the data as a string using the string semantics specified by the caller. The parameter STRING_DSC is dimensioned as an 8-byte array instead of CHARACTER. Just before returning to the caller, the FORTRAN subroutine copies the CHARACTER DATE_STR to the passed STRING_DSC.

```
SUBROUTINE RET_DATE_STR (STRING_DSC)
BYTE STRING_DSC(8)
CHARACTER*9 DATE_STR
.
.
.
CALL DATE (DATE_STR)        !Copy 9-character data to DATE_STR
CALL STR$COPY_DX (%DESC(STRING_DSC), DATE_STR)
RETURN
END
```

In MACRO, a typical call from procedure PROC would be:

```
        $DSCDEF                 ; define DSC$ descr symbols
DSTDSC: .WORD 0                 ; filled by STR$COPY_R
        .BYTE DSC$K_DTYPE_T     ; data type is ASCII text
        .BYTE DSC$K_CLASS_D     ; class is dynamic string
        .LONG 0                 ; adr of string filled in
SRC:    .ASCII /Fourscore and seven years ago/
SRCLEN= .-SRC                   ; length of source string
LEN:    .WORD   SRCLEN

        .ENTRY  PROC, ^M< >     ; save only what PROC uses
        .
        .
        .
        PUSHAB  SRC             ; par3 = adr of source string
        PUSHAW  LEN             ; par2 = adr of src str length
        PUSHAQ  DSTDSC          ; par1 = adr of dest descr
        CALLS   #3, STR$COPY_R
        BLBC    R0, TRUNC       ; test for truncation
```

The JSB form would be:

```
    .ENTRY  PROC, ^M<R2,R3,R4,R5,R6,R7,R8> ; save at least
                            ; R2:R8 in stack on entry
    .
    .
    .
    MOVAQ   DSTDSC, R0      ; R0 = adr of dest string descr
    MOVW    SRCLEN, R1      ; R1 = length of source string
    MOVAB   SRC, R2         ; R2 = adr of source string
    JSB     STR$COPY_R_R8   ; copy source to destination
    BLBC    R0, TRUNC       ; test for truncation
```

3.3.4.4 Extract a Substring of a String — The following procedures copy a substring of a source string into a destination string. Each procedure has a different method of defining the substring.

STR$LEN_EXTR defines the substring by specifying the relative starting position in the source string and the number of characters to be copied.

STR$POS_EXTR defines the substring by specifying the relative starting and ending positions in the source string.

STR$LEFT defines the substring by specifying the relative ending position in the source string. The relative starting position in the source string is one. This is a variation of STR$POS_EXTR.

STR$RIGHT defines the substring by specifying the relative starting position. The relative ending position is equal to the length of the source string. This is a variation of STR$POS_EXTR.

Format

ret-status = STR$LEN_EXTR (dst-adr, src-adr, start-pos, length)
JSB entry point: STR$LEN_EXTR_R8

ret-status = STR$POS_EXTR (dst-adr, src-adr, start-pos, end-pos)
JSB entry point: STR$POS_EXTR_R8

ret-status = STR$LEFT (dst-adr, src-adr, end-pos)
JSB entry point: STR$LEFT_R8

ret-status = STR$RIGHT (dst-adr, src-adr, start-pos)
JSB entry point: STR$RIGHT_R8

dst-adr
Address of destination string descriptor (fixed-length or dynamic).

src-adr
Address of source string descriptor.

start-pos
Address of a signed longword containing the relative starting position in the source string.

end-pos
Address of a signed longword containing the relative ending position in the source string.

length
Address of a longword containing the number of characters to be copied to the destination string.

Implicit Inputs (JSB entries only)

R0
Address of destination string descriptor.

R1
Address of source string descriptor.

R2
A longword containing the relative starting position in the source string except for STR$LEFT_R8 where it is a longword containing the relative ending position in the source string.

R3
For STR$LEN_EXTR_R8, a longword containing the number of characters to be copied to the destination string.
For STR$POS_EXTR_R8, a longword containing the relative ending position in the source string.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_ILLSTRPOS
Routine successfully completed. A character position parameter referenced a character position outside the appropriate string. A default value described in the string conventions was used.

STR$_ILLSTRSPE
Routine successfully completed. End-pos was less than start-pos or length was too long for appropriate string. Default values described in the string conventions section were used.

STR$_NEGSTRLEN
Routine successfully completed. The length parameter contained a negative value; zero was used.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters copied from the source string.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

Example

In BASIC, assuming SRC$ = 'ABCD', the following statements would yield, M$ = 'BC', N$ = 'BC', O$ = 'AB', and P$ = 'CD':

```
EXTERNAL INTEGER FUNCTION STR$LEN_EXTR, &
STR$POS_EXTR, STR$LEFT, STR$RIGHT
STATUS% = STR$LEN_EXTR (M$, SRC$, 2%, 2%)
STATUS% = STR$POS_EXTR (N$, SRC$, 2%, 3%)
STATUS% = STR$LEFT (O$, SRC$, 2%)
STATUS% = STR$RIGHT (P$, SRC$, 3%)
```

3.3.4.5 Generate a String — STR$DUPL_CHAR generates a string containing n duplicates of the input character.

Format

ret-status = STR$DUPL_CHAR (dst-adr [,length [,char]])
JSB entry point: STR$DUPL_CHARR8

dst-adr
Address of the destination string descriptor.

length
Optional address of a signed longword containing the number of times char will be duplicated. The default is one.

char
Optional address of a byte containing an ASCII character. The default is a space.

Implicit Inputs (JSB entries only)

R0
Address of the destination string descriptor.

R1
A signed longword containing the number of times char will be duplicated.

R2
<8:0> byte containing an ASCII character.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_NEGSTRLEN
Routine successfully completed. The length parameter contained a negative value; zero was used.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

STR$_STRTOOLON
String length exceeds 65,535 bytes.

Example

In BASIC, the following statements would yield X$ = 'AAAA' upon execution:

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

STR$PREFIX

3.3.4.6 Prefix a String — STR$PREFIX inserts the source string at the beginning of the destination string. The destination string must be dynamic.

Format

ret-status = STR$PREFIX (dst-str, src-str)

dst-str
Address of the destination string descriptor (dynamic).

src-str
Address of the source string descriptor.

Return Status

SS$_NORMAL
Routine successfully completed.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

Example

In BASIC, the following statements would yield D$ = 'ABCDEFG' on execution:

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

3.3.4.7 Replace a Substring — STR$REPLACE copies a source string to a destination string, replacing a substring with another substring. The replaced substring is specified by the starting and ending positions.

Format

ret-status = STR$REPLACE (dst-str, src-str, start-pos, end-pos, rpl-str)
JSB entry point: STR$REPLACE_R8

dst-str
Address of destination string descriptor (fixed-length or dynamic).

src-str
Address of source string descriptor.

start-pos
Address of a signed longword containing the relative starting position in the source string of the substring to be replaced.

end-pos
Address of a signed longword containing the relative ending position in the source string of the substring to be replaced.

rpl-str
Address of the replacement string descriptor.

Implicit Inputs (JSB entries only)

R0
Address of destination string descriptor.

R1
Address of source string descriptor.

R2
A signed longword containing the relative starting position in the source string of the substring to be replaced.

R3
A signed longword containing the relative ending position in the source string of the substring to be replaced.

R4
Address of the replacement string descriptor.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_ILLSTRPOS
Routine successfully completed. A character position parameter referenced a character position outside the appropriate string. A default value described in the string conventions was used.

STR$_ILLSTRSPE
Routine successfully completed. End-pos was less than start-pos or length was too long for appropriate string. Default values described in the string conventions were used.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

Example

In BASIC, the following statements would yield D$ = 'AXYZD' on execution:

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

3.3.4.8 Trim Trailing Blanks and Tabs — STR$TRIM copies a source string to a destination string and deletes the trailing blank and tab characters.

Format

ret-status = STR$TRIM (dst-str, src-str [,out-len])

dst-str
Address of the destination string descriptor (fixed-length or dynamic).

src-str
Address of the source string descriptor.

out-len
Optional address of a word to be set to the number of bytes written into dst-str, not counting padding in the case of a fixed string. If the input string is truncated to the size specified in the dst-str description, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of dst-str.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

Example

In BASIC, the following statements would yield D$ = 'ABC' on execution:

```
EXTERNAL INTEGER FUNCTION STR$TRIM
D$ = 'ABC'
STATUS% = STR$TRIM (D$, D$)
```

3.3.5 Translate String Functions

The following functions return a string that is an altered form of the source string.

3.3.5.1 Move Translated Characters — LIB$MOVTC moves the source string character-by-character to the destination string after translating each one using the specified translation table.

Each character in the source is used as an index into the translation table. The byte found is then placed into the destination string. The fill character is used if the destination string is longer than the source string. If the source is longer than the destination, the source string is truncated. Overlap of the source and destination strings does not affect execution.

Format

ret-status = LIB$MOVTC (src-str, fill-char, trans-tbl, dst-str)

src-str
Address of source string descriptor.

fill-char
Address of fill character descriptor.

trans-tbl
Address of translation table descriptor.

dst-str
Address of destination string descriptor (fixed-length or dynamic).

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_STRTRU
Procedure successfully completed; string truncated. Fixed-length destination string descriptor could not contain all of the characters.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_INVSTRDES
LIB$_STRIS_INT

Notes

FORTRAN passes arrays (trans-tbl) by reference as a one-origin array. In BASIC and PASCAL, the BY REF and %REF qualifiers must be appended to the trans-tbl parameter. In BASIC arrays are zero-origin.

Only the first character of fill is used, and the length is not checked.

The fill character is not translated.

Example

The following FORTRAN example uses LIB$MOVTC to translate ASCII code values 65 through 68 (decimal) from their usual value to W, X, Y and Z. The procedure will return a destination string of WXYZ #.

```
CHARACTER*6 DEST
CHARACTER TRTABLE (0:255)
DATA TRTABLE /65*' ','W','X','Y','Z',187*' '/
CALL LIB$MOVTC ('ABCDE', '#', TRTABLE, DEST)
END
```

LIB$MOVTUC

3.3.5.2 Move Translated Until Character — LIB$MOVTUC moves the source string character-by-character to the destination string after translating each one using the specified translation table. Each character in the source string is accessed and used as an index into the translation table.

If the table entry contains the specified stop character, the routine is terminated with the relative position of the source character returned. If the table entry is not the stop character, it is moved to the destination string.

If the source is longer than the destination, then truncation of the source string occurs. If the optional fill character is present, any remaining positions in the destination string are filled with the fill character. If the source or destination string is exhausted (without finding the stop character), a zero index is returned.

Format

stop-index = LIB$MOVTUC (src-str, stop-char, trans-tbl, dst-str [,fill-char])

src-str
Address of source string descriptor.

stop-char
Address of stop string descriptor.

trans-tbl
Address of translation table descriptor.

dst-str
Address of destination string descriptor (fixed-length or dynamic).

fill-char
Address of optional fill descriptor. If included, the remainder of the destination string (after the stop character) is filled with the fill character specified. If it is not included, the remainder of the destination string remains intact.

stop-index
Signed longword containing the relative position of the character in the source string that is translated to the stop character. Zero is returned, if the stop character is not found. Failure to allocate dst-str returns minus one.

Notes

Only the first character in the stop-char string and fill-char string, are used and the length is not checked. The fill character is not translated. The results are unpredictable if the source and destination strings overlap and have different starting addresses.

Example

The following FORTRAN example translates the ASCII symbols 48–58 into the decimal values 1 to 10:

```
CHARACTER*6 DEST
CHARACTER TRTABLE (0:255)
DATA TRTABLE /47*' ',',',',0',',1',',2',',3',',4',
1',5',',6',',7',',8',',9',198*' '/
CALL LIB$MOVTUC ('1-129/'', ',', TRTABLE, DEST, '#')
END
```

LIB$TRA_ASC_EBC

3.3.5.3 Translate ASCII to EBCDIC — LIB$TRA_ASC_EBC translates an ASCII string to an EBCDIC string. If the destination string is a fixed string, its length must match the length of the input string (no filling is done). The ASCII to EBCDIC translation table in LIB$AB_ASC_EBC can be specified in a routine using LIB$MOVTC, but no testing for untranslatable characters is done.

Format

ret-status = LIB$TRA_ASC_EBC (src-str, dst-str)

src-str
Address of the source (ASCII) string descriptor.

dst-str
Address of the destination (EBCDIC) string descriptor (fixed-length or dynamic).

Implicit Inputs

The ASCII to EBCDIC translation table at LIB$AB_ASC_EBC.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$__INVCHA
One or more occurrences of an untranslatable character has been detected in the course of the translation.

LIB$__INVARG
If the destination string is a fixed string and its length is not the same as the source string length; no translation is attempted.

LIB$AB__ASC__EBC is the ASCII to EBCDIC translation table, based on ANSI X3.26 - 1970. All ASCII graphics are translated to their equivalent EBCDIC graphic except for:

ASCII graphic	EBCDIC graphic
[(left square bracket)	cents sign
! (exclamation point)	short vertical bar
^ (circumflex)	logical not
] (right square bracket)	! (exclamation point)

The complete table in hexadecimal notation is:

ASCII to EBCDIC

	b7	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
column	b6	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	b5	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	b4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
row b3b2b1b0		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0 0 0 0	00	00	10	40	F0	7C	D7	79	97	3F	3F	3F	3F	3F	3F	3F	3F
0 0 0 1	01	01	11	4F	F1	C1	D8	81	98	3F	3F	3F	3F	3F	3F	3F	3F
0 0 1 0	02	02	12	7F	F2	C2	D9	82	99	3F	3F	3F	3F	3F	3F	3F	3F
0 0 1 1	03	03	13	7B	F3	C3	E2	83	A2	3F	3F	3F	3F	3F	3F	3F	3F
0 1 0 0	04	37	3C	5B	F4	C4	E3	84	A3	3F	3F	3F	3F	3F	3F	3F	3F
0 1 0 1	05	2D	3D	6C	F5	C5	E4	85	A4	3F	3F	3F	3F	3F	3F	3F	3F
0 1 1 0	06	2E	32	50	F6	C6	E5	86	A5	3F	3F	3F	3F	3F	3F	3F	3F
0 1 1 1	07	2F	26	7D	F7	C7	E6	87	A6	3F	3F	3F	3F	3F	3F	3F	3F
1 0 0 0	08	16	18	4D	F8	C8	E7	88	A7	3F	3F	3F	3F	3F	3F	3F	3F
1 0 0 1	09	05	19	5D	F9	C9	E8	89	A8	3F	3F	3F	3F	3F	3F	3F	3F
1 0 1 0	10	25	3F	5C	7A	D1	E9	91	A9	3F	3F	3F	3F	3F	3F	3F	3F
1 0 1 1	11	0B	27	4E	5E	D2	4A	92	C0	3F	3F	3F	3F	3F	3F	3F	3F
1 1 0 0	12	0C	1C	6B	4C	D3	E0	93	6A	3F	3F	3F	3F	3F	3F	3F	3F
1 1 0 1	13	0D	1D	60	7E	D4	5A	94	D0	3F	3F	3F	3F	3F	3F	3F	3F
1 1 1 0	14	0E	1E	4B	6E	D5	5F	95	A1	3F	3F	3F	3F	3F	3F	3F	3F
1 1 1 1	15	0F	1F	61	6F	D6	6D	96	07	3F	3F	3F	3F	3F	3F	3F	3F

where byte: b7b6b5b4 (column) b3b2b1b0 (row)

3.3.5.4 Translate EBCDIC to ASCII — LIB$TRA_EBC_ASC translates an EBCDIC string to an ASCII string. If the destination string is a fixed string, its length must match the length of the input string (no filling is done). The EBCDIC to ASCII translation table at LIB$AB_EBC_ASC can be specified in a routine using LIB$MOVTC, but no testing for untranslatable characters is done.

Format

ret-status = LIB$TRA_EBC_ASC (src-str, dst-str)

src-str
Address of the source (EBCDIC) string descriptor.

dst-str
Address of the destination (ASCII) string descriptor (fixed-length or dynamic).

Implicit Inputs

The EBCDIC to ASCII translation table at LIB$AB_EBC_ASC.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVCHA
One or more occurrences of an untranslatable character has been detected in the course of the translation.

LIB$_INVARG
If the destination string is a fixed string and its length is not the same as the source string length; no translation is attempted.

LIB$AB_EBC_ASC is the EBCDIC to ASCII translation table based on ANSI X3.26 - 1970. All EBCDIC graphics are translated to the identical ASCII graphic except for:

EBCDIC graphic	ASCII graphic
cents sign	[(left square bracket)
short vertical bar	! (exclamation point)
logical not	^ (circumflex)
! (exclamation point)	] (right square bracket)

Untranslatable codes map into 5C (hex) (the ASCII character "\"). Mapping them into 1A (hex) (the ASCII substitute character) would be more desirable, but could cause trouble with STREAM-ASCII files under RMS-11 which recognizes 1A (hex) as a CTRL/Z signifying an end-of-file. The complete table in hexadecimal notation is:

EBCDIC to ASCII

	b7	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
column	**b6**	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	b5	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	b4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
row b3b2b1b0		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0 0 0 0	00	00	10	5C	5C	20	26	2D	5C	5C	5C	5C	5C	7B	7D	5C	30
0 0 0 1	01	01	11	5C	5C	5C	5C	2F	5C	61	6A	7E	5C	41	4A	5C	31
0 0 1 0	02	02	12	5C	16	5C	5C	5C	5C	62	6B	73	5C	42	4B	53	32
0 0 1 1	03	03	13	5C	5C	5C	5C	5C	5C	63	6C	74	5C	43	4C	54	33
0 1 0 0	04	5C	5C	5C	5C	5C	5C	5C	5C	64	6D	75	5C	44	4D	55	34
0 1 0 1	05	09	5C	0A	5C	5C	5C	5C	5C	65	6E	76	5C	45	4E	56	35
0 1 1 0	06	5C	08	17	5C	5C	5C	5C	5C	66	6F	77	5C	46	4F	57	36
0 1 1 1	07	7F	5C	1B	04	5C	5C	5C	5C	67	70	78	5C	47	50	58	37
1 0 0 0	08	5C	18	5C	5C	5C	5C	5C	5C	68	71	79	5C	48	51	59	38
1 0 0 1	09	5C	19	5C	5C	5C	5C	5C	60	69	72	7A	5C	49	52	5A	39
1 0 1 0	10	5C	5C	5C	5C	5B	5D	7C	3A	5C	5C	5C	5C	5C	5C	5C	5C
1 0 1 1	11	0B	5C	5C	5C	2E	24	2C	23	5C	5C	5C	5C	5C	5C	5C	5C
1 1 0 0	12	0C	1C	5C	14	3C	2A	25	40	5C	5C	5C	5C	5C	5C	5C	5C
1 1 0 1	13	0D	1D	05	15	28	29	5F	27	5C	5C	5C	5C	5C	5C	5C	5C
1 1 1 0	14	0E	1E	06	5C	2B	3B	3E	3D	5C	5C	5C	5C	5C	5C	5C	5C
1 1 1 1	15	0F	1F	07	1A	21	5E	3F	22	5C	5C	5C	5C	5C	5C	5C	5C

where byte: b7b6b5b4 (column) b3b2b1b0 (row)

STR$TRANSLATE

3.3.5.5 Translate Matched Characters — STR$TRANSLATE 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, the source character moves 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. 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.

Format

ret-status = STR$TRANSLATE (des-str, src-str, trans-tbl, match-str)

des-str
Address of destination string descriptor (fixed-length or dynamic).

src-str
Address of source string descriptor.

trans-tbl
Address of translate string descriptor.

match-str
Address of match string descriptor.

STR$UPCASE

3.3.5.6 Uppercase Conversion — STR$UPCASE converts successive characters in a source string to uppercase and writes the converted character into the destination string. When you need to compare characters without regard to case, you can first convert both characters to uppercase. The routine only converts a to z. Foreign languages with accented letters should use STR$TRANSLATE.

Format

ret-status = STR$UPCASE (des-str, src-str)

des-str
Address of destination string descriptor (fixed-length or dynamic).

src-str
Address of source string descriptor.

Return Status

SS$_NORMAL
Routine successfully completed.

STR$_TRU
Warning. String truncated. Fixed-length destination string could not contain all of the characters.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_INSVIRMEM
STR$_STRIS_INT

Example

The following BASIC statements would result in D$ containing 'HELLO':

```
EXTERNAL INTEGER FUNCTION STR$UPCASE
STATUS% = STR$UPCASE (D$, 'Hello')
```

3.4 Formatted Input and Output Conversion Procedures

This section describes the formatted input and output conversion routines available as callable procedures. Input conversion procedures convert a fixed-length string of characters to a D_, G_, or H_floating or integer binary value. Output conversion procedures convert a D_, G_, or H_floating or integer binary value to the corresponding space-padded, fixed-length string of characters. String descriptors are used to specify the length and address of all strings.

The following floating input and output conversions are provided:

D FORTRAN D format (scientific notation with D exponent)
E FORTRAN E format (scientific notation with E exponent)
F FORTRAN F format
G FORTRAN G format (selects between E and F depending on the magnitude of the value)

The following integer input and output conversions are provided:

I Integer
L Logical
O Octal
Z Hexadecimal

While these procedures may be called from FORTRAN, they are provided for use by programs written in other languages. These procedures are called implicitly by the language support procedures to perform formatted and list-directed input/output statements. Input scalars are passed by immediate value, rather than by reference. Output strings are assumed to be static, and the class field in the descriptor is not checked.

NOTE

If you are interested in procedures that convert decimal, octal, or hexadecimal strings to binary values and pass the strings by count and address, see Section 3.4.1.6.

3.4.1 Input Conversions

3.4.1.1 Convert Text to Floating — OTS$CVT_T_x converts an ASCII text string representation of a numeric value to D_, G_, or H_floating. The routine supports FORTRAN D, E, F and G input type conversion as well as similar types for other languages.

For compatibility with previous releases, the name FOR$CNV_IN_DEFG is equivalent to OTS$CVT_T_D.

The syntax of a valid ASCII input string is:

<zero or more blanks>
<"+", "-", or nothing>
<zero or more decimal digits>
<"." or nothing>
<zero or more decimal digits>
<exponent or nothing, where exponent is:
< <<"E", "e", "D", "d", "Q", "q">
<zero or more blanks>
<"+", "-", or nothing>>
or
<"+", or "-">>
<zero or more decimal digits>>
<end of string>

NOTE

There is no difference in semantics between any of the six valid exponent letters. See discussion of flags.

Format

ret-status = OTS$CVT_T_x (inp-str, value [,digits-in-fract [,scale-factor [,flags [,ext-bits]]]])

where "x" is D for D_floating, G for G_floating or H for H_floating.

inp-str
Address of input string descriptor.

value
Address of the floating result.

digits-in-fract
An unsigned longword containing the number of digits in the fraction if the decimal point is in the input string. (This is an optional parameter, passed by immediate value. If omitted, the default is zero.)

scale-factor
A signed longword containing the scale factor. If flags bit 6 is clear, the result value is multiplied by 10**factor unless the exponent is present. If flags bit 6 is set, the scale factor is always applied. (This is an optional parameter, passed by immediate value. If omitted, the default is zero.)

flags
An unsigned longword containing caller-supplied flags defined as follows:

Bit 0 If set, blanks are ignored. If clear, blanks are equivalent to "0".

Bit 1 If set, only E or e exponents are allowed.

Bit 2 If set, underflow will cause an error.

Bit 3 If set, don't round the value.

Bit 4 If set, tabs are ignored. If clear, tabs are illegal.

Bit 5 If set, an exponent must begin with a valid exponent letter. If clear, the exponent letter may be omitted.

Bit 6 If set, the scale factor is always applied. If clear, it is only applied if there is no exponent present in the string.

(Flags is an optional parameter, passed by immediate value. If omitted, all bits are clear.)

ext-bits
Address of a byte or word to receive the extra precision bits. If present, the value is not rounded, and the first n bits after truncation are returned in this argument. For D_floating, n equals 8 and the bits are returned as a byte. For G_ and H_floating, n equals 11 and 15, respectively, and the bits are returned as a word, left-justified. These values are suitable for use as the extension operand in an EMOD instruction.

CAUTION

The bits returned for H_floating may not be precise because calculations are only carried to 128 bits. However, the error should be small. D_ and G_floating return guaranteed exact bits; they are not rounded.

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_INPCONERR
Input conversion error; an invalid character in input string, or value is outside the range that can be represented. Value is set to +0.0 (not reserved operand -0.0).

3.4.1.2 Convert Text (Signed Integer) to Longword — OTS$CVT_TI_L converts an ASCII text string representation of a decimal number to a signed byte, word, or longword. The result is a longword by default, but the calling program can specify a byte or a word value instead.

The syntax of a valid ASCII text input string is:

[+ or -][<integer-digits>]

Leading blanks are always ignored. Blanks after the sign or the first digit are ignored if flags bit 0 is set; otherwise, blanks are treated as zeroes. Tabs are ignored if flags bit 1 is set; otherwise, tabs are invalid. An implicit decimal point is assumed at the right of inp-str.

For compatibility with previous releases, the name FOR$CNV_IN_I is equivalent to OTS$CVT_TI_L.

Format

ret-status = OTS$CVT_TI_L (inp-str, value [,value-size [,flags]])

inp-str
Address of the input string descriptor.

value
Address of a signed byte, word, or longword to receive the integer value, depending on value-size. (This is an output parameter.)

value-size
A longword containing the number of bytes the value will occupy. (This is an optional parameter, passed by immediate value. The default is four.) Valid values are one, two and four. Invalid values return an error.

flags
An unsigned longword containing caller supplied flags defined as follows:

bit 0 If set, blanks are ignored. If clear, blanks after the first legal character are treated as zeroes.

bit 1 If set, tabs are ignored. If clear, tabs are invalid.

(Flags is an optional parameter, passed by immediate value. If omitted, all bits are clear.)

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_INPCONERR
Input conversion error; an invalid character in input string, or value overflows byte, word, or longword, or value-size is invalid; value is set to zero.

3.4.1.3 Convert Text (Logical) to Longword — OTS$CVT_TL_L converts an ASCII text string representation of a FORTRAN-77 L format to a byte, word, or longword value. The result is a longword by default, but the calling program can specify a byte or a word value instead.

For compatibility with previous releases, the name FOR$CNV_IN_L is equivalent to OTS$CVT_TL_L.

The syntax of a valid ASCII text string is:

```
<zero or more blanks>
<       <end of string>
        or
     <        <"." or nothing>
Letter:  <"T". "t". "F". "f">
              <zero or more of any character>
              <end of string>>>
```

The value returned by OTS$CVT_TL_L is minus one if the character denoted by "Letter:" is "T" or "t", zero otherwise.

Format

ret-status = OTS$CVT_TL_L (inp-str, value [,value-size])

inp-str
: Address of the input string descriptor.

value
: Address of a byte, word, or longword to receive the integer value, depending on value-size. (This is an output parameter.)

value-size
: A longword containing the number of bytes the value will occupy. (This is an optional parameter, passed by immediate value. The default is four.) Valid values are one, two and four. Invalid values return an error.

Return Status

SS$_NORMAL
: Routine successfully completed.

OTS$_INPCONERR
: Invalid character in the input string or invalid value-size; value set to zero.

3.4.1.4 Convert Text (Octal) to Longword — OTS$CVT_TO_L converts an ASCII text string representation of an unsigned octal value to an unsigned byte, word, or longword. The result is a longword by default, but the calling program can specify a byte or a word value instead. The valid input characters are the space and the digits 0 through 7. No sign is permitted.

For compatibility with previous releases, the name FOR$CNV_IN_O is equivalent to OTS$CVT_TO_L.

Format

ret-status = OTS$CVT_TO_L (inp-str, value [,value-size [,flags]])

inp-str
Address of input string descriptor.

value
Address of an unsigned byte, word, or longword to receive the result, depending on value-size.

value-size
A longword containing the number of bytes that the value will occupy. (This is an optional parameter, passed by immediate value. The default is four.) If input size is zero or negative, an error is returned.

flags
A longword containing caller supplied flags defined as follows:

Bit 0 If set, blanks are ignored; otherwise blanks are treated as zeroes.

(Flags is an optional parameter, passed by immediate value. If omitted, all bits are clear.)

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_INPCONERR
Input conversion error. An invalid character, overflow, or invalid value-size occurred.

3.4.1.5 Convert Text (Hexadecimal) to Longword — OTS$CVT_TZ_L converts an ASCII text string representation of an unsigned hexadecimal value to an unsigned byte, word, or longword. The result is a longword by default, but the calling program can specify a byte or a word value instead. Valid input characters are the space, the digits 0 through 9, and letters A through F. No sign is permitted. Lowercase letters a through f are acceptable.

For compatibility with previous releases, the name FOR$CNV_IN_Z is equivalent to OTS$CVT_TZ_L.

Format

ret-status = OTS$CVT_TZ_L (inp-str, value [,value-size [,flags]])

inp-str
Address of input string descriptor.

value
Address of an unsigned byte, word, or longword to receive the result, depending on value-size.

value-size
A longword containing the number of bytes that the value will occupy. (This is an optional parameter, passed by immediate value. The default is four.) If input size is zero or negative, an error is returned.

flags
A longword containing caller supplied flags defined as follows:

Bit 0 If set, blanks are ignored; otherwise blanks are treated as zeroes.

(Flags is an optional parameter, passed by immediate value. If omitted, all bits are clear.)

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_INPCONERR
Input conversion error. An invalid character, overflow, or invalid value-size occurred.

3.4.1.6 Convert Text to Binary

LIB$CVT__DTB — Decimal to Binary Conversion
LIB$CVT__HTB — Hexadecimal to Binary Conversion
LIB$CVT__OTB — Octal to Binary Conversion

These procedures return a binary representation of the ASCII text string representation of a decimal, octal, or hexadecimal number.

NOTE

These LIB$ procedures are unusual in that they expect input scalar parameters to be passed by immediate value and strings by reference and blanks are invalid characters.

Format

```
ret-status = LIB$CVT__DTB (count, string, result)
ret-status = LIB$CVT__OTB (count, string, result)
ret-status = LIB$CVT__HTB (count, string, result)
```

count
Byte count of input ASCII text string.

string
Address of input ASCII text string.

result
Address to receive longword result.

Return Status

SS$__NORMAL
Procedure successfully completed.

0
Nonradix character in the input string or a sign in any position other than the first character. Blanks and tabs are invalid characters. An overflow from 32 bits (unsigned) will cause an error.

NOTE

See Section 3.4.1.1 for more flexible and general input conversion routines.

3.4.2 Output Conversions

3.4.2.1 Convert Longword to Text (Signed Integer) — OTS$CVT_L_TI converts a signed integer to a decimal ASCII text string. This procedure supports FORTRAN Iw and Iw.m output and BASIC output conversion.

A separate entry point FOR$CNV_OUT_I is provided for compatibility with previous releases.

Format

ret-status = OTS$CVT_L_TI (value-adr, out-str [,int-digits [,value-size [,flags]]])

ret-status = FOR$CNV_OUT_I (value, out-str)

value-adr (OTS$CVT_L_TI only)
Address of the signed byte, word, or longword containing the integer value, depending on value-size.

value (FOR$CNV_OUT_I only)
A longword containing the signed integer value to be converted to text (passed by immediate value).

out-str
Address of output string descriptor to receive the ASCII text string. The string is assumed to be fixed-length (DSC$K_CLASS_S).

int-digits
A longword containing the minimum number of digits to be generated. If the actual number of significant digits is smaller, leading zeroes are produced. If int-digits is zero and value is zero, a blank field will result. (This is an optional parameter, passed by immediate value. The default value is one.)

value-size
A longword containing the number of bytes occupied by the value to be converted to text. The value-size must be either one, two or four. If value-size is 1 or 2, the value is sign extended to a longword before conversion. (This is an optional parameter, passed by immediate value. The default is four.)

flags
A longword containing caller supplied flags defined as follows:

Bit 0 If set, a plus sign (+) will be inserted before the first non-blank character in the output string; otherwise, the plus sign will be omitted.

(Flags is an optional parameter, passed by immediate value. If omitted, all bits are clear.)

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_OUTCONERR
Output conversion error. The result would have exceeded the fixed-length string; the output string is filled with asterisks.

OTS$CVT_L_TL

3.4.2.2 Convert Longword to Text (Logical) — OTS$CVT_L_TL converts an integer to the ASCII text string representation using FORTRAN L (logical) format.

The output string will consist of (length -1) blanks followed by:

The letter T if bit 0 is set
The letter F if bit 0 is clear

A separate entry point FOR$CNV_OUT_L is provided for compatibility with previous releases.

Format

ret-status = OTS$CVT_L_TL (value-adr, out-str)

ret-status = FOR$CNV_OUT_L (value, out-str)

value-adr (OTS$CVT_L_TL only)
Address of the longword containing the input value to be converted to text.

value (FOR$CNV_OUT_L only)
A longword containing the input value to be converted to text (passed by immediate value).

out-str
Address of output string descriptor to receive the ASCII text string. The string is assumed to be fixed-length (DSC$K_CLASS_S).

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_OUTCONERR
Output conversion error. The result would have exceeded the fixed-length string; the output string is of zero length (DSC$W_LENGTH=0).

3.4.2.3 Convert Longword to Text (Octal) — OTS$CVT_L_TO converts an unsigned integer to an octal ASCII text string. OTS$CVT_L_TO supports FORTRAN Ow and Ow.m output conversion formats.

A separate entry point FOR$CNV_OUT_O is provided for compatibility with previous releases.

Format

ret-status = OTS$CVT_L_TO (value-adr, out-str [,int-digits [,value-size]])

ret-status = FOR$CNV_OUT_O (value, out-str)

value-adr (OTS$CVT_L_TO only)
Address of the unsigned byte, word, or longword containing the integer value, depending on value-size.

value (FOR$CNV_OUT_O only)
A longword containing the integer value to be converted (passed by immediate value).

out-str
Address of output string descriptor to receive the ASCII text string. The string is assumed to be fixed-length (DSC$K_CLASS_S).

int-digits
A longword containing the minimum number of digits to be generated. If the actual number of significant digits is less, leading zeroes are produced. If int-digits is zero and value is zero, a blank string results. (This is an optional parameter, passed by immediate value. The default is one.)

value-size
A longword containing the size of value in bytes. (This is an optional parameter, passed by immediate value. The default is four.)

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_OUTCONERR
Output conversion error. The result would have exceeded the fixed-length string; the output string is filled with asterisks.

3.4.2.4 Convert Longword to Text (Hexadecimal) — OTS$CVT_L_TZ converts an unsigned integer to a hexadecimal ASCII text string. OTS$CVT_L_TZ supports FORTRAN Zw and Zw.m output conversion formats.

A separate entry point FOR$CNV_OUT_Z is provided for compatibility with previous releases.

Format

ret-status = OTS$CVT_L_TZ (value-adr, out-str [,int-digits [,value-size]])

ret-status = FOR$CNV_OUT_Z (value, out-str)

value-adr (OTS$CVT_L_TZ only)
Address of the unsigned byte, word, or longword containing the integer value, depending on value-size.

value (FOR$CNV_OUT_Z only)
A longword containing the integer value to be converted (passed by immediate value).

out-str
Address of output string descriptor to receive the ASCII text string. The string is assumed to be fixed-length (DSC$K_CLASS_S).

int-digits
A longword containing the minimum number of digits to be generated. If the actual number of significant digits is less, leading zeroes are produced. If int-digits is zero and value is zero, a blank string results. (This is an optional parameter, passed by immediate value. The default is one.)

value-size
A longword containing the size of value in bytes. (This is an optional parameter, passed by immediate value. The default is four.)

Return Status

SS$_NORMAL
Routine successfully completed.

OTS$_OUTCONERR
Output conversion error. The result would have exceeded the fixed-length string; the output string is filled with asterisks.

3.4.2.5 Convert Floating to Text — FOR$CVT_x_Ty are routines that convert floating values to ASCII text strings. They are divided according to VAX-11 data types and to FORTRAN format types.

FORTRAN format types are D/E (exponential), F (fixed point), and G (fixed or exponential). VAX-11 data types are D_, G_, and H_floating.

For compatibility with previous releases, the name FOR$CNV_OUT_y is equivalent to FOR$CVT_D_Ty.

Format

ret-status = FOR$CVT_x_Ty (value-adr, out-str, digits-in-fract [,scale-factor [,digits-in-int [,digits-in-exp [,flags]]]])

where:

x is the VAX-11 data type, either D_, G_, or H_floating and
y is the FORTRAN format, either D, E, F or G

value-adr
Address of the D_, G_, or H_floating value to be converted.

out-str
Address of the output string descriptor to receive the ASCII text string. The string is assumed to be fixed-length (DSC$K_CLASS_S).

digits-in-fract
An unsigned longword containing the number of digits in the fraction portion (passed by immediate value).

scale-factor
A longword containing the scale factor. The externally represented number equals the internally represented number multiplied by 10** scale-factor. If digits-in-int is not present, scale-factor indicates the true scale factor on F format or the digits-in-int for D, E and G formats. (This is an optional parameter, passed by immediate value. The default is zero.)

digits-in-int
An unsigned longword containing the number of digits in the integer part of an exponentially formatted value. Digits-in-int is ignored for F format. (This is an optional parameter, passed by immediate value. The default is zero.)

digits-in-exp
An unsigned longword containing the number of digits in the exponent field. If the exponent overflows this field by one digit, the exponent letter is removed. (This is an optional parameter, passed by immediate value. The default is two.)

flags
An unsigned longword containing the caller supplied flags defined as follows:

bit 0 If set, and the value is positive, insert a plus sign (+) before the first non-zero character in the output string.

(Flags is an optional parameter, passed by immediate value. If omitted, all bits are clear.)

Return Status

SS$_NORMAL
Routine successfully completed.

FOR$_OUTCONERR
Output conversion error. The result would have exceeded the fixed-length string; the output string is filled with asterisks.

Messages

SS$_ROPRAND
Reserved operand fault. A reserved floating operand was passed; out-str is not changed.

3.4.3 Convert Binary to Formatted ASCII

The Formatted ASCII Output system service ($FAO) converts binary values into ASCII characters and returns the converted characters in an output string. It can be used to:

- Insert variable character string data into an output string
- Convert binary values into the ASCII representations of their decimal, hexadecimal, or octal equivalents and substitute the results into an output string

The Formatted ASCII Output with List Parameter system service ($FAOL) provides an alternate way to specify input parameters for a call to the $FAO system service.

System service routines that return strings return only fixed-length strings and they are not blank filled. For some high-level languages, it is desirable to be able to return dynamic strings and for others, to blank fill fixed-length strings. Likewise, high-level languages generally pass parameters by reference, while system service routines pass by immediate value. The following procedures, LIB$SYS_FAO and LIB$SYS_FAOL, provide a convenient interface for higher level languages and the corresponding system services.

3.4.3.1 Formatted ASCII Output — LIB$SYS_FAO calls $FAO for the caller, returning a string using the semantics of the caller's string. If called with other than a fixed string for output, the length of the resultant string is limited to 256 bytes and truncation can occur.

See *VAX/VMS System Services Reference Manual* for a complete description of $FAO.

Format

ret-status = LIB$SYS_FAO (ctr-str [,out-len], out-buf [,p1 ... [,pn]])

ctr-str
Address of the ASCII control string descriptor. The control string consists of the fixed text of the output string and FAO directives.

out-len
Address of a word to receive the output string length. This is an optional parameter.

out-buf
Address of the fixed-length or dynamic output string descriptor to receive the fully formatted output string.

p1 - pn
Directive parameters contained in longwords. Depending on the directive, a parameter can be a value to be converted, the address of the string to be inserted, or a length or argument count. A maximum of 17 directive parameters can be specified. These are optional parameters. The passing mechanism for each of these parameters should be the one expected by the system service.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_STRTRU
Success, but source string was truncated on copy.

LIB$_INSVIRMEM
Insufficient virtual memory to allocate dynamic string.

LIB$_INVSTRDES
Invalid string descriptor.

SS$_BUFFEROVF
Successfully completed, but formatted output string overflowed the output buffer and has been truncated.

SS$_BADPARAM
An invalid directive was specified in the FAO control string.

3.4.3.2 Formatted ASCII Output with List Parameter — LIB$SYS_FAOL calls the system service routine $FAOL for the caller, returning the resultant string using the semantics of the caller's string. If called with other than a fixed string for output, the length of the resultant string is limited to 256 bytes and truncation may occur.

See the *VAX/VMS System Services Reference Manual* for a complete description of $FAOL.

Format

ret-status = LIB$SYS_FAOL (ctr-str [,out-len], out-buf, prm-lst)

ctr-str
Address of the ASCII control string descriptor. The control string consists of fixed text from the output string and conversion directives.

out-len
Address of a word to receive the output string length. (This is an optional parameter.)

out-buf
Address of the fixed-length or dynamic output string descriptor to receive the fully formatted output string.

prm-lst
Address of an array of longwords to be used as p1 through pn. The parameter list can be a data structure that already exists in a program and from which certain values are to be extracted.

Return Status

See LIB$SYS_FAO description in Section 3.4.3.1.

3.5 Variable Bit Field Instruction Procedures

The following procedures manipulate variable bit fields. The procedures are intended primarily for higher level languages. The MACRO programmer can perform the equivalent using a single machine instruction.

A variable bit field is specified by three scalar parameters:

- pos – the address of a signed longword containing the first bit position of the field with respect to the base address
- size – the address of a byte containing the size of the bit field, from 0 to 32
- base – the address of the base of the bit field

Bit fields are zero-origin, which means that the procedure regards the first bit in the field as being the zero position. For more detailed information on VAX-11 bit numbering and data formats, see the *VAX-11 Architecture Handbook*.

LIB$INSV

3.5.1 Insert a Variable Bit Field

LIB$INSV replaces the variable bit field specified by the base, position, and size parameters with bits zero through size-1 of the source. If the size of the bit field is zero, nothing is inserted.

Format

CALL LIB$INSV (src, pos, size, base)

src
Address of longword containing the source field to be inserted.

pos
Address of longword containing the first bit position of the field relative to the base address.

size
Address of unsigned byte containing the size of the bit field to be inserted.

base
Address of the base of the output field in which the source is to be inserted.

Messages

SS$_ROPRAND
A reserved operand fault is signaled if a size greater than 32 is specified.

Examples

In FORTRAN, to set bits 0 through 2 of longword COND_VALUE to 4:

```
INTEGER*4  COND_VALUE
CALL LIB$INSV (4, 0, 3, COND_VALUE)
```

In BASIC, to set bits 0 through 2 of longword COND_VALUE to 4:

```
DECLARE INTEGER COND_VALUE
CALL LIB$INSV (4%, 0%, 3%, COND_VALUE)
```

3.5.2 Extract and Sign-Extend a Field

LIB$EXTV returns a sign-extended, longword field that has been extracted from the specified variable bit field.

Format

field = LIB$EXTV (pos, size, base)

pos
Address of longword containing the beginning bit position (relative to the base address).

size
Address of unsigned byte containing the size of the bit field to be extracted. The maximum size is 32 bits.

base
Address of the base of the bit field to be extracted.

field
The field, sign-extended to a longword.

Messages

SS$_ROPRAND
A reserved operand fault occurs if a size greater than 32 is specified.

Example

In FORTRAN, if bits 3 to 0 of VALUE contain a 3 (0011), then SMALL_INT is set to 3 (00000003 hex) in the following example:

```
INTEGER*4  VALUE, SMALL_INT
SMALL_INT = LIB$EXTV (0, 4, VALUE)
```

If bits 3 to 0 of VALUE contain all ones, SMALL_INT is set to -1 (FFFFFFFF hex) in the preceding example.

3.5.3 Extract a Zero-Extended Field

LIB$EXTZV returns a longword, zero-extended field that has been extracted from the specified variable bit field.

Format

Field = LIB$EXTZV (pos, size, base)

pos
Address of longword containing the beginning bit position (relative to the base address).

size
Address of unsigned byte containing the size of the bit field to be extracted. The maximum size is 32 bits.

base
Address of the base of the bit field to be extracted.

field
The field, zero-extended to a longword.

Messages

SS$__ROPRAND
A reserved operand fault occurs if a size greater than 32 is specified.

Example

In this FORTRAN example, if bits 2 to 0 of COND__VALUE contain 4 (100), then SEVERITY will be set to 4:

```
INTEGER*4  COND_VALUE, SEVERITY
SEVERITY = LIB$EXTZV (0, 3, COND_VALUE)
```

3.5.4 Find First Clear Bit

LIB$FFC searches the field specified by the start position, size, and base for the first clear bit. If one is found, SS$_NORMAL is returned as well as the bit position (relative to start-pos) in the find-pos parameter. If a clear bit is not found or a size of zero is specified, a failure status is returned, and the find position is set to the size.

Format

ret-status = LIB$FFC (start-pos, size, base, find-pos)

start-pos
: Address of longword containing the starting bit position (relative to the base address).

size
: Address of unsigned byte containing the number of bits to be searched. The maximum size is 32 bits.

base
: Address of longword bit field to be searched.

find-pos
: Address of longword to receive bit position (relative to start-pos) of first clear bit. (This is an output parameter.)

Return Status

SS$_NORMAL
: Routine successfully completed. A clear bit was found.

LIB$_NOTFOU
: A clear bit was not found.

Messages

SS$_ROPRAND
: A reserved operand fault is signaled if a size greater than 32 is specified.

Example

In the following FORTRAN example, FPOS is set to 6, since bit 10 is the first clear bit and search started at bit 4 in BITS.

```
INTEGER*4 FPOS, BITS
BITS = 2**10-1
CALL LIB$FFC (4,28,BITS,FPOS)
```

3.5.5 Find First Set Bit

LIB$FFS searches the field specified by the start position, size, and base for the first set bit. If one is found, a success status is returned as well as the bit position (relative to start-pos) in the find-pos parameter. If a set bit is not found or a size of zero is specified, a failure status is returned and the find-pos is set to the size.

Format

ret-status = LIB$FFS (start-pos, size, base, find-pos)

start-pos
: Address of longword containing the start position (relative to the base address).

size
: Address of byte containing the number of bits to be searched. The maximum size is 32 bits.

base
: Address of the longword bit field.

find-pos
: Address of longword to receive the bit position (relative to start-pos) of first set bit. (This is an output parameter.)

Return Status

SS$_NORMAL
: Routine successfully completed. A set bit was found.

LIB$_NOTFOU
: A set bit was not found.

Messages

SS$_ROPRAND
: A reserved operand fault is signaled if a size greater than 32 is specified.

Example

In the following FORTRAN example, FPOS is set to 6, since bit 10 is the first set bit and the search is started at bit 4:

```
INTEGER*4 FPOS,BASE
BASE = 2**10
CALL LIB$FFS (4, 28, BASE, FPOS)
```

3.6 Performance Measurement Procedures

These procedures implement the Run-Time Library Timing Facility.

LIB$INIT_TIMER gets from VAX/VMS the current values of specified times and counts, and stores them for future use by LIB$SHOW_TIMER or LIB$STAT_TIMER.

LIB$SHOW_TIMER obtains the accumulated times/counts since the last call to LIB$INIT_TIMER as formatted ASCII text.

LIB$STAT_TIMER returns to its caller one of five available statistics. Unlike LIB$SHOW_TIMER, which formats the values for output, LIB$STAT_TIMER returns the values as unsigned integers to a location specified by a parameter.

LIB$FREE_TIMER frees the storage allocated by LIB$INIT_TIMER.

LIB$FREE_TIMER

3.6.1 Free Timer Storage

LIB$FREE_TIMER frees the storage allocated by LIB$INIT_TIMER. If the block referred to by "handle" was not allocated by LIB$INIT_TIMER, an error is returned.

Format

ret-status = LIB$FREE_TIMER (handle)

handle
A longword containing a pointer to the control block in which the times/counts are stored. The pointer must be the same value returned by a previous call to LIB$INIT_TIMER. On a successful return, "handle" is set to zero.

Implicit Inputs

It is assumed that "handle" has been returned by a previous call to LIB$INIT_TIMER.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid argument. "Handle" is invalid.

LIB$_BADBLOADR
Bad block address. "Handle" is invalid.

3.6.2 Initialize Times and Counts

LIB$INIT_TIMER stores the current values of specified times and counts for use by LIB$SHOW_TIMER or LIB$STAT_TIMER. The values are stored in one of three places, depending on the optional argument "handle."

Format

ret-status = LIB$INIT_TIMER ([handle])

handle
A longword containing a pointer to a control block where the values of times/counts will be stored. (This is an optional parameter.)

If missing, the times/counts will be stored in OWN storage. This call is neither AST-reentrant nor modular.

If zero, a control block will be allocated in dynamic heap storage by a call to LIB$GET_VM. The times/counts will be stored in that block and the address of the block returned in "handle." This method is AST-reentrant and modular.

If non-zero, it is considered to be the address of a storage block previously allocated by a call to LIB$INIT_TIMER. If so, the control block is reused, and fresh times and counts are stored in it.

Implicit Inputs

If "handle" is nonzero, the block of storage it refers to is assumed to have been initialized by a previous call to LIB$INIT_TIMER.

Implicit Outputs

Upon exit, the block of storage referred to by "handle" will contain the times/counts.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid argument. "Handle" is nonzero and the block it refers to was not initialized on a previous call to LIB$INIT_TIMER.

LIB$_INSVIRMEM
"Handle" is zero, and there is insufficient virtual memory to allocate a storage block.

3.6.3 Return Accumulated Times and Counts as a Statistic

LIB$STAT_TIMER returns to its caller one of five available statistics. Unlike LIB$SHOW_TIMER, which formats the values for output, LIB$STAT_TIMER returns the value as an unsigned longword or quadword.

Only one of the five statistics can be returned by a single call to LIB$STAT_TIMER. "Code" must be an integer from one to five.

NOTE

The elapsed time (code = 1) is returned in the system quadword format. Therefore the receiving area should be 8-bytes long. All other values are longwords.

Format

ret-status = LIB$STAT_TIMER (code-adr, value-adr [,handle-adr])

code-adr
Address of a longword or quadword containing a value which specifies the statistic to be returned. Allowed values are:

1 - Elapsed Time (quadword, in system time format)
2 - CPU Time (longword, in 10 millisecond increments)
3 - Buffered I/O (longword)
4 - Direct I/O (longword)
5 - Page Faults (longword)

NOTE

It is invalid to omit this parameter or to give a "code" of zero.

value-adr
Address of the area to store the result. All values are longword integers except elapsed time, which is a quadword. See the *VAX/VMS System Services Reference Manual* for more details on the system quadword time format.

handle-adr
Address of a longword containing a pointer to a block of storage. (This is an optional parameter.) If specified, the pointer must be the same value returned by a previous call to LIB$INIT_TIMER. Otherwise, OWN storage is used.

Implicit Inputs

It is assumed that LIB$INIT_TIMER has been called and that the "handle" argument to LIB$INIT_TIMER is the same as in the call to LIB$STAT_TIMER.

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid argument. Either "code" or "handle" is invalid.

LIB$SHOW_TIMER

3.6.4 Show Accumulated Times and Counts

LIB$SHOW_TIMER gets accumulated times/counts since the last call to LIB$INIT_TIMER. In the default mode, with neither CODE nor ACTION specified in the call, the routine outputs to SYS$OUTPUT a line giving the following five items of information:

ELAPSED = hhhh:mm:ss.cc - Elapsed real time
CPU = hhhh:mm:ss.cc - Elapsed CPU time
BUFIO = nnnn - Count of Buffered I/O operations
DIRIO = nnnn - Count of direct I/O operations
PAGEFLTS = nnnn - Count of page faults

Optionally, one or all five statistics can be output to SYS$OUTPUT or passed to a user-specified "action routine" for nondefault processing.

Format

ret-status = LIB$SHOW_TIMER ([[[[handle-adr], code-adr], action-adr], user-arg])

handle-adr
Address of a longword containing a pointer to a block of storage. (This is an optional parameter.) If specified, the pointer must be the same value returned by a previous call to LIB$INIT_TIMER. If omitted, the routine's OWN storage will be used. If handle-adr is omitted and LIB$INIT_TIMER has not been called previously, elapsed time will show the actual time-of-day, and the remaining values will be those accumulated since process log-in.

code-adr
Address of a longword value specifying a particular statistic. (This is an optional parameter.) It must be one of these values:

1 = Elapsed Time
2 = CPU Time
3 = Buffered I/O
4 = Direct I/O
5 = Page faults

If omitted or zero, all five statistics are returned on one line.

action-adr
Address of a function procedure to call. (This is an optional parameter.) The function should return either a success or failure condition value, which will be returned as the value of LIB$SHOW_TIMER.

Format for "action" routine:
ret-status = (action) (out-str[,user-arg])

out-str
Address of a descriptor of a fixed-length string containing the statistics you want. The string is formatted exactly as it would be if output to SYS$OUTPUT. The leading character is blank.

user-arg
If passed on to LIB$SHOW_TIMER, user-arg is passed directly on to the action routine. Note that this is passed by immediate value to both LIB$SHOW_TIMER and the action routine.

Implicit Inputs

It is assumed that LIB$INIT_TIMER has been previously called, and that the results of that call are stored in either OWN storage or a block pointed to be "handle."

Return Status

SS$_NORMAL
Routine successfully completed.

LIB$_INVARG
Invalid arguments. An invalid value was given for "code" or "handle." Other codes may be returned by LIB$PUT_OUTPUT or the user's action routine.

Example

The following FORTRAN code fragment could be used to time a loop and output the results to the terminal:

```
      INTEGER*4 HANDLE
      HANDLE = 0
      IF (.NOT. LIB$INIT_TIMER(HANDLE)) GO TO error
      DO 100 ...
      .
      .
      .
100   CONTINUE
      IF (.NOT. LIB$SHOW_TIMER(HANDLE)) GO TO error
```

3.7 Date/Time Utility Procedures

Some of the following procedures are provided primarily for use with FORTRAN built-in functions: DATE, SECNDS, and TIME. However, you can call them from programs written in any language. Input scalar parameters are passed by-reference. The FORTRAN compiler generates calls to the procedure you want depending on the data type of the parameter(s).

3.7.1 Convert Binary Date/Time to an ASCII String

LIB$SYS__ASCTIM calls the system service ASCTIM to convert a binary date and time value, returning the resultant ASCII string using the semantics of the caller's string. Parameter cnv-flg is presented to this routine by reference and is promoted to by immediate value for presentation to ASCTIM.

See the ASCTIM system service description in the *VAX/VMS System Services Reference Manual.*

Format

ret-status = LIB$SYS__ASCTIM ([out-len], dst-str, [user-time], [cnv-flg])

out-len
Optional addres of a word to receive the number of bytes written into dst-str, not counting padding in the case of a fixed string. If the input string is truncated to the size specified in the dst-str descriptor, out-len is set to this size. Therefore, out-len can always be used by the calling program to access a valid substring of dst-str.

dst-str
Address of a string descriptor to receive the string (fixed-length or dynamic).

user-time
Optional address of the quadword integer value to be converted. If zero or no address is specified, the current system date and time are returned. A positive value represents an absolute time. A negative value represents a delta time. If a delta time is specified, it must be less than 10,000 days.

cnv-flg
Optional address of an unsigned longword containing the conversion indicator. A value of one causes only the hour, minute, second, and hundredths of a second to be returned, depending on the length of the buffer. A value of zero (the default) causes the full date and time to be returned, depending on the length of the buffer.

Return Status

SS$__NORMAL
Routine succssfully completed.

LIB$__STRTRU
Routine successfully completed, but the source string was truncated.

LIB$__FATERRLIB
LIB$__INSVIRMEM
LIB$__INVSTRDES
LIB$__STRIS__INT

SS$__IVTIME
The specified delta time is greater than or equal to 10,000 days.

3.7.2 Return Month, Day, Year as INTEGER*2

FOR$IDATE uses the system service, Convert Binary Time to Numeric Time ($NUMTIM), to get date information. This information is converted to 16-bit integers and stored through the addresses passed as parameters.

Format

CALL FOR$IDATE (month, day, year)

month
Address of a word to receive month integer (range: 1 to 12). (This is an output parameter.)

day
Address of a word to receive day integer (range: 1 to 31). (This is an output parameter.)

year
Address of a word to receive year of century integer (range: 0 to 99). (This is an output parameter.)

3.7.3 Return Month, Day, Year as INTEGER*4

FOR$JDATE uses the system service, Convert Binary Time to Numeric Time ($NUMTIM), to get date information. This information is converted to 32-bit integers and stored through the addresses passed as parameters.

Format

CALL FOR$JDATE (month, day, year)

month
Address of longword to receive month integer (range: 1 to 12). (This is an output parameter.)

day
Address of a longword to receive day integer (range: 1 to 31). (This is an output parameter.)

year
Address of a longword to receive year of century integer (range: 0 to 99). (This is an output parameter.)

3.7.4 Return System Date as 9-Byte String

FOR$DATE returns the system date as a 9-byte string in the form DD-MMM-YY (for example, 01-Jun-78).

Format

CALL FOR$DATE (9-byte-array)

9-byte-array
Address of nine bytes where the string is to be placed. (This is an output parameter.)

Note

The string is passed by reference rather than by descriptor.

3.7.5 Return System Time in Seconds

FOR$SECNDS returns the system time in seconds as a F_floating value minus the value of its argument. This procedure will show the correct time difference through midnight into the next day.

Format

time-difference = FOR$SECNDS (time-origin)

time-difference
Address of location to receive the F_floating system time in seconds.

time-origin
Address of F_floating value of reference time.

Messages

SS$_FLTOVF
Floating overflow.

SS$_FLTUND
Floating underflow.

3.7.6 Return System Time as 8–Byte String

FOR$TIME returns the system time as an 8–byte string in the form HH:MM:SS.

Format

CALL FOR$TIME (8–byte-array)

8–byte-array
Address of eight bytes where the string is to be placed. (This is an output parameter.)

Notes

The 8–byte array is a string passed by reference.

The time of day is truncated to seconds using the system service Convert Binary Time to ASCII string ($ASCTIM).

3.7.7 Return Day Number as a Longword Integer

LIB$DAY returns the number of days since the system zero date of November 17, 1858. Optionally, the caller can supply a quadword by reference containing a time in system time format to be used instead of the system time.

NOTE

If the caller supplies a quadword time, it is not verified. If it is negative (bit 63 on), the day-number value returned is negative.

An optional return argument is a longword integer containing the number of 10 millisecond units since midnight.

Format

ret-status = LIB$DAY (day-number [,user-time [,day-time]])

day-number
Address of a longword containing the number of days since the system zero date.

user-time
Address of a quadword containing a time in 100 nanosecond units. (This is an optional parameter. The default is the current system time.)

day-time
Address of a longword containing the number of 10 millisecond units since midnight. (This is an optional output parameter.)

Return Status

SS$_NORMAL
Routine successfully completed.

SS$_INTOVF
The option argument user-time is present and represents a date past the year 8600.

Example

The following BASIC code fragment shows how you could use LIB$DAY to obtain the number of days between two dates.

```
100  EXTERNAL INTEGER FUNCTION SYS$BINTIM, LIB$DAY
110  COM INTEGER USER_TIME, FILL
120  DECLARE INTEGER RET_STATUS

300  DEF FNDAY%(DAY_TIME$)
 \        RET_STATUS = SYS$BINTIM(DAY_TIME$,USER_TIME)
 \        IF (RET_STATUS AND 1%) = 0% THEN
               CALL LIB$STOP(RET_STATUS BY VALUE)
320      RET_STATUS = LIB$DAY(DAY_TMP%,USER_TIME)
 \        IF (RET_STATUS AND 1%) = 0% THEN
               CALL LIB$STOP(RET_STATUS BY VALUE)
330      FNDAY% = DAY_TMP%
 \   FNEND

400  INPUT "Enter two dates(dd-mmm-yyyy)";DAY1$,DAY2$
410  PRINT "Number of days between is";
 \        FNDAY%(DAY2$) - FNDAY%(DAY1$)
999  END
```

LIB$DATE_TIME

3.7.8 Return System Date and Time as a String

LIB$DATE_TIME returns the VAX/VMS system date and time in the semantics of a user-provided string.

Format

ret-status = LIB$DATE_TIME (dst-str)

dst-str
Address of a fixed-length or dynamic destination string descriptor.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$__STRTRU
Success, but destination string was truncated.

LIB$__INSVIRMEM
Insufficient virtual memory.

LIB$__INVSTRDES
Invalid string descriptor.

3.8 Miscellaneous Procedures

The procedures in this section are those general utility procedures that do not belong to any group of related procedures.

LIB$AST__IN__PROG

3.8.1 AST in Progress

An asynchronous system trap (AST) is a VAX/VMS mechanism for providing a software interrupt when an external event occurs, such as the user typing CTRL/C. When an external event occurs, the current execution is interrupted and a user-declared AST procedure is called. While that procedure is active, the AST is said to be in progress. When the user AST procedure returns to the user program, the AST is disabled and execution continues where it left off.

LIB$AST__IN__PROG is provided for the convenience of programmers writing AST reentrant software (which takes different actions depending on whether an AST is in progress). For example, the procedure might have two separate statically allocated storage areas, one for AST level and one for non-AST level.

Format

in-progress = LIB$AST__IN__PROG ( )

in-progress
Indicator of whether an AST is currently in progress (value=1) or not (value=0).

3.8.2 Calculate Cyclic Redundancy Check (CRC)

LIB$CRC calculates the cyclic redundancy check (CRC) for a data stream. The CRC is returned for the data stream specified. See the *VAX-11 Architecture Handbook* for a description of the algorithms used in computing the CRC.

Format

crc = LIB$CRC (table, inicrc, stream)

table
Address of CRC table, which is an array of 16 longwords.

inicrc
Address of a longword containing the initial CRC.

stream
Address of a string descriptor for the data stream.

crc
Longword containing the computed cyclic redundancy check.

Example

The following FORTRAN code segment produces a DIGITAL Data Communications Message Protocol (DDCMP) CRC table and then computes the CRC for the string 'ABCDEFG'.

```
DIMENSION TABLE(16)
INTEGER*4 TABLE
CALL LIB$CRC_TABLE('120001'O, TABLE)
INTEGER*2 CRC
CRC = LIB$CRC (TABLE, 0, 'ABCDEFG')
```

In this example, only the low 16 bits of the result are used.

3.8.3 Construct Cyclic Redundancy Check (CRC) Table

LIB$CRC_TABLE constructs a 16-longword table that uses a CRC polynomial specification as a bit mask. This table can be passed to the LIB$CRC procedure for generating the CRC value for a stream of characters. See the *VAX-11 Architecture Handbook* for a description of how the table is generated.

Format

CALL LIB$CRC_TABLE (poly, table)

poly
Address of the longword containing a bit mask indicating which polynomial coefficients are to be generated.

table
Address of the 16-longword table that is to be produced. (This is an output parameter.)

Example

See Section 3.8.2.

3.8.4 Emulate VAX-11 Instructions

LIB$EMULATE intercepts "opcode reserved to DIGITAL" faults generated by attempts to execute VAX-11 instructions on processors which do not implement them, and simulates execution as if the processor did support the instruction. New instructions which are added to the VAX-11 architecture may not be implemented on all VAX-11 processors. LIB$EMULATE will emulate any non-privileged VAX-11 instruction if the processor does not support the instruction.

For this release of VMS, LIB$EMULATE will execute all instructions which manipulate the G_floating, H_floating and octaword data types. See the *VAX-11 Architecture Handbook* for more information on these instructions.

LIB$EMULATE is a condition handler that emulates execution of VAX-11 instructions that are not implemented on the host processor. If LIB$EMULATE can emulate the instruction, execution control never returns to the routine which called it; the exception essentially disappears.

Any exceptions that arise while emulating the instruction appear as if they were caused by the instruction itself. Floating overflow, underflow, and

divide-by-zero exceptions will be signaled as faults rather than traps. Processors that implement these instructions always fault. See the *VAX-11 Architecture Handbook* for more information on faults. LIB$SIM_TRAP (see Section 3.8.6) can be used to convert faults to traps.

Format

ret-status = LIB$EMULATE (sig-args-adr, mch-args-adr)

sig-args-adr
Address of the signal argument vector.

mch-args-adr
Address of the mechanism argument vector.

Return Status

SS$_RESIGNAL
Resignal condition to next handler. The exception was not one LIB$EMULATE could handle.

Notes

The preferred use of LIB$EMULATE is to establish it as a condition handler by the appropriate method for the source language. An alternative means is provided for users who do not want to modify the source program. The module LIB$ESTEMU in SYS$LIBRARY:STARLET.OLB uses the LIB$INITIALIZE facility to enable LIB$EMULATE as a condition handler before program execution begins. To use this method, link your program with LIB$ESTEMU as follows:

```
$ LINK program,SYS$LIBRARY:STARLET/INCLUDE=LIB$ESTEMU
```

If LIB$EMULATE is established this way, the new instructions will be available to all of 'program.'

LIB$ADDX

3.8.5 Multiple Precision Binary Procedures

The following routines can be used to perform addition and subtraction on signed two's-complement integers of arbitrary length. The integers are located in arrays of longwords. The higher addresses contain the higher precision parts of the values. The highest addressed longword contain the sign and 31-bits of precision. The remaining longwords contain 32-bits of precision. The number of longwords to be operated on is given by the optional argument, "len-adr." The default length is two which corresponds to the VAX-11 quadword data type.

The result is placed in the array addressed by the third argument. Any two or all three of the first three arguments can be the same. The operations performed are:

LIB$ADDX: result = a + b
LIB$SUBX: result = a - b

Format

ret-status = LIB$ADDX (a-adr, b-adr, result-adr [,len-adr])

ret-status = LIB$SUBX (a-adr, b-adr, result-adr [,len-adr])

a-adr
Address of an array containing a multiple precision signed, two's-complement integer.

b-adr
Address of an array containing a multiple precision signed, two's-complement integer.

result-adr
Address of an array to receive the result. (This is an output parameter.)

len-adr
Address of a longword containing the length in longwords of the arrays to be operated on. The length must be greater than one. (This is an optional parameter, the default is two.)

Return Status

SS$__NORMAL
Routine successfully completed.

SS$__INTOVF
Integer overflow. The result is correct, except the sign bit is lost.

LIB$__INVARG
Invalid argument. Length is less than two. The output array is unchanged.

Examples

In FORTRAN (where arrays by default are passed by reference):

```
INTEGER*4 A(2), B(2), C(2)
IF (.NOT. LIB$ADDX (A, B, C)) GO TO error
```

In BASIC (where arrays by default are passed by descriptor):

```
DIM A%(2%), B%(2%), C%(2%)
IF LIB$ADDX (A%() BY REF, B%() BY REF, &
     C%() BY REF) AND 1% <> 1% GOTO error
```

3.8.6 Simulate Floating Trap

LIB$SIM_TRAP converts floating faults to floating traps. It is designed to be enabled as a condition handler or be called by one.

LIB$SIM_TRAP intercepts floating overflow, underflow and divide-by-zero faults. When these conditions are detected, the routine simulates the instruction causing the condition up to the point where a fault should be signaled and signals the corresponding floating trap.

Since LIB$SIM_TRAP dissolves the condition handling for the original fault condition, the final condition signaled by the routine will be from the context of the instruction itself, rather than from the condition handler. The signaling path is identical to a hardware generated trap. The signal array is placed so the end of the table will be the user's stack pointer at the completion of the instruction (for traps), or at the beginning of the instruction (for faults). See the *VAX-11 Architecture Handbook* for more information on faults and traps.

Format

ret-status = LIB$SIM_TRAP (sig-args-adr, mech-args-adr)

sig-args-adr
: Address of the signal argument vector.

mech-args-adr
: Address of the mechanism argument vector.

Return Status

SS$_RESIGNAL
: Resignal condition to next handler. The exception was not one that LIB$SIM_TRAP could handle.

3.8.7 Extended Multiply and Integerize Procedures

The procedures described in this section provide the high-level language users with the capability to use the VAX hardware instructions EMODF, EMODD, EMODG and EMODH.

The floating-point multiplier extension operand (second operand) is concatenated with the floating-point multiplier (first operand) to gain "x" additional low order fraction bits. The multiplicand operand is multiplied by the extended multiplier operand. After multiplication, the integer portion is extracted and a "y"-bit floating-point number is formed from the fractional part of the product by truncating extra bits.

The multiplication yields a result equivalent to the exact product truncated to a fraction field of "y" bits. With respect to the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

"x" and "y" have the following values:

Instruction	x	bits	y	Procedure
EMODF	8	7:0	32	LIB$EMODF
EMODD	8	7:0	64	LIB$EMODD
EMODG	11	15:5	64	LIB$EMODG
EMODH	15	15:1	128	LIB$EMODH

Format

ret-status = LIB$EMODz (multiplier-adr, multext-adr, multiplicand-adr, int-adr, fract-adr)

where z = F for F_floating, D for D_floating, G for G_floating or H for H_floating.

multiplier-adr
Address of floating-point multiplier.

multext-adr
Address of the location containing the left-justified multiplier-extension bits. For F_ and D_floating, multext-adr points to an unsigned byte. For G_ and H_floating, multext-adr points to an unsigned word.

multiplicand-adr
Address of floating-point multiplicand.

int-adr
Address of a longword to receive the integer portion of the result.

fract-adr
Address of a floating-point value to receive the fractional portion of the result.

NOTE

The floating-point type referred to in the multiplier, multiplicand and the fractional portion of the result is either F_, D_, G_ or H_floating depending on the CALL entry-point.

Return Status

SS$_NORMAL
Routine successfully completed.

SS$__INTOVF
Integer overflow. The integer operand is replaced by the low order bits of the true result. Floating overflow is indicated by SS$__INTOVF also.

SS$__FLTUND
Floating underflow. The integer and fraction operands are replaced by zero.

SS$__ROPRAND
Reserved operand. The integer and fraction operands are unaffected.

3.8.8 Evaluate Polynomial Procedures

The procedures described in this section provide the high-level language users with the capability to use the VAX hardware instructions POLYF, POLYD, POLYG and POLYH.

The third operand is an array of floating-point coefficients. The coefficient of the highest order term of the polynomial is the lowest addressed element in the array. The data type of the coefficients is the same as the argument operand.

The evaluation is carried out by Horner's Method, and the result is stored at the location pointed to by the fourth operand. The result computed is:

if d = degree and x = argument, then
result = C[0]+X*(C[1]+X*(C[2]+...X*C[D]))

The unsigned word, degree operand, specifies the highest numbered coefficient to participate in the evaluation.

For further detail, refer to the *VAX-11 Architecture Handbook* for the description of POLY.

Format

ret-status = LIB$POLYz (arg-adr, degree-adr, coeff-adr, result-adr)

where z = F for F__floating, D for D__floating, G for G__floating or H for H__floating.

arg-adr
Address of the argument "X" in polynomial. "X" is either F__, D__, G__ or H__floating depending on the CALL entry-point.

degree-adr
Address of an unsigned word specifying the highest numbered coefficient to participate in the evaluation. If degree is zero, the result equals C[0].

coeff-adr
Address of an array of floating-point values. The data type of the coefficients is the same as the argument operand.

result-adr
Address of the floating-point result of the calculation. The data type of the result is the same as the argument operand. Intermediate multiplications are carried out using extended floating fractions (31 bits for POLYF, 63 bits for POLYD and POLYG and 127 bits for POLYH).

Return Status

SS$_NORMAL
Routine successfully completed.

SS$_FLTUND
Floating underflow. After rounding, the intermediate result is replaced by zero and the operation continues. If both overflow and underflow occur in the same instruction, the underflow condition is lost.

SS$_ROPRAND
Reserved operand. See the *VAX-11 Architecture Handbook* for details.

3.8.9 Queue Access Procedures

The following procedures are designed to give high-level languages access to the interlocked, self-relative queue instructions INSQHI, INSQTI, REMQHI and REMQTI. These instructions permit the user to insert a queue entry at the head or at the tail, or remove a queue entry from the head or from the tail of an interlocked, self-relative queue.

The remove queue instructions (REMQHI or REMQTI) have a particular problem with high-level languages that do not have pointers (BASIC, COBOL, and FORTRAN) since they return the address of the removed entry. One solution is to provide an optional action routine that is called with the address of the removed entry. Unfortunately this clean solution runs into another problem: FORTRAN passes procedures differently (ZEM: by reference to entry mask) from the other high-level languages (BPV: by reference to two longwords; address of the entry mask and address of the environment value). Also BASIC and COBOL do not allow procedures as parameters.

The user is restricted to what can be passed to the action routine. The BASIC and FORTRAN user can use the immediate value escape mechanisms to pass the address of the removed entry.

3.8.9.1 Queue Entry Inserted at Head — LIB$INSQHI inserts a queue entry at the head of the specified, self-relative, interlocked queue. The queues can be in process-private, processor-private, or processor-sharable memory to implement per-process, per-processor, or across-processor queues.

Format

ret-status = LIB$INSQHI (entry, header [,retry-cnt])

entry
Address of a quadword aligned array that must be at least 8–bytes long. Bytes following the first 8–bytes can be used for any purpose by the calling program.

header
Address of a quadword aligned quadword. It must be initialized to zero before first use of the queue; zero means an empty queue.

retry-cnt
Address of an unsigned longword integer containing the retry count to be used in case of secondary interlock failure of the queue instruction in a processor-shared memory application. This is an optional parameter, the default value is ten.

Return Status

SS$_NORMAL
Procedure successfully completed. Entry added to front of the queue, the resulting queue contains more than one entry.

LIB$_ONEENTQUE
Procedure successfully completed. Entry added to front of the queue, the resulting queue contains one entry.

LIB$_SECINTFAI
Secondary interlock failed (severe error) retry-cnt times in a row. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

Examples

In BASIC and FORTRAN, queues can be quadword aligned in a named COMMON block, say QUEUES, by using a linker option file to specify PSECT alignment. The linker command should contain ..., FILE/OPTIONS, ... where FILE.OPT is a linker option file containing the line:

```
PSECT = QUEUES, QUAD
```

For a FORTRAN application using processor-shared memory:

```
INTEGER*4 FUNCTION INSERT_Q (QENTRY)
COMMON/QUEUES/QHEADER
INTEGER*4 QENTRY(10), QHEADER(2)
INSERT_Q = LIB$INSQHI (QENTRY, QHEADER)
RETURN
END
```

A BASIC application using processor-shared memory:

```
COM (QUEUES) QENTRY%(9), QHEADER%(1)
EXTERNAL INTEGER FUNCTION LIB$INSQHI
IF LIB$INSQHI (QENTRY%() BY REF, QHEADER%() BY REF) AND 1%
     THEN GOTO 1000
1000 REM        INSERTED OK
```

LIB$INSQTI

3.8.9.2 Queue Entry Inserted at Tail — LIB$INSQTI inserts a queue entry at the tail of the specified, self-relative, interlocked queue. The queues can be in process-private, processor-private, or processor-sharable memory to implement per-process, per-processor, or across-processor queues.

Format

ret-status = LIB$INSQTI (entry, header [,retry-cnt])

entry
Address of a quadword aligned array that must be at least 8-bytes long. Bytes following the first 8-bytes can be used for any purpose by the calling program.

header
Address of a quadword aligned quadword. It must be initialized to zero before first use of the queue; zero means an empty queue.

retry-cnt
Address of an unsigned longword integer containing the retry count to be used in case of secondary interlock failure of the queue instruction in a processor-shared memory application. (This is an optional parameter, the default value is ten.)

Return Status

SS$_NORMAL
Procedure successfully completed. Entry added to tail of the queue, the resulting queue contains more than one entry.

LIB$_ONEENTQUE
Procedure successfully completed. Entry added to tail of the queue, the resulting queue contains one entry.

LIB$__SECINTFAI
Secondary interlock failed (severe error) retry-cnt times in a row. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

LIB$REMQHI

3.8.9.3 Queue Entry Removed at Head — LIB$REMQHI removes a queue entry from the head of the specified, self-relative, interlocked queue. The queues can be in process-private, processor-private, or processor-sharable memory to implement per-process, per-processor, or across-processor queues.

Format

ret-status = LIB$REMQHI (header, remque-adr [,retry-cnt])

header
Address of a quadword aligned quadword. It must be initialized to zero before first use of the queue; zero means an empty queue.

remque-adr
Address of a longword to receive the address of the removed entry. If the queue was empty, remque-adr is set to the address of the header.

retry-cnt
Address of an unsigned longword integer containing the retry count to be used in case of secondary interlock failure of the queue instruction in a processor-shared memory application. (This is an optional parameter, the default value is ten.)

Return Status

SS$__NORMAL
Procedure successfully completed. Entry removed from front of the queue, resulting queue contains one or more entries.

LIB$__ONEENTQUE
Procedure successfully completed. Entry removed from front of the queue, resulting queue is now empty.

LIB$__SECINTFAI
Secondary interlock failed (severe error) retry-cnt times in a row. The queue is not modified. This condition can only occur when the queue is in memory being shared between two or more processors.

LIB$__QUEWASEMP
Queue was empty. The queue is not modified.

3.8.9.4 Queue Entry Removed from Tail — LIB$REMQTI removes a queue entry from the tail of the specified, self-relative, interlocked queue. The queues can be in process-private, processor-private, or processor-sharable memory to implement per-process, per-processor, or across-processor queues.

Format

ret-status = LIB$REMQTI (header, remque-adr [,retry-cnt])

header
Address of a quadword aligned quadword. It must be initialized to zero before first use of the queue; zero means an empty queue.

remque-adr
Address of a longword to receive the address of the removed entry. If the queue was empty, remque-adr is set to the address of the header.

retry-cnt
Address of an unsigned longword integer containing the retry count to be used in case of secondary interlock failure of the queue instruction in a processor-shared memory application. (This is an optional parameter, the default value is ten.)

Return Status

SS$_NORMAL
Procedure successfully completed. Entry removed from tail of the queue, the resulting queue contains one or more entries.

LIB$_ONEENTQUE
Procedure successfully completed. Entry removed from tail of the queue, the resulting queue is empty.

LIB$_SECINTFAI
Secondary interlock failed (severe error) retry-cnt times in a row. The queue is not modified. This condition can occur only when the queue is in memory being shared between two or more processors.

LIB$_QUEWASEMP
Queue was empty. The queue is not modified.

Example

In FORTRAN, the address of the removed queue entry can be passed to another procedure as an array using the %VAL built-in function. In the following example, queue entries are ten longwords including the two longword pointers at the beginning of each entry.

```
COMMON/QUEUES/QHEADER
INTEGER*4 QHEADER(2), ISTAT
ISTAT = LIB$REMQHI (QHEADER, ADDR)
IF (ISTAT) THEN
     CALL PROC (%VAL (ADDR))!process removed entry
     GO TO ...
     ELSE IF (ISTAT .EQ. %LOC(LIB$_QUEWASEMP)) THEN
          GO TO ...          !queue was empty
          ELSE IF
               ...           !secondary interlock failed
END IF
.
.
.
END

SUBROUTINE PROC (QENTRY)
INTEGER*4 QENTRY(10)
.
.
.
RETURN
END
```

Chapter 4
Mathematics Procedures

4.1 The Mathematics Procedures

This chapter describes these mathematics procedures:

- Floating-point
- Complex functions
- Exponentiation code-support
- Complex exponentiation
- Random Number Generator
- Miscellaneous functions

In addition, it describes language-independent arithmetic expression evaluation code-support procedures.

4.1.1 Entry Point Names

The names of the mathematics procedures consist of the language processor-defined function names with MTH$ as a prefix.

In most cases, when function parameters and values are the same data type, the first letter of the name indicates the data type. When function parameters and values are different data types, the first letter indicates the data type of the result, and the second letter indicates the data type of the parameter(s).

The letters used as data type prefixes are:

Letter	Data Type	FORTRAN Declarator
I	word	INTEGER*2
J	longword	INTEGER*4
D	D_floating	REAL*8
G	G_floating	REAL*8
H	H_floating	REAL*16
C	F_complex	COMPLEX*8
CD	D_complex	COMPLEX*16
CG	G_complex	COMPLEX*16

F_floating data types have no letter designation.

For example, MTH$SIN returns an F_floating value of the sine of an F_floating parameter, and MTH$DSIN returns a D_floating value of the sine of a D_floating parameter.

Language-independent arithmetic expression evaluation procedures use the OTS$ prefix. In addition, the data type letters are the last letters of the entry point name, rather than the first, and the letter R indicates F_floating values. For example, OTS$POWRJ returns an F_floating value of an F_floating parameter raised to a longword power.

4.1.2 Calling Conventions

All mathematics procedures with an entry point name starting with MTH$ accept parameters passed by reference, except for the JSB entry points (see Chapter 2). All MTH$ routines return values in R0 or R0/R1 except those for which the values cannot fit in 64 bits, namely D_complex, G_complex or H_floating values. The latter procedures return their function values via the first argument in the argument list with the nominal argument list shifted one position to the right.

The notation JSB MTH$name_Rn, where n is the reference register number, indicates an equivalent JSB entry point. Procedures with JSB entry points accept a single parameter in R0, R0/R1 or R0:R3, and return a single value to R0, R0/R1, or R0:R3. No registers are saved; only registers R0:Rn are changed.

All mathematics procedures with an entry point name starting with OTS$ pass input parameters by immediate value, including double floating-point and complex numbers.

NOTE

This is a violation of the VAX-11 Procedure Calling Standard, which specifies that a maximum of 32 bits can be passed by immediate value. However, the standard exempts code support procedures.

For compactness of notation, double floating-point and complex parameters are indicated as a single parameter passed by immediate value. Function values are returned in R0, R0/R1 or R0:R3.

NOTE

Returning values in R0:R3 is also contrary to the VAX-11 Procedure Calling Standard.

All mathematics CALL entry points disable floating-point underflow, enable integer overflow, cause no floating-point overflow or other arithmetic traps, and preserve enable operations across the call. JSB entry points execute in the context of the caller with the enable operations as set by the caller. However, since the procedures do not cause arithmetic traps, their operation is not affected by the setting of the arithmetic trap enables, except as noted.

4.1.3 Algorithms

If the algorithm used by a procedure is not too extensive, it is included in the functional description. Otherwise, the algorithm is in Appendix D. The algorithms in Appendix D are in the same relative sequence as the procedure descriptions in this chapter.

4.1.4 Error Handling

Errors are indicated by both the MTH$ and OTS$ procedures using the VAX-11 signaling mechanism (see Chapter 6). All errors are signaled as SEVERE by calling LIB$SIGNAL, so that the default operation causes the image to exit after printing an error message. However, a user-established condition handler can cause execution to continue at the point of the error by returning SS$_CONTINUE. A mathematics procedure returns to its caller after R0/R1 have been restored from the signal mechanism vector CHF$L_MCH_SAVR0/R1. Thus, the user-established handler should correct CHF$L_MCH_SAVR0/R1 to the desired function value to be returned to the caller of the mathematics procedure.

Correcting D_complex, G_complex or H_floating cannot be done with complete generality since R2/R3 are not available in the mechanism vector.

Note that it is more reliable to correct R0 and R1 to resemble R0 and R1 of a double-precision, floating-point value. Then, the correction works for both single and double precision.

If the correction is not performed, the reserved operand -0.0 is returned. Accessing the -0.0 as a floating-point quantity will cause a reserved operand fault. See Chapter 6 for a complete description of how to write user handlers, including handlers for mathematics errors.

For a small number of mathematics procedures, floating underflow is signaled if the calling program (JSB or CALL) has enabled floating underflow faults (or traps); such a possibility is indicated in the Messages section of the procedure description.

All mathematics procedures access input parameters and the real and imaginary parts of complex numbers using floating-point instructions. Therefore, a reserved operand fault can occur in any mathematics procedure. To save repetition, the resulting message (SS$_ROPRAND) is not listed in the Messages section.

4.1.5 Summary of Mathematics Procedures

Table 4–1 lists all the mathematics procedures alphabetically by English name, ignoring the first word if it is a data type. The sections that follow the table describe the procedures in detail.

Table 4–1: Mathematics Procedures

Function	Call Entry Point Name	Type of Parameter	Type of Result	Section
Absolute	MTH$ABS	F_floating	F_floating	4.7
	MTH$CABS	F_complex	F_floating	4.3.1
|x|	MTH$DABS	D_floating	D_floating	4.7
	MTH$CDABS	D_complex	D_floating	4.3.1
	MTH$GABS	G_floating	G_floating	4.7
	MTH$CGABS	G_complex	G_floating	4.3.1
	MTH$HABS	H_floating	H_floating	4.7
	MTH$IIABS	Word	Word	4.7
	MTH$JIABS	Longword	Longword	4.7
Arc Cosine	MTH$ACOS	F_floating	F_floating	4.2.1
	MTH$DACOS	D_floating	D_floating	4.2.1
Arc Cos(x)	MTH$GACOS	G_floating	G_floating	4.2.1
	MTH$HACOS	H_floating	H_floating	4.2.1
Arc Sine	MTH$ASIN	F_floating	F_floating	4.2.2
	MTH$DASIN	D_floating	D_floating	4.2.2
Arc Sin (x)	MTH$GASIN	G_floating	G_floating	4.2.2
	MTH$HASIN	H_floating	H_floating	4.2.2
Arc Tangent	MTH$ATAN	F_floating	F_floating	4.2.3
	MTH$DATAN	D_floating	D_floating	4.2.3
Arc Tan (x)	MTH$GATAN	G_floating	G_floating	4.2.3
	MTH$HATAN	H_floating	H_floating	4.2.3
Arc Tangent with	MTH$ATAN2	F_floating	F_floating	4.2.4
Two Parameters	MTH$DATAN2	D_floating	D_floating	4.2.4
	MTH$GATAN2	G_floating	G_floating	4.2.4
Arc Tan (x1/x2)	MTH$HATAN2	H_floating	H_floating	4.2.4
Bitwise	MTH$IIAND	Word	Word	4.7
Logical AND	MTH$JIAND	Longword	Longword	4.7
Bitwise	MTH$INOT	Word	Word	4.7
Complement	MTH$JNOT	Longword	Longword	4.7
Bitwise	MTH$IIEOR	Word	Word	4.7
Exclusive OR	MTH$JIEOR	Longword	Longword	4.7
Bitwise	MTH$IIOR	Word	Word	4.7
Inclusive OR	MTH$JIOR	Longword	Longword	4.7

(continued on next page)

Table 4-1: Mathematics Procedures (Cont.)

Function	Call Entry Point Name	Type of Parameter	Type of Result	Section
Bitwise	MTH$IISHFT	Word	Word	4.7
Shift	MTH$JISHFT	Longword	Longword	4.7
Common	MTH$ALOG10	F_floating	F_floating	4.2.5
Logarithm	MTH$DLOG10	D_floating	D_floating	4.2.5
log 10 (x)	MTH$GLOG10	G_floating	G_floating	4.2.5
	MTH$HLOG10	H_floating	H_floating	4.2.5
Complex from	MTH$CMPLX	F_floating	F_complex	4.3.7
Two Parameters	MTH$DCMPLX	D_floating	D_complex	4.3.7
	MTH$GCMPLX	G_floating	G_complex	4.3.7
Conjugate of	MTH$CONJG	F_complex	F_complex	4.3.2
Complex Number	MTH$DCONJG	D_complex	D_complex	4.3.2
	MTH$GCONJG	G_complex	G_complex	4.3.2
Convert D_	MTH$CVT_D_G	D_floating	G_floating	4.7
to G_floating	MTH$CVT_DA_GA	D_array	G_array	4.7
Convert G_	MTH$CVT_G_D	G_floating	D_floating	4.7
to D_floating	MTH$CVT_GA_DA	G_array	D_array	4.7
Cosine	MTH$COS	F_floating	F_floating	4.2.6
	MTH$CCOS	F_complex	F_complex	4.3.3
	MTH$DCOS	D_floating	D_floating	4.2.6
	MTH$CDCOS	D_complex	D_complex	4.3.3
	MTH$GCOS	G_floating	G_floating	4.2.6
	MTH$CGCOS	G_complex	G_complex	4.3.3
	MTH$HCOS	H_floating	H_floating	4.2.6
Division of	OTS$DIVC	F_complex	F_complex	4.3.4
Complex Number	OTS$DIVCD_R3	D_complex	D_complex	4.3.4
	OTS$DIVCG_R3	G_complex	G_complex	4.3.4
Double from	MTH$DBLE	F_floating	D_floating	4.7
Single Precision	MTH$GDBLE	F_floating	G_floating	4.7
Exponential	MTH$EXP	F_floating	F_floating	4.2.7
e ** x	MTH$CEXP	F_complex	F_complex	4.3.5
	MTH DEXP	D_floating	D_floating	4.2.7
	MTH$CDEXP	D_complex	D_complex	4.3.5
	MTH$GEXP	G_floating	G_floating	4.2.7
	MTH$CGEXP	G_complex	G_complex	4.3.5
	MTH$HEXP	H_floating	H_floating	4.2.7
Exponentiation	OTS$POWCC	F_complex**F_complex	F_complex	4.5.1
Procedures	OTS$POWCDCD_R3	D_complex**D_complex	D_complex	4.5.1
	OTS$POWCDJ	D_complex**Longword	D_complex	4.5.2
(Complex)	OTS$POWCGCG_R3	G_complex**G_complex	G_complex	4.5.1
	OTS$POWCGJ	G_complex**Longword	G_complex	4.5.2
	OTS$POWCJ	F_complex**Longword	F_complex	4.5.2
Exponentiation	OTS$POWDD	D_float**D_float	D_floating	4.4.1
Procedures	OTS$POWDJ	D_float**Longword	D_floating	4.4.1
	OTS$POWDR	D_float**F_float	D_floating	4.4.1
(Real)	OTS$POWGG	G_float**G_float	G_floating	4.4.2
	OTS$POWGJ	G_float**Longword	G_floating	4.4.2

(continued on next page)

Table 4-1: Mathematics Procedures (Cont.)

Function	Call Entry Point Name	Type of Parameter	Type of Result	Section
	OTS$POWHH_R3	H_float**H_float	H_floating	4.4.3
	OTS$POWHJ_R3	H_float**Longword	H_floating	4.4.3
	OTS$POWII	Word**Word	Word	4.4.4
	OTS$POWJJ	Longword**Longword	Longword	4.4.5
	OTS$POWRD	F_float**D_float	D_floating	4.4.6
	OTS$POWRJ	F_float**Longword	F_floating	4.4.6
	OTS$POWRR	F_float**F_float	F_floating	4.4.6
Fix (Float-Int)	MTH$IIFIX	F_floating	Word	4.7
	MTH$JIFIX	F_floating	Longword	4.7
Float (Int-float)	MTH$FLOATI	Word	F_floating	4.7
	MTH$DFLOTI	Word	D_floating	4.7
	MTH$GFLOTI	Word	G_floating	4.7
	MTH$FLOATJ	Longword	F_floating	4.7
	MTH$DFLOTJ	Longword	D_floating	4.7
	MTH$GFLOTJ	Longword	G_floating	4.7
Greatest Integer Less than Input Value	MTH$FLOOR	F_floating	F_floating	4.7
	MTH$DFLOOR	D_floating	D_floating	4.7
	MTH$GFLOOR	G_floating	G_floating	4.7
	MTH$HFLOOR	H_floating	H_floating	4.7
Hyperbolic Cosine COSH(x)	MTH$COSH	F_floating	F_floating	4.2.8
	MTH$DCOSH	D_floating	D_floating	4.2.8
	MTH$GCOSH	G_floating	G_floating	4.2.8
	MTH$HCOSH	H_floating	H_floating	4.2.8
Hyperbolic Sine SINH (x)	MTH$SINH	F_floating	F_floating	4.2.9
	MTH$DSINH	D_floating	D_floating	4.2.9
	MTH$GSINH	G_floating	G_floating	4.2.9
	MTH$HSINH	H_floating	H_floating	4.2.9
Hyperbolic Tangent TANH (x)	MTH$TANH	F_floating	F_floating	4.2.10
	MTH$DTANH	D_floating	D_floating	4.2.10
	MTH$GTANH	G_floating	G_floating	4.2.10
	MTH$HTANH	H_floating	H_floating	4.2.10
Imaginary Part of Complex	MTH$AIMAG	F_complex	F_floating	4.3.6
	MTH$DIMAG	D_complex	D_floating	4.3.6
	MTH$GIMAG	G_complex	G_floating	4.3.6
Maximum (Returns the maximum value from among the input parameters list; there must be at least two input parameters.)	MTH$IMAX0	Word	Word	4.7
	MTH$AIMAX0	Word	F_floating	4.7
	MTH$JMAX0	Longword	Longword	4.7
	MTH$AJMAX0	Longword	F_floating	4.7
	MTH$AMAX1	F_floating	F_floating	4.7
	MTH$DMAX1	D_floating	D_floating	4.7
	MTH$GMAX1	G_floating	G_floating	4.7
	MTH$HMAX1	H_floating	H_floating	4.7
	MTH$IMAX1	F_floating	Word	4.7
	MTH$JMAX1	F_floating	Longword	4.7

(continued on next page)

Table 4-1: Mathematics Procedures (Cont.)

Function	Call Entry Point Name	Type of Parameter	Type of Result	Section
Minimum (Returns the minimum value from among the input parameter list; there must be at least two input parameters.)	MTH$IMIN0	Word	Word	4.7
	MTH$AIMIN0	Word	F_floating	4.7
	MTH$JMIN0	Longword	Longword	4.7
	MTH$AJMIN0	Longword	F_floating	4.7
	MTH$AMIN1	F_floating	F_floating	4.7
	MTH$DMIN1	D_floating	D_floating	4.7
	MTH$GMIN1	G_floating	G_floating	4.7
	MTH$HMIN1	H_floating	H_floating	4.7
	MTH$IMIN1	F_floating	Word	4.7
	MTH$JMIN1	F_floating	Longword	4.7
Multiplication of Complex Numbers	OTS$MULCD_R3	D_complex	D_complex	4.3.8
	OTS$MULCG_R3	G_complex	G_complex	4.3.8
Natural Logarithm $\log_e(x)$	MTH$ALOG	F_floating	F_floating	4.2.11
	MTH$CLOG	F_complex	F_complex	4.3.9
	MTH$DLOG	D_floating	D_floating	4.2.11
	MTH$CDLOG	D_complex	D_complex	4.3.9
	MTH$GLOG	G_floating	G_floating	4.2.11
	MTH$CGLOG	G_complex	G_complex	4.3.9
	MTH$HLOG	H_floating	H_floating	4.2.11
Nearest Integer [x+.5*sign(x)]	MTH$ANINT	F_floating	F_floating	4.7
	MTH$DNINT	D_floating	D_floating	4.7
	MTH$IIDNNT	D_floating	Word	4.7
	MTH$JIDNNT	D_floating	Longword	4.7
	MTH$GNINT	G_floating	G_floating	4.7
	MTH$IIGNNT	G_floating	Word	4.7
	MTH$JIGNNT	G_floating	Longword	4.7
	MTH$HNINT	H_floating	H_floating	4.7
	MTH$IIHNNT	H_floating	Word	4.7
	MTH$JIHNNT	H_floating	Longword	4.7
	MTH$ININT	F_floating	Word	4.7
	MTH$JNINT	F_floating	Longword	4.7
Positive Difference x1-(min(x1,x2))	MTH$DIM	F_floating	F_floating	4.7
	MTH$DDIM	D_floating	D_floating	4.7
	MTH$GDIM	G_floating	G_floating	4.7
	MTH$HDIM	H_floating	H_floating	4.7
	MTH$IIDIM	Word	Word	4.7
	MTH$JIDIM	Longword	Longword	4.7
Product of two Floating Numbers	MTH$DPROD	F_floating	D_floating	4.7
	MTH$GPROD	F_floating	G_floating	4.7
Random Number	MTH$RANDOM	Longword	F_floating	4.6.1
Real Part of Complex Number	MTH$REAL	F_complex	F_floating	4.3.10
	MTH$DREAL	D_complex	D_floating	4.3.10
	MTH$GREAL	G_complex	G_floating	4.3.10

(continued on next page)

Table 4-1: Mathematics Procedures (Cont.)

Function	Call Entry Point Name	Type of Parameter	Type of Result	Section
Remainder	MTH$AMOD	F_floating	F_floating	4.7
	MTH$DMOD	D_floating	D_floating	4.7
x1-x2*[x1/x2]	MTH$GMOD	G_floating	G_floating	4.7
	MTH$HMOD	H_floating	H_floating	4.7
	MTH$IMOD	Word	Word	4.7
	MTH$JMOD	Longword	Longword	4.7
Sign Function	MTH$SGN	F_floating	Longword	4.7
(Returns a 1	MTH$SGN	D_floating	Longword	4.7
if x is positive,				
a -1 if x is				
negative, and				
0 if x is 0)				
Sign Transfer	MTH$SIGN	F_floating	F_floating	4.7
	MTH$DSIGN	D_floating	D_floating	4.7
|x1|*sign(x2)	MTH$GSIGN	G_floating	G_floating	4.7
	MTH$HSIGN	H_floating	H_floating	4.7
	MTH$IISIGN	Word	Word	4.7
	MTH$JISIGN	Longword	Longword	4.7
Sine	MTH$SIN	F_floating	F_floating	4.2.12
	MTH$CSIN	F_complex	F_complex	4.3.11
Sin(x)	MTH$DSIN	D_floating	D_floating	4.2.12
	MTH$CDSIN	D_complex	D_complex	4.3.11
	MTH$GSIN	G_floating	G_floating	4.2.12
	MTH$CGSIN	G_complex	G_complex	4.3.11
	MTH$HSIN	H_floating	H_floating	4.2.12
Single from	MTH$SNGL	D_floating	F_floating	4.7
double	MTH$SNGLG	G_floating	F_floating	4.7
Square Root	MTH$SQRT	F_floating	F_floating	4.2.13
	MTH$CSQRT	F_complex	F_complex	4.3.12
x ** (1/2)	MTH$DSQRT	D_floating	D_floating	4.2.13
	MTH$CDSQRT	D_complex	D_complex	4.3.12
	MTH$GSQRT	G_floating	G_floating	4.2.13
	MTH$CGSQRT	G_complex	G_complex	4.3.12
	MTH$HSQRT	H_floating	H_floating	4.2.13
Tangent	MTH$TAN	F_floating	F_floating	4.2.14
	MTH$DTAN	D_floating	D_floating	4.2.14
Tan (x)	MTH$GTAN	G_floating	G_floating	4.2.14
	MTH$HTAN	H_floating	H_floating	4.2.14
Truncated Integer	MTH$AINT	F_floating	F_floating	4.7
[x]	MTH$DINT	D_floating	D_floating	4.7
	MTH$IIDINT	D_floating	Word	4.7
	MTH$JIDINT	D_floating	Longword	4.7
	MTH$GINT	G_floating	G_floating	4.7
	MTH$IIGINT	G_floating	Word	4.7
	MTH$JIGINT	G_floating	Longword	4.7
	MTH$HINT	H_floating	H_floating	4.7
	MTH$IIHINT	H_floating	Word	4.7
	MTH$JIHINT	H_floating	Longword	4.7
	MTH$IINT	F_floating	Word	4.7
	MTH$JINT	F_floating	Longword	4.7

4.2 Floating-Point Mathematical Functions

This section describes all floating-point mathematical functions. In the procedure names:

- No letter denotes F_floating
- D denotes D_floating
- G denotes G_floating
- H denotes H_floating

The following chart shows the data type formats:

Data Type	Letter	Size Binary Bits	Exponent Binary Bits	Fraction Binary Bits	Precision Decimal Digitals
F_floating	None	32	8	24	7
D_floating	D	64	8	56	16
G_floating	G	64	11	53	15
H_floating	H	128	15	113	33

The function values returned by these procedures have the same data type as the input arguments. The calls are standard, by reference calls.

MTH$xACOS

4.2.1 Arc Cosine

The arc cosine procedures return the angle expressed in radians whose cosine is given by the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
angle = MTH$ACOS (x-adr)	MTH$ACOS_R4
angle = MTH$DACOS (x-adr)	MTH$DACOS_R7
angle = MTH$GACOS (x-adr)	MTH$GACOS_R7
CALL MTH$HACOS (angle-adr, x-adr)	MTH$HACOS_R8

x-adr
Address of the area containing the cosine, x. The absolute value of x must be less than or equal to 1.

angle
Angle in radians; 0 to PI.

angle-adr
Address of an area to receive the angle in radians; 0 to PI.

Message

MTH$__INVARGMAT
Invalid argument: |X| > 1, LIB$SIGNAL copies the reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector.

4.2.2 Arc Sine

The arc sine procedures return the angle expressed in radians whose sine is given by the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
angle = MTH$ASIN (x-adr)	MTH$ASIN__R4
angle = MTH$DASIN (x-adr)	MTH$DASIN__R7
angle = MTH$GASIN (x-adr)	MTH$GASIN__R7
CALL MTH$HASIN (angle-adr, x-adr)	MTH$HASIN__R8

x-adr
Address of the area containing the sine, x. The absolute value of x must be less than or equal to 1.

angle
Angle in radians; -PI/2 to +PI/2.

angle-adr
Address of an area to receive the angle in radians; -PI/2 to +PI/2.

Message

MTH$__INVARGMAT
Invalid argument: |x| > 1, LIB$SIGNAL copies the reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector.

4.2.3 Arc Tangent

The arc tangent procedures return the angle expressed in radians whose tangent is given by the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
angle = MTH$ATAN (x-adr)	MTH$ATAN_R4
angle = MTH$DATAN (x-adr)	MTH$DATAN_R7
angle = MTH$GATAN (x-adr)	MTH$GATAN_R7
CALL MTH$HATAN (angle-adr, x-adr)	MTH$HATAN_R8

x-adr
Address of the area containing the tangent, x.

angle
Angle in radians; -PI/2 to +PI/2.

angle-adr
Address of an area to receive the angle in radians; -PI/2 to +PI/2.

4.2.4 Arc Tangent with Two Parameters

The arc tangent procedures with two parameters return the angle expressed in radians whose tangent is given by two input parameters, x and y. See Appendix D for algorithms used in the calculations.

Format

CALL

```
angle = MTH$ATAN2 (x-adr, y-adr)
angle = MTH$DATAN2 (x-adr, y-adr)
angle = MTH$GATAN2 (x-adr, y-adr)
CALL MTH$HATAN2 (angle-adr, x-adr, y-adr)
```

x-adr
Address of the area containing the dividend portion of the input parameter.

y-adr
Address of the area containing the divisor portion of the input parameter.

angle
Angle in radians.

angle-adr
Address of an area to receive the angle in radians.

Message

MTH$__INVARGMAT
Invalid argument. Both x and y are zero. LIB$SIGNAL copies the reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector.

4.2.5 Common Logarithm

The common logarithm procedures return the common (base 10) logarithm of the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
log10 = MTH$ALOG10 (x-adr)	MTH$ALOG10__R5
log10 = MTH$DLOG10 (x-adr)	MTH$DLOG10__R8
log10 = MTH$GLOG10 (x-adr)	MTH$GLOG10__R8
CALL MTH$HLOG10 (log10-adr, x-adr)	MTH$HLOG10__R8

x-adr
Address of area containing the input value, x.

log10
Common logarithm of x.

log10-adr
Address of an area to receive the common logarithm of x.

Message

MTH$__LOGZERNEG
Logarithm of zero or negative value. x <= 0.0; LIB$SIGNAL copies reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector.

4.2.6 Cosine

The cosine procedures return the cosine of the angle input in radians. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
cosine = MTH$COS (x-adr)	MTH$COS_R4
cosine = MTH$DCOS (x-adr)	MTH$DCOS_R7
cosine = MTH$GCOS (x-adr)	MTH$GCOS_R7
CALL MTH$HCOS (cosine-adr, x-adr)	MTH$HCOS_R5

x-adr
Address of the area containing the angle, x, in radians.

cosine
Cosine of x.

cosine-adr
Address of an area to receive the cosine of x.

Message

MTH$_SIGLOSMAT

Significance lost in Math Library. Occurs if the magnitude of the argument is so large that significance is lost from the result. The permitted argument ranges are:

MTH$COS	-2**30 < X < 2**30
MTH$DCOS	-2**31 < X < 2**31
MTH$GCOS	-2**31 < X < 2**31
MTH$HCOS	-2**31 < X < 2**31

4.2.7 Exponential

The exponential procedures return the exponential value of the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
exp = MTH$EXP (x-adr)	MTH$EXP_R4
exp = MTH$DEXP (x-adr)	MTH$DEXP_R6
exp = MTH$GEXP (x-adr)	MTH$GEXP_R6
CALL MTH$HEXP (exp-adr, x-adr)	MTH$HEXP_R6

x-adr
Address of area containing the input parameter, x.

exp
Exponential of x.

exp-adr
Address of an area to receive the exponential of x.

Message

MTH$_FLOOVEMAT
Floating-point overflow in Math Library: x > yyy; LIB$SIGNAL copies reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector. The values of yyy are:

MTH$EXP	88.028
MTH$DEXP	88.028
MTH$GEXP	709.08
MTH$HEXP	11355.83

MTH$_FLOUNDMAT
Floating-point underflow in Math Library: x =< yyy and caller (CALL or JSB) has set hardware floating-point underflow enable. Result is set to 0.0. If the caller has not enabled floating-point underflow (the default), a result of 0.0 is returned but no error is signaled. The values of yyy are:

MTH$EXP	-89.416
MTH$DEXP	-89.416
MTH$GEXP	-709.79
MTH$HEXP	-11356.52

4.2.8 Hyperbolic Cosine

The hyperbolic cosine procedures return the hyperbolic cosine of the angle input in radians. See Appendix D for algorithms used in the calculations.

Format

CALL

cosh = MTH$COSH (x-adr)

cosh = MTH$DCOSH (x-adr)

cosh = MTH$GCOSH (x-adr)

CALL MTH$HCOSH (cosh-adr, x-adr)

x-adr
Address of the area containing the angle, x, in radians.

cosh
Hyperbolic cosine of x.

cosh-adr
Address of an area to receive the hyperbolic cosine of x.

Message

MTH$__FLOOVEMAT

Floating-point overflow in Math Library: |x| > yyy; LIB$SIGNAL copies reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector. The values of yyy are:

MTH$COSH	88.028
MTH$DCOSH	88.028
MTH$GCOSH	709.08
MTH$HCOSH	11355.83

4.2.9 Hyperbolic Sine

The hyperbolic sine procedures return the hyperbolic sine of the angle input in radians. See Appendix D for algorithms used in the calculations.

Format

CALL

sinh = MTH$SINH (x-adr)
sinh = MTH$DSINH (x-adr)
sinh = MTH$GSINH (x-adr)
CALL MTH$HSINH (sinh-adr, x-adr)

x-adr
Address of the area containing the angle, x, in radians.

sinh
Hyperbolic sine of x.

sinh-adr
Address of an area to receive the hyperbolic sine of x.

Messages

See messages for the hyperbolic cosine.

4.2.10 Hyperbolic Tangent

The hyperbolic tangent procedures return the hyperbolic tangent of the angle input in radians. See Appendix D for algorithms used in the calculations.

Format

CALL

tanh = MTH$TANH (x-adr)
tanh = MTH$DTANH (x-adr)
tanh = MTH$GTANH (x-adr)
CALL MTH$HTANH (tanh-adr, x-adr)

x-adr
Address of the area containing the angle, x, in radians.

tanh
Hyperbolic tangent of x.

tanh-adr
Address of an area to receive the hyperbolic tangent of x.

4.2.11 Natural Logarithm

The natural logarithm procedures return the natural (base e) logarithm of the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
natlog = MTH$ALOG (x-adr)	MTH$ALOG_R5
natlog = MTH$DLOG (x-adr)	MTH$DLOG_R8
natlog = MTH$GLOG (x-adr)	MTH$GLOG_R8
CALL MTH$HLOG (natlog-adr, x-adr)	MTH$HLOG_R8

x-adr
Address of the area containing the input value, x.

natlog
Natural logarithm of x.

natlog-adr
Address of an area to receive the natural logarithm of x.

Message

MTH$_LOGZERNEG
Logarithm of zero or negative value: x <= 0.0; LIB$SIGNAL copies reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector.

4.2.12 Sine

The sine procedures return the sine of the angle input in radians. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
sine = MTH$SIN (x-adr)	MTH$SIN_R4
sine = MTH$DSIN (x-adr)	MTH$DSIN_R7
sine = MTH$GSIN (x-adr)	MTH$GSIN_R7
CALL MTH$HSIN (sine-adr, x-adr)	MTH$HSIN_R5

x-adr
Address of the area containing the angle, x, in radians.

sine
Sine of x.

sine-adr
Address of an area to receive the sine of x.

Messages

See messages for the cosine procedures.

4.2.13 Square Root

The square root procedures return the square root of the input parameter. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB entry point
sqrt = MTH$SQRT (x-adr)	MTH$SQRT_R3
sqrt = MTH$DSQRT (x-adr)	MTH$DSQRT_R5
sqrt = MTH$GSQRT (x-adr)	MTH$GSQRT_R5
CALL MTH$HSQRT (sqrt-adr, x-adr)	MTH$HSQRT_R8

x-adr
Address of the area containing the input parameter, x.

sqrt
Square root of x.

sqrt-adr
Address of an area to receive the square root of x.

Message

MTH$_SQUROONEG
Square Root of negative number. LIB$SIGNAL copies reserved operand to the signal mechanism vector. Result is reserved operand -0.0 unless a condition handler changes the signal mechanism vector.

4.2.14 Tangent

The tangent procedures return the tangent of the angle input in radians. See Appendix D for algorithms used in the calculations.

Format

CALL	JSB Entry Point
tangent = MTH$TAN (x-adr)	MTH$TAN_R4
tangent = MTH$DTAN (x-adr)	MTH$DTAN_R7
tangent = MTH$GTAN (x-adr)	MTH$GTAN_R7
CALL MTH$HTAN (tangent-adr, x-adr)	MTH$HTAN_R5

x-adr
Address of the area containing the angle, x, in radians.

tangent
Tangent of x.

tangent-adr
Address of an area to receive the tangent of x.

Messages

MTH$_SIGLOSMAT
Significance lost in Math Library. See messages for the cosine procedures.

MTH$_FLOOVEMAT
Floating-point overflow in Math Library.

4.3 Complex Functions

The following library procedures perform computations on complex numbers. MTH$ procedures pass the complex data type by reference. This means that the address of two contiguous floating-point numbers is passed, the first being the real part and the second the imaginary part. OTS$ procedures pass the complex data type by immediate value as two separate floating-point quantities.

MTH$CxABS

4.3.1 Absolute Value

These procedures return the absolute value of a complex number as follows:

result = (ABS(MAX*SQRT((MIN/MAX)**2+1)),MAX)

where MAX is the larger of r and i, and MIN is the smaller of r and i.

Format

absolute-value = MTH$CABS (complex-number-adr)
CALL MTH$CDABS (absolute-value-adr, complex-number-adr)
CALL MTH$CGABS (absolute-value-adr, complex-number-adr)

complex-number-adr
Address of an area containing a complex number (r,i) where r and i are both floating-point values.

absolute-value
Absolute-value of a complex-number.

absolute-value-adr
Address of an area to receive the absolute-value of a complex-number.

Messages

MTH$__FLOOVEMAT
Floating-point overflow in Math Library. Both r and i are large.

4.3.2 Conjugate of a Complex Number

These procedures return the complex conjugate of the complex input parameter (r,i), that is, the complex value (r,–i) is returned.

Format

```
complex-conjugate = MTH$CONJG (complex-number-adr)
CALL MTH$DCONJG (complex-conjugate-adr, complex-number-adr)
CALL MTH$GCONJG (complex-conjugate-adr, complex-number-adr)
```

NOTE

The first parameter of the D_ or G_complex procedures is considered the return value; however, since it cannot fit in 64–bits, it is returned as the first argument according to the VAX–11 Procedure Calling Standard.

complex-number-adr
Address of an area containing a complex number (r,i) where r and i are floating-point numbers.

complex-conjugate
Complex value (r,–i) expressed in F_floating notation.

complex-conjugate-adr
Address of an area to receive the complex value (r,–i).

4.3.3 Cosine

These procedures return the complex cosine of a complex number (r,i) as follows:

result = (COS(i)*COSH(r), -SIN(r)*SINH(–i))

Format

```
complex-cosine = MTH$CCOS (complex-number-adr)
CALL MTH$CDCOS (complex-cosine-adr, complex-number-adr)
CALL MTH$CGCOS (complex-cosine-adr, complex-number-adr)
```

complex-number-adr
Address of an area containing a complex number (r,i) where r and i are floating-point numbers.

complex-cosine
Complex cosine of the complex input number expressed in F_floating notation.

complex-cosine-adr
Address of an area to receive the complex cosine of the complex input number.

Messages

MTH$_SIGLOSMAT
Significance lost in Math Library: |r| > 2**30 (F_floating) or |r| > 2**31 (D_, G_floating).

MTH$_FLOOVEMAT
Floating-point overflow in Math Library: |i| > 88.028 (F_, D_floating) or |i| > 709.08 (G_floating).

OTS$DIVCx

4.3.4 Division of Complex Numbers

These procedures return a complex result of a complex division on complex numbers.

The complex result is computed as follows:

1. Let (a,b) represent the complex dividend.
2. Let (c,d) represent the complex divisor.
3. Let (r,i) represent the complex quotient.

Then:

r = (ac+bd)/(cc+dd)
i = (bc–ad)/(cc+dd)

Format

complex-quotient = OTS$DIVC (dividend, divisor)
complex-quotient = OTS$DIVCD_R3 (dividend, divisor)
complex-quotient = OTS$DIVCG_R3 (dividend, divisor)

complex-quotient
For F_floating, the complex value returned in R0,R1 is (a,b)/(c,d). For D_ and G_floating, the complex value returned in R0:R3 is (a,b)/(c,d).

dividend, divisor
The complex values of the dividend and divisor are in the argument list.

4.3.5 Exponential

This procedure returns the complex exponential of the complex number (r,i). The result of the operation e**(r,i) is computed by:

complex-exp = (EXP(r)*COS(i), EXP(r)*SIN(i))

Format

complex-exp = MTH$CEXP (x-adr)
CALL MTH$CDEXP (complex-exp-adr,x-adr)
CALL MTH$CGEXP (complex-exp-adr,x-adr)

x-adr
Address of an area containing the input complex number (r,i) where both r and i are floating-point numbers.

complex-exp
Complex exponential of the complex input number expressed in F_floating notation.

complex-exp-adr
Address of an area to receive the complex exponential of x.

Messages

MTH$_SIGLOSMAT
Significance lost in Math Library: |i| > 2**30 (F_floating) or |i| > 2**31 (D_, G_floating).

MTH$_FLOOVEMAT
Floating-point overflow in Math Library: |r| >88.028 (F_, D_floating) or |r| > 709.08 (G_floating).

4.3.6 Imaginary Part of a Complex Number

These procedures return the imaginary part of a complex number.

Format

imag-part = MTH$AIMAG (complex-number-adr)
imag-part = MTH$DIMAG (complex-number-adr)
imag-part = MTH$GIMAG (complex-number-adr)

complex-number-adr
Address of an area containing the input complex number.

imag-part
The imaginary part of the input complex number.

4.3.7 Make Complex from Floating-Point

These procedures return a complex number from two floating-point values.

Format

```
complx = MTH$CMPLX (real-part-adr, imag-part-adr)
CALL MTH$DCMPLX (dcmplx-adr, real-part-adr, imag-part-adr)
CALL MTH$GCMPLX (gcmplx-adr, real-part-adr, imag-part-adr)
```

real-part-adr
: Address of an area containing the floating-point value to become the real part of a complex number.

imag-part-adr
: Address of an area containing the floating-point value to become the imaginary part of a complex number.

cmplx
: F_floating complex value of a complex number.

dcmplx-adr
: Address of an area to receive the D_floating complex value of a complex number.

gcmplx-adr
: Address of an area to receive the G_floating complex value of a complex number.

4.3.8 Multiplication

These procedures calculate the complex product of two complex values.

The complex product is computed as follows:

1. Let (a,b) represent the complex multiplier.
2. Let (c,d) represent the complex multiplicand.
3. Let (r,i) represent the complex product.

Then:

r = ac-bd and i = ad+bc

Format

```
product = OTS$MULCD_R3 (multiplier, multiplicand)
product = OTS$MULCG_R3 (multiplier, multiplicand)
```

multiplier
The multiplier is passed by immediate value.

multiplicand
The multiplicand is passed by immediate value.

product
D__ or G__complex value returned in registers R0:R3.

MTH$CxLOG

4.3.9 Natural Logarithm

These procedures return the complex natural logarithm of the complex number (r,i) computed as follows:

CLOG(arg) = (LOG(CABS(arg)), ATAN2(arg))

Format

complex-natlog = MTH$CLOG (arg-adr)
CALL MTH$CDLOG (complex-natlog-adr, arg-adr)
CALL MTH$CGLOG (complex-natlog-adr, arg-adr)

arg-adr
Address of an area containing the complex number.

complex-natlog
Natural logarithm of the complex number.

complex-natlog-adr
Address of an area to receive the complex natural logarithm of arg.

MTH$xREAL

4.3.10 Real Part of a Complex Number

These procedures return the real part of a complex number.

Format

real-part = MTH$REAL (complex-number-adr)
real-part = MTH$DREAL (complex-number-adr)
real-part = MTH$GREAL (complex-number-adr)

complex-number-adr
Address of an area containing the complex number.

real-part
The real part of the complex number.

4.3.11 Sine

These procedures return the complex sine of the complex number (r,i) computed as follows:

complex-sine = (sin(r)*COSH(i), +COS(r)*SINH (i))

Format

complex-sine = MTH$CSIN (complex-number-adr)
CALL MTH$CDSIN (complex-sine-adr, complex-number-adr)
CALL MTH$CGSIN (complex-sine-adr, complex-number-adr)

complex-number-adr
Address of an area containing a complex number (r,i) where r and i are floating-point numbers.

complex-sine
Complex sine of the complex input number.

complex-sine-adr
Address of an area to receive the complex sine of the complex number.

Messages

MTH$__SIGLOSMAT
Significance lost in Math Library: |r| > 2**30 (F__floating) or |r| > 2**31 (D__ or G__floating).

MTH$__FLOOVEMAT
Floating-point overflow in Math Library: |i| > 88.028 (F__, D__floating) or |i| > 709.08 (G__floating).

4.3.12 Square Root

These procedures return the complex square root of the complex number (r,i) computed as follows:

ROOT = SQRT ((ABS(r) + (ABS((r,i)))/2)
Q = i/(2* ROOT)

r	i	CSQRT((r,i))
>=0	any	(ROOT,Q)
<0	>=0	(Q, ROOT)

Format

```
complex-sqrt = MTH$CSQRT(x-adr)
CALL MTH$CDSQRT (complex-sqrt-adr, x-adr)
CALL MTH$CGSQRT (complex-sqrt-adr, x-adr)
```

x-adr
Address of an area containing the complex number (r,i).

complex-sqrt
The complex square root of x.

complex-sqrt-adr
Address of an area to receive the complex square root.

4.4 Exponentiation Code-Support Procedures

The following procedures support all high-level, language-compiled expressions that use the exponential operator ** or ^. BASIC, FORTRAN and PASCAL programs call these procedures implicitly in arithmetic expressions containing the ** or ^ operator. These procedures raise a base of one data type to a power of either the same data type or that of the exponent (whichever has greater range); for example, longword has greater range than word, and floating-point has greater range than longword. OTS$ procedures pass input parameters including complex numbers, by immediate value. Therefore, complex and double floating-point numbers actually occupy two longwords in the argument list; H_Floating numbers occupy four longwords.

Table 4-2 lists the exponentiation features described in this section. (See Section 4.5 for the complex exponentiation procedures.)

Table 4-2: Exponentiation Procedures

Procedure Name	Operation Base ** Exponent	Resulting Data
OTS$POWDD	D_floating ** D_floating	D_floating
OTS$POWDJ	D_floating ** Longword	D_floating
OTS$POWDR	D_floating ** F_floating	D_floating
OTS$POWGG	G_floating ** G_floating	G_floating
OTS$POWGJ	G_floating ** Longword	G_floating
OTS$POWHH_R3	H_floating ** H_floating	H_floating
OTS$POWHJ_R3	H_floating ** Longword	H_floating
OTS$POWII	Word ** Word	Word
OTS$POWJJ	Longword ** Longword	Longword
OTS$POWRD	F_floating ** D_floating	D_floating
OTS$POWRJ	F_floating ** Longword	F_floating
OTS$POWRR	F_floating ** F_floating	F_floating

4.4.1 D_floating Base

These procedures raise a D_floating base to a D_floating, longword or F_floating power. See Appendix D for algorithms used in the calculations.

Format

result = OTS$POWDD (base, exponent)
result = OTS$POWDJ (base, exponent)
result = OTS$POWDR (base, exponent)

base
D_floating (D) base (passed by immediate value).

exponent
D_floating (D), signed longword (J) or F_floating (R) exponent (passed by immediate value).

result
D_floating base ** specified exponent returned as a D_floating result.

Messages

SS$_FLTOVF
Arithmetic Trap. This error is signaled by the hardware if a floating-point overflow occurs.

MTH$_FLOOVEMAT
Floating-point overflow in Math Library.

MTH$_FLOUNDMAT
Floating-point underflow in Math Library.

MTH$_UNDEXP
Undefined exponentiation. This error is signaled if base is zero and exponent is zero or negative.

4.4.2 G_floating Base

These procedures raise a G_floating base to a G_floating or a longword power. See Appendix D for algorithms used in the calculations.

Format

result = OTS$POWGG (base, exponent)
result = OTS$POWGJ (base, exponent)

base
G_floating (G) base (passed by immediate value).

exponent
G_floating (G) or signed longword (J) exponent (passed by immediate value).

result
G_floating base ** specified exponent returned as a G_floating r.

Messages

SS$_FLTOVF
Arithmetic Trap. This error is signaled by the hardware if a floating-point overflow occurs.

MTH$_FLOOVEMAT
Floating-point overflow in Math Library.

MTH$_FLOUNDMAT
Floating-point underflow in Math Library.

MTH$_UNDEXP
Undefined exponentiation. This error is signaled if base is zero and exponent is zero or negative.

4.4.3 H_floating Base

These procedures raise an H_floating base to an H_floating or longword power. See Appendix D for algorithms used in the calculations.

Format

result = OTS$POWHH_R3 (base, exponent)
result = OTS$POWHJ_R3 (base, exponent)

base
H_floating (H) base (passed by immediate value).

exponent
H_floating (H) or signed longword (J) exponent (passed by immediate value).

result
H_floating base ** specified exponent returned as an H_floating result in registers R0:R3.

Messages

SS$_FLTOVF
Arithmetic Trap. This error is signaled by the hardware if a floating-point overflow occurs.

MTH$_FLOOVEMAT
Floating-point overflow in Math Library.

MTH$_FLOUNDMAT
Floating-point underflow in Math Library.

MTH$_UNDEXP
Undefined exponentiation. This error is signaled if base is zero and exponent is zero or negative.

4.4.4 Word Base

This procedure raises a word base to a word power. See Appendix D for algorithms used in the calculations.

Format

result = OTS$POWII (base, exponent)

base
Word base (passed by immediate value).

exponent
Word exponent (passed by immediate value).

result
Word base ** word exponent returned as a word result.

Messages

SS$_FLTDIV
Arithmetic Trap. This error is signaled by the hardware if a floating-point zero divide occurs.

SS$_FLTOVF
Arithmetic Trap. This error is signaled by the hardware if a floating-point overflow occurs.

MTH$_UNDEXP
Undefined exponentiation. This error is signaled if base is zero and exponent is zero or negative.

4.4.5 Longword Base

This procedure raises a signed longword base to a signed longword power. See Appendix D for algorithms used in the calculations.

Format

result = OTS$POWJJ (base, exponent)

base
Signed longword base (passed by immediate value).

exponent
Signed longword exponent (passed by immediate value).

result
Signed longword base ** longword exponent returned as a longword result.

Messages

SS$_FLTDIV
Arithmetic Trap. This error is signaled by the hardware if a floating-point zero divide occurs.

SS$_FLTOVF
Arithmetic Trap. This error is signaled by the hardware if a floating-point overflow occurs.

MTH$_UNDEXP
Undefined exponentiation. This error is signaled if base is zero and exponent is zero or negative.

4.4.6 F_floating Base

These procedures raise an F_floating base to a D_floating, longword or F_floating power. See Appendix D for algorithms used in the calculations.

Format

result = OTS$POWRD (base, exponent)
result = OTS$POWRJ (base, exponent)
result = OTS$POWRR (base, exponent)

base
F_floating (R) base (passed by immediate value).

exponent
D_floating (D), signed longword (J) or F_floating (R) exponent (passed by immediate value).

result
F_floating base ** D_floating exponent returned as a D_floating result. F_floating base ** signed longword or F_floating exponent returned as an F_floating result.

Messages

SS$_FLTOVF
Arithmetic Trap. This error is signaled by the hardware if a floating-point overflow occurs.

MTH$_FLOOVEMAT
Floating-point overflow Math Library.

MTH$_FLOUNDMAT
Floating-point underflow Math Library.

MTH$_UNDEXP
Undefined exponentiation. This error is signaled if base is zero and exponent is zero or negative.

4.5 Complex Exponentiation Procedures

The algorithms used for exponentiation of complex numbers depend on the exponent data type. Therefore, the procedures in this section are grouped by exponent rather than by base.

Table 4–3 lists the exponentiation features described in this section.

Table 4–3: Complex Exponentiation Procedures

Procedure Name	Operation Base ** Exponent	Resulting Data
OTS$POWCC	F_complex ** F_complex	F_complex
OTS$POWCDCD_R3	D_complex ** D_complex	D_complex
OTS$POWCGCG_R3	G_complex ** G_complex	G_complex
OTS$POWCJ	F_complex ** Longword	F_complex
OTS$POWCDJ_R3	D_complex ** Longword	D_complex
OTS$POWCGJ_R3	G_complex ** Longword	G_complex

4.5.1 Complex Floating-Point Power

These procedures return the result of raising a complex base to a complex power. The ANS FORTRAN X3.9-1978 Standard defines complex exponentiation as:

X ** Y = CEXP(Y * CLOG(X))
where X and Y are type COMPLEX.

Format

result = OTS$POWCC (base, exponent)
result = OTS$POWCDCD_R3 (base, exponent)
result = OTS$POWCGCG_R3 (base, exponent)

In each format, the result, base and exponent are of the same data type.

base
F_complex (C), D_complex (CD), or G_complex (CG) base (passed by immediate value).

exponent
F_complex (C), D_complex (CD), or G_complex (CG) exponent (passed by immediate value).

result
F_complex, D_complex, or G_complex result.

NOTE

For OTS$POWCDCD_R3 and OTS$POWCGCG_R3, the result is returned in registers R0:R3.

Messages

MTH$_INVARGMAT
Invalid argument in Math Library. Base is (0.,0.).

MTH$_FLOOVEMAT
Floating-point overflow in Math Library.

MTH$_SIGLOSMAT
Significance Lost in Math Library. Absolute value of the imaginary part of (Y * CLOG(X)) > 2**30 for F_complex or > 2**31 for D_ or G_complex.

SS$_ROPRAND
Reserved operand.

4.5.2 Signed Longword Integer Power

These procedures return a complex result of raising a complex base to an integer power. The complex result is given by:

Base	Exponent	Result
any	>0	product (base * 2 ** i) where i is each non-zero bit in lexponentl.
(0.,0.)	<=0	Undefined exponentiation.
not (0.,0.),	<0	product (base * 2 ** i) where i is each non-zero bit in lexponentl.
not (0.,0.)	=0	(1.0,0.0)

Format

result = OTS$POWCJ (base, exponent)
result = OTS$POWCDJ_R3 (base, exponent)
result = OTS$POWCGJ_R3 (base, exponent)

In each format, the result and base are of the same data type.

base
F_complex (C), D_complex (CD), or G_complex (CG) base (passed by immediate value).

exponent
Signed longword integer exponent (passed by immediate value).

result
F_, D_, or G_complex result.

NOTE

For OTS$POWCDJ_R3 and OTS$POWCGJ_R3, the result is returned in registers R0:R3.

Messages

SS$_FLTDIV
Floating-point zero divide occurred.

SS$_FLTOVF
Floating-point overflow occurred.

MTH$_UNDEXP
Undefined exponentiation.

4.6 Random Number Generators

MTH$RANDOM

4.6.1 Uniform Pseudorandom Number Generator

MTH$RANDOM is a general random number generator that is multiplicative congruential. This procedure is called again to obtain the next pseudorandom number. The seed is updated automatically. The result is a floating-point number that is uniformly distributed between 0.0 inclusively and 1.0 exclusively. There are no restrictions on the seed, although it should be initialized to different values on separate runs in order to obtain different random sequences. MTH$RANDOM uses the following to update the seed passed as the parameter:

SEED = 69069 * SEED + 1 (MOD 2**32)

The value of SEED is a 32-bit number whose high order 24 bits are converted to F_floating and returned as the result.

Format

result = MTH$RANDOM (seed)

seed
Address of unsigned longword.

result
F_floating random number.

Notes

Because the result is never 1.0, a simple way to get a uniform random integer selector is to multiply by the number of cases. For example, if a uniform choice among five situations is to be made, then the following FORTRAN sequence will work:

```
REAL MTH$RANDOM
   .
   .
   .
   .

GO TO (1,2,3,4,5) 1 + INT(5.*MTH$RANDOM(SEED))
```

Note that the explicit INT is necessary before adding 1 in order to avoid a possible rounding during the normalization after the addition of F_floating numbers.

The BASIC RND function explicitly invokes MTH$RANDOM.

4.7 Processor-Defined Mathematical Procedures

Processor-defined procedures include both the intrinsic and basic external functions defined in ANSI FORTRAN, which are treated in a uniform manner.

Table 4-4 presents these functions in condensed form to conserve space and facilitate ease of use. The procedures are divided into two groups: floating/integer conversion functions and miscellaneous functions. They are arranged in alphabetical order by English name within each group.

All procedures in this section pass input parameters by-reference. In the formats for each procedure, the names of the input parameter and function return values are the data types themselves — that is, word, longword, F_floating, D_floating, G_floating and H_floating. The English name starts with the data type of the function value if it is different from the data type of the input parameter or parameters.

The "Notes" column at the far right of Table 4-4 lists numbers that refer to the notes following the table. As with all procedures in this chapter, a reserved operand fault can occur for any floating-point input parameter and thus is not indicated in the notes.

Table 4-4: Miscellaneous Mathematics Functions

Name	Function / Format	Notes*
FLOATING/INTEGER CONVERSION FUNCTIONS		
MTH$CVT_D_G	Convert D_floating to G_floating (rounded) g-floating = MTH$CVT_D_G (d-floating)	
MTH$CVT_DA_GA	Convert D_float array to G_float array (rounded) CALL MTH$CVT_DA_GA (d-flt-array, g-flt-array [,cnt])	7
MTH$CVT_G_D	Convert G_floating to D_floating (exact) d-floating = MTH$CVT_G_D (g-floating)	2,6
MTH$CVT_GA_DA	Convert G_float array to D_float array (exact) CALL MTH$CVT_GA_DA (g-flt-array, d-flt-array [,cnt])	2,6,7
MTH$DBLE	Convert F_floating to D_floating (exact) d-floating = MTH$DBLE (f-floating)	
MTH$GDBLE	Convert F_floating to G_floating (exact) g-floating = MTH$GDBLE (f-floating)	
MTH$IIFIX	Convert to word (truncated) word = MTH$IIFIX (f-floating)	3
MTH$JIFIX	Convert to longword (truncated) longword = MTH$JIFIX (f-floating)	3
MTH$FLOATI	Convert word to F_floating (exact) f-floating = MTH$FLOATI (word)	

* The "Notes" column at the far right of Table 4-4 lists numbers that refer to the notes following the table.

Table 4-4: Miscellaneous Mathematics Functions (Cont.)

MTH$DFLOTI	Convert word to D_floating (exact) d-floating = MTH$DFLOTI (word)	
MTH$GFLOTI	Convert word to G_floating (exact) g-floating = MTH$GFLOTI (word)	
MTH$FLOATJ	Convert longword to F_floating (rounded) f-floating = MTH$FLOATJ (longword)	
MTH$DFLOTJ	Convert longword to D_floating (exact) d-floating = MTH$DFLOTJ (longword)	
MTH$GFLOTJ	Convert longword to G_floating (exact) g-floating = MTH$GFLOTJ (longword)	
MTH$FLOOR	Convert F_floating to greatest F_floating integer greatest-f-float-int = MTH$FLOOR (f-floating)	
MTH$DFLOOR	Convert D_floating to greatest D_floating integer greatest-d-float-int = MTH$DFLOOR (d-floating)	
MTH$GFLOOR	Convert G_floating to greatest G_floating integer greatest-g-float-int = MTH$GFLOOR (g-floating)	
MTH$HFLOOR	Convert H_floating to greatest H_floating integer CALL MTH$HFLOOR (greatest-h-float-int, h-floating)	5
MTH$AINT	Convert F_floating to truncated F_floating truncated_f-floating = MTH$AINT (f-floating)	2
MTH$DINT	Convert D_floating to truncated D_floating truncated-d-floating = MTH$DINT (d-floating)	
MTH$IIDINT	Convert D_floating to word (truncated) word = MTH$IIDINT (d-floating)	3
MTH$JIDINT	Convert D_floating to longword (truncated) longword = MTH$JIDINT (d-floating)	3
MTH$GINT	Convert G_floating to truncated G_floating truncated-g-floating = MTH$GINT (g-floating)	3
MTH$IIGINT	Convert G_floating to truncated word truncated-word = MTH$IIGINT (g-floating)	3
MTH$JIGINT	Convert G_floating to truncated longword truncated-longword = MTH$JIGINT (g-floating)	3
MTH$HINT	Convert H_floating to truncated H_floating CALL MTH$HINT (truncated-h-floating, h-floating)	2,5
MTH$IIHINT	Convert H_floating to truncated word truncated-word = MTH$IIHINT (h-floating)	3
MTH$JIHINT	Convert H_floating to truncated longword truncated-longword = MTH$$JIHINT (h-floating)	3
MTH$IINT	Convert F_floating to truncated word truncated-word = MTH$IINT (f-floating)	3
MTH$JINT	Convert F_floating to truncated longword truncated-longword = MTH$JINT (f-floating)	3

* The "Notes" column at the far right of Table 4-4 lists numbers that refer to the notes following the table.

Table 4-4: Miscellaneous Mathematics Functions (Cont.)

MTH$ANINT	Convert F_floating to nearest F_floating integer nearest-f-float-int = MTH$ANINT (f-floating)	
MTH$DNINT	Convert D_floating to nearest D_floating integer nearest-d-float-int = MTH$DNINT (d-floating)	2
MTH$IIDNNT	Convert D_floating to nearest word integer nearest-word-integer = MTH$IIDNNT (d-floating)	
MTH$JIDNNT	Convert D_floating to nearest longword integer nearest-long-int = MTH$JIDNNT (d-floating)	
MTH$GNINT	Convert G_floating to nearest G_floating integer nearest-g-float-int = MTH$GNINT (g-floating)	2
MTH$IIGNNT	Convert G_floating to nearest word integer nearest-word-integer = MTH$IIGNNT (g-floating)	3
MTH$JIGNNT	Convert G_floating to nearest longword integer nearest-longword-integer = MTH$JIGNNT (g-floating)	3
MTH$HNINT	Convert H_floating to nearest H_floating integer CALL MTH$HNINT (nearest-h-float-int, h-floating)	5
MTH$IIHNNT	Convert H_floating to nearest word integer nearest-word-integer = MTH$IIHNNT (h-floating)	3
MTH$JIHNNT	Convert H_floating to nearest longword integer nearest-longword-integer = MTH$JIHNNT (h-floating)	3
MTH$ININT	Convert F_floating to nearest word integer nearest-word-integer = MTH$ININT (f-floating)	3
MTH$JNINT	Convert F_floating to nearest longword integer nearest-long-int = MTH$JNINT (f-floating)	2,4
MTH$SNGL	Convert D_floating to F_floating (rounded) f-floating = MTH$SNGL (d-floating)	2
MTH$SNGLG	Convert G_floating to F_floating (rounded) f-floating = MTH$SNGLG (g-floating)	2,6
Miscellaneous Functions		
MTH$ABS	F_floating absolute value f-absolute-value = MTH$ABS (f-floating)	
MTH$DABS	D_floating absolute value d-absolute-value = MTH$DABS (d-floating)	
MTH$GABS	G_floating absolute value g-absolute value = MTH$GABS (g-floating)	
MTH$HABS	H_floating absolute value CALL MTH$HABS (h-absolute-value, h-floating)	5
MTH$IIABS	Word absolute value absolute-value-word = MTH$IIABS (word)	3
MTH$JIABS	Longword absolute value absolute-value-longword = MTH$JIABS (longword)	3

* The "Notes" column at the far right of Table 4-4 lists numbers that refer to the notes following the table.

Table 4-4: Miscellaneous Mathematics Functions (Cont.)

MTH$IIAND	Bitwise AND of two word parameters word = MTH$IIAND (word1, word2)	
MTH$JIAND	Bitwise AND of two longword parameters longword = MTH$JIAND (longword1, longword2)	
MTH$DIM	Positive difference of two F_floating parameters f-floating = MTH$DIM (f-floating1, f-floating2)	2,6
MTH$DDIM	Positive difference of two D_floating parameters d-floating = MTH$DDIM (d-floating1, d-floating2)	2,6
MTH$GDIM	Positive difference of two G_floating parameters g-floating = MTH$GDIM (g-floating1, g-floating2)	2,6
MTH$HDIM	Positive difference of two H_floating parameters CALL MTH$HDIM (h-floating, h-floating1, h-floating2)	2,5,6
MTH$IIDIM	Positive difference of two word parameters word = MTH$IIDIM (word1, word2)	3
MTH$JIDIM	Positive difference of two longword parameters longword = MTH$JIDIM (longword1, longword2)	3
MTH$IIEOR	Bitwise exclusive OR of two word parameters word = MTH$IIEOR (word1, word2)	
MTH$JIEOR	Bitwise exclusive OR of two longword parameters longword = MTH$JIEOR (longword1, longword2)	
MTH$IIOR	Bitwise inclusive OR of two word parameters word = MTH$IIOR (word1, word2)	
MTH$JIOR	Bitwise inclusive OR of two longword parameters longword = MTH$JIOR (longword1, longword2)	
MTH$AIMAX0	F_floating maximum of n word parameters f-floating-max = MTH$AIMAX0 (word, ...)	
MTH$AJMAX0	F_floating maximum of n longword parameters f-floating-max = MTH$AJMAX0 (longword, ...)	
MTH$IMAX0	Word maximum of n word parameters word-max = MTH$IMAX0 (word, ...)	
MTH$JMAX0	Longword maximum of n longword parameters longword-max = MTH$JMAX0 (longword, ...)	
MTH$AMAX1	Maximum of n F_floating parameters f-floating-max = MTH$AMAX1 (f-floating, ...)	3
MTH$DMAX1	Maximum of n D_floating parameters d-floating-max = MTH$DMAX1 (d-floating, ...)	
MTH$GMAX1	Maximum of n G_floating parameters g-floating-max = MTH$GMAX1 (g-floating, ...)	
MTH$HMAX1	Maximum of n H_floating parameters CALL MTH$HMAX1 (h-floating-max, h-floating, ...)	5
MTH$IMAX1	Word maximum of n F_floating parameters word-max = MTH$IMAX1 (f-floating, ...)	3

* The "Notes" column at the far right of Table 4-4 lists numbers that refer to the notes following the table.

Table 4–4: Miscellaneous Mathematics Functions (Cont.)

MTH$JMAX1	Longword maximum of n F_floating parameters longword-max = MTH$JMAX1 (f-floating, ...)	3
MTH$AIMIN0	F_floating minimum of n word parameters f-floating-min = MTH$AIMIN0 (word, ...)	
MTH$AJMIN0	F_floating minimum of n longword parameters f-floating-min = MTH$AJMIN0 (longword, ...)	
MTH$IMIN0	Minimum of n word parameters word-min = MTH$IMIN0 (word, ...)	
MTH$JMIN0	Minimum of n longword parameters longword-min = MTH$JMIN0 (longword, ...)	
MTH$AMIN1	Minimum of n F_floating parameters f-floating-min = MTH$AMIN1 (f-floating, ...)	3
MTH$DMIN1	Minimum of n D_floating parameters d-floating-min = MTH$DMIN1 (d-floating, ...)	
MTH$GMIN1	Minimum of n G_floating parameters g-floating-min = MTH$GMIN1 (g-floating, ...)	
MTH$HMIN1	Minimum of n H_floating parameters CALL MTH$HMIN1 (h-floating-min, h-floating, ...)	5
MTH$IMIN1	Word minimum of n F_floating parameters word-min = MTH$IMIN1 (f-floating, ...)	3
MTH$JMIN1	Longword minimum of n F_floating parameters longword-min = MTH$JMIN1 (f-floating, ...)	3
MTH$AMOD	Remainder of two F_floating parameters, arg1/arg2 f-floating = MTH$AMOD (f-floating1, f-floating2)	2
MTH$DMOD	Remainder of two D_floating parameters, arg1/arg2 d-floating = MTH$DMOD (d-floating1, d-floating2)	2
MTH$GMOD	Remainder of two G_floating parameters, arg1/arg2 g-floating = MTH$GMOD (g-floating1, g-floating2)	2
MTH$HMOD	Remainder of two H_floating parameters, arg1/arg2 CALL MTH$HMOD (h-floating, h-floating1, h-floating2)	2,5
MTH$IMOD	Remainder of two word parameters, arg1/arg2 word = MTH$IMOD (word1, word2)	1
MTH$JMOD	Remainder of two longword parameters, arg1/arg2 longword = MTH$JMOD (longword1, longword2)	1
MTH$INOT	Bitwise complement of a word parameter word = MTH$INOT (word)	
MTH$JNOT	Bitwise complement of a longword parameter longword = MTH$JNOT (longword)	
MTH$DPROD	D_floating product of two F_floating parameters d-floating = MTH$DPROD (f-floating1, f-floating2)	2
MTH$GPROD	G_floating product of two F_floating parameters g-floating = MTH$GPROD (f-floating1, f-floating2)	2

* The "Notes" column at the far right of Table 4–4 lists numbers that refer to the notes following the table.

Table 4-4: Miscellaneous Mathematics Functions (Cont.)

MTH$SGN	F_floating sign function longword = MTH$SGN (f-floating)	
MTH$SGN	D_floating sign function longword = MTH$SGN (d-floating)	
MTH$IISHFT	Bitwise shift of a word by shift-count-word places word = MTH$IISHFT (word, shift-count-word)	
MTH$JISHFT	Bitwise shift of longword1 by longword2 places longword = MTH$JISHFT (longword1, longword2)	
MTH$SIGN	F_floating transfer of sign of y to sign of x f-floating = MTH$SIGN (f-floating-y, f-floating-x)	
MTH$DSIGN	D_floating transfer of sign of y to sign of x d-floating = MTH$DSIGN (d-floating-x, d-floating-y)	
MTH$GSIGN	G_floating transfer of sign of y to sign of x g-floating = MTH$GSIGN (g-floating-x, g-floating-y)	
MTH$HSIGN	H_floating transfer of sign of y to sign of x CALL MTH$HSIGN (h-float, h-float-x, h-float-y)	5
MTH$IISIGN	Word transfer of sign of y to sign of x word = MTH$IISIGN (word-y, word-x)	
MTH$JISIGN	Longword transfer of sign of y to sign of x longword = MTH$JISIGN (longword-y, longword-x)	

Notes

1. Divide-by-zero exceptions can occur.
2. Floating overflow exceptions can occur.
3. Integer overflow exceptions can occur.
4. Returns contents of R0 if a negative parameter is input.
5. Returns value to the first parameter; value exceeds 64–bits.
6. Floating underflow exceptions can occur.
7. Returns an array of values to the output parameter. The number of elements converted is given by the optional parameter; the default number is 1.

Chapter 5
Process-Wide Resource Allocation Procedures

The process-wide resource allocation procedures provide coordinated allocation and deallocation of process-wide resources in a single VMS process. The process-wide resources include dynamic virtual memory, dynamic string memory, VMS local event flags and BASIC/FORTRAN logical unit numbers. These resources exist for the duration of the execution of the program image. These resource-allocation procedures are provided so other procedures can use the process-wide resources without conflicting with one another.

In general, you must use these procedures when you need to allocate process-wide resources within your program. This allows Run-Time Library procedures, DIGITAL-supplied procedures, and user procedures that you write to perform properly together within a process.

Table 5-1 lists all the process-wide resource allocation procedures. The sections that follow this table describe the procedures in detail.

Table 5-1: Process-Wide Resource Allocation Procedures

Section	Entry Point Name	Title
Dynamic Allocation of Virtual Memory		
5.1.5	LIB$GET_VM	Allocate Virtual Memory
5.1.6	LIB$FREE_VM	Deallocate Virtual Memory
5.1.7	LIB$STAT_VM	Fetch VM Statistics
5.1.8	LIB$SHOW_VM	Show VM Statistics

(continued on next page)

Section	Entry Point Name	Title
BASIC/FORTRAN Logical Unit Allocation		
5.2.1	LIB$GET_LUN	Allocate next arbitrary LUN
5.2.2	LIB$FREE_LUN	Deallocate a specific LUN
Event Flag Allocation		
5.3.1	LIB$GET_EF	Allocate a local event flag
5.3.2	LIB$FREE_EF	Free a local event flag
5.3.3	LIB$RESERVE_EF	Reserve a local event flag
String Resource Allocation		
5.4.1	LIB$SGET1_DD OTS$SGET1_DD STR$GET1_DX	Allocate One Dynamic String
5.4.2	LIB$SFREE1_DD OTS$SFREE1_DD STR$FREE1_DX	Deallocate One Dynamic String
5.4.3	LIB$SFREEN_DD OTS$SFREEN_DD	Deallocate n Dynamic Strings

NOTE

LIB$ procedures indicate errors by return status and pass input scalars by reference.

OTS$ procedures indicate errors by signaling and pass input scalars by immediate value.

STR$ procedures indicate serious errors by signaling and pass input scalars by reference. The destination descriptor must be dynamic. STR$ procedures should be used for new programs, when manipulating strings because they do not assume the string is dynamic.

5.1 Allocation of Virtual Memory

The virtual address space of an executing process consists of three regions:

1. A per-process program region that contains the image to be executed, including both instructions and data.
2. A per-process control region that contains system control information and the process stack.
3. A common system region that contains VAX/VMS; this region is not writable from the user access mode.

There are two ways to allocate storage in the program region (that is to assign positions in main memory for the purpose of holding information):

1. Statically at link time (static storage)
2. Dynamically at run time (heap storage)

There is one way to allocate storage in the control region: dynamically at run time in the stack frame (stack storage).

NOTE

Great care must be used with any of the preceding methods to avoid conflict between your procedures and library procedures, procedures written by other users, or system services. See the *VAX-11 Guide to Creating Modular Library Procedures* for an explanation of how to use each storage form in modular fashion.

5.1.1 Static Storage

Static storage, or statically allocated storage, is the simplest form of storage. It is allocated at link time by the Linker.

- **MACRO**

  The .BLKB, .BLKW, .BLKL, .BLKQ, and .BLKO directives allocate static storage in the current program section. The .BYTE, .WORD, .LONG, .QUAD, and .OCTA directives let you initialize static storage to arbitrary values at link time.

- **BASIC**

  Variables in COM and MAP statements are allocated in static storage in named, overlaid PSECTS.

- **FORTRAN**

  All declared variables and arrays are allocated in static storage in concatenated PSECTS. FORTRAN COMMON variables are allocated in named, overlaid PSECTS. The DATA declaration permits you to initialize static storage to arbitrary values at link time.

- **PASCAL**

  Variables declared at the module or program level are allocated in static storage in named, overlaid PSECTS.

Static storage not initialized otherwise is initialized to zero by the Linker. (See the *VAX-11 Guide to Creating Modular Procedures* for more discussion on using storage.)

Static storage has some disadvantages:

1. When you write your main program or user procedure, you must specify the maximum amount of storage you will ever need.
2. Your user procedure can have obscure bugs if it inadvertently uses values left behind from previous calls.
3. Your user procedure may not execute properly if it is called by an AST-level routine during your procedure's normal execution.
4. If overlaid PSECTS are used, one module can inadvertently affect another's storage.

5.1.2 Stack Storage

Stack storage avoids the preceding disadvantages of static storage.

- **MACRO**

  Stack space is allocated by decrementing the stack pointer (SP) by the number of bytes required. This can happen a number of times during the execution of a single procedure. Because each procedure has its own stack frame, different procedures do not conflict with one another. All of the stack space is reclaimed automatically when the procedure returns to its caller using a return instruction (RET). On a subsequent call, the procedure must allocate any space needed again. At that time, the contents of allocated stack space is indeterminate. Therefore, a procedure must initialize the stack space properly each time it is allocated.

- **BASIC and PASCAL**

  All procedure local variables and arrays are allocated on the stack.

- **FORTRAN**

  Stack space is not accessible to the FORTRAN programmer. (However, a compiled program may use it for temporary storage while evaluating complicated expressions.)

5.1.3 Heap Storage

A procedure can allocate heap storage dynamically at run time as it is needed. There is no constraint on when a procedure must deallocate the storage. However, if a user procedure is to retain heap storage after returning control to its caller, it must allocate some static storage as well. The procedure uses static storage to remember where the heap storage was allocated; this enables the procedure to use the heap storage later and eventually return it to image free storage.

- **BASIC and STR$**

  Strings are automatically allocated in heap storage.

- **PASCAL**

  The NEW function allocates heap storage.

- **Other languages**

  Heap storage can be used by explicit calls to Run-Time Library procedures.

5.1.4 Use of System Services

The following system services let you change the size of program or control regions and allocate or deallocate virtual memory space dynamically at run time:

- Expand Program/Control Region ($EXPREG) — expands the program or control region in page increments (512 bytes)
- Contract Program/Control Region ($CNTREG) — contracts the program or control region in page increments (512 bytes)
- Create Virtual Address Space ($CRETVA) — allocates specific virtual pages in the program or control region
- Delete Virtual Address Space ($DELTVA) — deallocates specific virtual pages in the program or control region

However, if you use any of these four system services, your procedures may conflict with other user-written procedures and/or DIGITAL-supplied procedures (including those in the Run-Time Library). For example, if your procedure assumes that the space beyond the last allocated data location is available and you use the $CRETVA system service to allocate the next page, you may discover that it was already allocated for some other purpose by the linker, the Run-Time Library, VAX-11 RMS (for additional buffer space), or by another procedure that also wanted space. Thus, your program would not operate correctly.

Rather than using any of the preceding system services to allocate virtual space, you should use the Allocate Virtual Memory (LIB$GET_VM) and Deallocate Virtual Memory (LIB$FREE_VM) procedures. These procedures dynamically allocate and deallocate virtual space to procedures in an image. The requested virtual space can be smaller than, equal to, or greater than a page. You can use LIB$GET_VM and LIB$FREE_VM with:

- All Run-Time Library procedures
- All modular reentrant procedures
- All software that calls the Run-Time Library
- All user-written procedures that use the library, including the language support procedures
- VAX-11 RMS, which also dynamically allocates buffer space in the program region

To summarize:

- If you are allocating storage but not deallocating it later, you can use either the $EXPREG System Service or LIB$GET_VM. The storage area is maintained throughout execution.

- If you are allocating and deallocating storage, you should use LIB$GET_VM for allocation and LIB$FREE_VM for deallocation. The storage area is returned to free storage after use.

5.1.5 Allocate Virtual Memory in Program Region

LIB$GET_VM allocates a specified number of virtually contiguous bytes somewhere in the program region and returns the virtual address of the first byte so allocated. The number of bytes allocated is rounded up so that the smallest possible number of whole quadwords (eight bytes) is allocated starting at a quadword boundary. LIB$GET_VM usually allocates the bytes at the end of the program region. However, if sufficient storage space exists elsewhere in the program region, LIB$GET_VM will allocate that space instead.

The space allocated in successive calls to LIB$GET_VM may be noncontiguous because another procedure can call LIB$GET_VM between your calls. In fact, if AST interrupts occur, space may be allocated to another procedure between execution of any pair of instructions in your program.

When virtual memory is needed, LIB$GET_VM allocates the area and removes reference to the area from a list of available free storage areas that LIB$GET_VM maintains. (This list is called the image free storage list.) If more virtual memory is required than is available in the program region, LIB$GET_VM calls the Expand Program Region system service $EXPREG to expand the program region in steps of 128 pages. LIB$GET_VM links this new area (by deallocating it) into the image free storage list. The requested memory is then allocated from this list. The image free storage list is therefore initialized on the first allocation call.

Format

ret-status = LIB$GET_VM (num-bytes, base-adr)

num-bytes
: Address of an unsigned longword integer specifying the number of virtually contiguous bytes to be allocated. Sufficient pages are allocated to satisfy the request. However, the program should not reference an address before the first byte address allocated (base-adr) or beyond the last byte allocated (base-adr+num-bytes - 1) since that space may be assigned to another procedure.

base-adr
: Address of a longword that is set to the first virtual address of the newly allocated contiguous block of bytes. (This is an output parameter.)

Return Status

SS$__NORMAL
Procedure successfully completed.

LIB$__INSVIRMEM
Insufficient virtual memory. The request required more dynamic memory than was available from the operating system. No partial assignment (allocation) is made in this case.

LIB$__BADBLOSIZ
Bad block size. The block size was zero.

Notes

The calling procedure must retain the address of the allocated area. This allows the procedure to access or deallocate it later.

LIB$GET__VM may be called at AST-level.

Examples

In FORTRAN, the address of dynamically allocated memory can be passed to another procedure as an array using the %VAL built-in function.

```
INTEGER*4 DYNARRAYADR          ! Pointer to array
NLONGWORDS = 500               ! Size of array in longwords
IF (LIB$GETVM (NLONGWORDS*4, DYNARRAYADR)) THEN
    CALL SUB(NLONGWORDS, %VAL(DYNARRAYADR))
.
.
.
END

SUBROUTINE SUB(SIZE, ARRAY)
INTEGER*4 SIZE, ARRAY(SIZE)
.
.
.
RETURN
END
```

When SUB is called, it is passed the address of an adjustable dimensioned array of SIZE longwords. The %VAL built-in function is needed in order to pass the address of the dynamically allocated area rather than the address of the longword variable DYN__ARRAY__ADR.

In MACRO, the following code allocates 100 bytes of dynamic memory:

```
BASADR:  .LONG   0                ; Receives address of allocated area
SIZE:    .LONG   100              ; Size to allocate in bytes
         PUSHAL  BASADR           ; 2nd parameter = adr. of longword
                                  ; To receive address of allocated area
         PUSHAL  SIZE             ; 1st parameter = adr of longword
                                  ; Containing the size to be allocated
         CALLS   #2, LIB$GET_VM   ; Allocate
         BLBC    R0, ERROR        ; Branch if error
```

5.1.6 Deallocate Virtual Memory from Program Region

LIB$FREE_VM deallocates an entire block of virtually contiguous bytes that had been allocated by LIB$GET_VM. The parameters passed are the same as for LIB$GET_VM.

Format

ret-status = LIB$FREE_VM (num-bytes, base-adr)

num-bytes
: Address of an unsigned longword integer specifying the number of virtually contiguous bytes to be deallocated. Rounding of byte counts is performed in the same manner as in LIB$GET_VM.

base-adr
: Address of a longword containing the address of the first byte to be deallocated. (This is an input parameter.)

Return Status

SS$_NORMAL
: Procedure successfully completed.

LIB$_BADBLOADR
: Base_adr contained a bad block address. This might be an address that was outside of the area allocated by LIB$GET_VM, or the contents of base_adr was not quadword aligned (as returned by LIB$GET_VM), or part of the space being deallocated was previously deallocated.

Notes

This procedure returns the indicated block(s) to the image free storage list so that it is available on a subsequent call to LIB$GET_VM.

No partial blocks or multiple blocks can be freed.

LIB$FREE_VM can be called at AST level. Blocks obtained at non-AST level can be freed at AST level and vice-versa.

Examples

The following FORTRAN code fragment deallocates the virtual memory allocated in the FORTRAN example in Section 5.1.5:

```
CALL LIB$FREE_VM (NLONGWORDS*4, DYN_ARRAY_ADR)
```

The following MACRO code fragment deallocates the virtual memory allocated in the MACRO example in the LIB$GET_VM procedure description:

```
PUSHAL    BASADR              ; par2 = adr of longword
                              ; containing adr to deallocate
PUSHAL    SIZE                ; par1 = adr of longword
                              ; containing size to deallocate
CALLS     #2, LIB$FREE_VM     ; deallocate virtual memory
BLBC      R0, ERROR
```


5.1.7 Fetch Virtual Memory Statistic

LIB$STAT_VM returns to its caller one of three statistics available from calls to LIB$GET_VM and LIB$FREE_VM. Unlike LIB$SHOW_VM, which produces ASCII values for output, LIB$STAT_VM returns the value in binary form to a location specified as a parameter.

Only one of the three statistics can be returned by one call to LIB$STAT_VM. A "code" of zero is invalid.

Format

ret-status = LIB$STAT_VM (code, value)

code
Address of a longword containing the code that specifies which statistic is to be returned. Allowed values are:

1 = Number of calls to LIB$GET_VM
2 = Number of calls to LIB$FREE_VM
3 = Number of bytes allocated by LIB$GET_VM but not yet freed by LIB$FREE_VM

It is invalid to omit "code" or to give a "code" of zero.

Value
Address of a longword to receive the result. All values are longword integers.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_INVARG
Invalid argument. Code was not in range 1:3 inclusive.

5.1.8 Show Virtual Memory Statistics

LIB$SHOW__VM is used to obtain the accumulated statistics from calls to LIB$GET__VM and LIB$FREE__VM. In the default mode, with neither "code" nor "action-routine" specified in the call, the routine will output to SYS$OUTPUT a line giving the following three items of information:

mmm calls to LIB$GET__VM, nnn calls to LIB$FREE__VM, ppp bytes still allocated

Optionally, only one of the three statistics can be output to SYS$OUTPUT and/or the line of information can be passed to a user-specified "action-routine", for processing different from the default.

Format

ret-status = LIB$SHOW__VM ([code] [,action-routine] [,user-arg])

code
Address of a longword containing the code which specifies the particular statistic desired. This is an optional parameter. If omitted or zero, all three statistics are returned on one line. If given, it must be one of the following values:

1 = Number of calls to LIB$GET__VM
2 = Number of calls to LIB$FREE__VM
3 = Number of bytes allocated by LIB$GET__VM but not yet deallocated by LIB$FREE__VM

action-routine
Address of a function procedure to call. This is an optional parameter. The function should return either a success or failure condition value, which will be returned as the return value of LIB$SHOW__VM. The arguments to this function follow:

ret-status = (action-routine) (out-str [,user-arg])

out-str
Address of the output string descriptor. The string is formatted exactly as it would be if output to SYS$OUTPUT. The leading character is blank. No embedded CR/LFs are included.

user-arg
A longword integer passed to LIB$SHOW__VM. This is an optional parameter. If given, it is passed directly on to the action routine without interpretation. That is, the contents of the arg list entry user-arg is copied to the arg list entry for action-routine.

Return Status

SS$__NORMAL
Procedure successfully completed.

LIB$__INVARG
Invalid arguments. This can be caused by an invalid value for "code".

NOTE

Other codes may be returned by LIB$PUT__OUTPUT or the user's action routine.

5.2 Logical Unit Allocation

Logical unit numbers are used in BASIC and FORTRAN to define a logical unit upon which I/O is done. For routines to be modular, they must have no knowledge of their run time environment. That is, they cannot rely on a knowledge of what logical unit numbers are being used in routines with which they coexist. This independence is maintained by allocating and deallocating logical units at run time.

The entire resource allocation logic and data is contained in a single module, named LIB$LUN. LIB$LUN contains two entry points, LIB$GET__LUN and LIB$FREE__LUN. The central data base consists of a single variable in which individual bit positions indicate whether or not a logical unit number is currently allocated. Logical unit numbers 100 to 119 are available to modular programs through these entry points.

LIB$GET__LUN

5.2.1 Allocate One Logical Unit Number

LIB$GET__LUN allocates one logical unit number from a process-wide pool. If a unit is available, its number is returned to the caller. Otherwise, an error is returned as the function value.

Format

ret-status = LIB$GET__LUN (log-unit-num)

log-unit-num
Address of a longword that is set to the number of the allocated logical unit or a -1, if none were available. (This is an output parameter.)

Return Status

SS$__NORMAL
Procedure successfully completed.

LIB$__INSLUN
Insufficient logical unit numbers. No logical unit numbers were available.

Example

In BASIC, a logical unit number could be allocated as follows:

```
100  CALL LIB$GET_LUN(A%)
200  IF A% >= 0% THEN
300        OPEN 'FOO' AS FILE #A%
```

5.2.2 Deallocate One Logical Unit Number

LIB$FREE_LUN is the complement of LIB$GET_LUN. When a logical unit number allocated by calling LIB$GET_LUN is no longer needed, it should be released for use by other routines. If successful, LIB$FREE_LUN releases the specified logical unit number back to the pool for available numbers.

Format

ret-status = LIB$FREE_LUN (log-unit-num)

log-unit-num
Address of a longword that contains the logical unit number to be deallocated. This is the value returned to the user by LIB$GET_LUN. (This is an input parameter.)

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_LUNALRFRE
Logical unit number already free.

LIB$_LUNRESSYS
Logical unit number reserved to system. This occurs if the specified logical unit number is outside the range of 100 to 119.

Example

In BASIC, a logical unit number could be deallocated as follows:

```
100  CLOSE #A%
200  CALL LIB$FREE_LUN(A%)
```

5.3 Event Flag Resource Allocation Procedures

This section describes the event flag resource allocation procedures provided by the Run-Time Library. These procedures allocate and deallocate local event flag numbers. Using these procedures allows use of local event flags by multiple procedures without conflicts.

5.3.1 Allocate One Local Event Flag

LIB$GET_EF allocates one local event flag from a process-wide pool. If a flag is available for use, its number is returned to the caller. If no flags are available, an error is returned as the function value.

The 64 local event flags are:

0 Never used by these procedures; always available

1–23 Initially marked as in use (for compatibility with the PDP–11 which had no allocation routines)

24–31 Reserved to VMS

32–63 Initially free

Format

ret-status = LIB$GET_EF (event-flag-num)

event-flag-num
Address of a longword that is set to the number of the allocated local event flag or -1 if none were available. (This is an output parameter.)

Return Status

SS$_NORMAL

Procedure successfully completed.

LIB$_INSEF
Insufficient event flags. There were no more event flags available for allocation.

5.3.2 Deallocate One Local Event Flag

LIB$FREE_EF is the complement of LIB$GET_EF. When a local event flag, allocated by calling LIB$GET_EF, is no longer needed, LIB$FREE_EF should be called to free the event flag for use by other routines.

Format

ret-status = LIB$FREE_EF (event-flag-num)

event-flag-num
Address of a longword containing the event flag number to be deallocated. This is the value returned to the user by LIB$GET_EF. (This is an input parameter).

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_EF_ALRFRE
Event flag already free.

LIB$_EF_RESSYS
Event flag reserved to system. This occurs if the event flag number is outside the ranges of 1–23 and 32–63.

5.3.3 Reserve a Local Event Flag

LIB$RESERVE_EF allocates a particular local event flag number. This differs from LIB$GET_EF, which allocates an arbitrary event flag. Use LIB$FREE_EF to deallocate an event flag reserved with LIB$RESERVE_EF.

Format

ret-status = LIB$RESERVE_EF (event-flag-num)

event-flag-num

Address of a longword containing the event flag number to be allocated. (This is an input parameter.)

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_EF_ALRRES
Event flag already reserved.

LIB$_EF_RESSYS
Event flag reserved to system. This occurs if the event flag number is outside the ranges of 1–23 and 32–63.

5.4 String Resource Allocation Procedures

This section describes string resource allocation procedures provided by the Run-Time Library. These procedures accept as parameters strings of any standard class.

NOTE

Chapter 3 contains procedures for copying and manipulating strings; Chapter 7 contains procedures for syntactically analyzing strings.

Dynamic strings are the most convenient type to write, since you need not specify length (or maximum length) or position for them. The library procedures that allocate dynamic strings also allocate virtual memory for them. They are thus resource allocation procedures and must be used whenever a dynamic string descriptor is modified.

The caller of the procedures described in this section must allocate space for the string descriptor itself before making the call. Such allocation can be done statically at compile time, or dynamically in local stack storage or heap storage.

Programs that allocate dynamic string descriptors in the stack must free the associated dynamic string areas by calling the LIB$SFREE1_DD, OTS$SFREE1_DD, or STR$FREE1_DX procedures before executing a RET instruction. Otherwise, the dynamic string area becomes unavailable when the RET instruction removes the descriptors which point to the string area. Similarly, before executing a RET instruction, a program that allocates dynamic string descriptors in heap storage must free the associated dynamic string areas by calling the LIB$SREE1_DD, OTS$SFREE1_DD, or STR$FREE1_DX procedure.

When a procedure might be unwound (see Chapter 6), it should establish a handler that will free the associated dynamic string areas when the SS$_UNWIND condition is signaled. The handler can free these areas by calling the LIB$SFREE1_DD, OTS$SFREE1_DD, or STR$FREE1_DX procedure.

The string resource allocation procedures can be called from any access mode at AST or non-AST level.

Eight string resource allocation entry points are provided, each with slightly different input parameters, calling techniques, or methods of indicating errors. In all cases, destination strings are passed by descriptor. The following parts of the entry point name indicate the differences among the entry points:

fac$[S]GET_abxyn

fac$
: LIB$, OTS$, or STR$. Table 5–2 compares the parameter passing conventions for these facilities.

ab

DX means any type of source descriptor.

R_ means a source string that is passed by reference with a pair of parameters. The first parameter is the length of the string; the second parameter is the address of the string.

xy

DX means any type of destination descriptor. The class field (that is, DSC$B_CLASS) determines what actions the procedure will take for each type of string input (either unspecified, fixed length, or dynamic).

DD means that the destination descriptor is assumed to be dynamic and is not checked.

n

A number or Rn is appended to distinguish the JSB entry point from CALL entry points. JSB entry points modify registers R0:Rn.

Table 5-2: LIB$, OTS$, & STR$ Parameter Passing Conventions

	LIB$	OTS$	STR$
CALL input scalars	reference	immediate value	reference
JSB input scalars	immediate value	immediate value	immediate value
Severe errors	return status	signal	signal
Truncation errors	return status (success or severe)	value	return status (warning)
JSB output R0–R5	status(R0)	MOVC5 registers	status(R0)

5.4.1 Allocate One Dynamic String

LIB$SGET1_DD
OTS$SGET1_DD
STR$GET1_DX

LIB$SGET1_DD allocates a specified number of bytes of dynamic virtual memory to a specified string descriptor. This procedure is identical to LIB$SCOPY_DXDX except that no source string is copied. It is provided so you can write anything you want in the allocated area.

Format

ret-status = LIB$SGET1_DD (len, str)

JSB entry point: LIB$SGET1_DD_R6

len
Address of a word containing the unsigned number of bytes to be allocated; the amount of storage allocated may automatically be rounded up. If the number of bytes is zero, a small amount of space is allocated.

str
Address of a dynamic string descriptor to which the area is to be allocated. The class field is not checked but it is set to dynamic (DSC$B_CLASS = 2). The length field (DSC$W_LENGTH) is set to len, and the address field (DSC$A_POINTER) is set to the string area allocated (first byte beyond the header).

Implicit Inputs (JSB entry)

R0<15:0>
Unsigned number of bytes to be allocated.

R1
Address of dynamic string descriptor to which the area is to be allocated.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_FATERRLIB
LIB$_INSVIRMEM
LIB$_STRIS_INT

Note

In the event that the specified string descriptor already has dynamic memory allocated to it, but the amount allocated is less than len, that space is deallocated before LIB$SGET1_DD allocates new space.

Examples

See LIB$SFREE1_DD.

OTS$GET1_DD allocates a specified number of bytes of dynamic virtual memory to a specified string descriptor. This procedure is identical to OTS$SCOPY_DXDX except that no source string is copied. It is provided so you can write anything you want in the allocated area.

Format

CALL OTS$SGET1_DD (len, str)

JSB entry point: OTS$SGET1_DD_R6

len
Unsigned number of bytes to be allocated; the amount of storage allocated may automatically be rounded up. If the number of bytes is zero, a small number of bytes is allocated. Only the low-order word of the longword parameter is used by the called procedure (passed by immediate value).

str
Address of a dynamic string descriptor to which the area is to be allocated. The class field is not checked but it is set to dynamic (DSC$B_CLASS = 2). The length field (DSC$W_LENGTH) is set to len and the address field (DSC$A_POINTER) is set to the string area allocated (first byte beyond the header).

Implicit Inputs (JSB entry)

R0<15:0>
Unsigned number of bytes for which areas are to be allocated.

R1
Address of dynamic string descriptor to which the area is to be allocated.

Messages

OTS$_FATINTERR
OTS$_INSVIRMEM
OTS$_STRIS_INT

Note

In the event that the specified string descriptor already has dynamic memory allocated to it, but the amount allocated is less than len, that space is deallocated before OTS$SGET1_DD allocates new space.

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

Format

ret-status = STR$GET1_DX (len,str)
JSB entry point: STR$GET1_DX_R4

len
Address of a word containing the unsigned number of bytes to be allocated.

str
Address of a dynamic string descriptor to which the area is to be allocated. The class field (DSC$B_CLASS) is checked.

Implicit Inputs (JSB entry)

R0 <15:0>
Unsigned number of bytes to be allocated.

R1
Address of dynamic string descriptor to which the area is to be allocated.

Return Status

SS$_NORMAL
Procedure successfully completed.

Messages

STR$__FATINTERR
STR$__ILLSTRCLA
STR$__INSVIRMEM
STR$__STRIS__INT

Note

If the specified string descriptor already has dynamic memory allocated to it, but the amount allocated is less than len, that space is deallocated before STR$GET1__DX allocates new space.

xxx$SFREE1__DX

5.4.2 Deallocate One Dynamic String

LIB$SFREE1__DD
OTS$SFREE1__DD
STR$FREE1__DX

LIB$SFREE1__DD returns one dynamic string area to free storage. Before a procedure deallocates a dynamic descriptor, it must use LIB$SFREE1__DD or LIB$SFREEn__DD to deallocate the string storage space specified by the dynamic descriptor. Otherwise, string storage is lost.

This procedure deallocates the described string space and flags the descriptor as describing no string at all (that is, DSC$A__POINTER = 0 and DSC$W__LENGTH = 0).

Format

ret-status = LIB$SFREE1__DD (dsc-adr)

JSB entry point: LIB$SFREE1__DD6

dsc-adr
Address of a dynamic descriptor which specifies the area to be deallocated. The descriptor is assumed to be dynamic and its class field is not checked.

Implicit Inputs (JSB entry)

R0
Address of the descriptor specifying the area to be deallocated.

Return Status

SS$__NORMAL
Procedure successfully completed.

LIB$__FATERRLIB
LIB$__STRIS__INT

OTS$SFREE1_DD returns one dynamic string area to free storage.

Format

CALL OTS$SFREE1_DD (dsc-adr)

JSB entry point: OTS$SFREE1_DD6

dsc-adr
Address of a dynamic descriptor. The descriptor is assumed to be dynamic and its class field is not checked.

Implicit Inputs (JSB entry)

R0
Address of the descriptor whose string area is to be deallocated.

Messages

OTS$_FATINTERR
OTS$_STRIS_INT

Note

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

STR$FREE1_DX deallocates one dynamic string.

Format

CALL STR$FREE1_DX (dsc-adr)

JSB entry point STR$FREE1_DX_R4

dsc-adr
Address of a dynamic string descriptor. The class field (DSC$B_CLASS) is checked.

Implicit Inputs
None

Return Status

SS$_NORMAL
Procedure successfully completed.

Messages

STR$_FATINTERR
STR$_ILLSTRCLA
STR$_STRIS_INT

Note

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

Example

The following MACRO procedure allocates a dynamic string descriptor on the stack, allocates 100 bytes of dynamic string memory, and frees it just before return.

```
$DSCDEF                                 ; Define DSC$... symbols
.ENTRY  PROC, ^M<R2,R3,R4,R5>           ; Save registers used by JSBs
ASHQ    #16, DSC$K_CLASS_D, -(SP)       ; Allocate descriptor and set
                                        ; class field to dynamic
MOVB    #DSC$K_DTYPE_T, DSC$B_DTYPE(SP) ; Set type to text
MOVZBW  #100, R0                        ; R0 = no. of bytes
MOVL    SP, R1                          ; R1 = adr of descriptor
JSB     STR$GET1_DX_R4                  ; Allocate storage
.
.
.
MOVAQ   -8(FP), R0                      ; R0 = adr of descriptor
JSB     STR$FREE1_DX_R4                 ; Deallocate storage
RET                                     ; Return to caller
```


5.4.3 Deallocate n Dynamic Strings

LIB$SFREEN__DD
OTS$SFREEN__DD

LIB$SFREEN__DD returns one or more dynamic strings to free storage.

Before a procedure that allocates space returns to its caller, it must use LIB$SFREE1__DD or LIB$SFREEN__DD to deallocate the string storage space specified by any descriptors located in the stack.

This procedure deallocates the described string space and flags each descriptor as describing no string at all (DSC$A__POINTER = 0 and DSC$W__LENGTH = 0).

Format

ret-status = LIB$SFREEN__DD (dsc-num, first-dsc-adr)

JSB entry point: LIB$SFREEN__DD6

dsc-num
: Address of a longword containing the number of adjacent descriptors to be flagged as having no allocated area (DSC$A__POINTER = 0 and DSC$W__LENGTH = 0) and to have their allocated area returned to free storage.

first-dsc-adr
: Address of the first descriptor of an array of descriptors. The descriptors are assumed to be dynamic, and their class fields are not checked.

Implicit Inputs (JSB entry)

R0
Unsigned number of adjacent descriptors for which areas are to be deallocated.

R1
Address of the first descriptor for which an area is to be deallocated.

Return Status

SS$_NORMAL
Procedure successfully completed.

LIB$_FATERRLIB
LIB$_STRIS_INT

OTS$SFREEN_DD takes as input a vector of one or more dynamic string areas and returns them to free storage.

Format

CALL OTS$SFREEN_DD (dsc-num, first-dsc-adr)

JSB entry point: OTS$SFREEN_DD6

dsc-num
Number of adjacent descriptors to be flagged as having no allocated area (DSC$A_POINTER = 0 and DSC$W_LENGTH = 0) and to have their allocated areas returned to free storage (passed by immediate value)

first-dsc-adr
Address of the first descriptor of an array of descriptors. The descriptors are assumed to be dynamic, and their class fields are not checked.

Implicit Inputs (JSB entry)

R0
Unsigned number of adjacent descriptors for which areas are to be deallocated.

R1
Address of the first descriptor for which the area is to be deallocated.

Messages

OTS$_FATINTERR
OTS$_STRIS_INT

Notes

This procedure deallocates the described string space and flags each descriptor as describing no string at all (DSC$A_POINTER = 0 and DSC$W_LENGTH = 0).

Chapter 6
Signaling and Condition Handling Procedures

This chapter describes the signaling and condition handling procedures as well as the related system services that together comprise the VAX-11 Condition Handling Facility. The facility is language-independent and provides a single, unified method for:

- Printing error messages
- Indicating the occurrence of error conditions
- Changing the error behavior from the system default, such as altering or suppressing the error message, correcting a result, or changing the flow of control
- Enabling/disabling detection of certain hardware errors

Appendix C contains the functional specification for the VAX-11 Condition Handling Facility. This chapter introduces condition handling in a tutorial manner for the programmer using a language that does not have condition handling as part of the language, and for programmers using languages that incorporate error handling, such as BASIC. However, some of the procedures described herein cannot be called explicitly in languages that have error handlers to avoid conflict with the language-support routine.

6.1 Summary of VAX-11 Condition Handling Facility

The specific functions provided by the VAX-11 Condition Handling Facility are:

- *Establish a condition handler procedure.* A condition handler is associated with the currently executing procedure by placing an address pointing to the handler in the executing procedure's stack frame. The condition handler is called on errors that are not returned by means of the completion status normally used. (See Section 6.3.)
- *Remove an established condition handler procedure.* If a condition handler has been established, it can be removed by setting the address pointing to the condition handler in the currently executing procedure's stack frame to zero. (See Section 6.3.)
- *Enable or disable the detection of certain arithmetic hardware exceptions.* Detection of floating-point underflow, integer overflow, and decimal overflow can be enabled or disabled under software control. (See Section 6.2 and Section 6.5.)
- *Signal a condition.* Signaling a condition initiates a search for an established condition handler from top to bottom of the procedure stack. (See Section 6.6.)
- *Print an error message.* A default catch-all handler is established by the system before it calls the main program. This handler formats and outputs signaled conditions using the Put Message $PUTMSG system service, and the system message file. Signaling is the standard way to output any error message on VAX-11. (See Section 6.4 and 6.9.)
- *Unwind the stack.* A condition handler can indicate that when it returns, one or more pre-signal frames are to be removed (unwound) from the stack. During the unwinding operation, the stack is scanned. If a condition handler is associated with a frame, that handler is called before the frame is removed. Unwind allows a procedure to perform application-specific cleanup, such as recovery from noncontinuable errors. (See Section 6.8.)
- *Log error messages to an arbitrary file.* The Put Message $PUTMSG system service also permits any user-written handler to obtain a copy of the formatted message for any purpose, such as inclusion in a listing file. Such message logging can be completely supplemental to the default messages the user receives. (See Section 6.9.)
- *Print a stack traceback on errors.* The default operations of the LINK and RUN commands provide a system-supplied handler to print a symbolic stack traceback. The traceback shows the state of the procedure stack up to the point of the occurrence of the condition. (See Section 6.4.1.)

Table 6–1 lists all the signaling and condition handling procedures. The sections that follow this table describe how to write a condition handler, then describe the various procedures and how to use them in detail.

Table 6–1: Signaling and Condition Handling Procedures

Section	Entry Point Name	Title
Establishing a Condition Handler		
6.3.1	LIB$ESTABLISH	Establish Condition Handler
6.3.2	LIB$REVERT	Delete Condition Handler
Enable/Disable Hardware Conditions		
6.5.1	LIB$DEC_OVER	Enable/Disable Decimal Overflow
6.5.2	LIB$FLT_UNDER	Enable/Disable Floating Underflow
6.5.3	LIB$INT_OVER	Enable/Disable Integer Overflow
Signal Generators		
6.7.2	LIB$SIGNAL	Signals Exception Condition
6.7.3	LIB$STOP	Stop Execution via Signaling
Condition Handler Support		
6.10.2	LIB$MATCH_COND	Match Condition Value
6.10.3	LIB$FIXUP_FLT	Fixup Floating Reserved Operand
6.10.4	LIB$SIG_TO_RET	Convert Any Signal to Return Status

6.2 Exception Conditions

An exception condition is a hardware- or software-detected event that changes the normal flow of instruction execution. It usually indicates a failure, although it is not restricted to error situations.

There are two standard methods for a DIGITAL- or user-written procedure to indicate that an exception condition has occurred:

1. Return a completion code to the calling program as a function value (bit 0 clear in R0) that indicates which exception condition occurred
2. Signal the exception condition

In the first method, described in Chapter 2, the calling program explicitly associates the error recovery action with the call to the procedure that detected the error. Of the two methods, the first allows better programming structure because each call site explicitly indicates the flow of control when an error occurs.

In the second method, described in this chapter, the calling program associates the same error recovery action with all calls to all procedures.

Method 2 is the only way to handle hardware exceptions, and is the normal way to output error messages of any kind. This method makes it possible for a calling program to establish a condition handler to perform any of the following:

- Change the message to a more suitable one for the application
- Suppress the message
- Correct the result
- Continue execution at the same or at a different point

A technique called *signaling* propagates the indication of an error condition along the stack starting with the procedure called most recently. Therefore, each procedure has the opportunity to perform any of the above condition handling actions in the reverse order from that in which the procedures were called. In other words, condition handling "nests," so that each caller can potentially override the action of the procedure it called.

The following classes of exception conditions can occur while a program is executing:

1. *Hardware Processor detected*
   - Arithmetic trap in a user-written program (for example, floating overflow)
   - Arithmetic trap in a math library routine (for example, floating underflow)
   - Program fault (for example, invalid address)
   - Processor error (for example, memory parity error)
2. *Run-Time Library (software) detected*
   - Error in a user parameter to a math routine (for example, a negative SQRT)
   - Error in an I/O call (FORTRAN) where the user has supplied an ERR= (for example, direct access not specified, record too small for I/O list)
   - Error in an I/O call where the user has not supplied an ERR= (for example, direct access not specified, record too small for I/O list)
   - Error in a compiled code support routine due to an error in a user operation
3. *Other hardware and software detected*
   - I/O transfer error (for example, parity error)

- VAX-11 RMS detected errors
- Executive detected errors
- Application-specific errors

6.2.1 Condition Value

A condition value is a longword that includes fields to describe the software component detecting the error, the cause of the error, and the error severity status. It is used in both methods of indicating exception conditions.

A condition value consists of a 32-bit quantity that uniquely identifies an exception condition. Each condition value has a unique system-wide symbol and an associated message.

The 32-bit condition value is divided into several fields. The FAC_NO field (bits 27 through 17) indicates the system facility in which the condition occurred. The MSG_NO field (bits 15 through 3) indicates the particular condition that occurred. The SEVERITY field (bits 2 through 0) indicates whether the condition is a success (bit 0 = 1) or a failure (bit 0 = 0) as well as the severity. See Section C.4 of Appendix C for a more complete description of the fields in a condition value.

Software components return condition values when they complete execution. When a severity code of WARNING, ERROR, or SEVERE has been generated, the status code returned describes the nature of the problem. This value can be tested to change the flow of control of a procedure and/or to generate a message. User procedures can also generate condition values to be examined by other procedures and by the command language interpreter. User-generated condition values should set bits 27 and 15 so that these values will not conflict with values generated by DIGITAL.

For more detailed information about condition values, see Section C.4 of Appendix C.

6.2.2 Hardware Processor Detected Exception Conditions

When a hardware exception occurs, the VAX/VMS executive examines any primary and/or secondary exception vectors and calls these vectored condition handlers if any are present. (See the Set Exception Vector ($SETEXV) system service in the *VAX/VMS System Services Reference Manual.*)

NOTE

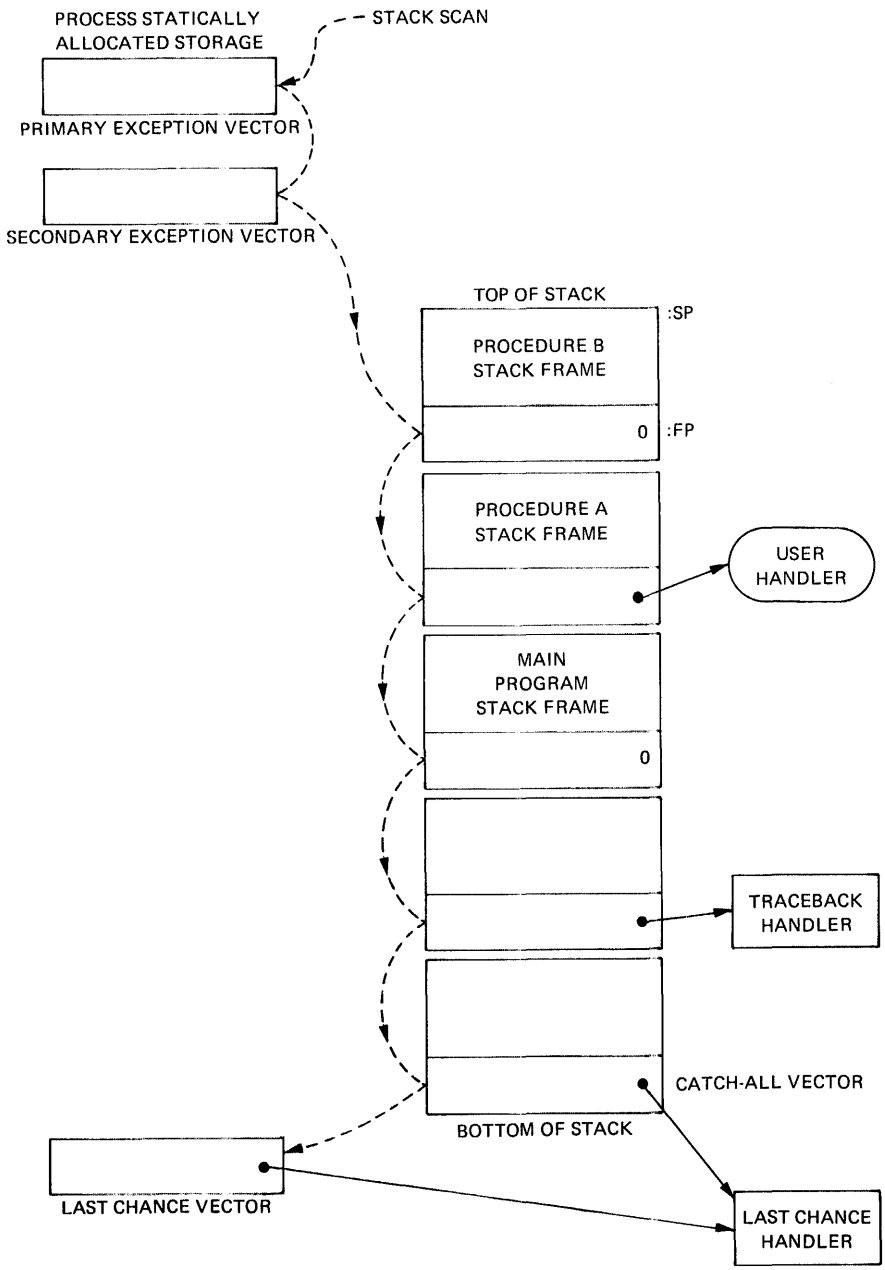
> The primary vector is used by the debugger, the secondary vector is reserved to customers for performance monitoring and/or testing, and the last-chance vector is used by the system and the debugger.

If a vectored condition handler is called and resignals, or is not present, the stack is scanned frame by frame from the currently executing procedure to the

beginning of the stack. If the entire stack is scanned without finding a pointer to a condition handler, a last-chance vectored condition handler is called. The process of searching for handlers and calling them is referred to as signaling a condition. (See Section 6.6.)

Figure 6–1 illustrates a stack scan for condition handlers in which the main program calls procedure A, which calls procedure B. A stack scan will be performed when a hardware exception occurs or a call is made to LIB$SIGNAL or LIB$STOP.

Figure 6–1: Sample Stack Scan for Condition Handlers



6.2.3 Language–Support Procedures Exception Conditions

Some languages have specifications for actions to be taken if an exception condition occurs. An example of this is the optional "ERR=" construct in a FORTRAN OPEN statement, because it specifies an address to which control is to be transferred. In these cases, exceptions are indicated by returning a completion status rather than by using the signaling mechanism. The calling program contains a branch instruction, which tests the low bit of the completion status returned in R0.

Some calls to the Run–Time Library do not or cannot specify an action to be taken. In this case, the Run–Time Library will signal the proper exception condition using the VAX–11 signaling mechanism. The same search for a condition handler is performed as with a hardware exception (see Section C.9 of Appendix C and Section 6.2.2).

The use of exception vectors, which are process-wide data locations, violates Run–Time Library modularity principles. Therefore, the Run–Time Library itself does not establish handlers using the primary, secondary, or last-chance exception vectors.

6.2.4 Mathematics Procedure Exception Conditions

All mathematics procedures return a function value in register R0 or registers R0/R1. This means that mathematics procedures cannot return a completion status, and therefore must signal all errors. In addition, all mathematics routines signal a mathematics procedure-specific error rather than a general hardware error.

6.2.4.1 Integer Overflow and Floating Overflow — All mathematics procedures are programmed with a software check to avoid integer overflow and floating-point overflow conditions. If an integer or floating-point overflow occurs in a CALL or JSB procedure, it signals a mathematics-specific error such as MTH$__FLOOVEMAT (FLOATING OVERFLOW IN MATH LIBRARY) by calling LIB$SIGNAL explicitly. (See Appendix B for a list of the mathematics procedures errors.)

The software check is needed because JSB routines cannot set up condition handlers. The check permits the JSB mathematics procedures to add an extra stack frame so that the error message and stack traceback will appear as if a CALL instruction had been performed. Because of the software check, JSB procedures will not cause a hardware exception condition even when the calling program has enabled integer overflow. Floating-point overflow detection is always enabled and cannot be disabled.

NOTE

The BASIC and FORTRAN compilers use the JSB entries instead of the equivalent CALL entries for those procedures that have JSB entry points.

6.2.4.2 Floating Underflow — All mathematics procedures are programmed to avoid floating underflow conditions. Software checks are made to determine if a floating-point underflow condition would occur. If so, the software makes an additional check. If the immediate calling program (CALL or JSB) has enabled floating-point underflow traps, a mathematics-specific error condition is signaled. Otherwise, the result is corrected to zero and execution continues with no error condition. The user can enable or disable floating-point underflow detection at run time by calling the LIB$FLT_UNDER procedure (see Section 6.5.2).

6.2.5 VAX-11 RMS and Executive Detected Errors

An exception condition detected when a language support procedure calls VAX-11 RMS or some other VAX/VMS service is always returned as a condition value in a function return. The language support procedure then performs one of the following (in order of preference):

1. Recovers from the error
2. Returns the error to the calling program if this has been explicitly indicated (for example, in an ERR= statement)
3. Signals the exception condition, with additional arguments, such as the VAX-11 RMS error status; the VAX/VMS error status; the BASIC/FORTRAN logical unit number, the resultant file name string; and the user PC, following the call to the library

6.3 Establishing a Condition Handler

Each procedure uses the first longword in its stack frame (longword 0) to specify a condition handler. When a procedure is called, the CALL instruction automatically initializes longword 0 to zero to indicate the absence of a condition handler.

Your procedure can establish a condition handler for itself by moving the address pointing to the condition handler to longword 0 of your procedure's stack frame. LIB$ESTABLISH is used to accomplish this for higher level languages. LIB$REVERT deletes the handler established by LIB$ESTABLISH.

LIB$ESTABLISH

6.3.1 Establish a Condition Handler

LIB$ESTABLISH moves the address of a condition handling routine (which can be a user-written or a library procedure) to longword 0 of the stack frame of the caller of LIB$ESTABLISH. This routine then becomes the caller's condition handler. At the same time, the previous contents of longword 0 are returned as old-handler. This can either be the address of the caller's previous condition handler or zero if none existed.

The new condition handler remains in effect for your procedure until a call to LIB$REVERT or control returns to the caller of the caller of LIB$ESTABLISH. Once this happens, LIB$ESTABLISH must be called again if the same (or a new) condition handler is to be associated with the caller of LIB$ESTABLISH.

Format

old-handler = LIB$ESTABLISH (new-handler)

new-handler
Address of routine to be set up as the condition handler.

old-handler
Previous contents of SF$A_HANDLER (longword 0) of the caller's stack frame.

Notes

This procedure modifies caller's stack frame.

This procedure is provided primarily for use with FORTRAN.

Use of this procedure with other VAX languages, such as BASIC, may modify the behavior of your procedure in certain situations. The language-support library depends on language-specific handlers to be established already. The handler address is also used to identify the stack frames of procedures written in these languages.

For use of LIB$ESTABLISH with PASCAL, see the *VAX-11 PASCAL User's Guide.*

In MACRO, the programmer merely uses the following instruction instead of calling LIB$ESTABLISH:

```
MOVAB HANDLER, (FP)             ; set handler address
                                ; in current stack frame
```

Example

In FORTRAN, the following code fragment establishes the condition handler procedure, HANDLER, for the current procedure activation of the program unit (whether it is a main program, subroutine, or function):

```
EXTERNAL HANDLER
CALL LIB$ESTABLISH (HANDLER)
```

NOTE

In BASIC, the user should use the ON ERROR GO TO statement.

6.3.2 Delete Handler Associated with Procedure Activation

LIB$REVERT deletes the condition handler established by LIB$ESTABLISH by clearing the address pointing to the condition handler from the activated procedure's stack frame. This address is returned as old-handler. LIB$REVERT is only used if your procedure is to establish and then cancel a condition handler for a portion of its execution.

Format

old-handler = LIB$REVERT ( )

old-handler
Previous contents of SF$A_HANDLER (longword 0) of the caller's stack frame.

Notes

This procedure modifies caller's stack frame.

This procedure is provided primarily for use with FORTRAN.

Use of this procedure with other VAX-11 languages, such as BASIC, may modify the behavior of your procedure in certain situations. In BASIC, the user should use the ON ERROR GO TO statement.

For use of LIB$REVERT with PASCAL, see the *VAX-11 PASCAL User's Guide*.

In MACRO, the programmer merely uses the following instruction rather than calling LIB$REVERT:

```
CLRL      (FP)          ; set handler address to 0
                        ; in current stack frame
```

Example

In FORTRAN, LIB$ESTABLISH and LIB$REVERT can be used to bracket a small section of code where a particular recoverable error could occur. This is a good practice, since unanticipated errors in other parts of the same program unit will not inadvertently invoke the handler procedure.

```
EXTERNAL HANDLER
.
.
.
CALL LIB$ESTABLISH (HANDLER)
Y = X/B
CALL LIB$REVERT
.
.
.
END
```

HANDLER will get control only if a hardware exception occurs in the Y = X/B statement.

To write a better structured program, you should save the old value and restore it using LIB$ESTABLISH. Then, the sequence of code can be embedded in a larger sequence that also has established a handler for its duration. Thus, the sequence should be:

```
SAV_HANDLER = LIB$ESTABLISH (HANDLER) ! Establish handler
.
.
.
CALL LIB$ESTABLISH (SAV_HANDLER) ! Restore to previous handler
```

6.4 Default Handlers

VAX/VMS establishes default condition handlers any time a new image is started. The following default handlers are established and are shown in the order they are encountered while processing a signal. These three handlers are the only handlers that should output error messages.

6.4.1 Traceback Handler

The traceback handler is established on the stack after the catch-all handler. This enables the traceback handler to get control first. This handler performs three functions in the order shown:

1. It outputs an error message using the Put Message (SYS$PUTMSG) system service. SYS$PUTMSG formats the message using the Formatted ASCII Output (SYS$FAO) system service. The message is output to device SYS$ERROR (and SYS$OUTPUT if it differs from SYS$ERROR).
2. It outputs a symbolic traceback which shows the module and procedure where each nested call was made at the time of the exception.
3. It decides whether to continue execution of the image or to force an exit based on the severity field of the condition value:

Value of Bits 2:0	Error Type	Action
1	SUCCESS	continue
3	INFO	continue
0	WARNING	continue
2	ERROR	continue
4	SEVERE	exit

The traceback handler can be eliminated at link-time by using the /NOTRACEBACK qualifier in the link command.

6.4.2 Catch–All Handler

The catch-all handler is established in the first stack frame by the operating system and hence is called last. This handler performs the same functions as the traceback handler except that no stack traceback is accomplished. (Functions 1 and 3 in Section 6.4.1)

6.4.3 Last-Chance Handler

The last-chance handler is established by a system exception vector. It is called only if the stack is invalid or all the handlers on the stack have resignaled. Note that if the debugger is present, the system last-chance handler will be replaced with the debugger's own last-chance handler.

6.4.4 Using Default Handlers to Output Messages

The system-supplied default handlers are the only handlers that should output error messages. This means that:

- The details of formatting and the choice of natural language is centralized.
- The system utility programs that run as commands (such as COPY or PRINT) may be called as procedures. By linking with /NO TRACEBACK, such programs can output error messages without traceback.
- Any set of procedures may be called by any application program to alter or hide the messages from the application user. For example, an application may choose to output a stock error message of the form:

```
Internal System Error, Please Start Over
```

or

```
Internal System Error, Please Call System Operator
```

Any applications procedures called in this manner should also signal any changed messages, rather than outputting them directly, so they, in turn, can be called by other applications that might want to change the message again.

6.5 Overflow/Underflow Detection Enabling Procedures

The following procedures permit a program to enable or disable the reporting of hardware detection of decimal overflow, floating-point underflow, and integer overflow. These are the only hardware detected exception conditions that can be disabled. Integer divide-by-zero, floating-point overflow, and floating-point/decimal divide-by-zero cannot be disabled. When a hardware condition is enabled, the occurrence of the condition causes a hardware exception to occur; the operating system signals the exception condition as a severe error. When a hardware condition is disabled, the occurrence of the condition is ignored and the processor executes the next instruction in the sequence.

The setting of overflow and underflow enables is independent for each procedure activation, since the call instruction saves the state of the calling program's hardware enables in the stack and then initializes the enables for the called procedure. A return instruction restores the calling program's enables.

The following procedures are intended primarily for higher-level languages, since the MACRO programmer can achieve the same effect by the single Bit Set PSW (BISPSW) or Bit Clear PSW (BICPSW) instructions.

These procedures allow you to enable and disable detection of decimal overflow, floating-point underflow, and integer overflow for a portion of your procedure's execution. Note that the BASIC and FORTRAN compilers provide a compile-time qualifier that permits you to enable or disable integer overflow for your entire procedure.

LIB$DEC_OVER

6.5.1 Enable/Disable Decimal Overflow Detection

LIB$DEC_OVER enables or disables decimal overflow detection for the calling procedure activation. The previous setting is returned as a value.

Format

old-setting = LIB$DEC_OVER (new-setting)

new-setting
: Address of byte containing the new decimal overflow enable setting. Bit 0 = 1 means enable, bit 0 = 0 means disable.

old-setting
: The old decimal overflow enable setting. (The previous contents of SF$W_PSW[PSW$V_DV] in the caller's frame.)

Notes

The caller's stack frame will be modified by this procedure.

A call to LIB$DEC_OVER affects only the current procedure activation and does not affect any of its callers or any procedures that it may call. However, the setting does remain in effect for any procedures which are entered through a JSB entry point.

LIB$FLT_UNDER

6.5.2 Enable/Disable Floating-Point Underflow Detection

LIB$FLT_UNDER enables or disables floating-point underflow detection for the calling procedure activation. The previous setting is returned as a value.

Format

old-setting = LIB$FLT_UNDER (new-setting)

new-setting
: Address of byte containing new floating-point underflow enable setting. Bit 0 = 1 means enable; bit 0 = 0 means disable.

old-setting
: The old floating-point underflow enable setting. (The previous contents of the SF$W_PSW[PSW$V_FU] in the caller's frame.)

Notes

The caller's stack frame will be modified by this procedure.

LIB$FLT_UNDER affects only the current procedure activation and does not affect any of its callers or any procedures that it may call. However, the setting does remain in effect for any procedures entered through a JSB entry point.

Examples

In FORTRAN, the following main program enables reporting of floating-point underflow. If a floating-point underflow occurs in the main program, a severe error condition is signaled, and the process exits. Any underflow occurring in any procedure called by the main program is undetected, unless that procedure also calls LIB$FLT_UNDER.

```
      PROGRAM MAIN
      CALL LIB$FLT_UNDER(1)
      .
      .
      .
      END
```

In MACRO, the equivalent main program is:

```
        .TITLE MAIN
MAIN:   .ENTRY MAIN, ^M<,,,>
        BISPSW #M^<FU>          ; enable floating underflow
        .
        .
        .
        MOVL #1, R0             ; return success
RET     ; end of main program
.END MAIN                       ; start at MAIN
```


6.5.3 Enable/Disable Integer Overflow Detection

LIB$INT_OVER enables or disables integer overflow detection for the calling procedure activation. The previous setting is returned as a value.

Format

old-setting = LIB$INT_OVER (new-setting)

new-setting
: Address of byte containing the new integer overflow enable setting. Bit 0 = 1 means enable, bit 0 = 0 means disable.

old-setting
: The old integer overflow enable setting. (The previous contents of SF$W_PSW[PSW$V_IV] in the caller's frame.

Notes

The caller's stack frame will be modified by this procedure.

LIB$INT_OVER affects only the current procedure activation and does not affect any of its callers or any procedures that it may call. However, the setting does remain in effect for any procedures which are entered through a JSB entry point.

6.6 Generating Signals

This section describes the procedures available for explicitly signaling an exception condition.

Signaling is the method a procedure uses to indicate to the user or the calling program that an exception condition has occurred. When a program wishes to issue a message and optionally continue execution after handling the condition, it calls the standard procedure:

CALL LIB$SIGNAL (condition-value, parameters...)

When a program wishes to issue a message and stop unconditionally, it calls the procedure:

CALL LIB$STOP (condition-value, parameters...)

In both cases, condition-value indicates the condition that is being signaled. However, LIB$STOP always forces the severity of condition-value to SEVERE. The parameter list describes the details of the exception condition. These are the same parameters used to issue a system message.

Unlike most calls, LIB$SIGNAL and LIB$STOP preserve R0 and R1 as well as the other registers. Therefore, a call to LIB$SIGNAL allows the debugger to display the entire state of the process at the time of the exception. This is useful for debugging checks and statistics gathering.

Hardware exceptions behave in the same manner as a call to LIB$SIGNAL. That is, the same stack scan is used and the same parameters are passed to each condition handler. This allows a user to write a single condition handler to detect both hardware and software conditions.

LIB$SIGNAL

6.6.1 Signal Exception Condition

LIB$SIGNAL is called whenever it is necessary to indicate an exception condition or output a message rather than return a status code to the calling program.

LIB$SIGNAL examines the primary and secondary exception vectors and then scans the stack frame by frame, starting at the top of the stack. The

stack frames are found by using the frame pointer (FP) to chain back through the stack frames using the saved FP in each frame. (See the preceding Figure 6–1.) LIB$SIGNAL calls each condition handler encountered.

If an encountered handler returns a "continue" code (that is, any success completion code with bit 0 = 1), LIB$SIGNAL returns to its caller, which should be prepared to continue execution.

If an encountered handler returns a "resignal" code (that is, any failure completion code with bit 0 = 0) the stack scan is continued.

LIB$SIGNAL will, if necessary, scan up to 64K previous stack frames and then finally examine the last-chance exception vector.

Format

CALL LIB$SIGNAL (condition-value [,parameters...])

condition-value
: A standard signal name designating a VAX–11, system-wide, 32–bit condition value (passed immediate value).

parameters
: Optional additional FAO (formatted ASCII output) parameters for message. See Section 6.6.4 for the message format (passed immediate value).

Notes

The argument list is copied to the signal argument list vector, and the Program Counter (PC) and Program Status Longword (PSL) of the caller are appended.

If a handler indicates unwind by calling SYS$UNWIND, then control will not return to the caller of LIB$SIGNAL, thereby changing the flow of control. A handler can also modify the saved copy of R0/R1 in the mechanism vector. If a handler does neither of these things, then all registers including R0/R1 and the hardware condition codes are preserved.

Examples

In FORTRAN, the following code fragment would signal the standard system message ACCESS VIOLATION:

```
INCLUDE 'SYS$LIBRARY:SIGDEF'      ! define SS$... symbols

CALL LIB$SIGNAL (%VAL (SS$_ACCVIO))
```

The FORTRAN compile-time function %VAL is needed because LIB$SIGNAL expects parameters to be passed by-value.

In MACRO, the equivalent code is:

```
.EXTRN    SS$_ACCVIO       ; Declare external symbol
PUSHL     #SS$_ACCVIO      ; Condition value symbol
                           ; for access violation
CALLS     #1, LIB$SIGNAL   ; Signal the condition
```

In FORTRAN, the following code fragment would signal the FORTRAN FILE NOT FOUND message followed by unit number, file name, and user PC (but not VAX-11 RMS message):

```
INCLUDE 'SYS$LIBRARY:FORDEF'  ! define FOR$... SYMBOLS

CALL LIB$SIGNAL (%VAL(FOR$_FILNOTFOU), %VAL(3), %VAL(UNIT),
1MONDAY.DAT')
```

NOTE

The third FAO parameter (user PC) is supplied by LIB$SIGNAL itself.

In MACRO, the equivalent code is:

```
.EXTRN    FOR$_FILNOTFOU     ; Declare condition value
PUSHAQ    FILE_NAME_DSC      ; Address of string descriptor
PUSHL     UNIT               ; Logical unit
PUSHL     #3                 ; No. of FAO parameters following
                             ; (uses PC supplied by LIB$SIGNAL)
PUSHL     #FOR$_FILNOTFOU    ; Condition value
                             ; FILE NOT FOUND
CALLS     #4, LIB$SIGNAL     ; Signal the condition
```

In FORTRAN, the following user defined (bit 27 = 1 and bit 15 = 1) condition-value N is signaled:

```
CALL LIB$SIGNAL (%VAL(N + 2**27 + 2**15))
```

In MACRO, the preceding example is:

```
$STSDEF                    ; Define condition value
                           ; fields (STS$...)
PUSHL     #<N + STS$V_CUST_DEF + STS$V_FAC_SP>
CALLS     #1, LIB$SIGNAL   ; Signal the condition
```

6.6.2 Stop Execution Via Signaling

LIB$STOP is called whenever it is necessary to indicate an exception condition or output a message because it is impossible to continue execution or return a status code to the calling program. LIB$STOP scans the stack frame-by-frame, starting with the most recent frame calling each established handler (see the preceding Figure 6–1). LIB$STOP guarantees that control will not return to the caller.

Format

CALL LIB$STOP (condition-value [,parameters...])

condition-value
A standard signal name for a VAX–11 system-wide 32–bit condition value (passed immediate value).

parameters
Optional additional FAO parameters for message. See 6.6.4 for format for messages (passed immediate value).

Notes

The argument list is copied to the signal argument list vector and the PC and PSL of the caller are appended.

The severity of condition-value is forced to SEVERE before each call to a handler.

If any handler attempts to continue by returning a success completion code, the error message ATTEMPT TO CONTINUE FROM STOP is printed and the user's program exits.

If a handler indicates unwind by calling SYS$UNWIND, control will not return, thereby changing the flow of control. A handler can also modify the saved copy of R0/R1 in the mechanism vector.

NOTE

The only way a handler can prevent the image from exiting after a call to LIB$STOP is to unwind the stack using the SYS$UNWIND system service.

Examples

The same calling sequence as for LIB$SIGNAL.

6.6.3 Signaling Messages

To understand how to write a handler which obtains the error message text, you must understand the system-supplied default handlers (see Section 6.4) and the SYS$PUTMSG system service.

Most user-mode images (such as compilers, utilities, and user programs) need to send single or multi-line messages to the interactive or batch user. These messages can be informational and/or error messages. For example, the DCL COPY utility generates error sequences similar to the following in the event that a file cannot be opened:

```
%COPY-E-OPENIN, error opening <file name> as input
-RMS-F-FNF, file not found

%COPY-E-OPENOUT, error opening <file name> as output
-RMS-E-,PRV privilege violation (OS denies access)

%COPY-E-OPENOUT, error opening <file name> as output
-RMS-F-ATW, attribute write error
-SYSTEM-W-FCPWRITERR, file processor write error
```

6.6.4 Signal Argument List

This section describes the method for sending messages to a user through use of the signaling mechanism.

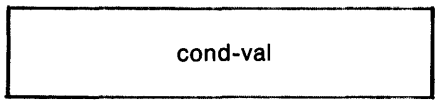
When any software detects an error, it sends a message to the user by calling LIB$SIGNAL or LIB$STOP. A signal argument list is made up of one or more message sequences.

LIB$SIGNAL and LIB$STOP copy the signal argument list and use it to create a signal argument vector. The signal argument vector serves as part of the input parameters to the user established handlers and the system default handlers. It also serves as input to the system service SYS$PUTMSG, which outputs the message. This section describes various formats for those parts of the signal argument list that could be interpreted by SYS$PUTMSG.

The system-supplied default handlers call SYS$PUTMSG to actually output the condition being signaled, provided that all intervening handlers have resignaled. SYS$PUTMSG interprets the signal argument vector as a series of one or more message sequences. Each message sequence starts with a 32-bit, VAX-11 system-wide condition value that identifies a message in the system message file. SYS$PUTMSG obtains the text of the message using SYS$GETMSG. The message text may contain embedded FAO (Formatted ASCII Output) directives (see SYS$FAO system service in *VAX/VMS System Services Reference Manual.*) SYS$PUTMSG calls SYS$FAO to format the message, substituting the values listed below from the signal argument list. Finally, SYS$PUTMSG outputs the message on device SYS$OUTPUT. It also outputs the message on device SYS$ERROR, if SYS$ERROR is different from SYS$OUTPUT, and the condition value severity field is anything but SUCCESS; that is: INFO, WARNING, ERROR, or SEVERE.

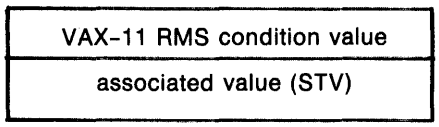
Each message sequence in the signal argument list produces a line of output. The format of a message sequence is one of the following:

- *No FAO arguments:*



Note that a condition value of 0 results in no message.

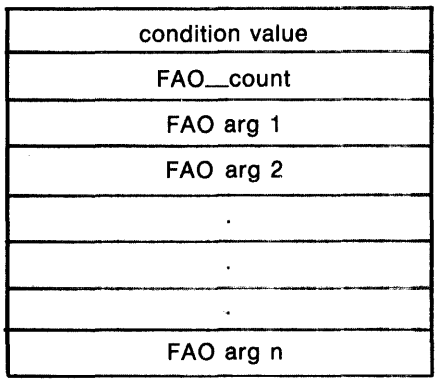
- *VAX-11 RMS error with STV value:*



condition value

1 FAO arg or SS$... cond value

- *Variable number of FAO arguments:*



condition value

number of FAO args

Section 6.7.1 describes the format of signal argument lists as passed to a condition handler.

VAX-11 RMS system services return two related completion values. The primary completion code is the returned value (R0 or function value) and is also placed in the associated VAX-11 RMS FAB, or RAB (FAB$L_STS or RAB$L_STS). The associated value is returned in the same FAB or RAB (FAB$L_STV or RAB$L_STV). The meaning of this secondary value is based on the corresponding STS value. It could be: (1) an operating system condition value of the form SS$_..., (2) a VAX-11 RMS value such as the size of a record which exceeds the buffer, or (3) zero.

Rather than have each caller determine the meaning of the STV value, SYS$PUTMSG performs the necessary processing. Therefore, this STV value must always be passed in place of the FAO argument count. In other words, a VAX-11 RMS message sequence always consists of two arguments (passed by immediate value): an STS value and an STV value.

6.7 Condition Handlers

This section describes how to write and call condition handling procedures. Section 6.8 describes the various options available upon returning from a condition handler. More information is available in Section C.11.1 of Appendix C.

The VAX-11 condition handling facility scans the stack until it finds a pointer to a condition handler.

Format

continue = handler (signal-args, mechanism-args)

signal-args
: The address of a vector of longwords that indicate the nature of the condition. The format of the vector has the same open-ended structure whether the condition was signaled by the operating system, by calling LIB$SIGNAL, or by calling LIB$STOP. In the last two cases, signal-args is a copy of the argument list passed to LIB$SIGNAL or LIB$STOP, with the caller's PC and PSL appended. See Section 6.7.1, Signal Argument Vector, for a detailed description.

mechanism-args
: The address of a vector of longwords that indicate the state of the process at the time of the signal. See Section 6.7.2.

continue
: A condition value. Success (bit 0 = 1) causes execution to continue at PC and failure (bit 0 = 0) causes the condition to be resignaled, that is, the stack scan for other handlers is resumed. If the SYS$UNWIND system service was called, the return value is ignored and the stack is unwound. See Section 6.8.3.

Notes

Handlers can modify the contents of either the signal-args vector or the mechanism-args vector. Generally, a BASIC handler cannot access the signal and mechanism vectors.

In high-level languages, a condition handler is a function that returns a longword integer value. You must provide two dummy arguments for a condition handler:

1. An array to reference signal arguments
2. An array to reference mechanism arguments

For example, you could define a condition handler in FORTRAN as follows:

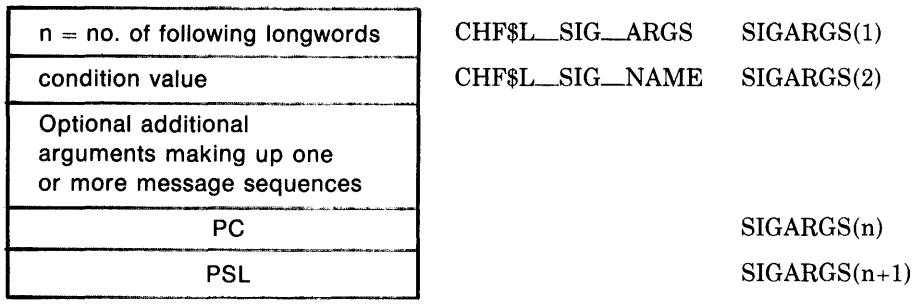
```
INTEGER*4 FUNCTION HANDLER (SIGARGS, MCHARGS)
INTEGER*4 SIGARGS (7), MCHARGS (5)
```

The dimension bounds for the SIGARGS array should specify as many entries as necessary to reference the optional arguments. (The value seven in this example is for the purpose of illustration only.)

In MACRO and BLISS, the symbols CHF$L_SIGARGLST (=4) and CHF$L_MCHARGLST (=8) can be used to obtain the addresses of the signal and mechanism argument vectors, respectively, relative to the argument pointer (AP).

6.7.1 Signal Argument Vector

The signal argument vector contains all of the information describing the nature of the hardware or software condition. It has the following open-ended structure, which can be from 4 to 258 longwords in length:



Each longword entry contains the following:

SIGARGS(1) Contains an unsigned integer (n) designating the number of longwords that follow in the vector, including PC and PSL. For example, the first entry of a four-longword vector would contain a three.

SIGARGS(2) Contains a condition value indicating the condition being signaled. (See Section 6.2.1.) Handlers should always check to see if the condition is the one that they expect by examining the STS$V_COND_ID field (bits 27:3). Bits 2:0 are the severity field and bits 31:28 are control bits which may have been changed by an intervening handler and so should not be included in the comparison. The LIB$MATCH_COND procedure is provided for matching the correct fields (see Section 6.10.1). If the condition is not expected, the handler should resignal by returning FALSE (bit 0 = 0).

SIGARGS(3 to n–1) Contain optional arguments that provide additional information about the condition. These arguments consist of one or more message sequences. The format of a message sequence is described in Section 6.6.4.

SIGARGS(n) Contains the PC of the next instruction to be executed should any handler (including the system-supplied handlers) return with continue TRUE. For hardware faults, the PC is that of the instruction that caused the fault.For hardware traps, the PC is that of the instruction following the one that caused the trap. For conditions signaled by calling LIB$SIGNAL or LIB$STOP, the PC is that location following the CALLS or CALLG instruction.

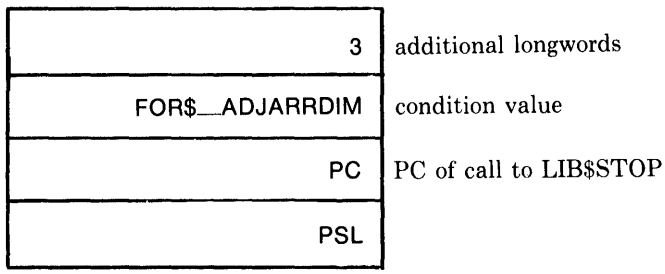
SIGARGS(n+1) Contains the PSL of the program at the time that the condition was signaled. See the *VAX-11 Architecture Handbook*.

NOTE

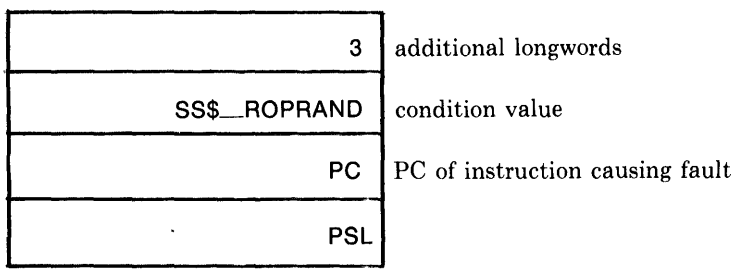
When called, LIB$SIGNAL and LIB$STOP copy the variable-length parameter list passed by the caller, then append the PC and PSL entries to the end of the list before calling handlers.

The formats for all conditions signaled by the operating system and some conditions signaled by the Run-Time Library follow:

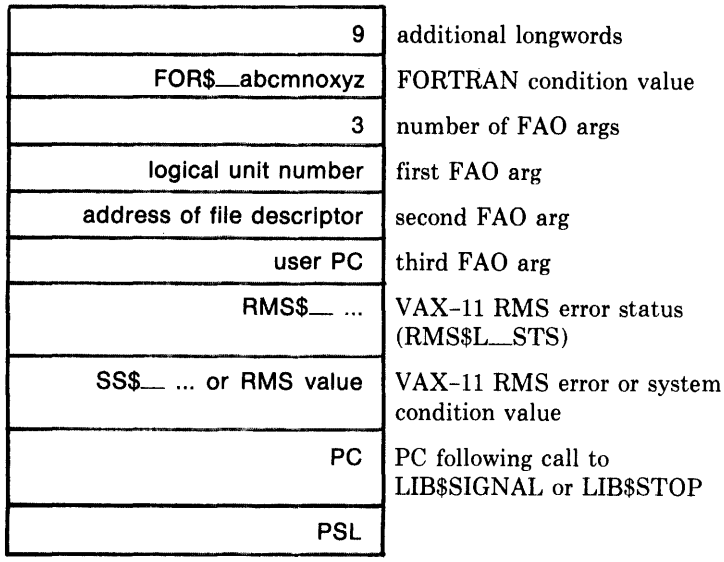
- *The signal argument vector for the reserved operand error condition* is:



- *The signal argument vector for the FORTRAN error condition ADJUSTABLE ARRAY DIMENSION ERROR* is:



- *Signal argument vector for FORTRAN I/O statement errors* is:



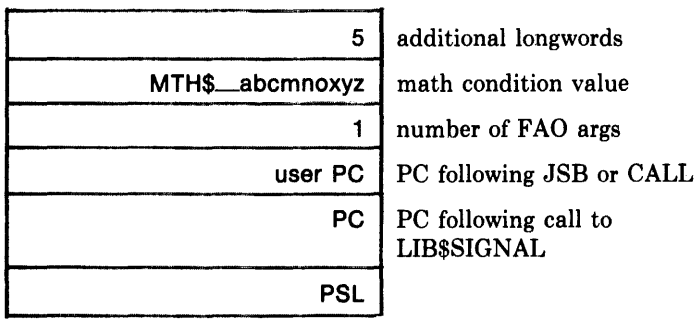
NOTE

In the future, additional FAO arguments may be added following the user PC with a correspondingly increased FAO argument count. Thus, user handlers accessing either RMS longword, should use the contents of SIGARGS(3) as part of the subscript. In FORTRAN, the VAX-11 RMS error status is accessed as:

= SIGARGS (SIGARGS(3) + 4)

If the error does not involve VAX-11 RMS and/or VAX/VMS, the corresponding signal vector entries are 0, which are skipped over by SYS$PUTMSG.

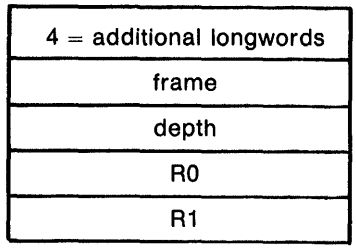
- *The signal argument vector for mathematics procedures errors* is:



The user PC is the PC that follows the user JSB or CALL to the mathematics procedure detecting the error. The PC is that following the call to LIB$SIGNAL.

6.7.2 Mechanism Argument Vector

The mechanism argument vector contains all of the information describing the state of the process at the time of the hardware or software signaled condition. It is a five-longword vector of the form:



	MACRO	FORTRAN
4 = additional longwords	CHF$L_MCH_ARGS	MCHARGS(1)
frame	CHF$L_MCH_FRAME	MCHARGS(2)
depth	CHF$L_MCH_DEPTH	MCHARGS(3)
R0	CHF$L_MCH_SAVR0	MCHARGS(4)
R1	CHF$L_MCH_SAVR1	MCHARGS(5)

The contents of each longword entry is:

MCHARGS(1) Contains an unsigned integer indicating the number of longwords that follow in the vector. Currently, this is always four.

MCHARGS(2) Contains the address of the stack frame of the procedure activation that established the handler being called. This address can be used as a base from which to reference the local stack allocated storage of the establisher as long as the restrictions in Section 6.7.3 are observed.

MCHARGS(3) Contains the stack depth, which is the number of stack frames between the establisher of the condition handler and the frame in which the condition was signaled. In other words, it indicates the number of calls from the establisher which have not yet returned. In order that calls to LIB$SIGNAL and LIB$STOP appear as similar as possible to hardware exception conditions, the call to LIB$SIGNAL or LIB$STOP is not included in the depth.

The depth is 0 for an exception handled by the procedure activation invoking the exception. That is, the activated procedure containing the hardware exception or calling LIB$SIGNAL or LIB$STOP. The depth is 1 for an exception handled by the immediate caller of the procedure activation in which the exception occurred, and so on. If a system service signals an exception, a handler established by the immediate caller is entered with a depth of 1.

The depth is -2 for a handler established using the primary exception vector, -1 for the secondary vector, and -3 for the last-chance vector.

MCHARGS(4)
MCHARGS(5) Contain copies of the contents of R0 and R1 at the time of the exception or the call to LIB$SIGNAL or LIB$STOP. When execution continues or a stack unwind occurs, these

values are restored to R0 and R1. Thus a handler can modify these values to change the function value returned to a caller.

When writing condition handlers, you should determine whether the error that has occurred is the one expected. This can be done by checking the condition value in the signal argument vector and/or the depth in the mechanism vector for the expected values.

Examples

The following FORTRAN program establishes a handler which corrects a SIGNIFICANCE LOST IN MATH LIBRARY error by setting the value to be returned in R0 or R0/R1 to 0 in the mechanism vector. It continues execution rather than resignaling. No error message is printed.

```
EXTERNAL HANDL
CALL LIB$ESTABLISH (HANDL)
.
.
.
Y = SIN(X)
.
.
.
END

INTEGER*4 FUNCTION HANDL (SIGARGS, MECHARGS)
INTEGER*4 SIGARGS(7), MECHARGS(5)
INCLUDE 'SYS$LIBRARY:MTHDEF.FOR' ! define MTH$... symbols
HANDL = 0                   ! Assume Resignal
IF (SIGARGS(2) .EQ. MTH$_SIGLOSMAT) THEN
     MECHARGS(4) = 0        ! set image of R0 to 0
     MECHARGS(5) = 0        ! set image of R1 to 0
     HANDL = 1              ! force continue instead of resignal
ENDIF
RETURN
END
```

When the handler is called, it tests to see if the error being signaled is MTH$__SIGLOSMAT (SIGNIFICANCE LOST IN MATH LIBRARY). If it is, the handler sets the saved copy of R0/R1 in the mechanism vector to 0, so that the function value result returned to the caller of the math routine will be +0.0, rather than the reserved operand -0.0. Then, rather than resignaling, it returns success so that execution continues. No error message is printed.

The equivalent MACRO code for this handler is:

```
$CHFDEF                         ; define CHF$... symbols
.EXTRN   MTH$_SIGLOSMAT         ; condition value
.ENTRY   HANDL, ^M< >
CLRL     R0                     ; assume resignal
MOVL     CHF$L_SIGARGLST(AP), R1   ; R1 = adr of signal arg vector
CMPL     CHF$L_SIG_NAME(R1), #MTH$_SIGLOSMAT
BNEQ     10$                    ; branch if not expected error
```

```
      MOVL      CHF$L_MCHARGSLST(AP), R1   ; R1 = adr of mech arg vector
      CLRQ      CHF$L_MCH_SAVR0(R1)   ; set math return value to +0.0
      MOVL      #1, R0                ; return SS$_CONTINUE
10$:  RET                             ; resignal or continue
```

The following FORTRAN code fragment asks the user for a file name. If an error occurs, the FORTRAN program signals the standard FORTRAN I/O statement error condition, using the same format as would have been signaled by the Run-Time Library if ERR= had been omitted. Thus, the user is told exactly why the file could not be opened.

```
      CHARACTER*40 FILE_NAME
      INTEGER*4 RMS_STS, RMS_STV, COND_VAL
      .
      .
      .
100   TYPE *, ' Type File Name '
      ACCEPT *, FILE_NAME
      OPEN (UNIT = 10, ERR = 200, TYPE = 'OLD', NAME= FILE_NAME)
      .
      .
      .
200   CALL ERRSNS (, RMS_STS, RMS_STV, COND_VAL)
      CALL LIB$INSV (3,0,3 COND_VAL)  ! Set Severity to INFO
      CALL LIB$SIGNAL (%VAL (COND_VAL),
      1    %VAL(3),                   ! No. of FAO args
      2    %VAL(10),                  ! Logical Unit No.
      3    FILE_NAME,                 ! File Name
      4    %VAL(0),                   ! User PC
      5    %VAL(RMS_STS),             ! RMS Error Status or 0
      7    %VAL(RMS_STV))             ! VAX/VMS Status, RMS value or 0
      GO TO 100                       ! try again
```

If any open error occurs, the ERR= transfer occurs (with no signaled condition). Statement 200 calls ERRSNS, which returns the FORTRAN condition value, the VAX-11 RMS value and VAX/VMS condition value. The severity field is then set to 3 to indicate INFO so that no stack traceback is printed. Finally, the condition is signaled using the same format as the Run-Time Library itself. However, since the condition is being signaled by LIB$SIGNAL instead of LIB$STOP and the severity has been changed from SEVERE to INFO, execution will continue after the message has been printed.

In BASIC, the user condition handler (ON ERROR GO TO line-number) can only intercept BASIC specific errors. All other errors are automatically resignaled without giving control to the BASIC error handler. The BASIC error number can be obtained using the BASIC built-in function, ERR.

6.7.3 Restrictions for Accessing Data from Handlers

In order not to affect compiler optimization, a handler and anything it calls are restricted to referencing only arguments explicitly passed to the handlers. They cannot reference COMMON or other external storage, nor can they

reference local storage in the procedure that established the handler. Compilers that relax this rule must ensure that any variables referenced by the handler are always kept in memory, not in a register.

6.8 Returning from a Condition Handler

There are three mutually exclusive possibilities for a handler when it returns control to its caller, the VAX-11 Condition Handling Facility:

1. Indicate that the condition is to be resignaled (R0<0> = 0)
2. Indicate that execution is to continue at the point of the signal (R0<0> = 1)
3. Indicate that the stack is to be unwound (call SYS$UNWIND)

6.8.1 Resignaling

All condition handlers should check for specific errors. If the signaled condition is not one of the expected errors, a handler should resignal. If a handler wants to resignal the condition, it returns with the function value SS$_RESIGNAL ("false", that is, with bit 0 clear). If a handler wants to alter the severity of the signal, it modifies the low three bits of the condition value and resignals.

For example, if a handler changes the severity of an error from SEVERE to ERROR and resignals, the default action is for the error message and a stack traceback to be printed by the default traceback handler and for execution to continue. If a handler changes the severity from SEVERE or ERROR to INFO, the default action is for the error message to be printed and for execution to continue. If a handler wants to alter the defined control bits of the signal, it modifies bits 31:28 of the condition value and resignals.

Example

The following FORTRAN example enables floating-point underflow after first establishing a handler. The handler changes the severity of any floating-point underflow condition from SEVERE to INFO and then resignals. Therefore, each floating-point underflow error gets a message, and then continues using the hardware fixup of zero.

```
SUBROUTINE CALC
EXTERNAL HANDLE_FU
CALL LIB$ESTABLISH (HANDLE_FU)
CALL LIB$FLT_UNDER(1)
.
.
.
RETURN
END

INTEGER*4 FUNCTION HANDLE_FU (SIGARGS, MECHARGS)
INTEGER*4 SIGARGS(3), MECHARGS (5)
INCLUDE 'SYS$LIBRARY:SIGDEF'          ! Define SS$_...symbols
HANDLE_FU = SS$_RESIGNAL              ! Always resignal
```

```
IF (SIGARGS(2) .EQ. SS$_FLTUND) THEN  ! If floating underflow
      CALL LIB$INSV (3,0,3,SIGARGS(2))  ! Set INFO, severity
ENDIF
RETURN                                ! Always Resignal
END
```

When any exception occurs in CALC, HANDLE_FU is called. It first determines if the condition value is the floating-point underflow arithmetic trap (SS$_FLTUND). If so, it changes the severity to INFO(3). Then, it always resignals the error so that an error message is always printed. Execution continues only for floating underflow errors.

NOTE

To resignal in BASIC, a user condition handler executes the statement, ON ERROR GO BACK.

6.8.2 Continuing

If a condition handler wants execution to continue from the instruction following the call to LIB$SIGNAL or the instruction following a hardware arithmetic trap (such as integer overflow), and also wants no error messages or traceback, it must return with the function value SS$_CONTINUE (bit 0 = 1). If, however, the condition was signaled with a call to LIB$STOP, the error message: ATTEMPT TO CONTINUE FROM STOP is printed and the image is exited. The only way to continue from a call to LIB$STOP is for the condition handler to request a stack unwind. If it wants to unwind, it calls SYS$UNWIND and then returns (see Section 6.8.3). In this case the handler function value is ignored.

If execution is to continue after a hardware fault (such as a reserved operand fault) has occurred, the condition handler must correct the cause of the condition before returning the function value SS$_CONTINUE or requesting a stack unwind. The correction is required; otherwise, the instruction that caused the fault will be executed again.

Examples

In FORTRAN, the following procedure first enables floating-point underflow detection, then establishes a handler which merely tallies each floating-point underflow trap and continues:

```
C  PROGRAM TO COUNT UNDERFLOWS
   EXTERNAL CNT_HANDLER, NO_UNDERFLOWS

   CALL LIB$ESTABLISH (CNT_HANDLER)
   CALL LIB$FLT_UNDER (1)
   .
   .
   .
   TYPE *, 'Number of Underflow Traps:', NO_UNDERFLOWS()
   END
```

```
      INTEGER*4 FUNCTION CNT_HANDLER (SIGARGS, MECHARGS)
      INTEGER*4 SIGARGS(3), MECHARGS(5), UNDER_FLO_COUNT
      INCLUDE 'SYS$LIBRARY:SIGDEF'        ! define system symbols
      SAVE UNDER_FLO_COUNT
      CNT_HANDLER = SS$_RESIGNAL          ! Assume resignal
      IF (SIGARGS(2) .EQ. SS$_FLTUND) THEN ! If float underflow
           UNDER_FLO_COUNT = UNDER_FLO_COUNT + 1
           CNT_HANDLER = SS$_CONTINUE     ! Change to continue
      ENDIF
      RETURN
C
C  Routine to return number of underflows
C
      ENTRY NO_UNDERFLOWS
      NO_UNDERFLOWS = UNDER_FLO_COUNT
      RETURN
      END
```

If an exception occurs during the execution of the main program, the condition handler (CNT__HANDLER) is called. This handler must determine whether the condition being signaled is one that it was expecting. A floating-point underflow trap is signaled as an arithmetic exception (SS$__FLTUND). Thus, the handler tests for the condition value (SIGARGS(2)). If the condition is floating-point underflow, the handler counts it and continues execution at the point of the underflow. If the condition is anything other than floating-point underflow, it is resignaled. In the case of underflow, the hardware automatically corrects the result to be +0.0.

In BASIC, a user condition handler cannot continue execution at the next instruction (see next section).

6.8.3 Request to Unwind

Stack unwinding is a way to remove one or more frames from the stack starting with the frame in which the condition occurred. It is a fairly drastic method of altering the flow of control. It may be used whether the condition was detected by hardware, or signaled by LIB$SIGNAL or LIB$STOP. Unwinding is the only way to continue execution after LIB$STOP has been called. In addition to specifying the number of pre-signal frames to be removed, a return PC that is different from the one in the last frame unwound can be specified.

If a handler wants to unwind, the handler, or any procedure it calls, executes the SYS$UNWIND system service as specified by:

Format

```
ret-status = SYS$UNWIND( [depth = handler depth + 1],
                         [new-PC = return PC] )
```

depth
: Address of a longword containing the number of frames to be removed, starting with the frame where the condition occurred. A depth of zero indicates the call frame that was active when the condition occurred, one indicates the caller of that frame, two indicates the caller of the caller of

the frame, and so on. If depth is specified as zero or less, no unwind occurs. If no address is specified, the unwind is performed to the caller of the frame that established the condition handler, that is the handler depth plus one.

new-PC
The address of the instruction to receive control when the unwind is complete. It is passed by-value. The default (new-PC = 0) is to continue execution with the instruction immediately following the CALLS or CALLG to the last procedure that is unwound.

Return Status

SS$_NORMAL
Service successfully completed.

SS$_NOSIGNAL
No signal was active.

SS$_UNWINDING
Already unwinding.

SS$_INSFRAME
Insufficient frame depth.

Notes

Because this is a system service, the comma is required if both optional arguments are omitted.

If the handler wants to specify the function value of the last function to be unwound, it should modify the saved copies of R0 and R1 (CHFL_MCH_SAVR0, CHFL_MCH_SAVR1) in the mechanism vector. R0 and R1 are restored from the mechanism argument vector at the end of the unwind.

Depending on the argument(s) to SYS$UNWIND, the unwinding operation will terminate as follows in FORTRAN:

- SYS$UNWIND(,) — unwind to the establisher's caller
- SYS$UNWIND(DEPTH,) — unwind to the establisher at the point of the call that resulted in the exception
- SYS$UNWIND(DEPTH,%VAL(LOCATION)) — unwind to a specified activation and transfer to a specified location

SYS$UNWIND can be called whether the condition was a software condition signaled by calling LIB$SIGNAL or LIB$STOP, or was a hardware exception.

Any function value from the handler is ignored. Therefore, a handler cannot both resignal and unwind. Consequently, the only way for a handler to both issue a message and unwind is to call LIB$SIGNAL and then call SYS$UNWIND. (See Section 6.11, Multiple Active Signals.)

The unwind will occur when the handler returns to its caller, the condition handling facility. Unwinding is done by scanning back through the stack and calling each handler that has been established in a frame. Each handler is called with a condition value of SS$_UNWIND to perform any application specific cleanup. In particular, if the depth specified includes unwinding the establisher's frame, then the current handler will be called again with this unwind exception. Handlers established by the primary, secondary, or last-chance vectors are not called, since they are not removed during an unwind operation.

The call to the handler is of the same form as described previously with the following values:

signal-args
    1
    condition_value = SS$_UNWIND

mechanism-args
    4
    frame establisher's frame
    depth 0 (that is, unwinding self)
    R0 R0 that unwind will restore
    R1 R1 that unwind will restore

When the handler returns, the return status from the handler is ignored. The stack is then cut back to the previous frame.

Example

This FORTRAN example shows a matrix inversion procedure, using the logical function INVERT to indicate success or failure. Thus, if the matrix can be inverted, the logical value returned in INVERT is .TRUE. If, however, the matrix is singular, and therefore cannot be inverted, the logical value .FALSE. is returned. A condition handler is provided to detect failure and return .FALSE. to the calling program. Note that the condition handler is an INTEGER*4 function.

```
LOGICAL FUNCTION INVERT (A,N)
DIMENSION A(N,N)
EXTERNAL HANDL
CALL LIB$ESTABLISH (HANDL)      ! ESTABLISH HANDLER
INVERT = .TRUE.                 ! ASSUME SUCCESS
         .
         .    (INVERT THE MATRIX)
         .
RETURN
END
```

```
 INTEGER*4 FUNCTION HANDL (SIGARGS, MECHARGS)
 INTEGER*4 SIGARGS(3), MECHARGS(5)
 INCLUDE 'SYS$LIBRARY:SIGDEF'
 HANDL = SS$_RESIGNAL            ! ASSUME RESIGNAL

 IF (SIGARGS(2) .EQ. SS$_FLTOVF .OR. SIGARGS(2)
1 .EQ. SS$_FLTDIV) THEN
       MECHARGS(4) = .FALSE.
       CALL SYS$UNWIND(%VAL(0),%VAL(0))
       ENDIF

RETURN
END
```

If an exception occurs during the execution of INVERT, the condition handler (HANDL) is called. The handler must first determine whether the condition being signaled is one that it can deal with. A floating-point overflow is signaled as an arithmetic exception with additional arguments indicating the specific arithmetic exception. Thus, the condition handler tests the condition value (SIGARGS(2)) and the optional third argument (SIGARGS(3)). If the condition is floating overflow, the condition handler causes a return to INVERT with the value .FALSE.

If the condition is floating-point underflow, the condition handler uses the unwind procedure to force a return to the procedure which called INVERT. The logical value .FALSE. is stored in the saved R0 element of the mechanism vector (MECHARGS(4)). This value is used as the function value for INVERT when the unwind occurs. The handler calls SYS$UNWIND and returns; the VAX-11 Condition Handling Facility then gets control and actually performs the unwind operation. Note that the function value from the user condition handler (HANDL = .FALSE.) is ignored if SYS$UNWIND is called.

If the exception condition is not a floating overflow, the condition handler returns a value of .FALSE., indicating that it is not able to deal directly with the condition. The immediately preceding procedure activation is then checked for a condition handler; the search continues until an established condition handler or the system condition handler is reached.

In BASIC, a user condition handler can restart the current statement in the same module using a RESUME statement or can start at an arbitrary statement in the same module using a RESUME line-number statement.

6.8.4 Summary of Interaction Between Handlers and Default Handlers

All combinations of interaction between condition handler actions, the default condition handlers, the type of signal, and the call to signal or stop are detailed in Table 6–2.

Table 6-2: Interaction Between Handlers and Default Handlers

CALL to:	Signaled Condition Severity <2:0>	Default Handler Gets Control	Handler Specifies Continue	Handler Specifies UNWIND	No Handler Is Found (bad stack)
LIB$SIGNAL or hardware exception	<4	condition message RET	RET	UNWIND	Call last chance handler EXIT
	=4	condition message EXIT	RET	UNWIND	Call last chance handler EXIT
LIB$STOP	force (=4)	condition message EXIT	"CAN'T CONTINUE" EXIT	UNWIND	Call last chance handler EXIT

In the table, CAN'T CONTINUE indicates an error which results in the error message ATTEMPT TO CONTINUE FROM STOP.

6.9 User Logging of Error Messages

A handler can obtain a copy of the text of a signaled error message in order to write the message into an auxiliary file such as a listing file. Thus, the user can receive identical messages at the terminal (or batch log file) and in the auxiliary file.

To log messages, a handler calls the system service SYS$PUTMSG, specifying a signal argument list and the address of an action routine. This routine is called by the handler with each line of the message passed as a single parameter consisting of a string descriptor. To understand how to write a handler which obtains the error message text, you must understand the system supplied default handlers and the SYS$PUTMSG system service.

6.9.1 SYS$PUTMSG Put Message System Service

This section describes the Put Message SYS$PUTMSG system service, which the system-supplied default handlers call to output all error messages to the user terminal or batch log file. To centralize this important function, no other means should be used to output error messages. Furthermore, rather than call this service directly, user programs should call LIB$SIGNAL or LIB$STOP to preserve modularity and permit calling programs to recover or change the error message. The signaled message arguments (Section 6.6.4) consist of one or more message sequences passed to SYS$PUTMSG by the system-supplied condition handler.

Each "message sequence" is processed as follows:

1. Special setup is performed for SYS (subsystem 0) messages and VAX-11 RMS (subsystem 1) messages.
2. The model message text is obtained from a file by calling SYS$GETMSG.
3. SYS$FAO is called, if necessary, to insert caller-supplied information into the model message.
4. The caller's action routine (see Section 6.9.2), if present, is called. If this routine returns a failure code, Steps 5 and 6 are skipped.
5. The message is sent to SYS$OUTPUT.
6. The message is also sent to SYS$ERROR if: (1) the severity of the primary message is not SUCCESS, (that is, bits 2:0 of condition value are not 1); and (2) SYS$ERROR is different from SYS$OUTPUT.

See the *VAX/VMS System Services Reference Manual* for a complete description of SYS$PUTMSG.

Format

ret-status = SYS$PUTMSG (msg-list, [action-routine], [fac-name])

msg-list
Address of array of longwords containing one or more message sets. The contents of the array is the same as that produced by the signal argument vector.

action-subroutine
Optional address of action subroutine called on each line of output.

fac-name
Optional address of a string descriptor for a facility name to replace the one specified in bits 27 to 17 of the condition value in the second longword of msg-list.

Notes

Because this is a system service, the two commas are always required even if both optional arguments are omitted. For example, in FORTRAN:

```
INTEGER COND_VAL, SIGARGS(7)
COND_VAL = SYS$PUTMSG (SIG_ARGS,,)
```

Call LIB$SIGNAL or LIB$STOP instead of SYS$PUTMSG, except when setting up a handler to log messages. Such a handler should return .FALSE. so that the condition is resignaled.

6.9.2 Caller-Supplied Action Subroutine

A caller to SYS$PUTMSG who wants to use the standard message mechanisms but needs to perform additional message processing may specify an

action subroutine to be called after each message line has been formatted but before the message is actually output.

The caller's action subroutine is passed the address of a string descriptor which contains the length and address of the formatted message. The action subroutine can scan the message and/or copy it into a log file.

If the action subroutine returns a success completion code (bit 0 = 1), SYS$PUTMSG puts the message line in file SYS$ERROR and/or SYS$OUTPUT. If a failure code is returned (bit 0 = 0), the remaining SYS$PUTMSG processing for that single message sequence is skipped, so that the message is not output to SYS$OUTPUT or SYS$ERROR. User supplied action routines should return a failure code so that the callers of your procedure can decide whether to output the message or not.

The following FORTRAN handler receives all errors occurring in the main program or any procedures called by the main program, and directs the associated error message text into file ERRLOG.DAT. Then it resignals so that the user also receives the error message on SYS$OUTPUT or SYS$ERROR.

```
C     MAIN PROGRAM
      EXTERNAL LOG_HANDL

      OPEN (UNIT=99, FILE = 'ERRLOG', STATUS = 'NEW')
      CALL LIB$ESTABLISH (LOG_HANDL)
      .
      .
      .
      END

      INTEGER*4 FUNCTION LOG_HANDL (SIGARGS, MECHARGS)
      INTEGER*4 SIGARGS(9), MECHARGS(5)
C     HANDLER TO JOURNAL ANY SIGNALED ERROR MESSAGES
      INCLUDE 'SYS$LIBRARY:SIGDEF'
      EXTERNAL PUT_LINE
      LOG_HANDL = .FALSE.                    ! Always resignal
      CALL SYS$PUTMSG (SIGARGS, PUT_LINE, )
      RETURN
      END
C     ACTION SUBROUTINE              ! Output string passed to unit
C                                    ! 99
      LOGICAL*4 FUNCTION PUT_LINE (LINE)
      CHARACTER*(*)LINE
      PUT_LINE = .FALSE.       ! Always suppress other output
100   WRITE (99,200) LINE
200   FORMAT(A)
      RETURN
      END
```

In this example, the main program opens file ERRLOG.DAT. Then the condition handler LOG_HANDL is established. Because LOG_HANDL is established after ERRLOG.DAT has been opened, LOG_HANDL will not be called if an error occurs while opening the file. When any error condition is signaled, the handler LOG_HANDL is called. It passes the signal argument vector to SYS$PUTMSG, along with the address of the action subroutine PUT_LINE. SYS$PUTMSG calls PUT_LINE once for each line in the error message. PUT_LINE writes the line on unit 99. Then it returns with an error

indication which causes SYS$PUTMSG not to output any lines to SYS$OUTPUT and SYS$ERROR. Finally, LOG_HANDL returns with a resignal so that the regular error message output and traceback will be performed by the system supplied default handlers. The normal VAX-11 RMS I/O rundown will close the log file.

Note that if an error occurs during the WRITE in statement 100, there will be multiple active signals (see Section 6.11). In this case, the stack scan skips frames which have already been scanned for the active signals, thereby avoiding loops. Thus, LOG_HANDL would not be called again. Instead, one of the system default handlers would get control and output the error to the user.

6.10 Signal Handling Procedures

This section describes procedures that can be established as condition handlers or called from handlers to handle signals. The programming examples illustrate common types of handlers.

LIB$MATCH_COND

6.10.1 Match Condition Values

Each handler must examine the signal parameter list vector to determine which error is being signaled. If the error is not one that the handler knows about, the handler should resignal. A handler should not assume that only one kind of error can occur in the procedure which established it or any procedures it calls. However, because a condition value may get modified by an intervening handler, each handler should only compare that part of the condition value that distinguishes it from another.

LIB$MATCH_COND is provided for programmers who want to match a list of one or more condition values. It is designed to be used in multi-way branch statements available in most higher level languages.

LIB$MATCH_COND checks for a match between the condition value addressed by cond-val and the condition values addressed by the subsequent parameters. Each parameter is the address of a longword containing a condition value.

LIB$MATCH_COND takes a portion (STS$V_COND_ID) of the condition value pointed to by the first parameter and compares it to the same portion of the condition value pointed to by the second through nth parameters. Furthermore, if the facility-specific bit (STS$V_FAC_SP = bit 15) is clear in cond-val (meaning that the condition value is system-wide rather than facility specific), the facility code field (STS$V_FAC_NO = bits 27:17) is ignored and only the STS$V_MSG_ID fields (bits 15:3) are compared. (See Section C.4 Condition Values for more details.) The routine returns a 0 if a match is not found, a 1 if the second parameter matches, a 2 if the third parameter matches, and so on. A check is made for null parameter entries in the parameter list.

Format

index = LIB$MATCH_COND (cond-val, cond-val-i...)

cond-val
Address of a longword containing the condition value to be matched.

cond-val-i
Address of longwords containing condition value(s) to be compared to cond-val.

index
A 0, if no match found; i, for match between the first and (i+1)st parameter.

Notes

When LIB$MATCH_COND is called with only two parameters, the possible values for index are .TRUE. (1) or .FALSE. (0).

Examples

The following FORTRAN program fragment tests for File Not Found:

```
INCLUDE 'SYS$LIBRARY:FORDEF.FOR'  ! Declare FOR$... symbols
IF (LIB$MATCH_COND (SIG_ARGS(2), FOR$_FILNOTFOU)) THEN
.
.
.
```

If a match occurs, a true value is returned, if not, a false value is returned.

The following FORTRAN program uses a computed GOTO to dispatch on a condition value:

```
      INCLUDE 'SYS$LIBRARY:FORDEF.FOR' ! Define FOR$... symbols
      INTEGER*4 SIG_ARGS(9)
      I = LIB$MATCH_COND (SIG_ARGS(2), FOR$_FILNOTFOU,
     1FOR$_NO_SUCDEV, FOR$_FILNAMSPE, FOR$_OPEFAI)
      GO TO (100, 200, 300, 400), I
      ! (if Some Other Error)
      .
      .
      .
100   ! (if File Not Found)
200   ! (if No Such Device)
300   ! (if File Name Specification Error)
400   ! (if Open Failure)
      .
      .
      .
```

6.10.2 Fixup Floating Reserved Operand

LIB$FIXUP_FLT finds the reserved operand of any F_, D_, G_, or H_floating instruction (with exceptions stated in the next paragraph) after a reserved operand fault has been signaled. LIB$FIXUP_FLT changes the reserved operand from -0.0 to the parameter, new-operand, if present; or to +0.0 if new-operand is absent.

LIB$FIXUP_FLT cannot handle the following cases and will return a status of SS$_RESIGNAL if any of them occur:

1. The currently active signaled condition is not SS$_ROPRAND.
2. The reserved operand's data type is not F_, D_, G_, or H_floating.
3. The reserved operand is an element in a POLYx coefficient table.

Format

ret-status = LIB$FIXUP_FLT (sig-args-adr, mch-args-adr [,new-operand])

sig-args-adr
Address of signal argument vector.

mch-args-adr
Address of mechanism argument vector.

new-operand
Address of an F_floating value to replace the reserved operand. This is an optional parameter, the default value is +0.0.

Return Status

SS$_NORMAL
Routine successfully completed. The reserved operand was found and has been fixed up.

SS$_ACCVIO
Access violation. An argument to LIB$FIXUP_FLT or an operand of the faulting instruction could not be read or written.

SS$_RESIGNAL
The signaled condition was not SS$_ROPRAND or the reserved operand was not a floating point value or was an element in a POLYx table.

SS$_ROPRAND
Reserved operand fault/abort. The optional argument new-operand was supplied but was itself an F_floating reserved operand.

LIB$_BADSTA
Bad Stack. The stack frame linkage has been corrupted since the time of the reserved operand exception.

Notes

If the status value returned from LIB$FIXUP_FLT is seen by the condition handling facility, (as would be the case if LIB$FIXUP_FLT was the handler), any success value is equivalent to SS$_CONTINUE, which causes the instruction to be restarted. Any failure value is equivalent to SS$_RESIGNAL, which causes the condition to be resignaled to the next handler. This is because the condition handler (LIB$FIXUP_FLT) failed to handle the condition correctly.

Examples

The following FORTRAN program permits 15 floating-point overflows to occur before exiting, thereby overriding the system default action of exiting after the first overflow. The program converts the floating-point overflow condition value from SEVERE to ERROR and resignals. Thus, the error message and stack traceback is printed by the default handler, but execution continues. When the program references the reserved operand stored by the hardware on floating-point overflow, it fixes up the reserved operand and continues without an error message.

```
C  MAIN PROGRAM
   EXTERNAL HANDL
   CALL LIB$ESTABLISH (HANDL)     ! establish handler
   .
   .
   .
   CALL ...
   .
   .
   .
   END

   INTEGER*4 FUNCTION HANDL (SIGARGS, MECHARGS)
   INTEGER*4 SIGARGS (3), MECHARGS (5), ERROR_COUNT
   INCLUDE 'SYS$LIBRARY:SIGDEF'  ! define SS$_... symbols
   HANDL = SS$_RESIGNAL          ! Assume resignal
   IF (LIB$MATCH_COND (SIGARGS(2), SS$_FLTOVF)) THEN ! Float ovf?
      ERROR_COUNT = ERROR_COUNT + 1
      IF (ERROR_COUNT .LT. 15) THEN
         CALL LIB$INSV (2, 0, 3, SIGARGS(2)) ! Set to ERROR
      ENDIF
   ELSE
      HANDL = LIB$FIXUP_FLT (SIGARGS, MECHARGS)
   ENDIF
   RETURN
   END
```

If an exception occurs during execution of the main program, any procedure which it calls, or any procedure which they call, the condition handler (HANDL) is called. The handler must first determine whether the condition being signaled is one that it can act upon. A floating-point overflow is signaled as an arithmetic trap. Thus, the condition handler tests the condition value (SIGARGS(2)) for a match with SS$_FLTOVF by calling LIB$MATCH_COND. If the condition is SS$_FLTOVF, the condition handler increments the count of floating-point overflows. If the

count is still less than 15, the severity field (bits 2 to 0) of the condition value are changed from SEVERE (=4) to ERROR (=2) using the insert field library procedure, LIB$INSV. Changing the severity will cause the program image to continue after printing the message, rather than exiting.

If the condition being signaled was not an arithmetic exception, then LIB$FIXUP_FLT is called to check for a reserved operand condition and if so to correct the reserved operand (if present) to +0.0. If the correction was successful, LIB$FIXUP_FLT returns SS$_NORMAL which, when assigned to HANDL, causes execution to continue with no error message when HANDL returns. If the condition was other than the reserved operand, LIB$FIXUP_FLT returns an error condition which, when assigned to HANDL, causes the condition to be resignaled when HANDL returns.

In MACRO the equivalent code is as follows:

```
    .TITLE FLT_CONT - Continue after floating overflow
    .ENTRY FLT_CONT, ^M<...>
    MOVAL    HANDL, (FP)            ; Establish Handler
             .
             .
             .
    CALL     ...                    ; Call other procedures
             .
             .
             .
    MOVL #1, R0                     ; return success since there
                                    ; were less than 15 errors
    RET                             ; return from main program
    .END FLT_CONT

    .TITLE HANDL - Handler to continue for 15 overflows
    $CHFDEF                         ; def cond hand symbols (CHF$...)
    $STSDEF                         ; & cond value symbols (STS$...)
    $SSDEF                          ; define system symbols (SS$...)
    .PSECT $DATA, RED, WRT, NOEXE
ERROR_COUNT:
    .LONG 0                         ; error count initialized to 0
    .PSECT $CODE, RED, NOWRT, EXE, PIC
    .ENTRY HANDL, ^M< >
    MOVL CHF$L_SIG_NAME(AP), R1     ; R1 = adr of signal vector
    CMPV     #STS$V_COND_ID, -      ; pos of cond ident
             #STS$S_COND_ID, -      ; size of cond ident
             CHF$L_SIG_NAME(R1), - ; the signaled condition value
             #<SS$_FLTOVF@-STS$V_COND_ID> ; arithmetic exception
                                    ; condition shifted right
                                    ; to line up with condition id,
                                    ; so severity field is
                                    ; ignored in case it is already
                                    ; changed by an intervening
                                    ; handler.
    BNEQ NOT_FLT_OVER               ; branch if not floating overflow
    INCL ERROR_COUNT                ; count this floating overflow
    CMPL ERROR_COUNT, #15           ; exceeded maximum limit yet?
    BGEQ RESIGNAL                   ; branch if it has
    INSV     #STS$K_ERROR, -        ; severity code of ERROR
             #STS$V_SEVERITY, -     ; position of severity field
             #STS$S_SEVERITY, -     ; size of severity field
             CHF$L_SIG_NAME(R1)     ; change severity field of
                                    ; signaled condition
    MOVL      SS$_RESIGNAL, R0      ; R0 = resignal status
    RET                             ; return & resignal SEVERE or ERROR
```

```
;+
; Here if not floating overflow - if reserved operand fault, fixup and
; continue execution; otherwise resignal
;-
NOT_FLT_OVER:
    CALLG (AP), LIB$FIXUP_FLT       ; pass signal & mech args along
                                    ; if floating reserved operand,
                                    ; fixup & return R0 = SS$_CONTINUE
                                    ; otherwise R0 = error code so
                                    ; return
    RET
```


6.10.3 Convert any Signal to a Return Status

LIB$SIG_TO_RET converts any signaled condition to a function value to be returned to the caller of the user procedure containing LIB$SIG_TO_RET. It may be established as or called from a condition handler. LIB$SIG_TO_RET is called with the argument list passed to a condition handler by the condition handling facility. The signaled condition is converted into a return to the program that called the procedure that established the handler. The stack is unwound to the caller of the establisher and the condition code is returned as the value in R0.

Format

ret-status = LIB$SIG_TO_RET (sig-args-adr, mch-args-adr)

sig-args-adr
: Address of the signal arguments vector.

mch-args-adr
: Address of mechanism arguments vector.

Return Status

SS$_NORMAL

Procedure successfully completed; SS$_UNWIND completed. Otherwise, the error code from SS$_UNWIND is returned.

Notes

LIB$SIG_TO_RET causes the stack to be marked to be unwound as far back as the caller of the procedure that established the handler which was called on this signal.

Example

This FORTRAN example shows a matrix inversion procedure that uses the integer function INVERT to indicate success or failure. Thus, if the matrix can be inverted, the logical value returned is .TRUE. If, however, the matrix is singular (causing a division by zero) or any other error occurs, the standard system condition value is returned.

```
INTEGER*4 FUNCTION INVERT (A,N)
DIMENSION A (N,N)
EXTERNAL LIB$SIG_TO_RET
CALL LIB$ESTABLISH (LIB$SIG_TO_RET)    ! Establish handler
INVERT = .TRUE.                        ! Assume success
.
. (Invert the matrix with no checks for divide by zero)
.
RETURN
END
```

If an exception occurs during the execution of INVERT, the condition handler (LIB$SIG_TO_RET) is called. The handler copies the condition value being signaled to the image of R0 in the mechanism vector (CHF$L_MCH_SAVR0). Then it calls the system service SYS$UNWIND with defaults set so that the stack is unwound to the caller of INVERT with the error condition value in R0. The caller of INVERT can check for success or failure by an IF test on the returned value. Thus:

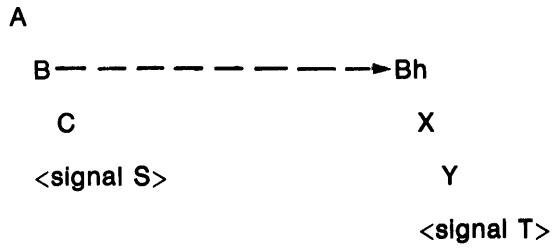
```
IF (.NOT. INVERT (ARRAY, 100)) THEN GO TO error
```

6.11 Multiple Active Signals

A signal is said to be active until the signaler regains control or the stack is unwound. A signal can occur while a condition handler or a procedure it has called is executing. Consider the following example. For each procedure (A, B, C, ...), let the condition handler it establishes be (Ah, Bh, Ch, ...). If A calls B calls C which signals "S" and Ch resignals, then Bh gets control. If Bh calls X calls Y which signals "T", the stack is:

```
<signal T>   :top of stack
    Y
    X
    Bh
<signal S>
    C
    B
    A
```

which was programmed:



The desired order to search for handlers is Yh, Xh, <Bh>h, Ah. Note that Ch should not be called because it is a structural descendant of B. Bh should not be called again because that would require it to be recursive. If it were recursive, then handlers could not be coded in nonrecursive languages such

as FORTRAN. Instead, Bh can establish itself or another procedure as its handler (Bhh).

To implement this, the following algorithm is used. The primary and secondary exception vectors are checked. Then, however, the search backward in the process stack is modified. In effect, the stack frames traversed in the first search are skipped over in the second search. Thus, the stack frame preceding the first condition handler up to and including the frame of the procedure that has established the handler is skipped. Despite this skipping, depth is not incremented. The stack frames traversed in the first and second search are skipped over in a third search, etc. Note that if a condition handler SIGNALs, it will not automatically be invoked recursively. However, if a handler itself establishes a handler, this second handler will be invoked. Thus, a recursive condition handler should start by establishing itself. Any procedures invoked by the handler are treated in the normal way; that is, exception signaling follows the stack up to the condition handler.

For proper hierarchical operation, an exception occurring during execution of a condition handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. This is the vectored condition handler's responsibility. It is most easily accomplished by the vectored handler establishing a catch-all handler.

The following FORTRAN procedure asks the user for a file name and opens that file on the logical unit passed as a parameter. If any kind of OPEN error occurs, the usual FORTRAN, RMS, and VAX/VMS error messages are printed, but execution continues and the user is asked again for a file name. Recovery from a fatal error is achieved by a handler, which signals the error again with LIB$SIGNAL and severity changed to INFO so that execution will continue and no traceback will occur.

```
      SUBROUTINE FILE_OPEN (UNIT)
      EXTERNAL DO_OPEN
      INTEGER*4 DO_OPEN, UNIT
10    IF .NOT. (DO_OPEN (UNIT)) THEN GO TO 10
      RETURN
      END
C     PROCEDURE TO DO OPEN
      INTEGER*4 FUNCTION DO_OPEN (UNIT)
      EXTERNAL HANDLE_OPEN
      INTEGER*4 UNIT
      CHARACTER*15 FILE
      CALL LIB$ESTABLISH (HANDLE_OPEN)
      DO_OPEN = 1              ! Assume success
100   TYPE *, 'Type File Name'
      ACCEPT *, FILE
      OPEN (UNIT=UNIT, TYPE='OLD' NAME=FILE)
      RETURN          ! success unless handler is called
      END

C     HANDLER FOR OPEN ERRORS
      INTEGER*4 FUNCTION HANDLE_OPEN (SIGARGS, MECHARGS)
      INTEGER*4 SIGARGS(9), MECHARGS(5)
      CALL LIB$INSV (3, 0, 3, SIGARGS(2)) ! Set severity to INFO
      CALL LIB$SIGNAL (%VAL(7), ! 7 following longwords
```

```
1     %VAL(SIGARGS(2)),  ! Signaled condition value
2     %VAL(3),           ! # of following FAO args, assume 3
3     %VAL(SIGARGS(4)),  ! Unit number
4     %VAL(SIGARGS(5)),  ! Adr of resultant file descr
5     %VAL(SIGARGS(7)),  ! User PC
6     %VAL(SIGARGS(3)+4), ! RMS STS no matter how many FAO args
7     %VAL(SIGARGS(3)+5)) ! RMS STV no matter how many FAO args
MECHARGS(4) = 0          ! Image of R0 set to indicate error
CALL SYS$UNWIND(,)       ! Set to unwind
                         ! last call made by establisher
RETURN                   ! Resignal
END
```

If an error occurs in the OPEN statement, then the condition handler HANDLE__OPEN is called, which calls LIB$SIGNAL with the same signal argument list except: (1) PC and PSL are omitted from the end, and (2) the severity is changed to WARNING (0). The FAO arg count is assumed to be three, although the count could be larger. The RMS STS and STV are obtained in a manner independent of the actual number of FAO arguments, which could be larger than three for some conditions in the future (see Section 6.7.1). Then the handler sets the image of R0 to a failure code and unwinds to the caller of the establisher, namely to FILE__OPEN, which tests function value of DO__OPEN; finding it .FALSE., the handler loops back and recalls DO__OPEN.

Chapter 7
Syntax Analysis Procedures

This chapter describes the use of procedures that perform string syntax analysis, and pass complex instructions in a computer language such as command languages. Table 7–1 contains the names and titles of the syntax analysis procedures.

Table 7–1: String Syntax Procedures

Section	Entry Point Name	Title
7.1	LIB$TPARSE	A Table-driven Finite-state Parser
7.12	LIB$LOOKUP_KEY	Scan Keyword Table

Chapter 3 contains procedures for manipulating strings; Chapter 5 contains procedures for writing and allocating dynamic strings.

This chapter describes LIB$TPARSE from an assembly language, or BLISS viewpoint. LIB$TPARSE can also be called from programs written in FORTRAN. However, the LIB$TPARSE state tables must be generated with a set of assembler or BLISS macros. Appendix G contains sample programs in MACRO and BLISS using LIB$TPARSE.

7.1 LIB$TPARSE — A Table-Driven Finite-State Parser

LIB$TPARSE is a general purpose table-driven parser. It is implemented as a finite-state automaton, with extensions that make it suitable for a wide range of applications, including command lines, most programming languages, and commands for special purpose utilities. TPARSE has built-in features to allow convenient implementation of commonly used command grammars; and the flexibility to handle special problems.

7.2 Fundamentals of a Finite-State Parser

This section presents the basic principles of a finite-state, table-driven parser.

A finite-state machine is a processor consisting of a set of states. The total memory available to the processor is the knowledge of which state it is in currently. (As an example, think of a computer with its program in read-only memory and its program counter in the only writable storage.) A string of symbols is input to this processor. Only the first symbol in the string is visible to the machine. For each state, there is a list of particular symbols that can be accepted in that state. Each symbol accepted by a state causes the machine to enter some other state. As the state transition is made, the symbol that caused the transition is removed from the front of the input string.

The symbol in the input string which is recognized by a single state transition is generally referred to as a *token*. A token can consist of one or more characters. The machine runs through a sequence of state transitions as it processes the consecutive tokens of the input string.

The complete list of symbols that appear in the state transition lists of the machine is called the machine's *alphabet*. The machine will recognize a subset of all possible strings that could be generated from the alphabet. Certain input strings will cause the machine to enter a state whose list of acceptable symbols does not include the next token in the string. Such strings are not accepted by the machine. A string is accepted by the machine if, in processing the string, the machine enters a state designated as a final state. A final state causes the machine to halt. Any portion of the input string that has not been processed already remains ignored.

A finite-state machine can be used to check if a string of characters constitutes a valid input in a language (such as the command language for a utility program). LIB$TPARSE is a general purpose finite-state machine simulator. A program uses LIB$TPARSE by calling it with the string to be analyzed and a tabular description of the finite-state machine (called a state table). LIB$TPARSE reads the string, executes the state transitions of the machine, and returns a status indicating whether the machine halted in a final state or not. A string not accepted by the machine is said to contain a syntax error. The location of the syntax error is the position in the string at which the machine halted.

LIB$TPARSE checks a string for valid syntax. LIB$TPARSE lets its caller extract the meaning of a string (the semantics, as opposed to the syntax) by calling an optional user-written action routine each time it makes a state transition.

Action routines link semantics with the syntax defined by the state transitions in a state table. They are also useful for providing additional memory and computational ability that otherwise are not available to the basic finite-state machine. A different action routine can be called for each state transition in the state table. LIB$TPARSE makes available to the action routine additional information that can be useful in determining the meaning of the state

transition. This includes the characters and the position in the input string of the current token. The action routine can use whatever global data base the user wants to provide.

7.3 The Alphabet of LIB$TPARSE

LIB$TPARSE provides an alphabet of symbols that can be used in constructing state tables. This includes all of the basic building blocks needed for constructing a grammar using the ASCII character set. There are also symbols that represent the more complex constructions found in programming and command language grammar.

This section describes the types of symbols that can be recognized by LIB$TPARSE. It also describes how each symbol is represented in a state table. The complete set of macro calls used to construct a state table is described in a Section 7.4.

7.3.1 'x' – Any Particular Character

'x' matches the particular ASCII character. In a state table, it is expressed by enclosing the character in single quotation marks. The character can be any member of the 8-bit ASCII code set. Note that this symbol type matches the exact code only. Uppercase and lowercase alphabetics, and codes with bit 7 different are not equivalenced.

7.3.2 TPA$_ANY – Any Single Character

TPA$_ANY matches any single character. (The actual matching character is available to the action routine.) In a state table, it is expressed as the symbolic name TPA$_ANY.

7.3.3 TPA$_ALPHA – Any Alphabetic Character

TPA$_ALPHA matches any character in the English alphabet, that is, uppercase and lowercase A through Z.

7.3.4 TPA$_DIGIT – Any Numeric Character

TPA$_DIGIT matches any numeric character, that is, 0 through 9.

7.3.5 TPA$_STRING – Any Alphanumeric String

TPA$_STRING matches any string of one or more alphanumeric characters, that is, uppercase or lowercase A through Z, and 0 through 9. The string can be any length; it is bounded on the right by the first non-alphanumeric character seen in the input string (or by the end of the string). A descriptor of the matching string is available to the action routine.

7.3.6 TPA$_SYMBOL - Any Symbol Constituent String

TPA$_SYMBOL matches any string of one or more characters of the standard VAX-11 symbol constituent set, that is, uppercase and lowercase A through Z, 0 through 9, the dollar sign ($), and the underscore (_). The string must be bounded on the right by some character not in the symbol constituent set (or by the end of the string).

7.3.7 TPA$_BLANK - Any Blank String

TPA$_BLANK matches any string of one or more blanks and/or tabs.

7.3.8 TPA$_DECIMAL - Any Decimal Number

TPA$_DECIMAL matches any decimal number (that is, any string of one or more digits 0 through 9) whose magnitude is less than 2**32. The binary value of the number, converted in decimal radix, is available to the action routine.

7.3.9 TPA$_OCTAL - Any Octal Number

TPA$_OCTAL matches any octal number (that is, any string of one or more digits 0 through 7) whose magnitude is less than 2**32. The binary value of the number, converted in octal radix, is available to the action routine.

7.3.10 TPA$_HEX - Any Hexadecimal Number

TPA$_HEX matches any hexadecimal number (that is, any string of one or more digits 0 through 9, A through F) whose magnitude is less than 2**32. The binary value of the number, converted in hexadecimal radix, is available to the action routine.

7.3.11 'keyword' - A Particular Keyword String

'keyword' matches the string of characters enclosed in single quotes. A keyword can consist of one or more characters of the VAX-11 symbol constituent set. Note, uppercase and lowercase alphabetics are treated as different characters. Programs that want to treat uppercase and lowercase as equivalent should code keywords in state tables in uppercase and capitalize the input string before calling LIB$TPARSE. (See Section 3.3.5.1: LIB$MOVTC, for a description of character translation tables. See also Section 3.3.5.6: STR$UPCASE, for a description of a routine to translate lowercase to uppercase.)

A state table can contain up to 220 keywords. The keyword, as it appears in the string being parsed, must be bounded on the right by a character not in the symbol constituent set (or by the end of the string). At the caller's option, keywords appearing in the string being parsed can be abbreviated. (A full description of the abbreviation facility appears in Section 7.9.)

Keywords that are one character in length are expressed in the form 'x*' to distinguish them from the single-character symbol ('x'). They must be differentiated since they are not the same in operation. For example, in the input string AB+C, the single character 'A' would match the first character of this string, whereas the keyword 'A*' would not, since B in the string is in the symbol constituent set.

7.3.12 TPA$_LAMBDA – The Empty String

TPA$_LAMBDA matches the empty string (and therefore always matches). As the transition is taken, no characters are removed from the input string. LAMBDA transitions are useful in getting action routines called under otherwise awkward circumstances, providing unconditional GOTOs to link portions of a state table together, and providing default actions in certain cases.

7.3.13 TPA$_EOS – End of Input String

TPA$_EOS matches the end of the input string. That is, a transition naming the TPA$_EOS symbol is taken if the entire input string has been processed.

7.3.14 !label – Complex Subexpression

!label matches any string that is matched by entering the state table at the indicated label and executing state transitions until a final state is entered. Roughly, this corresponds to calling a subroutine in the state table. If the state table subroutine fails (that is, if it encounters a syntax error in the input string), the input string is backed up to the point at which the subroutine started, and the subexpression simply fails to match. The subexpression facility permits complex syntactic constructs that appear in many places in a grammar to appear only once in the state table. It also permits a degree of non-deterministic and/or push down parsing with a parser that is otherwise deterministic and finite-state. Subexpressions are described in more detail in Section 7.10.

7.4 Coding a State Table in MACRO

A set of assembler macros is available from the VAX/VMS system macro library to allow convenient and readable coding of a LIB$TPARSE state table. Macros exist to initialize the LIB$TPARSE macro system, define the states in the state table, and define the transitions to other states within each state. These macros generate symbol definitions and tables; they do not produce any executable code or routine calls.

7.4.1 $INIT_STATE – Initialize the TPARSE Macros

The $INIT_STATE macro declares the beginning of a state table. It initializes the internals of the table generator macros and declares the locations of the state table and the keyword table. The state table is the structure containing the definitions of the states and the transitions between them. The keyword table contains the text of the keywords used in the state table.

Format

$INIT_STATE state-table,key-table

state-table
The name assigned to the state table. This label is equated to the start of the first state in the state table.

key-table
The name assigned to the keyword table. This label is equated to the start of the keyword table.

Both the address of the state table and the address of the keyword table must be supplied in the call to LIB$TPARSE to perform a parse. The $INIT_STATE macro can appear multiple times in a program. Each occurrence defines a separate state table; no part of any state table can make reference to part of any other state table.

7.4.2 $STATE – Define a State

The $STATE macro declares the beginning of a state.

Format

$STATE [label]

label
An optional label for the state. If present, the label is equated to the starting address of the state.

7.4.3 $TRAN – Define a State Transition

The $TRAN macro defines a transition from the state in which it appears to some other (or even the same) state. The parameters of the macro define, among other things, the symbol type that causes the transition to be taken, the state to transfer to, and the action routine to call, if any.

Format

$TRAN type[,label][,action][,mask][,msk-adr][,parameter]

type
The symbol type recognized by this transition. The transition is taken if the characters at the front of the input string match the symbol specified. The symbol can be any of the constructs discussed in Section 7.3.

The assembler will not permit all characters to be entered in the 'x' format (such as single quote and all of the control characters). Such characters can be specified as the symbol type with any assembler expression that

evaluates to the ASCII code of the desired character, not including the single quotes. For example, a transition to match a backspace character could be coded as:

```
BACKSPACE = 8
        :
        :
$TRAN BACKSPACE, .....
```

label

The optional target state of this transition. If present, it must be the label assigned to some state in the state table. If no label is present in the transition, control is transferred to the next state immediately following in the state table. If the label is the expression TPA$_EXIT, it denotes a transition to the final state. A transition to TPA$_EXIT terminates the parsing operation in progress. If the label is the expression TPA$_FAIL, the parsing operation is terminated with a failure status as if a syntax error had occurred.

action

The optional address of a user-supplied action routine. If this parameter is present, the named action routine is called before the transition is taken. The calling sequence of action routines and the information available to them is described in Section 7.6.

mask

An optional 32-bit mask value used with the msk-adr parameter. If the mask is present, its value is inclusive ORed into the longword specified by msk-adr. Use of the mask parameter allows the state table to flag the fact that a certain transition was taken without the expense and overhead of calling an action routine.

msk-adr

The optional address associated with the preceding mask parameter. This parameter specifies the address into which the mask is to be ORed. If the mask parameter is present, the msk-adr parameter must also be present.

The msk-adr parameter can also be present without the preceding mask parameter. In this case it is used to specify an address into which information about the matching token is stored. The information stored depends on the nature of the symbol.

If the symbol is a number (that is, if the type code in the transition is TPA$_DECIMAL, TPA$_OCTAL, or TPA$_HEX), the 32-bit binary value of the number is stored at the address (an unsigned longword).

If the symbol is a single character (that is, if the type code in the transition is 'x', TPA$_ANY, TPA$_ALPHA, or TPA$_DIGIT) the eight-bit matching character is stored at the address (an unsigned byte).

If the symbol is of any other type, the 64-bit string descriptor of the matching token is stored at the address (an unsigned quadword; class and data type fields in descriptor are undefined).

The use of the msk-adr alone lets a parser program extract the most commonly needed information from the input string without the use of action routines. Note that the information is stored, not ORed as is the preceding mask.

parameter
An optional 32-bit parameter which, if specified, is made available to the action routine. This parameter can be an identifier number, an address, or anything else that a user written action routine might find useful. It allows a single action routine to serve many transitions for which similar, but slightly varying, actions must be performed. Note that the parameter appears in the state table in its absolute form; if it is used as an address, the resulting parsing program containing this state table will not be PIC.

7.4.4 $END_STATE - End the State Table

The $END_STATE macro declares the end of the state table. Its presence is mandatory to permit the orderly cleanup of the TPARSE macro system. The $END_STATE macro has no arguments. It is coded as:

```
$END_STATE
```

7.5 Coding a State Table in BLISS

A set of BLISS macros is available in the file SYS$LIBRARY:TPAMAC.L32 to allow convenient and readable coding of TPARSE state tables in BLISS. The macros are made available to the program by including the declaration:

```
LIBRARY 'SYS$LIBRARY:TPAMAC';
```

in the module containing the state tables. The names and functions of the macros are the same as those provided for MACRO; the following sections detail the syntactic differences.

7.5.1 $INIT_STATE - Initialize the TPARSE Macros

The $INIT_STATE macro initializes the TPARSE macro system in the same manner as it does for the assembler.

Format

```
$INIT_STATE (state-table, key-table);
```

state-table
The name assigned to the state table. This label is equated to the start of the first state in the state table.

key-table
The name assigned to the keyword table. This label is equated to the start of the keyword table.

Both names are declared as global vectors of length zero. As with the assembler macros, $INIT_STATE can be invoked multiple times to declare multiple state tables within a single module.

7.5.2 $STATE – Declare a State

The $STATE macro is used in BLISS to declare a state in its entirety.

Format

```
$STATE ([label],
        ( transition ),
        ( transition ),
              :
              :
         ( transition )
         );
```

label
: Optional address of the start of the state. It is declared as a local vector of length zero. Note that the comma following the optional label is mandatory.

transition
: Each transition appears within the parentheses in the same form as the transition parameter list for the assembler $TRAN macro:

  type[,label][,action][,mask][,msk-adr][,parameter]

  The individual parameters of each transition are expressed in exactly the same format as in the assembler macros. The one exception to this is the subexpression type, expressed as !label in the assembler macros. In the BLISS macros, this type is coded in the form (label).

As in MACRO, not all characters can be included in quoted strings in BLISS. To build a transition matching such a single character, you can use the %CHAR lexical function as follows:

```
LITERAL BACKSPACE = 8;
          :
          :
$STATE (label,
        (%CHAR (BACKSPACE), ..... )
        );
```

7.5.3 $TRAN and $END_STATE

There are no $TRAN or $END_STATE macros in the BLISS macro system. The former is absorbed into the $STATE macro; the latter is not needed.

7.5.4 BLISS Coding Considerations

The BLISS TPARSE table generator macros interact with the BLISS module environment in some ways that require explanation. To allow references between $STATE macros, no BEGIN or END statements are used. However, the macros do use PSECT declarations; all storage is generated with OWN declarations. Thus, if a state table appears at the front of a module with other module data declarations, the PSECT declarations for OWN and GLOBAL are modified coming out of the TPARSE macros. They cannot be surrounded

with BEGIN and END statements, since this would constitute an expression; no declarations (in particular, no ROUTINE declarations) can follow any expression. There are four acceptable techniques of including TPARSE state tables in BLISS modules:

1. Following the state table with explicit redeclarations of the OWN and GLOBAL PSECTs
2. Confining the state table within a separate module
3. Placing the state table within BEGIN and END statements after the declarations within a routine body
4. Placing the state table within BEGIN and END statements at the end of a module

In all cases, of course, all action routines, masks, addresses, and parameters must be defined with suitable declarations (which can be FORWARD or EXTERNAL). The TPARSE macros handle the necessary FORWARD declarations for forward references to labels in the state table.

7.6 Calling LIB$TPARSE

LIB$TPARSE is called giving the address of a parameter block, the address of the state table, and the address of the keyword table. The input string is specified by part of the parameter block. LIB$TPARSE reads the input string, interprets the transitions in the state table, and calls the action routines until:

1. A transition to TPA$_EXIT or TPA$_FAIL is executed at main level (that is, while LIB$TPARSE is not processing a subexpression call).
2. An error occurs at main level. The error can be either a syntax error, in which case all of the transitions in the current state fail to match the current input string, or a state table format error.

Format

ret-status = LIB$TPARSE (param-blk, state-table, key-table)

param-blk
: Address of the LIB$TPARSE parameter block. This block contains information about the state of the parse operation. It becomes the argument list presented to all action routines. The contents of the parameter block are detailed below.

state-table
: Address of the starting state in the state table. Usually, the name appearing as the first parameter of the $INIT_STATE macro is used.

key-table
: Address of the keyword table. The name appearing as the second parameter of the $INIT_STATE macro must be supplied.

Return Status

SS$_NORMAL
Procedure successfully completed. LIB$TPARSE has executed a transition to TPA$_EXIT at main level (not within a subexpression).

LIB$_SYNTAXERR
Parse completed with syntax error. LIB$TPARSE has encountered a state at main level in which none of the transitions match the input string, or a transition to TPA$_FAIL was executed.

LIB$_INVTYPE
State table error. LIB$TPARSE has encountered an invalid entry in the state table.

other
If an action routine returns a failure status other than zero, and the parse consequently fails, LIB$TPARSE returns the status returned by the action routine.

Note that LIB$TPARSE generates no signals and establishes no condition handler; user-written action routines can signal through LIB$TPARSE back to the calling program.

7.6.1 The LIB$TPARSE Parameter Block

The parameter block is the impure data base upon which LIB$TPARSE operates. It contains the descriptor of the string being parsed and option flags for LIB$TPARSE: It also contains the data about the current token that is available to action routines. When an action routine is called, the parameter block becomes the argument list of the action routine, allowing efficient and ready reference by the routine.

The fields in the parameter block have symbolic names. Assembly language programs can define these names by invoking the macro $TPADEF (automatically loaded from the system macro library). The field names define the byte offset of the field from the start of the block, with the exception of the bit fields ($V_names), which are defined as bit offsets from the start of the containing field. In addition, bitmask values ($M_names) are available for the bit fields.

The same field names are available to BLISS programs from the system macro library SYS$LIBRARY:STARLET.L32. Each name (except for the $M_names) is defined as a fixed reference macro that operates on a byte-based block. The $M_names are defined as literals.

The parameter block contains the following fields:

TPA$L_COUNT — A longword containing the number of longwords that make up the rest of the parameter block. This longword functions as the argument count when the parameter block becomes the argument list to an

	action routine. This field must contain the value TPA$K_COUNT0 (whose numeric value is 8).
TPA$L_OPTIONS	A longword containing various option and flag bits.
TPA$V_BLANKS	Setting this bit causes LIB$TPARSE to process blanks and tabs explicitly, rather than treating them as invisible separators (see Section 7.8 on blank processing).
TPA$V_ABBRFM	Setting this bit causes LIB$TPARSE to allow the abbreviation of keywords to any length. If an abbreviated keyword string is ambiguous, it is matched by the first eligible transition listed in the state.
TPA$V_ABBREV	Setting this bit causes LIB$TPARSE to allow the abbreviation of keywords in the input string to the shortest length that is unambiguous in that state (see Section 7.7 on keyword abbreviation).
TPA$V_AMBIG	This bit is set by LIB$TPARSE when an ambiguous keyword string has been detected in the current state.
TPA$B_MCOUNT	This byte, when non-zero, contains the minimum number of characters that keywords can be abbreviated to. Preventing ambiguity is the responsibility of the state table designer. If TPA$V_ABBRFM or TPA$V_ABBREV is set, this value is ignored.
TPA$M_BLANKS TPA$M_ABBRFM TPA$M_ABBREV TPA$M_AMBIG	These names define bitmasks that correspond to the location of the corresponding $V_ fields in the options longword.
TPA$L_STRINGCNT	A longword containing the number of characters remaining in the parser input string.
TPA$L_STRINGPTR	A longword containing the address of the remainder of the string being parsed. Together with TPA$L_STRINGCNT, a descriptor of the input string is formed. The caller initializes this descriptor with the string to be parsed. When an action routine is called, this descriptor describes the remainder of the input string. When LIB$TPARSE returns, this descriptor describes the portion of the input string that was not processed. (This occurs whether TPARSE returns success or failure.)

The following elements of the parameter block are primarily of use to action routines called by LIB$TPARSE:

TPA$L_TOKENCNT	A longword containing the number of characters in the current token.
TPA$L_TOKENPTR	A longword containing the address of the current token. Together with TPA$L_TOKENCNT, a descriptor of the current token string is formed. The current token string is the set of characters of the input string that are being matched by the transition currently being taken. If TPARSE encounters a syntax error (fails to match a transition), then this descriptor describes whatever portion of the current input string would have been matched by a TPA$_SYMBOL symbol type; if none would have matched, it describes the first remaining character in the input string. A transition to TPA$_FAIL leaves the descriptor describing the token matched by that transition.
TPA$B_CHAR	A byte containing the character matched by a single character symbol type ('x', TPA$_ANY, TPA$_ALPHA, or TPA$_DIGIT). The remainder of the longword is not used.
TPA$L_NUMBER	A longword containing the binary value of a numeric token (TPA$_DECIMAL, TPA$_OCTAL, or TPA$_HEX), converted in the appropriate radix.
TPA$L_PARAM	A longword containing the 32-bit parameter supplied by the state transition.

The three preceding fields (TPAL_CHAR, TPAL_NUMBER, and TPA$L_PARAM) are only modified when an action routine is about to be called from a transition of the relevant type (or containing an explicit parameter). While transitions of unrelated types are executed, the fields are not modified.

TPA$K_LENGTH0	This symbol represents the number of bytes in the basic LIB$TPARSE parameter block. A parameter block of at least this length (containing a count field of TPA$K_COUNT0 in TPA$L_COUNT) must be presented to TPARSE as the first argument.

7.6.2 Interface to TPARSE Action Routines

User-supplied action routines are called by LIB$TPARSE using a CALL instruction. When an action routine is specified by a state transition, the action

routine is called when the transition is found to be able to execute successfully (that is, when its symbol type matches a leading portion of the input string). The action routine is called before the mask and/or msk-adr parameters of the state transition have been processed.

The argument list presented to the action routine is the LIB$TPARSE parameter block. This allows an action routine written in assembly language, for example, to reference fields in the parameter block by their symbolic offsets relative to the AP register.

The action routine returns a value to LIB$TPARSE in R0 that controls execution of the state transition currently being processed. If the action routine returns success (low bit set in R0) then LIB$TPARSE proceeds with the execution of the state transition. If the action routine returns failure (low bit clear in R0), LIB$TPARSE rejects the transition that was being processed and acts as if the symbol type of that transition had not matched. It proceeds to evaluate other transitions in that state for eligibility. In keeping with efficient design, LIB$TPARSE calls action routines with R0 set to one, allowing most action routines to return success by simply not modifying R0.

If an action routine returns a non-zero failure status to TPARSE and no subsequent transitions in that state match, TPARSE will return the status of the action routine, rather than the status LIB$__SYNTAXERR.

The mechanism of allowing action routines to reject a state transition provides a powerful facility for implementing symbol types specific to a particular application. To recognize a specialized symbol type, the state table designer codes a state transition using a LIB$TPARSE symbol type that describes a superset of the set of possible tokens that is desired. The associated action routine then performs the additional discrimination necessary and returns success or failure to LIB$TPARSE, which then accordingly executes or fails the transition. Simple examples of symbol type discrimination that are cumbersome using a pure finite-state machine include recognizing only strings that are shorter than some maximum length, or accepting numeric values confined to some particular range.

7.7 LIB$TPARSE State Table Processing

In a theoretical finite-state machine, when a state is entered, the symbol types given by all of the transitions out of that state are compared simultaneously with the front of the input string. The one transition whose symbol type matches is then taken. Since LIB$TPARSE is executed by an ordinary sequential computer, the evaluation of a LIB$TPARSE state table differs somewhat from the theoretical model. Note also that the set of symbol types implemented by LIB$TPARSE matches overlapping sets of tokens. For example, the token 123 could be matched by TPA$__DECIMAL, TPA$__OCTAL, TPA$__STRING, or one of several others.

In a LIB$TPARSE state table, each state consists of a list of the transitions to other states. The transitions appear in the order in which they were written in the source program. LIB$TPARSE evaluates the transitions in the order in

which they appear in the state. For each transition, it tests whether the symbol type specified matches the leftmost portion of the input string. If it does not match, it proceeds to attempt to match the next transition, until it runs out of transitions in the state. If a transition matches, LIB$TPARSE stores the optional parameter longword, if any, into the parameter block and calls the action routine. If the action routine returns failure, LIB$TPARSE continues attempting to match successive transitions. If the action routine returns success (or if no action routine was specified), LIB$TPARSE executes the transition. The mask or other value is stored at the mask address, if specified, and control passes to the specified target state. If no target state is given, control passes to the next state following in the state table. In either case, the remaining transitions in the state are not evaluated.

What this means is that where there are multiple transitions out of a state whose symbol types match overlapping sets of tokens, they must be carefully ordered. For example, all keyword strings are matched by the TPA$_SYMBOL symbol type; keyword transitions appearing in a state following a TPA$_SYMBOL transition will in general never be executed. A good rule of thumb is to order transitions of different types in order of increasing generality, as follows:

'keyword'
'x'
TPA$_EOS
TPA$_ALPHA
TPA$_DIGIT
TPA$_BLANK
TPA$_OCTAL
TPA$_DECIMAL
TPA$_HEX
TPA$_STRING
TPA$_SYMBOL
TPA$_ANY
TPA$_LAMBDA

Note that subexpressions are not in this list; their placement depends on the symbol types recognized within the subexpression. Also note that the use of transition rejection can alter the generality of a symbol type and affect its placement in the preceding order. However, the first transition listed in a state that is permitted to match the leftmost portion of the input string is the one that will be executed.

7.8 Blanks In the Input String

The default mode of operation in LIB$TPARSE is to treat blanks as invisible separators. That is, they can appear between any two tokens in the string

being parsed without being called for by transitions in the state table. Since situations in which blanks are significant exist, LIB$TPARSE enables the explicit processing of blanks if the bit TPA$V_BLANKS is set in the options longword of the parameter block. The following input string illustrates the difference in operation:

```
ABC  DEF
```

The string is recognized by the following sequences of state transitions, depending on the state of the blanks control flag:

TPA$V_BLANKS set		TPA$V_BLANKS clear	
$STATE			
$TRAN	TPA$_STRING	$STATE	
		$TRAN	TPA$_STRING
$STATE			
$TRAN	TPA$_BLANK	$STATE	
		$TRAN	TPA$_STRING
$STATE			
$TRAN	TPA$_STRING		

The action routines in a parsing program can set or clear the blanks control flag as sections of the state table in which blanks are significant are entered and left. LIB$TPARSE always checks the blanks control flag as it enters a state; if the flag is clear it removes any space or tab characters present at the front of the input string before it proceeds to evaluate transitions. Note that when the TPA$V_BLANKS flag is clear, the TPA$_BLANK symbol type will never match.

7.9 Abbreviating Keywords

Many languages (command languages in particular) allow their keywords to be abbreviated. LIB$TPARSE has three abbreviation facilities to permit the recognition of abbreviated keywords when only the full spellings are listed in the state table.

The default mode of LIB$TPARSE is exact match. All keywords in the input string must exactly match their spelling and length in the state table.

By setting a value in TPA$B_MCOUNT in the LIB$TPARSE parameter block, the calling program (or action routine) specifies that all keywords can be abbreviated to the number of characters given. For example, setting the byte to the value four would allow the keyword DEASSIGN to appear in an input string as DEAS (or DEASS or DEASSI ...). All characters of the keyword strings in the input string are checked; incorrect spellings beyond the minimum abbreviation are not permitted.

If TPA$V_ABBRFM is set in the options longword (by caller or action routine), LIB$TPARSE will recognize any leftmost substring of a keyword as a match for that keyword. No check is made for ambiguity; LIB$TPARSE will match the first keyword listed in the state table of which the input token is a subset.

If TPA$V_ABBREV is set in the options longword (by the caller or action routine), TPARSE will permit any abbreviation of a keyword to be recognized as long as it is unambiguous among the keywords in that state. If LIB$TPARSE finds that the front of the input string contains an ambiguous keyword string, it sets the bit TPA$V_AMBIG in the options longword and refuses to recognize any keyword transitions in that state (other symbol types are still accepted). The TPA$V_AMBIG flag can be checked by an action routine called coming out of that state, or by the calling program should TPARSE return with a syntax error status. The flag is cleared upon entering the next state.

Proper recognition of ambiguous keywords requires that the keywords in each state be arranged in alphabetical order by an ASCII collating sequence. The sequence runs:

1. $
2. numerics
3. uppercase alphabetics
4. _
5. lowercase alphabetics

Use of this feature must be made with some care, since permitting minimal abbreviation tends to restrict the extensibility of a language. Often, adding a new keyword can make a formerly valid abbreviation ambiguous.

If both TPA$V_ABBRFM and TPA$V_ABBREV are set, then TPA$V_ABBRFM takes precedence.

7.10 Using Subexpressions

LIB$TPARSE subexpressions are analogous to subroutines within the state table. A subexpression call, indicated with the MACRO expression !label or the BLISS expression (label), causes LIB$TPARSE to call itself recursively,

using: (1) the same parameter block and keyword table, and (2) the specified label as a starting state. LIB$TPARSE processes the state transitions, consuming the portion of the input string called for. When a transition to TPA$__EXIT is executed, LIB$TPARSE returns success to itself. The subexpression call is thus considered to match, the action routine is called, and the transition is taken. If the parse of the subexpression fails; LIB$TPARSE backs up the input string to where it was prior to the call and proceeds to evaluate the remaining transitions in the state.

Subexpressions are a very powerful and useful mechanism. They are usable in the same way one would use subroutines in any program: to avoid replication of complex expressions. They can also be used in a limited form of push down parsing, in which the state table contains recursively nested subexpressions. Finally, subexpressions can be used for non-deterministic parsing, that is, parsing where some number of states of look-ahead is needed. This is done by placing each path of look-ahead in a separate subexpression and calling the subexpressions in the transitions of the state that needs the look-ahead. When a look-ahead path fails, the subexpression failure mechanism causes LIB$TPARSE to back out and try another one.

Some care must be exercised in the design of subexpressions which contain calls to action routines or use the mask and msk-adr transition parameters. As the state transitions of a subexpression are processed, the specified action routines are called and the mask and msk-adr stores are performed. Should the subexpression fail, LIB$TPARSE will back up the input string and resume processing in the calling state. However, any effects that the action routines have had on the caller's data base cannot be undone. If subexpressions are simply being used as state table subroutines, this tends to be harmless, since in this mode of operation, when a subexpression fails, the parse will generally fail. This is not true of push down or non-deterministic parsing. In applications where subexpressions are expected to fail, action routines should be designed to store results in temporary storage. These results can then be made permanent at the main level, where the flow of control is deterministic.

Sections 7.10.1 and 7.10.2 show two uses of subexpressions.

7.10.1 Use of Subexpressions and Transition Rejection

The following example is an excerpt of a state table that parses a string quoted by an arbitrary character. The first character to appear is interpreted as a quote character. This sort of construction turns up in many text editors, and in some programming languages. Execution of this set of state transitions leaves a descriptor for the string in the two longwords at Q__DESCRIPTOR, and the quoting character at location Q__CHAR.

```
;
; Main level state table. The first transition accepts and
; stores the quoting character.
;
        $STATE  STRING
        $TRAN   TPA$_ANY,,,,Q_CHAR
;
; Call the subexpression to accept the quoted string and store
; the string descriptor. Note that the descriptor spans all
; the characters accepted by the subexpression.
;
        $STATE
        $TRAN   !Q_STRING,,,,Q_DESCRIPTOR
;
; Accept the trailing quote character, left behind by the
; subexpression
;
        $STATE
        $TRAN   TPA$_ANY,NEXT
;
; Subexpression to scan the quoted string. The first transition
; matches until it is rejected by the action routine.
;
        $STATE  Q_STRING
        $TRAN   TPA$_ANY,Q_STRING,TEST_Q
        $TRAN   TPA$_LAMBDA,TPA$_EXIT
;
; The following MACRO subroutine compares the current character
; with the quoting character and returns failure if it matches.
;
TEST_Q: .WORD   0                       ; null entry mask
        CMPB    TPA$B_CHAR(AP),Q_CHAR   ; check the character
        BNEQ    10$                     ; note R0 is already 1
        CLRL    R0                      ; match - reject transition
10$:    RET
```

7.10.2 Using Subexpressions to Parse Complex Grammars

The following example is an excerpt from a state table that shows how subexpressions are used to parse complex grammars. The state table accepts a number followed by a keyword qualifier. Depending on the keyword, the number is interpreted as either decimal, octal, or hexadecimal. These strings are examples of what is accepted by executing the state table:

```
10/OCTAL
32768/DECIMAL
77AF/HEX
```

This sort of grammar is difficult to parse with a deterministic finite-state machine. Using a subexpression look-ahead of two states permits the state tables to be expressed more simply.

```
;
; Main state table entry. Accept a number of some type and store
; its value at the location NUMBER.
;
        $STATE
        $TRAN    !OCT_NUM,NEXT,,,NUMBER
        $TRAN    !DEC_NUM,NEXT,,,NUMBER
        $TRAN    !HEX_NUM,NEXT,,,NUMBER
;
; Subexpressions to accept an octal number followed by the OCTAL
; qualifier.
;
        $STATE   OCT_NUM
        $TRAN    TPA$_OCTAL
        $STATE
        $TRAN    '/'
        $STATE
        $TRAN    'OCTAL',TPA$_EXIT
;
; Subexpression to accept a decimal number followed by the DECIMAL
; qualifier.
;
        $STATE   DEC_NUM
        $TRAN    TPA$_DECIMAL
        $STATE
        $TRAN    '/'
        $STATE
        $TRAN    'DECIMAL',TPA$_EXIT
;
; Subexpression to accept a hex number followed by the HEX
; qualifier.
;
        $STATE   HEX_NUM
        $TRAN    TPA$_HEX
        $STATE
        $TRAN    '/'
        $STATE
        $TRAN    'HEX',TPA$_EXIT
```

Note that the TPA$__NUMBER longword is not disturbed by the transitions following the numeric token, allowing it to be retrieved by the main level subexpression call.

7.11 State Table Object Representation

This section describes the binary representation of a LIB$TPARSE state table. Each state consists of its transitions concatenated in memory; the state label is equated to the address of the first byte of the first transition. The end of the state is identified by a marker in the last transition. The state table is built by the LIB$TPARSE table macros in the PSECT __LIB$STATE$.

Each transition in a state consists of from 2 to 23 bytes containing the parameters of the transition. Storage is not allocated for parameters not specified in the transition macro. This allows simple transitions to be represented efficiently. For example, the transition:

```
$TRAN    '?'
```

which simply accepts the character ? and falls through to the next state is represented in 2 bytes.

In this section, pointers described as self-relative are signed displacements from the address following the end of the pointer (this is identical to branch displacements in the VAX instruction set).

A state transition consists of the following elements:

- *Symbol Type - One Byte.* The first byte of a transition contains the binary coding of the symbol type accepted by this transition. It is always present. The interpretation of the type byte is controlled by flag bit 0 in the flags byte (described in Section 7.11.2). If the flag is clear, then the type byte represents a single character (the 'x' construct). If the flag bit is set, then the type byte is one of the other type codes (keyword, number, and so forth). The various symbol types accepted by TPARSE are encoded as follows:

'x'	= ASCII code of the character (8 bits)	
'keyword'	= the keyword index (0 up to 219)	
TPA$_ANY	= 237	
TPA$_ALPHA	= 238	
TPA$_DIGIT	= 239	
TPA$_STRING	= 240	
TPA$_SYMBOL	= 241	
TPA$_BLANK	= 242	
TPA$_DECIMAL	= 243	
TPA$_OCTAL	= 244	
TPA$_HEX	= 245	
TPA$_LAMBDA	= 246	
TPA$_EOS	= 247	
TPA$_SUBEXPR	= 248	(subexpression call)
		(other codes are reserved for expansion)

- *Flags - One Byte.* This byte contains bits that describe the presence of the optional components of the transition. It is always present. The bits are used as follows:

Bit 0	Set if the type byte is a keyword, and so forth
Bit 1	Set if the second flags byte is present
Bit 2	Set if this is the last transition in the state
Bit 3	Set if a subexpression pointer is present
Bit 4	Set if an explicit target state is present
Bit 5	Set if the mask longword is present
Bit 6	Set if the msk-adr longword is present
Bit 7	Set if an action routine address is present

- *Second Flags Byte - One Byte.* This byte is present if any of its flag bits are set. It contains additional flags describing the transition. They are used as follows:

  Bit 0 Set if the action routine parameter is present

- *Subexpression Pointer - Two Bytes.* This word is present in transitions which are subexpression calls. It is a 16-bit signed, self-relative pointer to the starting state of the subexpression.

- *Parameter Longword - Four Bytes.* This longword contains the 32-bit action routine parameter, when specified.

- *Action Routine Address - Four Bytes.* This longword contains a self-relative pointer to the action routine, when specified.

- *Bit Mask - Four Bytes.* This longword contains the mask parameter, when specified.

- *Mask Address - Four Bytes.* This longword, when specified, contains a self-relative pointer through which the mask, or symbol type dependent data, is to be stored. Because the pointer is self-relative, using it to point to an absolute location causes the state table to be non-position independent code.

- *Transition Target - Two Bytes.* This word, when specified, contains the address of the target state of the transition. The address is stored as a 16-bit signed, self-relative pointer. The final state TPA$__EXIT is coded as a word of -1; the failure state TPA$__FAIL is coded as a -2.

- *Keyword Table.* This table is the structure to which the $INIT__STATE macro equates its second parameter. The table is a vector of 16-bit, signed pointers into the keyword string area, relative to the start of the keyword vector. As keywords are generated from the state table source, the TPARSE macros assign an index number to each keyword. The index number is stored in the symbol type byte in the transition; it locates the associated keyword vector entry. The keyword strings are stored in the order encountered in the state table. Each keyword string is terminated by a byte containing the value -1; between the keywords of adjacent states is an additional -1 byte to stop the ambiguous keyword scan.

  To ensure that the keyword vector is adjacent to the keyword string area, the keyword vector is located in PSECT __LIB$KEY0$ and the keyword strings and stored in PSECT __LIB$KEY1$. User programs should not use any of the three PSECTs used by TPARSE (__LIB$STATE$, __LIB$KEY0$, and __LIB$KEY1$) to avoid interfering with the state table structure. These PSECTs refer to each other using 16-bit displacements, so user PSECTS inserted between them can cause truncation errors from the Linker.

7.12 LIB$LOOKUP_KEY — Scan Keyword Table

This procedure scans a table of keywords to find one that matches a caller-specified keyword or keyword abbreviation. It is intended to be an aid for programmers writing utilities that have command qualifiers with values.

LIB$LOOKUP_KEY locates a matching keyword or keyword abbreviation by comparing the first n characters of each keyword in the keyword table with the supplied string, where n is the length of the supplied string.

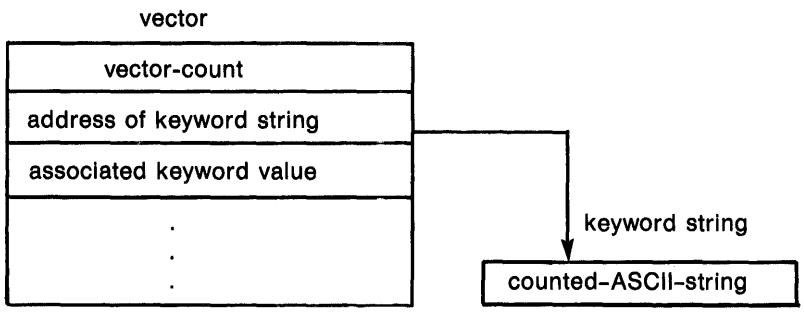
When a keyword match is found, the following information is optionally returned to the caller:

- The longword value associated with the matched keyword
- The full keyword string (any descriptor type)

An exact match is found if the length of the keyword found is equal to the length of the supplied string.

If an exact keyword match is found, no further processing is performed, and a normal return status is returned to the caller. Otherwise, after a match has been found, the rest of the keyword table is scanned. If an additional match is found, a "not enough characters" return status is returned to the caller. If the keyword table contains a keyword that is an abbreviation of another keyword in the table, an exact match can occur for short abbreviations.

The keyword table, which the caller creates for this procedure, has the following structure:



Vector-count is the number of longwords that follow, and counted-ASCII-string starts with a byte that is the unsigned count of the number of ASCII characters that follow.

Format

ret-status = LIB$LOOKUP-KEY (str-dsc-adr, key-table-adr [,key-value-adr [,full-dsc-adr [,out-len]]])

str-dsc-adr
Address of search string descriptor.

key-table-adr
Address of keyword table.

key-value-adr
Address of longword to receive the keyword value. (This is an optional output parameter.)

full-dsc-adr
Address of string descriptor to receive the full keyword matched. (This is an optional output parameter.)

out-len
Address of a word to receive the number of characters in the keyword, independent of padding. (This is an optional output parameter.)

Return Status

SS$_NORMAL
Procedure successfully completed. Unique keyword match found.

LIB$_AMBKEY
Multiple keyword match found (that is, not enough characters specified for unique match).

LIB$_UNRKEY
No keyword match found.

LIB$_INVARG
Invalid arguments, not enough arguments, and/or bad keyword table.

LIB$_INSVIRMEM
Insufficient virtual memory to return keyword string. This is only possible if full-dsc-adr is a dynamic string.

Notes

Because of the format of the keyword table, this procedure cannot be called easily from high-level languages.

Chapter 8
Cross–Reference Procedures

8.1 Introduction

The cross-reference procedures are contained in a separate, sharable image capable of creating a cross-reference analysis of symbols. They accept cross-reference data, summarize it, and format it for output. Two facilities that use the cross-reference procedures are the VAX/VMS Linker and the MACRO assembler. They are sufficiently general, however, to be used by any native-mode utility.

The user provides cross-reference information to the cross-reference procedures as it is acquired. The cross-reference procedures build tables of the data supplied in virtual memory. When all the information has been accumulated in the tables, the user calls the cross-reference output routine to summarize the data and format output lines. The actual printing of the output file is performed by a user-supplied routine that the cross-reference output procedure calls to print each line. Allowing a user-written routine to produce the output provides the user with control over the number of lines per page and the header lines, as well as error handling and recovery.

The interface to the cross-reference procedures is by way of a set of control blocks, format definition tables, and a set of callable entry points. Macros are provided for assembly language and BLISS initialization of the control blocks and format definition tables.

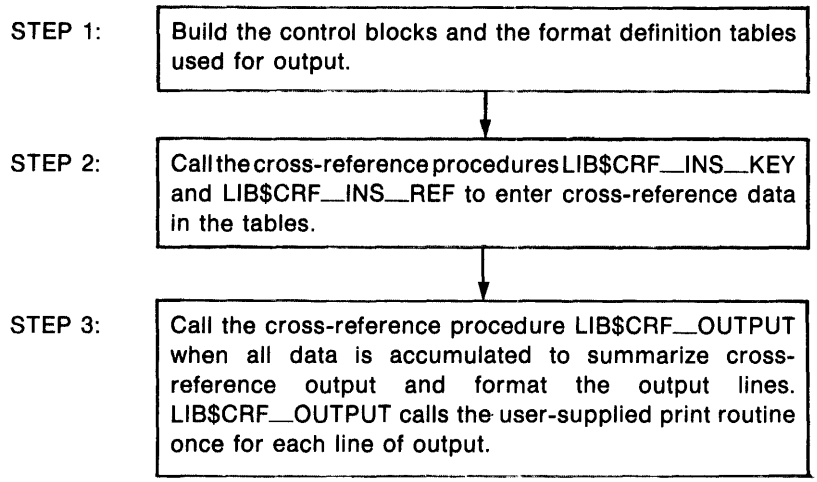
The three entry points provide the user with the following services:

1. Entering a symbol in a cross-reference table
2. Entering a reference to a symbol in a cross-reference table
3. Summarizing accumulated data by symbol name and formatting output lines

A user can create multiple cross-reference tables concurrently.

Figure 8–1 illustrates the steps required of the user to accumulate cross-reference information and prepare it for output using the cross-reference procedures.

Figure 8–1: Producing a Cross-Reference Listing



8.2 Cross-Reference Output

LIB$CRF_OUTPUT is capable of formatting output lines for any of three types of cross-reference listings:

1. A summary of symbol names and their values, as illustrated in Figure 8–2.
2. A summary of symbol names, their values, and the names of modules that refer to the symbol, as illustrated in Figure 8–3.
3. A summary of symbol names, their values, the name of the definer, and the names of those modules that refer to the symbol, as illustrated in Figure 8–4.

Figure 8–2: Summary of Symbol Names and Values

Symbols By Name

Symbol	Value	Symbol	Value
BAS$INSTR	000020B0-RU	BAS$SCRATCH	00002308-RU
BAS$IN_D_R	000021F0-RU	BAS$STATUS	00002338-RU
BAS$IN_F_R	000021E8-RU	BAS$STR_D	000020C0-RU
BAS$IN_L_R	000021E0-RU	BAS$STR_F	000020B8-RU
BAS$IN_T_DX	000021F8-RU	BAS$STR_L	000020C8-RU
BAS$IN_W_R	000021D8-RU	BAS$UNLOCK	00002310-RU
BAS$IO_END	000021D0-RU	BAS$UPDATE	000022E8-RU

(continued on next page)

Symbol	Value	Symbol	Value
BAS$LINKAGE	00001674-R	BAS$UPDATE_COUN	000022F0-RU
BAS$LINPUT	000021A8-RU	BAS$VAL_D	00002110-RU
BAS$MAT_INPUT	00002268-RU	BAS$VAL_F	00002108-RU

Figure 8-3: Summary of Symbol Names, Values, and Name of Referring Modules

Symbol	Value	Reference By ...	
BAS$K_DIVBY_ZER	0000003D	ALLGBL	BAS$ERROR
		BAS$POWDJ	BAS$POWII
		BAS$POWRJ	BAS$POWRR
BAS$K_DUPKEYDET	00000086	ALLGBL	BAS$$SIGNAL_IO
BAS$K_ENDFILDEV	0000000B	ALLGBL	BAS$$REC_PROC
		BAS$$UDF_RL	
BAS$K_ENDOF_STA	0000006C	ALLGBL	

Figure 8-4: Summary Indicating Defining Module

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

Regardless of the format of the output, LIB$CRF__OUTPUT considers the output line as consisting of six different field types:

- A KEY1 field is the first column on the page and contains a symbol name.
- A KEY2 field is the second column of the page and consists of flags (for example, -R) to provide information about the symbol.
- A VAL1 field is the third column of the page and contains the value of the symbol.
- A VAL2 field is the fourth column of the page and consists of flags.
- A number of REF1 and REF2 fields. Each REF1 and REF2 pair provides flags and the name of a module that references the symbol, respectively.

Any of these fields can be omitted from the output.

8.3 Table Initialization Macros

Three macros are used to initialize the data structures used by the cross-reference procedures:

1. $CRFCTLTABLE defines a table of control information.
2. $CRFFIELD defines each field of the output format definition table. Multiple $CRFFIELD macro instructions can be issued in defining one particular field.
3. $CRFFIELDEND indicates the end of a set of $CRFFIELD macro instructions; that is, the end of the format table.

8.3.1 $CRFCTLTABLE Macro

The $CRFCTLTABLE macro initializes a cross-reference control table. One $CRFCTLTABLE macro must be issued for each cross-reference table to be built. The cross-reference procedures let you accumulate information for more than one cross-reference at a time. Each cross-reference must have its own control table defined. The $CRFCTLTABLE macro instruction has the following format:

```
label: $CRFCTLTABLE  keytype,output,error,memexp,key1table,
                     key2table,val1table,val2table,ref1table,ref2table
```

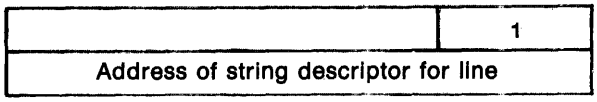
label — Address of the control table. A control table address is specified in all calls to the cross-reference procedures.

keytype — Indicator for the type of key to be entered in the table. The following key types are defined:

- ASCIC — keys are counted ASCII strings, with a maximum of 31 characters (symbol name).
- BIN_U32 — keys are 32-bit unsigned binary values; refer to Section 8.4.4 for its use.

error — Address of an error routine to execute if the called cross-reference procedure encounters an error. A value of zero indicates that no error routine is supplied.

output — Address of a user-supplied routine that prints a formatted output line. The routine is called with the following arguments list:



memexp — Number of pages to expand region when needed (default = 50).

key1table — Address of the field descriptor table for the KEY1 field. The field descriptor table is created by a number of $CRFFIELD macro instructions. A value of zero indicates that the field is not to be included in the output line.

The remaining arguments provide the address of the field descriptor tables for the KEY2, VAL1, VAL2, REF1, and REF2 fields of the output line, respectively.

Argument names (for example, keytype) can be used as keywords in the macros.

8.3.2 $CRFFIELD Macro

One or more $CRFFIELD macros is used to define each field in the output line. The macro identifies the field, supplies an FAO command string to control the printing of the field, and provides flag information. FAO is described in Chapter 6. The $CRFFIELD macro has the following format:

label: $CRFFIELD bit_mask,fao_string,field_width,set_clear

label — Address of the field descriptor table being generated as a result of this set of one or more $CRFFIELD macro instructions. The label field can be omitted after the first macro of the set. These addresses correspond to the field descriptor table addresses in the $CRFCTLTABLE macro.

bit_mask — 16-bit mask with which the flags specified in calls for key processing are to be ANDed when determining which table entry to use in printing this field. Multiple $CRFFIELD macro instructions are used to define multiple bit patterns for a flag field; refer to the discussion of flags in Section 8.3.2.1. Note: the high order bit is reserved to the cross-reference procedures.

fao_string — FAO command string to be used when formatting this field for output.

set_clear — Indicator used to determine whether the bit mask is to be tested as set or clear when determining which flag to use, as follows:

SET indicates test for set.

CLEAR indicates test for clear.

field_width — Maximum width of the output field.

Argument names can be used as keywords.

8.3.2.1 Flag Usage — The KEY2, VAL2, and REF1 fields of cross-reference output can provide special characters, or flags, to indicate additional information about an associated KEY1, VAL1, or REF2 field. For example, the character -R is appended to a symbol name (KEY1) field by the Linker to indicate

that the symbol is relocatable. When the user enters a key or reference, the cross-reference procedure stores flag information with the entry. When preparing the output line, LIB$CRF_OUTPUT ANDs the flag bit mask in the field descriptor table with the flag stored with the entry. Any number of bit masks can be defined for a field. LIB$CRF_OUTPUT searches the list of entries for each flag field. It retains the last entry that has a matching bit pattern. If no match occurs, the first descriptor is used. In the following example, one bit pattern is defined twice: once indicating a string that is to be printed if the pattern is set, and once indicating that spaces are to appear if the pattern is clear.

```
$CRFFIELD    BIT_MASK=SYM$M_REL,FAO_STRING=3\  \,-
             SET_CLEAR=CLEAR,FIELD_WIDTH=2

$CRFFIELD    BIT_MASK=SYM$M_REL,FAO_STRING=\-R\,-
             SET_CLEAR=SET,FIELD_WIDTH=2
```

If more than one set of flags is defined for a field, each FAO string must print the same number of characters; otherwise, the output is not aligned in columns.

The fields for the symbol name, symbol value, and referrers are always formatted using the first descriptor in the corresponding table.

8.3.3 $CRFFIELDEND Macro

The $CRFFIELDEND macro instruction specifies the end of a set of macros that describe one field of the output line. It is used once to end each set of field descriptors. It has the following format:

```
$CRFFIELDEND
```

8.4 Entry Points to Cross-Reference Procedures

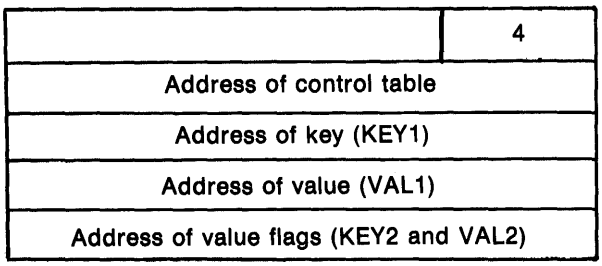
The cross-reference procedures have three entry points which users can call:

1. Insert key information entry point
2. Insert reference information entry point
3. Summarize output and format output lines entry point

8.4.1 Insert Key Entry Point — LIB$CRF_INS_KEY

The user calls LIB$CRF_INS_KEY to store information to be printed in the KEY1, KEY2, VAL1, and VAL2 fields. When the insert key entry point is called, an entry for the key is made in the cross-reference table if the key is not already present in the table. If it is present, only the value address and value flag fields are updated. Figure 8–5 illustrates the format of the argument list used in the call.

Figure 8–5: Argument List for Entering a Key



The first argument is the address of the control table associated with this cross-reference.

The second argument is the address of the key. The address of the key points to a counted ASCII string that contains a symbol name or the unsigned binary longword if BIN_U32 was specified as the key type.

The third argument is the address of the symbol value.

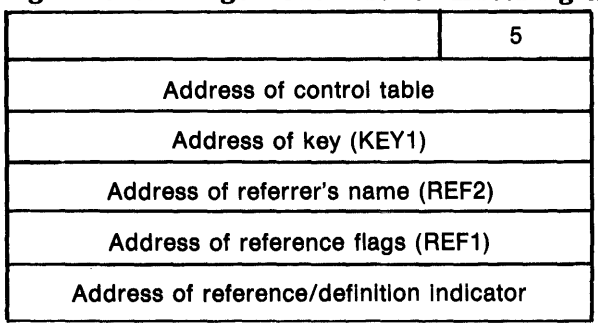
Both the key and value addresses must be permanent addresses in the user's symbol table. LIB$CRF_INS_KEY does not store its own copy of the symbol or value.

The fourth argument is the address of the 16-bit value flags, used in selecting the contents of the KEY2 and VAL2 fields. The flags specified in this argument are copied by LIB$CRF_INS_KEY and are ANDed with the bit mask specified in the field descriptor tables produced by $CRFFIELD macro information for the KEY2 and VAL2 fields. Note: the high-order bit of the 16-bit value is reserved for LIB$CRF_INS_KEY.

8.4.2 Insert Reference Entry Point — LIB$CRF_INS_REF

The user calls LIB$CRF_INS_REF to insert a reference to a key in the cross-reference symbol table. Figure 8–6 illustrates the format of the argument list used in the call.

Figure 8–6: Argument List for Entering a Reference



The first argument is the address of the control table associated with this cross-reference.

The second argument is the address of the key referred to. The address of the key must be a permanent address in the user's symbol table. LIB$CRF_INS_REF does not store its own copy of the key.

The third address is the address of the referrer's name. The address must point to a counted ASCII string with a maximum of 31 characters, not including the byte count. Maintaining the referrer's name as a counted string permits the Linker to pass a module name to identify a reference, and permits compilers to specify a line number to identify a reference. The reference data is stored by LIB$CRF_INS_REF; this data does not have to be stored permanently by the user. LIB$CRF_INS_REF sorts referrer names into alphabetical order and places them in the cross-reference output.

When a table for a synopsis by value is being built, the symbol name associated with a value is specified instead of the referrer's name. That is, the KEY1 field contains the value, and the REF2 fields contain the names of symbols with that value.

The fourth argument is the address of the reference flags used in selecting the contents of the REF1 field. The flags specified in this argument are copied by LIB$CRF_INS_REF and are ANDed with the bit mask specified in the field descriptor table produced by the $CRFFIELD macro instruction for the REF1 field. For example, the assembler may wish to indicate whether a reference is one that modifies a labeled location or whether the name is a macro name, an equated symbol, or a variable name (location label). Note: the high-order bit of the 16-bit value is reserved for LIB$CRF_INS_REF.

The fifth argument is the address of the reference/definition indicator. It is used to distinguish between a reference to a symbol and the definition of the symbol. The indicator can have either of the following values:

CRF$K_REF for a reference to a symbol
CRF$K_DEF for the definition of a symbol

The only difference between processing a symbol reference and a symbol definition is the location where LIB$CRF_INS_REFstores the information. Storing references and definitions in differentplaces provides a means for printing the defining reference first in the cross-reference output line (see Figure 8–4, Section 8.2).

If the user makes two calls, both specifying defining references, the second call overlays the first. Multiple definitions should be entered in the tables as references. The special print characters specified by the reference flags can be used to indicate that the reference is a redefinition of the key.

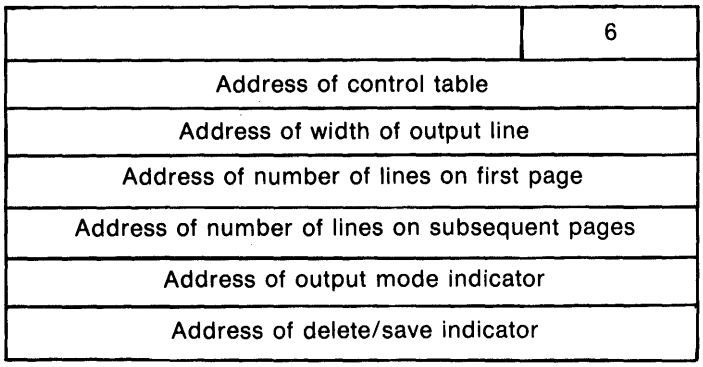
If no defining field is specified for a symbol, and the user requests a definition field in the output, the first REF1 and REF2 fields are space-filled.

**8.4.2.1 Using LIB$CRF_INS_REF to Insert a Key —** If the user attempts to insert reference information for a key that was not specified in a call to LIBCRF_INS_KEY, LIBCRF_INS_REF uses the address of the key (the second argument) to locate the symbol name and set the KEY1 field. Once set, either as a result of LIB$CRF_INS_KEY or LIB$CRF_INS_REF, the KEY1 field is never changed. A KEY1 field set by LIB$CRF_INS_REF has a space-filled VAL1 field associated with it unless it is overridden by a subsequent call to LIB$CRF_INS_KEY.

8.4.3 Output Entry Point — LIB$CRF_OUTPUT

The user calls LIB$CRF_OUTPUT to extract the information from the cross-reference tables. Figure 8-7 shows the format of the argument list used in the call.

Figure 8-7: Argument List for Output of Cross-Reference



The address of the control table points to the control table used to enter key and reference information for the cross-reference. The control table contains the address of the user-supplied routine that prints the lines formatted by LIB$CRF_OUTPUT.

The width of the output line is used by LIB$CRF_OUTPUT in formatting output lines.

Specifying the number of lines on the first page and the number of lines on subsequent pages allows the user to reserve space to print header information on the first page of the cross-reference.

The output mode indicator allows the user to select which of three output modes is desired:

CRF$K_VALUES	indicates that only the value and key fields are to be printed. For this mode, LIB$CRF_OUTPUT creates multiple columns across the page. Each column consists of the KEY1, KEY2, VAL1, and VAL2 fields. A minimum of one space between each column is guaranteed.
CRF$K_VALS_REFS	requests a cross-reference summary. It has no column space saved for a defining reference. If the user inserted a reference with the CRF$K_DEF indicator, the entry is ignored.
CRF$K_DEFS_REFS	requests a cross-reference summary with the first REF1 and REF2 fields used only for definition references. If no definition reference is provided, the fields are space filled.

The delete/save indicator allows the user to specify whether the tables built in accumulating symbol information are to be saved or deleted once the cross-reference is produced. The indicator can be either of the following:

CRF$K_SAVE	to preserve the tables for subsequent processing
CRF$K_DELETE	to delete the tables

8.4.4 Synopsis by Value

LIB$CRF_OUTPUT can also produce a synopsis by symbol value. The following differences exist between producing a synopsis by symbol and a synopsis by value:

1. The KEY1 field of a synopsis by value contains the value, not the symbol name.
2. The VAL1 and VAL2 fields are omitted.
3. The REF2 fields contain the names of symbols with the associated value, not the names of referrers.
4. The control-table macro instruction ($CRFCTLTABLE) specifies a key-type of BIN_U32 to indicate that the KEY1 field is to be handled as a 32-bit, unsigned binary value. The binary-to-ASCII conversion is done by FAO using the format string for the KEY1 field.
5. Calls to LIB$CRF_INS_KEY are made to place a symbol value in the cross-reference table.
6. Calls to LIB$CRF_INS_REF are made to place a symbol name in the cross-reference table.
7. CRF$K_REFS is used in all calls to LIB$CRF_INS_REF.

8.5 User Example

This section contains an example of the use of the cross-reference procedures by the VAX/VMS Linker. The following subsections provide sample macros and entry point calls.

8.5.1 Control Table Initialization

The Linker defines two control tables. The first table defines the output for a symbol-by-name synopsis and also the cross-reference synopsis. The following macro instructions are used:

```
LNK$NAMTAB:
            $CRFCTLTABLE KEYTYPE=ASCIC,ERROR=LNK$ERR_RTN,-
                         OUTPUT=LNK$MAPOUT,KEY1TABLE=LNK$KEY1,-
                         KEY2TABLE=LNK$KEY2,VAL1TABLE=LNK$VAL1,-
                         VAL2TABLE=LNK$VAL2,REF1TABLE=LNK$REF1,-
                         REF2TABLE=LNK$REF2
```

```
LNK$KEY1:
            $CRFFIELD     BIT_MASK=0, FAO_STRING=\!15AC\,-
                          SET_CLEAR=SET,FIELD_WIDTH=15
            $CRFFIELDEND

LNK$KEY2:
            $CRFFIELD     BIT_MASK=0,FAO_STRING=\ \,-
                          SET_CLEAR=SET, FIELD_WIDTH=1
            $CRFFIELDEND

LNK$VAL1:
            $CRFFIELD     BIT_MASK=0,FAO_STRING=\!XL\,-
                          SET_CLEAR=SET,FIELD_WIDTH=8
            $CRFFIELDEND

LNK$VAL2:
            $CRFFIELD     BIT_MASK=0, FAO_STRING=\!2* \,-
                          SET_CLEAR=SET,FIELD_WIDTH=2
            $CRFFIELD     BIT_MASK=SYM$M_REL,FAO_STRING=\-R\,-
                          SET_CLEAR=SET,FIELD_WIDTH=2
            $CRFFIELD     BIT_MASK=SYM$M_DEF, FAO_STRING=\-*\,-
                          SET_CLEAR=CLEAR,FIELD_WIDTH=2
            $CRFFIELDEND

LNK$REF1:
            $CRFFIELD     BIT_MASK=0,FAO_STRING=\!6* \,-
                          SET_CLEAR=SET,FIELD_WIDTH=6
            $CRFFIELD     BIT_MASK=SYM$M_WEAK,FAO_STRING=\!3* WK-\,-
                          SET_CLEAR=SET,FIELD_WIDTH=6
            $CRFFIELDEND

LNK$REF2:
            $CRFFIELD     BIT_MASK=0,FAO_STRING=\!16AC\,-
                          SET_CLEAR=SET,FIELD_WIDTH=16
            $CRFFIELDEND
```

A second control table defines the output for a symbol-by-value synopsis. For this output, the value fields are eliminated. The symbol value becomes a binary longword key. The symbols having this value are entered as reference indicators. None is specified as the defining reference. The control table uses the field descriptors set up previously. The following macro instructions are used:

```
LNK$VALTAB:
            $CRFCTLTABLE KEYTYPE=BIN_U32, ERROR=LNK$ERR_RTN,-
                         OUTPUT=LNK$MAPOUT,KEY1TABLE=LNK$VAL1,-
                         KEY2TABLE=LNK$VAL2,VAL1TABLE=0,-
                         VAL2TABLE=0,REF1TABLE=LNK$REF1,-
                         REF2TABLE=LNK$REF2
```

The FAO control strings for each field above are defined to produce an output of the maximum character size. For example, !15AC produces the variable symbol name left-aligned and right-filled with spaces. Another example is the three sets of characters to be printed for field VAL2. Each FAO control string produces two characters, which is the maximum size of the field. This is essential in producing columnar output.

8.5.2 Sample Calls

After initializing the format data for the symbol tables, the Linker enters data into the cross-reference tables by calling LIB$CRF_INS_KEY.

8.5.2.1 Symbol Processing — As the Linker processes the first object module, MAPINITIAL, it encounters a symbol definition for $MAPFLG. The following is an example of a call to enter the symbol, MAPINITIAL, as a key in the cross-reference symbol table:

```
        PUSHAB  VALUE_FLAGS
        PUSHAB  VALUE_ADDR
        PUSHAB  SYMBOL_ADDR
        PUSHAB  LNK$NAMTAB
        CALLS   #4,G^LIB$CRF_INS_KEY
```

where:

LNK$NAMTAB	is the address of the control table.
SYMBOL_ADDR	is the address of the counted ASCII string $MAPFLG.
VALUE_ADDR	is the address of the symbol value.
VALUE_FLAGS	is the address of a word whose bits are used to select special characters to print beside the value.

The Linker then calls LIB$CRF_INS_REF to process the defining reference indicator:

```
DEF:    .LONG CRF$K_DEF
        PUSHAB  DEF
        PUSHAB  REF_FLAGS
        PUSHAB  REF_ADDR
        PUSHAB  SYMBOL_ADDR
        PUSHAB  LNK$NAMTAB
        CALLS   #5,G^LIB$CRF_INS_REF
```

where:

LNK$NAMTAB	is the address of the control table.
SYMBOL_ADDR	is the address of the counted string $MAPFLG.
REF_ADDR	is the address of the referrer's counted ASCII string.
REF_FLAGS	is the address of a word whose bits are used to select special characters to print beside the reference.

Further on in the input module, the Linker encounters a global symbol reference to CS$GBL. The call to store data for this reference is:

```
REF:    .LONG CRF$K_REF
        PUSHAB  REF
        PUSHAB  REF_FLAGS
        PUSHAB  REF_ADDR
        PUSHAB  SYMBOL_ADDR
        PUSHAB  LNK$NAMTAB
        CALLS   #5,G^LIB$CRF_INS_REF
```

The parameters are similar to the previous example, except CRF$K_REF, which indicates that this is not the defining reference.

After it has performed symbol relocation for the module being bound, the Linker calls LIB$CRF_INS_REF to build a table ordered by value:

```
PUSHAB  REF
PUSHAB  REF_FLAGS
PUSHAB  REF_ADDR
PUSHAB  VAL_ADDR
PUSHAB  LNK$VALTAB
CALLS   #5,G^LIB$CRF_INS_REF
```

where:

LNK$VALTAB	is the address of the control table for the symbol synopsis by value.
VAL_ADDR	is the address of the value (binary longword key).
REF_ADDR	is the address of the symbol name having the value contained in VAL-ADDR.
REF_FLAGS	is the address of a word whose bits are used to select special characters to print beside the value.
CRF$K_REF	is the indicator that this is not a defining reference.

8.5.2.2 Output — After all the input modules are entirely processed, the Linker requests the information for the map. It calls LIB$CRF_OUTPUT once for each type of output. A call to list the symbols and their values would be:

```
LNWID:    .LONG 132
LNSP1:    .LONG LINES_PAGE1
LNSOP:    .LONG LINES_OTHR_PAGE
SAVE:     .LONG CRF$K_SAVE
VAL:      .LONG CRF$K_VALUES
          PUSHAB  VAL
          PUSHAB  SAVE
          PUSHAB  LNSOP
          PUSHAB  LNSP1
          PUSHAB  LNWID
          PUSHAB  LNK$NAMTAB
          CALLS   #6,G^LIB$CRF_OUTPUT
```

The type of output produced by this call is shown in Section 8.2, Figure 8–2.

In the previous example, CRF$K_VALUES means that no reference indicators are to be printed, while CRF$K_SAVE means that the cross-reference table is to be saved. Alternatively, all cross-reference data can be listed. The following call produces such a summary and releases the storage at the same time:

```
LNWID:    .LONG 132
LNSP1:    .LONG LINES_PAGE1
LNSOP:    .LONG LINES_OTHR_PAGE
```

(continued on next page)

```
DELETE:  .LONG CRF$K_DELETE
DEFREF:  .LONG CRF$K_DEF_REF
         PUSHAB  DELETE
         PUSHAB  DEFREF
         PUSHAB  LNSOP
         PUSHAB  LNSP1
         PUSHAB  LNWID
         PUSHAB  LNK$NAMTAB
         CALLS   #6,G^LIB$CRF_OUTPUT
```

The type of output produced by this call is shown in Section 8.2, Figure 8–4.

CRF$K__DEFS__REFS indicates that the first two reference fields are to be used for the defining references, and CRF$K__DELETE indicates that the table is to be deleted.

Another call is made to list the symbol by value synopsis, as follows:

```
LNWID:   .LONG 132
LNSP1:   .LONG LINES_PAGE1
LNSOP:   .LONG LINES_OTHR_PAGE
VALREF:  .LONG CRF$K_VALS_REF
DELETE:  .LONG CRF$K_DELETE
         PUSHAB  DELETE
         PUSHAB  VALREF
         PUSHAB  LNSOP
         PUSHAB  LNSP1
PUSHAB   LNWID
         PUSHAB  LNK$VALTAB
         CALLS   #6,G^LIB$CRF_OUTPUT
```

This is similar to the previous call in that it produces a complete cross-reference output by value, but it does not have the defining reference fields.

8.6 How to Link the Cross-Reference Sharable Image

To link the cross-reference sharable image include a Linker option file with the following line in it:

SYS$LIBRARY:CRFSHR/SHARE

For example, A.COM containing:

```
$LINK X, SYS$INPUT/OPT
!+
! option input
!-
SYS$LIBRARY:CRFSHR/SHARE
```

Appendix A
Summary of Run–Time Library Entry Points

This appendix summarizes the entry points defined by the Run–Time Library. The entry points are described using the procedure parameter notation, a shorthand notation you can use to describe procedure parameters.

The order of the entry points in this appendix is identical to the order of the procedure descriptions in this manual.

A.1 Summary of Procedure Parameter Notation

Procedure parameter notation provides a compact means of specifying the access type, data type, passing mechanism, and parameter form of each parameter.

Subroutine references take the form:

CALL Procedure_name (par1, par2, ... parn)

Function references take the form:

ret-status = PREFIX$PROCEDURE_NAME (par1, par2, ... parn)

func-value = PREFIX$PROCEDURE_NAME (par1, par2, ... parn)

where par1...parn, and func-value characteristics take the form:

<parameter-name>.<access type><data type>.<passing mechanism><parameter form>

In the example that follows, the parameter get-str has the shorthand notation (wt.dx) which translates to <write><text string>.<passed by descriptor><of any data type in descriptor>:

ret-status.wlc.v = LIB$GET_INPUT (get-str.wt.dx
[,prompt-str.rt.dx [,out-len.wwu.r]])

The following notation is used to define these characteristics:

<access type>

c	Call after stack unwind
f	Function call (before return)
j	JMP (after unwind) access
m	Modify access
r	Read-only access
s	Call without stack unwinding
w	Write-only access

<passing mechanism>

d	By descriptor
r	By reference
v	By immediate value

<parameter form>

-	Scalar
a	Array reference or descriptor
d	Dynamic string descriptor
nca	Non-contiguous array desc.
p	Procedure ref. or desc.
s	Fixed-length string descriptor
sd	Scalar decimal descriptor
x	Class type in descriptor

<data type>

a	Virtual address
arb	8-bit relative virtual address
arl	32-bit relative virtual address
arw	16-bit relative virtual address
b	Byte integer (signed)
bpv	Bound procedure value
bu	Byte logical (unsigned)
c	Single character
cit	COBOL intermediate temporary
cp	Character pointer
d	D_floating
dc	D_floating complex
dsc	Descriptor (used by descriptors)
f	F_floating
fc	F_Floating complex
g	G_floating
gc	G_floating complex
h	H_floating
hc	H_floating complex
l	Longword integer (signed)
lc	Longword return status
lu	Longword logical (unsigned)
nu	Num. string, unsigned
nl	Num. string, lt. separate sign
nlo	Num. string, lt. overpunched sign
nr	Num. string, rt. separate sign
nro	Num. string, rt. overpunched sign
nz	Num. string, zoned sign
o	Octaword integer (signed)
ou	Octaword logical (unsigned)
p	Packed decimal string
q	Quadword integer (signed)
qu	Quadword logical (unsigned)
t	Text (character) string
u	Smallest addressable storage unit
v	Bit (variable bit field)
w	Word integer (signed)
wu	Word logical (unsigned)
x	Data type in descriptor
z	Unspecified
zi	Sequence of instruction
zem	Procedure entry mask

The notation xy.z means that the argument is only passed to a user-supplied procedure, and so can have any access type (x), data type (y) and passing mechanism (z).

The order of parameters is generally:

1. Required input (read, jump, function access)
2. Required input/output (modify access)
3. Required output (write access)
4. Optional input (read, jump, function access)
5. Optional input/output (modify access)
6. Optional output (write access)

NOTE

JSB entry points accept parameters in the preceding order in registers starting at R0.

A.2 General Utility Procedures

A.2.1 Common Control Input/Output Procedures

LIB$ASN_WTH_MBX	Assign channel with Mailbox ret-status.wlc.v = LIB$ASN_WTH_MBX (dev-nam.rt.dx, max-msg.rl.v, buf-quo.rl.v, dev-chn.ww.r, mbx-chn.ww.r)
LIB$RUN_PROGRAM	Chain to Program ret-status.wlc.v = LIB$RUN_PROGRAM (pgm-name.rt.dx)
LIB$DO_COMMAND	Execute Command ret-status.wlc.v = LIB$DO_COMMAND (cmd-text.rt.dx)
LIB$GET_COMMAND	Get Command Line from SYS$COMMAND ret-status.wlc.v = LIB$GET_COMMAND (get-str.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]])
LIB$GET_INPUT	Get Command Line from SYS$INPUT ret-status.wlc.v = LIB$GET_INPUT (get-str.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]])
LIB$GET_FOREIGN	Get Foreign Command Line ret-status.wlc.v = LIB$GET_FOREIGN (get-str.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]])
LIB$GET_COMMON	Get String from Common ret-status.wlc.v = LIB$GET_COMMON (dst-str.wt.dx [,chars-copied.ww.r])
LIB$SYS_GETMSG	Get system message ret-status.wlc.v = LIB$SYS_GETMSG (msg-id.rl.r, [msg-len.ww.r], dst-str.wt.dx [,flags.rl.r [,out-arr.wa.ra]])
LIB$CURRENCY	Get currency symbol ret-status.wlc.v = LIB$CURRENCY (currency-str.wt.dx [,out-len.wwu.r])
LIB$DIGIT_SEP	Get digit separator symbol ret-status.wlc.v = LIB$DIGIT_SEP (digit-sep-str.wt.dx [,out-len.wwu.r])

LIB$LP_LINES	Get default number of lines on a line-printer page page-len.wl.v = LIB$LP_LINES ( )
LIB$RADIX_POINT	Get system's radix point symbol ret-status.wlc.v = LIB$RADIX_POINT (radix-point-str.wt.dx [,out-len.wwu.r])
LIB$PUT_OUTPUT	Put Line to SYS$OUTPUT ret-status.wlc.v = LIB$PUT_OUTPUT (msg-str.rt.dx)
LIB$PUT_COMMON	Put String to Common ret-status.wlc.v = LIB$PUT_COMMON (src-str.rt.dx [,chars-copied.ww.r])
LIB$SYS_TRNLOG	Translate Logical name ret-status.wlc.v = LIB$SYS_TRNLOG (logical-name.rt.dx [,dst-len], dst-str.wt.dx [,table.wb.r] [,acc-mode.wb.r] [,dsb-msk.rbu.r])

A.2.2 Terminal Independent Screen Procedures

LIB$ERASE_LINE	Erase Line ret-status.wlc.v = LIB$ERASE_LINE ([line-no.rw.r, col-no.rw.r]) ret-status.wlc.v = SCR$ERASE_LINE ([line-no.rw.v, col-no.rw.v])
LIB$ERASE_PAGE	Erase Page ret-status.wlc.v = LIB$ERASE_PAGE ([line-no.rw.r, col-no.rw.r]) ret-status.wlc.v = SCR$ERASE_PAGE ([line-no.rw.v, col-no.rw.v])
LIB$SCREEN_INFO	Get Screen Information ret-status.wlc.v = LIB$SCREEN_INFO (flags.wl.r [,dev-type.wb.r [,line-width.ww.r [,lines-per-page.ww.r]]]) ret-status.wlc.v = SCR$SCREEN_INFO (control-block.wl.r)
LIB$GET_SCREEN	Get Text from Screen ret-status.wlc.v = LIB$GET_SCREEN (input-text.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]]) ret-status.wlc.v = SCR$GET_SCREEN (input-text.wt.dx [,prompt-str.rt.dx [,out-len.wwu.r]])
LIB$DOWN_SCROLL	Move Cursor up one line, Scroll down if at top ret-status.wlc.v = LIB$DOWN_SCROLL () ret-status.wlc.v = SCR$DOWN_SCROLL ()
LIB$PUT_BUFFER	Put Current Buffer to Screen or Previous Buffer ret-status.wlc.v = LIB$PUT_BUFFER ([old-buffer.wt.ds]) ret-status.wlc.v = SCR$PUT_BUFFER ([old-buffer.wt.ds])
LIB$PUT_SCREEN	Put Text to Screen ret-status.wlc.v = LIB$PUT_SCREEN (text.rt.dx [,line-no.rw.r, col-no.rw.r]) ret-status.wlc.v = SCR$PUT_SCREEN (text.rt.dx [,line-no.rw.v, col-no.rw.v])
LIB$SET_BUFFER	Set/Clear Buffer Mode ret-status.wlc.v = LIB$SET_BUFFER (buffer.mt.ds [,old-buffer.wl.r]) ret-status.wlc.v = SCR$SET_BUFFER (buffer.mt.ds [,old-buffer.wl.r])
LIB$SET_CURSOR	Set Cursor to character position on screen ret-status.wlc.v = LIB$SET_CURSOR (line-no.rw.r, col-no.rw.r) ret-status.wlc.v = SCR$SET_CURSOR (line-no.rw.v, col-no.rw.v)

A.2.3 String Manipulation Procedures

STR$COMPARE		Compare two strings match.wlu.v = STR$COMPARE (src1-str.rt.dx, src2-str.rt.dx)
STR$COMPARE_EQL		Compare two strings for equal match.wlu.v = STR$COMPARE_EQL (src1-str.rt.dx, src2-str.rt.dx)
LIB$LOCC		Locate Character index.wlu.v = LIB$LOCC (char-str.rt.dx, src-str.rt.dx)
LIB$LEN		Return Length of String as Longword Value str-len.wlu.v = LIB$LEN (src-str.rt.dx)
LIB$INDEX		Return Relative Position of Substring index.wlu.v = LIB$INDEX (src-str.rt.dx, sub-str.rt.dx)
LIB$MATCHC		Match Characters index.wlu.v = LIB$MATCHC (sub-str.rt.dx, src-str.rt.dx)
STR$POSITION		Return Relative Position of Substring index.wlu.v = STR$POSITION (src-str.rt.dx, sub-str.rt.dx [,start-pos.rl.r])
	JSB	index.wlu.v = STR$POSITION_R6 (src-str.rt.dx, sub-str.rt.dx, start-pos.rl.v)
LIB$SCANC		Scan Characters index.wlu.v = LIB$SCANC (src-str.rt.dx, table-arr.rbu.ra, mask.rbu.r)
LIB$SKPC		Skip Character index.wlu.v = LIB$SKPC (char-str.rt.dx, src-str.rt.dx)
LIB$SPANC		Span Characters index.wlu.v = LIB$SPANC (src-str.rt.dx, table-arr.rbu.ra, mask.rbu.r)
LIB$CHAR		Transform Byte to First Character of a String ret-status.wlc.v = LIB$CHAR (one-char-str.wt.dx, ascii-code.rbu.r)
LIB$ICHAR		Transform First Character of String to Longword value first-char-value.wlu.v = LIB$ICHAR (src-str.rt.dx)
STR$ADD		Add Two Decimal Strings ret-status.wlc.v = STR$ADD (asign.rv.r, aexp.rl.r, adigits.rnu.dx, bsign.rv.r, bexp.rl.r, bdigits.rnu.dx, csign.wl.r, cexp.wl.r, cdigits.wnu.dx)
STR$MUL		Multiply Two Decimal Strings ret-status.wlc.v = STR$MUL (asign.rv.r, aexp.rl.r, adigits.rnu.dx, bsign.rv.r, bexp.rl.r, bdigits.rnu.dx, csign.wl.r, cexp.wl.r, cdigits.wnu.dx)
STR$RECIP		Reciprocal of a Decimal String ret-status.wlc.v = STR$RECIP (asign.rv.r, aexp.rl.r, adigits.rnu.dx, bsign.rv.r, bexp.rl.r, bdigits.rnu.dx, csign.wl.r, cexp.wl.r, cdigits.wnu.dx)
STR$ROUND		Round or Truncate a Decimal String ret-status.wlc.v = STR$ROUND (places.rl.r, trunc-flg.rv.r, asign.rv.r, aexp.rl.r, adigits.rnu.dx, csign.wl.r, cexp.wl.r, cdigits.wnu.dx)
STR$APPEND		Append a String ret-status.wlc.v = STR$APPEND (dst-str.wt.dx, src-str.rt.dx)
STR$CONCAT		Concatenate Two or more Strings ret-status.wlc.v = STR$CONCAT (dst-str.wt.dx, src1-str.rt.dx, src2-str.rt.dx [,src3-str.rt.dx ... ,srcn-str.rt.dx])

LIB$SCOPY_DXDX		Copy Any Class String Passed by Descriptor to Any Class String ret-status.wlc.v = LIB$SCOPY_DXDX (src-str.rt.dx, dst-str.wt.dx)
	JSB	ret-status.wlc.v = LIB$SCOPY_DXDX6 (src-str.rt.dx, dst-str.wt.dx)
OTS$SCOPY_DXDX		Copy Any Class String Passed by Descriptor to Any Class String unmoved-src.wlu.v = OTS$SCOPY_DXDX (src-str.rt.dx, dst-str.wt.dx)
	JSB	unmoved-src.wlu.v = OTS$SCOPY_DXDX6 (src-str.rt.dx, dst-str.wt.dx)
STR$COPY_DX		Copy any Class String Passed by Descriptor ret-status.wlc.v = STR$COPY_DX (dst-str.wt.dx, src-str.rt.dx)
	JSB	ret-status.wlc.v = STR$COPY_DX_R8 (dst-str.wt.dx, src-str.rt.dx)
LIB$SCOPY_R_DX		Copy Any Class String Passed by Reference to Any Class String ret-status.wlc.v = LIB$SCOPY_R_DX (src-len.rwu.r, src-adr.ra.v, dst-str.wt.dx)
	JSB	ret-status.wlc.v = LIB$SCOPY_R_DX6 (src-len.rwu.v, src-adr.ra.v, dst-str.wt.dx)
OTS$SCOPY_R_DX		Copy Any Class String Passed by Reference to Any Class String unmoved-src.wlu.v = OTS$SCOPY_R_DX (src-len.rwu.v, src-adr.ra.v, dst-str.wt.dx)
	JSB	unmoved-src.wlu.v = OTS$SCOPY_R_DX6 (src-len.rwu.v, src-adr.ra.v, dst-str.wt.dx)
STR$COPY_R		Copy any Class String Passed by Reference ret-status.wlc.v = STR$COPY_R (dst-str.wt.dx, src-len.rwu.r, src-adr.ra.v)
	JSB	ret-status.wlc.v = STR$COPY_R_R8 (dst-str.wt.dx, src-len.rwu.v, src-adr.ra.v)
STR$LEN_EXTR		Extract a substring of a string ret-status.wlc.v = STR$LEN_EXTR (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.r, length.rl.r)
	JSB	ret-status.wlc.v = STR$LEN_EXTR_R8 (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.v, length.rl.v)
STR$POS_EXTR		Extract a substring of a string ret-status.wlc.v = STR$POS_EXTR (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.r, end-pos.rl.r)
	JSB	ret-status.wlc.v = STR$POS_EXTR_R8 (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.v, end-pos.rl.v)
STR$LEFT		Extract a substring of a string ret-status.wlc.v = STR$LEFT (dst-str.wt.dx, src-str.rt.dx, end-pos.rl.r)
	JSB	ret-status.wlc.v = STR$LEFT_R8 (dst-str.wt.dx, src-str.rt.dx, end-pos.rl.v)
STR$RIGHT		Extract a substring of a string ret-status.wlc.v = STR$RIGHT (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.r)
	JSB	ret-status.wlc.v = STR$RIGHT_R8 (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.v)
STR$DUPL_CHAR		Generate a String ret-status.wlc.v = STR$DUPL_CHAR (dst-str.wt.dx [,length.rl.r [,char.rbu.r]])
	JSB	ret-status.wlc.v = STR$DUPL_CHARR8 (dst-str.wt.dx, length.rl.v, char.rbu.v)
STR$PREFIX		Prefix a String ret-status.wlc.v = STR$PREFIX (dst-str.wt.dx, src-str.rt.dx)
STR$REPLACE		Replace a Substring ret-status.wlc.v = STR$REPLACE (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.r, end-pos.rl.r, rpl-str.rt.dx)

	JSB	ret-status.wlc.v = STR$REPLACE_R8 (dst-str.wt.dx, src-str.rt.dx, start-pos.rl.v, end-pos.rl.v, rpl-str.rt.dx)
STR$TRIM		Trim trailing blanks and tabs ret-status.wlc.v = STR$TRIM (dst-str.wt.dx, src-str.rt.dx [,out-len.wwu.r])
LIB$MOVTC		Move Translated Characters ret-status.wlc.v = LIB$MOVTC (src-str.rt.dx, fill-char.rt.dx, trans-tbl.rt.dx, dst-str.wt.dx)
LIB$MOVTUC		Move Translated until Character stop-index.wlu.v = LIB$MOVTUC (src-str.rt.dx, stop-char.rt.dx, trans-tbl.rt.dx, dst-str.wt.dx [,fill-char.rt.dx])
LIB$TRA_ASC_EBC		Translate ASCII to EBCDIC ret-status.wlc.v = LIB$TRA_ASC_EBC (src-str.rt.dx, dst-str.wbu.dx)
LIB$TRA_EBC_ASC		Translate EBCDIC to ASCII ret-status.wlc.v = LIB$TRA_EBC_ASC (src-str.rbu.dx, dst-str.wt.dx)
STR$TRANSLATE		Translate Matched Characters ret-status.wlc.v = STR$TRANSLATE (dst-str.wt.dx, src-str.rt.dx, trans-tbl.rt.dx, match-str.rt.dx)
STR$UPCASE		Uppercase Conversion ret-status.wlc.v = STR$UPCASE (dst-str.wt.dx, src-str.rt.dx)

A.2.4 Formatted Input Conversion Procedures

OTS$CVT_T_z	Convert text to floating (where z = D, G, or H) ret-status.wlc.v = OTS$CVT_T_z (inp-str.rt.dx, value.wz.r [,digits-in-fract.rlu.v [,scale-factor.rl.v [,flags.rlu.v [,ext-bits.wz.r]]]])
FOR$CNV_IN_DEFG	FORTRAN Data Types D, E, F, G Floating-point Input Conversion ret-status.wlc.v = FOR$CNV_IN_DEFG (inp-str.rt.dx, value.wd.r, [,digits-in-fract.rl.v [,scale-factor.rl.v]])
OTS$CVT_TI_L	Convert text (signed integer) to Longword (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_TI_L (inp-str.rt.dx, value.wz.r [,value-size.rl.v [,flags.rlu.v]])
FOR$CNV_IN_I	FORTRAN Integer I Format Input Conversion ret-status.wlc.v = FOR$CNV_IN_I (inp-str.rt.dx, value.wl.r)
OTS$CVT_TL_L	Convert text (logical) to Longword (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_TL_L (inp-str.rt.dx, value.wz.r [,value-size.rl.v])
FOR$CNV_IN_L	FORTRAN Logical L Format Input Conversion ret-status.wlc.v = FOR$CNV_IN_L (inp-str.rt.dx, value.wl.r)
OTS$CVT_TO_L	Convert text (octal) to Longword (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_TO_L (inp-str.rt.dx, value.wz.r [,value-size.rl.v [,flags.rlu.v]])
FOR$CNV_IN_O	FORTRAN Octal O Format Input Conversion ret-status.wlc.v = FOR$CNV_IN_O (inp-str.rt.dx, value.wl.r)
OTS$CVT_TZ_L	Convert text (hexadecimal) to Longword (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_TZ_L (inp-str.rt.dx, value.wz.r [,value-size.rl.v [,flags.rlu.v]])

FOR$CNV_IN_Z	FORTRAN Hexadecimal Z Format Input Conversion ret-status.wlc.v = FOR$CNV_IN_Z (inp-str.rt.dx, value.wl.r)
LIB$CVT_DTB	Decimal to Binary Conversion ret-status.wlc.v = LIB$CVT_DTB (count.rl.v, string.rt.r, result.wl.r)
LIB$CVT_OTB	Octal to Binary Conversion ret-status.wlc.v = LIB$CVT_OTB (count.rl.v, string.rt.r, result.wl.r)
LIB$CVT_HTB	Hexadecimal to Binary Conversion ret-status.wlc.v = LIB$CVT_HTB (count.rl.v, string.rt.r, result.wl.r)

A.2.5 Formatted Output Conversion Procedures

OTS$CVT_L_TI	Convert Longword to Text (signed integer) (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_L_TI (value.rz.r, out-str.wt.ds [,int-digits.rl.v [,value-size.rl.v [,flags.rlu.v]]])
FOR$CNV_OUT_I	FORTRAN Integer I Output Conversion ret-status.wlc.v = FOR$CNV_OUT_I (value.rl.v, out-str.wt.ds)
OTS$CVT_L_TL	Convert Longword to Text (logical) ret-status.wlc.v = OTS$CVT_L_TL (value.rl.r, out-str.wt.ds)
FOR$CNV_OUT_L	FORTRAN Logical L Output Conversion ret-status.wlc.v = FOR$CNV_OUT_L (value.rl.v, out-str.wt.ds)
OTS$CVT_L_TO	Convert Longword to Text (octal) (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_L_TO (value.rz.r, out-str.wt.ds [,int-digits.rl.v [,value-size.rl.v]])
FOR$CNV_OUT_O	FORTRAN Octal O Output Conversion ret-status.wlc.v = FOR$CNV_OUT_O (value.rl.v, out-str.wt.ds)
OTS$CVT_L_TZ	Convert Longword to Text (hexadecimal) (where z = b, w, or l) ret-status.wlc.v = OTS$CVT_L_TZ (value.rz.r, out-str.wt.ds [,int-digits.rl.v [,value-size.rl.v]])
FOR$CNV_OUT_Z	FORTRAN Hexadecimal Z Output Conversion ret-status.wlc.v = FOR$CNV_OUT_Z (value.rl.v, out-str.wt.ds)
FOR$CVT_z_TD	Convert Floating to Text (D format) (where z = D, G, or H) ret-status.wlc.v = FOR$CVT_z_TD (value.rz.r, out-str.wt.ds, digits-in-fract.rlu.v [,scale-factor.rl.v [,digits-in-int.rlu.v [,digits-in-exp.rlu.v [,flags.rlu.v]]]])
FOR$CNV_OUT_D	FORTRAN D Format Output Conversion ret-status.wlc.v = FOR$CNV_OUT_D (value.rd.r, out-str.wt.ds [,digits-in-fract.rlu.v [,scale-factor.rl.v]])
FOR$CVT_z_TE	Convert Floating to Text (E format) (where z = D, G, or H) ret-status.wlc.v = FOR$CVT_z_TE (value.rz.r, out-str.wt.ds, digits-in-fract.rlu.v [,scale-factor.rl.v [,digits-in-int.rlu.v [,digits-in-exp.rlu.v [,flags.rlu.v]]]])
FOR$CNV_OUT_E	FORTRAN E Format Output Conversion ret-status.wlc.v = FOR$CNV_OUT_E (value.rd.r, out-str.wt.ds [,digits-in-fract.rlu.v [,scale-factor.rl.v]])
FOR$CVT_z_TF	Convert Floating to Text (F format) (where z = D, G, or H) ret-status.wlc.v = FOR$CVT_z_TF (value.rz.r, out-str.wt.ds, digits-in-fract.rlu.v [,scale-factor.rl.v [,digits-in-int.rlu.v [,digits-in-exp.rlu.v [,flags.rlu.v]]]])

FOR$CNV_OUT_F	FORTRAN F Format Output Conversion ret-status.wlc.v = FOR$CNV_OUT_F (value.rd.r, out-str.wt.ds [,digits-in-fract.rlu.v [,scale-factor.rl.v]])
FOR$CVT_z_TG	Convert Floating to Text (G format) (where z = D, G, or H) ret-status.wlc.v = FOR$CVT_z_TG (value.rz.r, out-str.wt.ds, digits-in-fract.rlu.v [,scale-factor.rl.v [,digits-in-int.rlu.v [,digits-in-exp.rlu.v [,flags.rlu.v]]]])
FOR$CNV_OUT_G	FORTRAN G Format Output Conversion ret-status.wlc.v = FOR$CNV_OUT_G (value.rd.r, out-str.wt.ds [,digits-in-fract.rlu.v [,scale-factor.rl.v]])
LIB$SYS_FAO	Formatted ASCII output ret-status.wlc.v = LIB$SYS_FAO (ctr-str.rt.dx, [out-len.ww.r], out-buf.wt.dx [,p1.xy.z ... [,pn.xy.z]])
LIB$SYS_FAOL	Formatted ASCII output with List parameter ret-status.wlc.v = LIB$SYS_FAOL (ctr-str.rt.dx, [out-len.ww.r], out-buf.wt.dx, prm-lst.ra.r)

A.2.6 Variable Bit Field Instruction Procedures

LIB$INSV	Insert a Variable Bit Field CALL LIB$INSV (src.rl.r, pos.rl.r, size.rbu.r, base.wv.r)
LIB$EXTV	Extract and Sign-extend a Field field.wlu.v = LIB$EXTV (pos.rl.r, size.rbu.r, base.ra.v)
LIB$EXTZV	Extract a Zero-extended Field field.wlu.v = LIB$EXTZV (pos.rl.r, size.rbu.r, base.ra.v)
LIB$FFC	Find First Clear Bit ret-status.wlc.v = LIB$FFC (start-pos.rl.r, size.rbu.r, base.ra.r, find-pos.wl.r)
LIB$FFS	Find First Set Bit ret-status.wlc.v = LIB$FFS (start-pos.rl.r, size.rbu.r, base.ra.r, find-pos.wl.r)

A.2.7 Performance Measurement Procedures

LIB$FREE_TIMER	Free Timer Storage ret-status.wlc.v = LIB$FREE_TIMER (handle.ml.v)
LIB$INIT_TIMER	Initialize Times and Counts ret-status.wlc.v = LIB$INIT_TIMER ([handle.ml.v])
LIB$STAT_TIMER	Return Accumulated Times and Counts as a Statistic ret-status.wlc.v = LIB$STAT_TIMER (code.rl.r, value.wx.r [,handle.rl.r])
LIB$SHOW_TIMER	Show Accumulated Times and Counts ret-status.wlc.v = LIB$SHOW_TIMER ([[[[handle.rl.r], code.rl.r], action.flc.rp], user-arg.rl.v])

A.2.8 Date/Time Utility Procedures

LIB$SYS_ASCTIM	Convert Binary Date/Time to an ASCII String ret-status.wlc.v = LIB$SYS_ASCTIM (length.ww.r, dst-str.wt.dx, [,user-time.rq.r [,cnvflg.rlu.r]])

FOR$IDATE	Return Month, Day, Year as INTEGER*2 CALL FOR$IDATE (month.ww.r, day.ww.r, year.ww.r)
FOR$JDATE	Return Month, Day, Year as INTEGER*4 CALL FOR$JDATE (month.wl.r, day.wl.r, year.wl.r)
FOR$DATE	Return System Date as 9-Byte String CALL FOR$DATE (9-byte-array.wb.ra)
FOR$SECNDS	Return System Time in Seconds time-difference.wf.v = FOR$SECNDS (time-origin.rf.r)
FOR$TIME	Return System Time as 8-Byte String CALL FOR$TIME (8-byte-array.wb.ra)
LIB$DAY	Return Day Number as a Longword Integer ret-status.wlc.v = LIB$DAY (day-number.wl.r [,user-time.rq.r [,day-time.wl.r]])
LIB$DATE_TIME	Return the System Date and Time as a String ret-status.wlc.v = LIB$DATE_TIME (dst-str.wt.dx)

A.2.9 Miscellaneous General Utility Procedures

LIB$AST_IN_PROG	AST in Progress in-progress.wlu.v = LIB$AST_IN_PROG ( )
LIB$CRC	Calculate Cyclic Redundancy Check crc.wlu.v = LIB$CRC (table.rlu.ra,inicrc.rlu.r, stream.rt.dx)
LIB$CRC_TABLE	Construct Cyclic Redundancy Check Table CALL LIB$CRC_TABLE (poly.rlu.r, table.wl.ra)
LIB$EMULATE	Emulate VAX-11 Instructions ret-status.wlc.v = LIB$EMULATE (sig-args.ma.r. mech-args.ma.r)
LIB$ADDX	Multiple Precision Binary Add ret-status.wlc.v = LIB$ADDX (a.rl.ra, b.rl.ra, result.wl.ra [,len.rl.r])
LIB$SUBX	Multiple Precision Binary Subtract ret-status.wlc.v = LIB$SUBX (a.rl.ra, b.rl.ra, result.wl.ra [,len.rl.r])
LIB$SIM_TRAP	Simulate Floating Trap ret-status.wlc.v = LIB$SIM_TRAP (sig-args.ma.r, mech-args.ma.r)
LIB$EMODF	Extended Multiply and Integerize (F_floating) ret-status.wlc.v = LIB$EMODF (multiplier.rf.r, multext.rb.r, multiplicand.rf.r, int.wl.r, fract.wf.r)
LIB$EMODD	Extended Multiply and Integerize (D_floating) ret-status.wlc.v = LIB$EMODD (multiplier.rd.r, multext.rb.r, multiplicand.rd.r, int.wl.r, fract.wd.r)
LIB$EMODG	Extended Multiply and Integerize (G_floating) ret-status.wlc.v = LIB$EMODG (multiplier.rg.r, multext.rb.r, multiplicand.rg.r, int.wl.r, fract.wg.r)
LIB$EMODH	Extended Multiply and Integerize (H_floating) ret-status.wlc.v = LIB$EMODH (multiplier.rh.r, multext.rb.r, multiplicand.rh.r, int.wl.r, fract.wh.r)
LIB$POLYF	Evaluate Polynomial (F_floating) ret-status.wlc.v = LIB$POLYF (arg.rf.r, degree.rw.r, coeff.rf.ra, result.wf.r)

LIB$POLYD	Evaluate Polynomial (D_floating) ret-status.wlc.v = LIB$POLYD (arg.rd.r, degree.rw.r, coeff.rd.ra, result.wd.r)
LIB$POLYG	Evaluate Polynomial (G_floating) ret-status.wlc.v = LIB$POLYG (arg.rg.r, degree.rw.r, coeff.rg.ra, result.wg.r)
LIB$POLYH	Evaluate Polynomial (H_floating) ret-status.wlc.v = LIB$POLYH (arg.rh.r, degree.rw.r, coeff.rh.ra, result.wh.r)
LIB$INSQHI	Queue Entry Inserted at Head ret-status.wlc.v = LIB$INSQHI (entry.mq.ra, header.mq.r [,retry-cnt.rlu.r])
LIB$INSQTI	Queue Entry Inserted at Tail ret-status.wlc.v = LIB$INSQTI (entry.mq.ra, header.mq.r [,retry-cnt.rlu.r])
LIB$REMQHI	Queue Entry Removed at Head ret-status.wlc.v = LIB$REMQHI (header.mq.r, remque-adr.wl.r [,retry-cnt.rlu.r])
LIB$REMQTI	Queue Entry Removed at Tail ret-status.wlc.v = LIB$REMQTI (header.mq.r, remque-adr.wl.r [,retry-cnt.rlu.r])

A.3 Mathematics Procedures

A.3.1 Floating-Point Mathematical Functions

MTH$ACOS		Arc Cosine (F_floating) acos.wf.v = MTH$ACOS (x.rf.r)
	JSB	acos.wf.v = MTH$ACOS_R4 (x.rf.v)
MTH$DACOS		Arc Cosine (D_floating) dacos.wd.v = MTH$DACOS (x.rd.r)
	JSB	dacos.wd.v = MTH$DACOS_R7 (x.rd.v)
MTH$GACOS		Arc Cosine (G_floating) gacos.wg.v = MTH$GACOS (x.rg.r)
	JSB	gacos.wg.v = MTH$GACOS_R7 (x.rg.v)
MTH$HACOS		Arc Cosine (H_floating) CALL MTH$HACOS (hacos.wh.r, x.rh.r)
	JSB	hacos.wh.v = MTH$HACOS_R8 (x.rh.v)
MTH$ASIN		Arc Sine (F_floating) asin.wf.v = MTH$ASIN (x.rf.r)
	JSB	asin.wf.v = MTH$ASIN_R4 (x.rf.v)
MTH$DASIN		Arc Sine (D_floating) dasin.wd.v = MTH$DASIN (x.rd.r)
	JSB	dasin.wd.v = MTH$DASIN_R7 (x.rd.v)
MTH$GASIN		Arc Sine (G_floating) gasin.wg.v = MTH$GASIN (x.rg.r)
	JSB	gasin.wg.v = MTH$GASIN_R7 (x.rg.v)

MTH$HASIN		Arc Sine (H_floating)
		CALL MTH$HASIN (hasin.wh.r, x.rh.r)
	JSB	hasin.wh.v = MTH$HASIN_R8 (x.rh.v)
MTH$ATAN		Arc Tangent (F_floating)
		atan.wf.v = MTH$ATAN (x.rf.r)
	JSB	atan.wf.v = MTH$ATAN_R4 (x.rf.v)
MTH$DATAN		Arc Tangent (D_floating)
		datan.wd.v = MTH$DATAN (x.rd.r)
	JSB	datan.wd.v = MTH$DATAN_R7 (x.rd.v)
MTH$GATAN		Arc Tangent (G_floating)
		gatan.wg.v = MTH$GATAN (x.rg.r)
	JSB	gatan.wg.v = MTH$GATAN_R7 (x.rg.v)
MTH$HATAN		Arc Tangent (H_floating)
		CALL MTH$HATAN (hatan.wh.r, x.rh.r)
	JSB	hatan.wh.v = MTH$HATAN_R8 (x.rh.v)
MTH$ATAN2		Arc Tangent - 2 parameters (F_floating)
		atan2.wf.v = MTH$ATAN2 (x.rf.r, y.rf.r)
MTH$DATAN2		Arc Tangent - 2 parameters (D_floating)
		datan2.wd.v = MTH$DATAN2 (x.rd.r, y.rd.r)
MTH$GATAN2		Arc Tangent - 2 parameters (G_floating)
		gatan2.wg.v = MTH$GATAN2 (x.rg.r, y.rg.r)
MTH$HATAN2		Arc Tangent - 2 parameters (H_floating)
		CALL MTH$HATAN2 (hatan2.wh.r, x.rh.r, y.rh.r)
MTH$ALOG10		Common Logarithm (F_floating)
		alog10.wf.v = MTH$ALOG10 (x.rf.r)
	JSB	alog10.wf.v = MTH$ALOG10_R5 (x.rf.v)
MTH$DLOG10		Common Logarithm (D_floating)
		dlog10.wd.v = MTH$DLOG10 (x.rd.r)
	JSB	dlog10.wd.v = MTH$DLOG10_R8 (x.rd.v)
MTH$GLOG10		Common Logarithm (G_floating)
		glog10.wg.v = MTH$GLOG10 (x.rg.r)
	JSB	glog10.wg.v = MTH$GLOG10_R8 (x.rg.v)
MTH$HLOG10		Common Logarithm (H_floating)
		CALL MTH$HLOG10 (hlog10.wh.r, x.rh.r)
	JSB	hlog10.wh.v = MTH$HLOG10_R8 (x.rh.v)
MTH$COS		Cosine (F_floating)
		cosine.wf.v = MTH$COS (x.rf.r)
	JSB	cosine.wf.v = MTH$COS_R4 (x.rf.v)
MTH$DCOS		Cosine (D_floating)
		dcosine.wd.v = MTH$DCOS (x.rd.r)
	JSB	dcosine.wd.v = MTH$DCOS_R7 (x.rd.v)
MTH$GCOS		Cosine (G_floating)
		gcosine.wg.v = MTH$GCOS (x.rg.r)
	JSB	gcosine.wg.v = MTH$GCOS_R7 (x.rg.v)
MTH$HCOS		Cosine (H_floating)
		CALL MTH$HCOS (hcosine.wh.r, x.rh.r)
	JSB	hcosine.wh.v = MTH$HCOS_R5 (x.rh.v)
MTH$EXP		Exponential (F_floating)
		exp.wf.v = MTH$EXP (x.rf.r)
	JSB	exp.wf.v = MTH$EXP_R4 (x.rf.v)

MTH$DEXP		Exponential (D_floating)
		dexp.wd.v = MTH$DEXP (x.rd.r)
	JSB	dexp.wd.v = MTH$DEXP_R6 (x.rd.v)
MTH$GEXP		Exponential (G_floating)
		gexp.wg.v = MTH$GEXP (x.rg.r)
	JSB	gexp.wg.v = MTH$GEXP_R6 (x.rg.v)
MTH$HEXP		Exponential (H_floating)
		CALL MTH$HEXP (hexp.wh.r, x.rh.r)
	JSB	hexp.wh.v = MTH$HEXP_R6 (x.rh.v)
MTH$COSH		Hyperbolic Cosine (F_floating)
		cosh.wf.v = MTH$COSH (x.rf.r)
MTH$DCOSH		Hyperbolic Cosine (D_floating)
		dcosh.wd.v = MTH$DCOSH (x.rd.r)
MTH$GCOSH		Hyperbolic Cosine (G_floating)
		gcosh.wg.v = MTH$GCOSH (x.rg.r)
MTH$HCOSH		Hyperbolic Cosine (H_floating)
		CALL MTH$HCOSH (hcosh.wh.r, x.rh.r)
MTH$SINH		Hyperbolic Sine (F_floating)
		sinh.wf.v = MTH$SINH (x.rf.r)
MTH$DSINH		Hyperbolic Sine (D_floating)
		dsinh.wd.v = MTH$DSINH (x.rd.r)
MTH$GSINH		Hyperbolic Sine (G_floating)
		gsinh.wg.v = MTH$GSINH (x.rg.r)
MTH$HSINH		Hyperbolic Sine (H_floating)
		CALL MTH$HSINH (hsinh.wh.r, x.rh.r)
MTH$TANH		Hyperbolic Tangent (F_floating)
		tanh.wf.v = MTH$TANH (x.rf.r)
MTH$DTANH		Hyperbolic Tangent (D_floating)
		dtanh.wd.v = MTH$DTANH (x.rd.r)
MTH$GTANH		Hyperbolic Tangent (G_floating)
		gtanh.wg.v = MTH$GTANH (x.rg.r)
MTH$HTANH		Hyperbolic Tangent (H_floating)
		CALL MTH$HTANH (htanh.wh.r, x.rh.r)
MTH$ALOG		Natural Logarithm (F_floating)
		log.wf.v = MTH$ALOG (x.rf.r)
	JSB	log.wf.v = MTH$ALOG_R5 (x.rf.v)
MTH$DLOG		Natural Logarithm (D_floating)
		dlog.wd.v = MTH$DLOG (x.rd.r)
	JSB	dlog.wd.v = MTH$DLOG_R8 (x.rd.v)
MTH$GLOG		Natural Logarithm (G_floating)
		glog.wg.v = MTH$GLOG (x.rg.r)
	JSB	glog.wg.v = MTH$GLOG_R8 (x.rg.v)
MTH$HLOG		Natural Logarithm (H_floating)
		CALL MTH$HLOG (hlog.wh.r, x.rh.r)
	JSB	hlog.wh.v = MTH$HLOG_R8 (x rh.v)
MTH$SIN		Sine (F_floating)
		sine.wf.v = MTH$SIN (x.rf.r)
	JSB	sine.wf.v = MTH$SIN_R4 (x.rf.v)

MTH$DSIN		Sine (D_floating)
		dsine.wd.v = MTH$DSIN (x.rd.r)
	JSB	dsine.wd.v = MTH$DSIN_R7 (x.rd.v)
MTH$GSIN		Sine (G_floating)
		gsine.wg.v = MTH$GSIN (x.rg.r)
	JSB	gsine.wg.v = MTH$GSIN_R7 (x.rg.v)
MTH$HSIN		Sine (H_floating)
		CALL MTH$HSIN (hsine.wh.r, x.rh.r)
	JSB	hsine.wh.v = MTH$HSIN_R5 (x.rh.v)
MTH$SQRT		Square Root (F_floating)
		sqrt.wf.v = MTH$SQRT (x.rf.r)
	JSB	sqrt.wf.v = MTH$SQRT_R3 (x.rf.v)
MTH$DSQRT		Square Root (D_floating)
		dsqrt.wd.v = MTH$DSQRT (x.rd.r)
	JSB	dsqrt.wd.v = MTH$DSQRT_R5 (x.rd.v)
MTH$GSQRT		Square Root (G_floating)
		gsqrt.wg.v = MTH$GSQRT (x.rg.r)
	JSB	gsqrt.wg.v = MTH$GSQRT_R5 (x.rg.v)
MTH$HSQRT		Square Root (H_floating)
		CALL MTH$HSQRT (hsqrt.wh.r, x.rh.r)
	JSB	hsqrt.wh.v = MTH$HSQRT_R8 (x.rh.v)
MTH$TAN		Tangent (F_floating)
		tangent.wf.v = MTH$TAN (x.rf.r)
	JSB	tangent.wf.v = MTH$TAN_R4 (x.rf.v)
MTH$DTAN		Tangent (D_floating)
		dtangent.wd.v = MTH$DTAN (x.rd.r)
	JSB	dtangent.wd.v = MTH$DTAN_R7 (x.rd.v)
MTH$GTAN		Tangent (G_floating)
		gtangent.wg.v = MTH$GTAN (x.rg.r)
	JSB	gtangent.wg.v = MTH$GTAN_R7 (x.rg.v)
MTH$HTAN		Tangent (H_floating)
		CALL MTH$HTAN (htangent.wh.r, x.rh.r)
	JSB	htangent.wh.v = MTH$HTAN_R5 (x.rh.v)

A.3.2 Complex Functions

MTH$CABS	Absolute Value (F_floating) absolute-value.wf.v = MTH$CABS (complex-number.rfc.r)
MTH$CDABS	Absolute Value (D_floating) CALL MTH$CDABS (absolute-value.wd.r, complex-number.rdc.r)
MTH$CGABS	Absolute Value (G_floating) CALL MTH$CGABS (absolute-value.wg.r, complex-number.rgc.r)
MTH$CONJG	Conjugate of a F_complex number complex-conjugate.wfc.v = MTH$CONJG (complex-number.rfc.r)
MTH$DCONJG	Conjugate of a D_complex number CALL MTH$DCONJG (result.wdc.r, complex-number.rdc.r)
MTH$GCONJG	Conjugate of a G_complex number CALL MTH$GCONJG (result.wgc.r, complex-number.rgc.r)
MTH$CCOS	Cosine (F_complex) complex-cosine.wfc.v = MTH$CCOS (complex-number.rfc.r)

MTH$CDCOS	Cosine (D_complex) CALL MTH$CDCOS (result.wdc.r, complex-number.rdc.r)
MTH$CGCOS	Cosine (G_complex) CALL MTH$CGCOS (result.wgc.r, complex-number.rgc.r)
OTS$DIVC	Division of F_complex numbers complex-quotient.wfc.v = OTS$DIVC (dividend.rfc.v, divisor.rfc.v)
OTS$DIVCD_R3	Division of D_complex numbers complex-quotient.wdc.v = OTS$DIVCD_R3 (dividend.rdc.v, divisor.rdc.v)
OTS$DIVCG_R3	Division of G_complex numbers complex-quotient.wgc.v = OTS$DIVCG_R3 (dividend.rgc.v, divisor.rgc.v)
MTH$CEXP	Exponentiation (F_complex) complex-exp.wfc.v = MTH$CEXP (x.rfc.r)
MTH$CDEXP	Exponentiation (D_complex) CALL MTH$CDEXP (result.wdc.r, x.rdc.r)
MTH$CGEXP	Exponentiation (G_complex) CALL MTH$CGEXP (result.wgc.r, x.rgc.r)
MTH$AIMAG	Imaginary Part of a F_complex number aimag.wf.v = MTH$AIMAG (complex-number.rfc.r)
MTH$DIMAG	Imaginary Part of a D_complex number dimag.wd.v = MTH$DIMAG (complex-number.rdc.r)
MTH$GIMAG	Imaginary Part of a G_complex number gimag.wg.v = MTH$GIMAG (complex-number.rgc.r)
MTH$CMPLX	Make F_complex from F_floating cmplx.wfc.v = MTH$CMPLX (real-part.rf.r, imag-part.rf.r)
MTH$DCMPLX	Make D_complex from D_floating CALL MTH$DCMPLX (dcmplex.wdc.r, real-part.rd.r, imag-part.rd.r)
MTH$GCMPLX	Make G_complex from G_floating CALL MTH$GCMPLX (gcmplex.wgc.r, real-part.rg.r, imag-part.rg.r)
OTS$MULCD_R3	Multiplication of D_complex numbers product.wdc.v = OTS$MULCD_R3 (multiplier.rdc.v, multiplicand.rdc.v)
OTS$MULCG_R3	Multiplication of G_complex numbers product.wgc.v = OTS$MULCG_R3 (multiplier.rgc.v, multiplicand.rgc.v)
MTH$CLOG	Natural Logarithm (F_complex) complex-nat-log.wfc.v = MTH$CLOG (arg.rfc.r)
MTH$CDLOG	Natural Logarithm (D_complex) CALL MTH$CDLOG (result.wdc.r, arg.rdc.r)
MTH$CGLOG	Natural Logarithm (G_complex) CALL MTH$CGLOG (result.wgc.r, arg.rgc.r)
MTH$REAL	Real Part of a F_complex number real-part.wf.v = MTH$REAL (complex-number.rfc.r)
MTH$DREAL	Real Part of a D_complex number dreal-part.wd.v = MTH$DREAL (complex-number.rdc.r)
MTH$GREAL	Real Part of a G_complex number greal-part.wg.v = MTH$GREAL (complex-number.rgc.r)
MTH$CSIN	Sine (F_complex) complex-sine.wfc.v = MTH$CSIN (complex-number.rfc.r)
MTH$CDSIN	Sine (D_complex) CALL MTH$CDSIN (result.wdc.r, complex-number.rdc.r)

MTH$CGSIN	Sine (G_complex) CALL MTH$CGSIN (result.wgc.r, complex-number.rgc.r)
MTH$CSQRT	Square Root (F_complex) complex-sqrt.wfc.v = MTH$CSQRT (x.rfc.r)
MTH$CDSQRT	Square Root (D_complex) CALL MTH$CDSQRT (result.wdc.r, x.rdc.r)
MTH$CGSQRT	Square Root (G_complex) CALL MTH$CGSQRT (result.wgc.r, x.rgc.r)

A.3.3 Exponentiation Procedures

OTS$POWDD	D_floating base to D_floating power result.wd.v = OTS$POWDD (base.rd.v, exponent.rd.v)
OTS$POWDJ	D_floating base to longword power result.wd.v = OTS$POWDJ (base.rd.v, exponent.rl.v)
OTS$POWDR	D_floating base to F_floating power result.wd.v = OTS$POWDR (base.rd.v, exponent.rf.v)
OTS$POWGG	G_floating base to G_floating power result.wg.v = OTS$POWGG (base.rg.v, exponent.rg.v)
OTS$POWGJ	G_floating base to longword power result.wg.v = OTS$POWGJ (base.rg.v, exponent.rl.v)
OTS$POWHH_R3	H_floating base to H_floating power result.wh.v = OTS$POWHH_R3 (base.rh.v, exponent.rh.v)
OTS$POWHJ_R3	H_floating base to longword power result.wh.v = OTS$POWHJ_R3 (base.rh.v, exponent.rl.v)
OTS$POWII	Word base to word power result.ww.v = OTS$POWII (base.rw.v, exponent.rw.v)
OTS$POWJJ	Longword base to longword power result.wl.v = OTS$POWJJ (base.rl.v, exponent.rl.v)
OTS$POWRD	F_floating base to D_floating power result.wd.v = OTS$POWRD (base.rf.v, exponent.rd.v)
OTS$POWRJ	F_floating base to longword power result.wf.v = OTS$POWRJ (base.rf.v, exponent.rl.v)
OTS$POWRR	F_floating base to F_floating power result.wf.v = OTS$POWRR (base.rf.v, exponent.rf.v)

A.3.4 Complex Exponentiation Procedures

OTS$POWCC	F_complex base to F_complex power result.wfc.v = OTS$POWCC (base.rfc.v, exponent.rfc.v)
OTS$POWCDCD_R3	D_complex base to D_complex power result.wdc.v = OTS$POWCDCD_R3 (base.rdc.v, exponent.rdc.v)
OTS$POWCGCG_R3	G_complex base to G_complex power result.wgc.v = OTS$POWCGCG_R3 (base.rgc.v, exponent.rgc.v)
OTS$POWCJ	F_complex base to longword power result.wfc.v = OTS$POWCJ (base.rfc.v, exponent.rl.v)

OTS$POWCDJ_R3		D_complex base to longword power result.wdc.v = OTS$POWCDJ_R3 (base.rdc.v, exponent.rl.v)
OTS$POWCGJ_R3		G_complex base to longword power result.wgc.v = OTS$POWCGJ_R3 (base.rgc.v, exponent.rl.v)

A.3.5 Random Number Generators

MTH$RANDOM		Universal Pseudo-Random Number Generator result.wf.v = MTH$RANDOM (seed.mlu.r)

A.3.6 Floating/Integer Conversion Procedures

MTH$CVT_D_G		Convert D_floating to G_floating (rounded) g-floating.wg.v = MTH$CVT_D_G (d-floating.rd.r)
MTH$CVT_DA_GA		Convert D_floating array to G_floating array (rounded) CALL MTH$CVT_DA_GA (d-floating.rd.ra, g-floating.wg.ra [,count.rl.v])
MTH$CVT_G_D		Convert G_floating to D_floating (exact) d-floating.wd.v = MTH$CVT_G_D (g-floating.rg.r)
MTH$CVT_GA_DA		Convert G_floating array to D_floating array (exact) CALL MTH$CVT_GA_DA (g-floating.rg.ra, d-floating.wd.ra [,count.rl,v])
MTH$DBLE		Convert F_floating to D_floating (exact) d-floating.wd.v = MTH$DBLE (f-floating.rf.r)
MTH$GDBLE		Convert F_floating to G_floating (exact) g-floating.wg.v = MTH$GDBLE (f-floating.rf.r)
MTH$IIFIX		Convert F_floating to word (truncated) word.ww.v = MTH$IIFIX (f-floating.rf.r)
MTH$JIFIX		Convert F_floating to longword (truncated) longword.wl.v = MTH$JIFIX (f-floating.rf.r)
MTH$FLOATI		Convert word to F_floating (exact) f-floating.wf.v = MTH$FLOATI (word.rw.r)
MTH$DFLOTI		Convert word to D_floating (exact) d-floating.wd.v = MTH$DFLOTI (word.rw.r)
MTH$GFLOTI		Convert word to G_floating (exact) g-floating.wg.v = MTH$GFLOTI (word.rw.r)
MTH$FLOATJ		Convert longword to F_floating (exact) f-floating.wf.v = MTH$FLOATJ (longword.rl.r)
MTH$DFLOTJ		Convert longword to D_floating (exact) d-floating.wd.v = MTH$DFLOTJ (longword.rl.r)
MTH$GFLOTJ		Convert longword to G_floating (exact) g-floating.wg.v = MTH$GFLOTJ (longword.rl.r)
MTH$FLOOR		Convert F_floating to greatest F_floating integer result-int.wf.v = MTH$FLOOR (input.rf.r)
	JSB	result-int.wf.v = MTH$FLOOR_R1 (input.rf.v)
MTH$DFLOOR		Convert D_floating to greatest D_floating integer result-int.wd.v = MTH$DFLOOR (input.rd.r)
	JSB	result-int.wd.v = MTH$DFLOOR_R3 (input.rd.v)

MTH$GFLOOR		Convert G_floating to greatest G_floating integer
		result-int.wg.v = MTH$GFLOOR (input.rg.r)
	JSB	result-int.wg.v = MTH$GFLOOR_R3 (input.rg.v)
MTH$HFLOOR		Convert H_floating to greatest H_floating integer
		CALL MTH$HFLOOR (result-int.wh.r, input.rh.r)
	JSB	result-int.wh.v = MTH$HFLOOR_R7 (input.rh.v)
MTH$AINT		Convert F_floating to truncated F_floating
		truncated-f-floating.wf.v = MTH$AINT (f-floating.rf.r)
	JSB	truncated-f-floating.wf.v = MTH$AINT_R2 (f-floating.rf.v)
MTH$DINT		Convert D_floating to truncated D_floating
		truncated-d-floating.wd.v = MTH$DINT (d-floating.rd.r)
	JSB	truncated-d-floating.wd.v = MTH$DINT_R4 (d-floating.rd.v)
MTH$IIDINT		Convert D_floating to word (truncated)
		word.ww.v = MTH$IIDINT (d-floating.rd.r)
MTH$JIDINT		Convert D_floating to longword (truncated)
		longword.wl.v = MTH$JIDINT (d-floating.rd.r)
MTH$GINT		Convert G_floating to G_floating (truncated)
		truncated-g-floating.wg.v = MTH$GINT (g-floating.rg.r)
	JSB	truncated-g-floating.wg.v = MTH$GINT_R4 (g-floating.rg.v)
MTH$IIGINT		Convert G_floating to word (truncated)
		truncated-word.ww.v = MTH$IIGINT (g-floating.rg.r)
MTH$JIGINT		Convert G_floating to longword (truncated)
		truncated-longword.wl.v = MTH$JIGINT (g-floating.rg.r)
MTH$HINT		Convert H_floating to H_floating (truncated)
		CALL MTH$HINT (truncated-h-floating.wh.r, h-floating.rh.r)
	JSB	truncated-h-floating.wh.v = MTH$HINT_R8 (h-floating.rh.v)
MTH$IIHINT		Convert H_floating to truncated word
		truncated-word.ww.v = MTH$IIHINT (h-floating.rh.r)
MTH$JIHINT		Convert H_floating to truncated longword
		truncated-longword.wl.v = MTH$JIHINT (h-floating.rh.r)
MTH$IINT		Convert F_floating to word (truncated)
		truncated-word.ww.v = MTH$IINT (f-floating.rf.r)
MTH$JINT		Convert F_floating to longword (truncated)
		truncated-longword.wl.v = MTH$JINT (f-floating.rf.r)
MTH$ANINT		Convert F_floating to nearest F_floating integer
		nearest-f-float-int.wf.v = MTH$ANINT (f-floating.rf.r)
MTH$DNINT		Convert D_floating to nearest D_floating integer
		nearest-d-float-int.wd.v = MTH$DNINT (d-floating.rd.r)
MTH$IIDNNT		Convert D_floating to nearest word integer
		nearest-word-int.ww.v = MTH$IIDNNT (d-floating.rd.r)
MTH$JIDNNT		Convert D_floating to nearest longword integer
		nearest-long-int.wl.v = MTH$JIDNNT (d-floating.rd.r)
MTH$GNINT		Convert G_floating to nearest G_floating integer
		nearest-g-float-int.wg.v = MTH$GNINT (g-floating.rg.r)
MTH$IIGNNT		Convert G_floating to nearest word integer
		nearest-word-int.ww.v = MTH$IIGNNT (g-floating.rg.r)
MTH$JIGNNT		Convert G_floating to nearest longword integer
		nearest-long-int.wl.v = MTH$JIGNNT (g-floating.rg.r)

MTH$HNINT	Convert H_floating to nearest H_floating integer CALL MTH$HNINT (nearest-h-float-int.wh.v, h-floating.rh.r)
MTH$IIHNNT	Convert H_floating to nearest word integer nearest-word-int.ww.v = MTH$IIHNNT (h-floating.rh.r)
MTH$JIHNNT	Convert H_floating to nearest longword integer nearest-long-int.wl.v = MTH$JIHNNT (h-floating.rh.r)
MTH$ININT	Convert F_floating to nearest word integer nearest-word-int.ww.v = MTH$ININT (f-floating.rf.r)
MTH$JNINT	Convert F_floating to nearest longword integer nearest-long-int.wl.v = MTH$JNINT (f-floating.rf.r)
MTH$SNGL	Convert D_floating to F_floating (rounded) f-floating.wf.v = MTH$SNGL (d-floating.rd.r)
MTH$SNGLG	Convert G_floating to F_floating (rounded) f-floating.wf.v = MTH$SNGLG (g-floating.rg.r)

A.3.7 Miscellaneous Functions

MTH$ABS	F_floating Absolute Value absolute-value.wf.v = MTH$ABS (f-floating.rf.r)
MTH$DABS	D_floating Absolute Value d-absolute-value.wd.v = MTH$DABS (d-floating.rd.r)
MTH$GABS	G_floating Absolute Value g-absolute-value.wg.v = MTH$GABS (g-floating.rg.r)
MTH$HABS	H_floating Absolute Value CALL MTH$HABS (h-absolute-value.wh.r, h-floating.rh.r)
MTH$IIABS	Word Absolute Value absolute-value.ww.v = MTH$IIABS (word.rw.r)
MTH$JIABS	Longword Absolute Value absolute-value.wl.v = MTH$JIABS (longword.rl.r)
MTH$IIAND	Bitwise AND of two word parameters word-value.ww.v = MTH$IIAND (word1.rw.r, word2.rw.r)
MTH$JIAND	Bitwise AND of two longword parameters longword-value.wl.v = MTH$JIAND (longword1.rl.r, longword2.rl.r)
MTH$DIM	Positive Difference of two F_floating parameters f-floating.wf.v = MTH$DIM (f-floating1.rf.r, f-floating2.rf.r)
MTH$DDIM	Positive Difference of two D_floating parameters d-floating.wd.v = MTH$DDIM (d-floating1.rd.r, d-floating2.rd.r)
MTH$GDIM	Positive Difference of two G-floating parameters g-floating.wg.v = MTH$GDIM (g-floating1.rg.r, g-floating2.rg.r)
MTH$HDIM	Positive Difference of two H_floating parameters CALL MTH$HDIM (h-floating.wh.r, h-floating1.rh.r, h-floating2.rh.r)
MTH$IIDIM	Positive Difference of two word parameters word.ww.v = MTH$IIDIM (word1.rw.r, word2.rw.r)
MTH$JIDIM	Positive Difference of two longword parameters longword.wl.v = MTH$JIDIM (longword1.rl.r, longword2.rl.r)
MTH$IIEOR	Bitwise Exclusive OR of two word parameters word.ww.v = MTH$IIEOR (word1.rw.r, word2.rw.r)

MTH$JIEOR	Bitwise Exclusive OR of two longword parameters longword.wl.v = MTH$JIEOR (longword1.rl.r, longword2.rl.r)
MTH$IIOR	Bitwise Inclusive OR of two word parameters word.ww.v = MTH$IIOR (word1.rw.r, word2.rw.r)
MTH$JIOR	Bitwise Inclusive OR of two longword parameters longword.wl.v = MTH$JIOR (longword1.rl.r, longword2.rl.r)
MTH$AIMAX0	F_floating Maximum of n word parameters f-floating-max.wf.v = MTH$AIMAX0 (word.rf.r, ...)
MTH$AJMAX0	F_floating Maximum of n longword parameters f-floating-max.wf.v = MTH$AJMAX0 (longword.rf.r, ...)
MTH$IMAX0	Word Maximum of n word parameters word-max.wf.v = MTH$IMAX0 (word.rf.r, ...)
MTH$JMAX0	Longword Maximum of n longword parameters longword-max.wf.v = MTH$JMAX0 (longword.rf.r, ...)
MTH$AMAX1	F_floating Maximum of n F_floating parameters f-floating-max.wf.v = MTH$AMAX1 (f-floating.rf.r, ...)
MTH$DMAX1	D_floating Maximum of n D_floating parameters d-floating-max.wf.v = MTH$DMAX1 (d-floating.rf.r, ...)
MTH$GMAX1	G_floating Maximum of n G_floating parameters g-floating-max.wg.v = MTH$GMAX1 (g-floating.rg.r, ...)
MTH$HMAX1	H_floating Maximum of n H_floating parameters CALL MTH$HMAX1 (h-floating-max.wh.r, h-floating.rh.r, ...)
MTH$IMAX1	Word Maximum of n F_floating parameters word-max.ww.v = MTH$IMAX1 (f-floating.rf.r, ...)
MTH$JMAX1	Longword Maximum of n F_floating parameters longword-max.wl.v = MTH$JMAX1 (f-floating.rf.r, ...)
MTH$AIMIN0	F_floating Minimum of n word parameters f-floating-min.wf.v = MTH$AIMIN0 (word.rw.r, ...)
MTH$AJMIN0	F_floating Minimum of n longword parameters f-floating-min.wf.v = MTH$AJMIN0 (longword.rl.r, ...)
MTH$IMIN0	Word Minimum of n word parameters word-min.ww.v = MTH$IMIN0 (word.rw.r, ...)
MTH$JMIN0	Longword Minimum of n longword parameters longword-min.wl.v = MTH$JMIN0 (longword.rl.r, ...)
MTH$AMIN1	F_floating Minimum of n F_floating parameters f-floating-min.wf.v = MTH$AMIN1 (f-floating.rf.r, ...)
MTH$DMIN1	D_floating Minimum of n D_floating parameters d-floating-min.wd.v = MTH$DMIN1 (d-floating.rd.r, ...)
MTH$GMIN1	G_floating Minimum of n G_floating parameters g-floating-min.wg.v = MTH$GMIN1 (g-floating.rg.r, ...)
MTH$HMIN1	H_floating Minimum of n H_floating parameters CALL MTH$HMIN1 (h-floating-min.wh.r, h-floating.rh.r, ...)
MTH$IMIN1	Word Minimum of n F_floating parameters word-min.ww.v = MTH$IMIN1 (f-floating.rf.r, ...)
MTH$JMIN1	Longword Minimum of n F_floating parameters longword-min.wl.v = MTH$JMIN1 (f-floating.rf.r, ...)

MTH$AMOD	Remainder of two F_floating parameters, arg1/arg2 f-floating.wf.v = MTH$AMOD (f-floating1.rf.r, f-floating2.rf.r)
MTH$DMOD	Remainder of two D_floating parameters, arg1/arg2 d-floating.wd.v = MTH$DMOD (d-floating1.rd.r, d-floating2.rd.r)
MTH$GMOD	Remainder of two G_floating parameters, arg1/arg2 g-floating.wg.v = MTH$GMOD (g-floating1.rg.r, g-floating2.rg.r)
MTH$HMOD	Remainder of two H_floating parameters, arg1/arg2 CALL MTH$HMOD (h-floating.wh.r, h-floating1.rh.r, h-floating2.rh.r)
MTH$IMOD	Remainder of two word parameters, arg1/arg2 word.ww.v = MTH$IMOD (word1.rw.r, word2.rw.r)
MTH$JMOD	Remainder of two longword parameters, arg1/arg2 longword.wl.v = MTH$JMOD (longword1.rl.r, longword2.rl.r)
MTH$INOT	Bitwise Complement of a word parameter word.ww.v = MTH$INOT (word.rw.r)
MTH$JNOT	Bitwise Complement of a longword parameter longword.wl.v = MTH$JNOT (longword.rl.r)
MTH$DPROD	D_floating Product of two F_floating parameters d-floating.wd.v = MTH$DPROD (f-floating1.rf.r, f-floating2.rf.r)
MTH$GPROD	G_floating Product of two F_floating parameters g-floating.wg.v = MTH$GPROD (f-floating1.rf.r, f-floating2.rf.r)
MTH$SGN	F_floating sign function longword.wl.v = MTH$SGN (f-floating.rf.r)
MTH$SGN	D_floating sign function longword.wl.v = MTH$SGN (d-floating.rd.r)
MTH$IISHFT	Bitwise Shift of a word word.ww.v = MTH$IISHFT (word1.rwu.r, shift-count.rw.r)
MTH$JISHFT	Bitwise Shift of a longword longword.wl.v = MTH$JISHFT (longword1.rlu.r, shift-count.rl.r)
MTH$SIGN	F_floating Transfer of Sign of y to Sign of x f-floating.wf.v = MTH$SIGN (f-floating-x.rf.r, f-floating-y.rf.r)
MTH$DSIGN	D_floating Transfer of Sign of y to Sign of x d-floating.wd.v = MTH$DSIGN (d-floating-x.rd.r, d-floating-y.rd.r)
MTH$GSIGN	G_floating Transfer of Sign of y to Sign of x g-floating.wg.v = MTH$GSIGN (g-floating-x.rg.r, g-floating-y.rg.r)
MTH$HSIGN	H_floating Transfer of Sign of y to Sign of x CALL MTH$HSIGN (h-floating.wh.r, h-floating-x.rh.r, h-floating-y.rh.r)
MTH$IISIGN	Word Transfer of Sign of y to Sign of x word.ww.v = MTH$IISIGN (word-x.rw.r, word-y.rw.r)
MTH$JISIGN	Longword Transfer of Sign of y to Sign of x longword.wl.v = MTH$JISIGN (longword-x.rl.r, longword-y.rl.r)

A.4 Resource Allocation Procedures

A.4.1 Dynamic Allocation of Virtual Memory Procedures

LIB$GET_VM	Allocate Virtual Memory in Program Region ret-status.wlc = LIB$GET_VM (num-bytes.rlu.r, base-adr.wa.r)

LIB$FREE_VM		Deallocate Virtual Memory from Program Region ret-status.wlc = LIB$FREE_VM (num-bytes.rlu.r, base-adr.ra.r)
LIB$STAT_VM		Fetch Virtual Memory Statistics ret-status.wlc = LIB$STAT_VM (code.rl.r, value.wl.r)
LIB$SHOW_VM		Show Virtual Memory Statistics ret-status.wlc = LIB$SHOW_VM ([code.rl.r [,action.flc.rp [,user-arg.xy.z]]])
LIB$GET_LUN		Allocate One Logical Unit Number ret-status.wlc = LIB$GET_LUN (base-adr.wl.r)
LIB$FREE_LUN		Deallocate One Logical Unit Number ret-status.wlc = LIB$FREE_LUN (base-adr.rl.r)
LIB$GET_EF		Allocate One Event Flag ret-status.wlc = LIB$GET_EF (base-adr.wl.r)
LIB$FREE_EF		Deallocate One Event Flag ret-status.wlc = LIB$FREE_EF (base-adr.rl.r)
LIB$RESERVE_EF		Reserve One Event Flag ret-status.wlc = LIB$RESERVE_EF (base.adr.rl.r)

A.4.2 String Resource Allocation Procedures

Note that all LIB$ procedures indicate errors by return status, and all OTS$ and STR$ procedures indicate errors by signaling.

LIB$SGET1_DD		Allocate One Dynamic String ret-status.wlc = LIB$SGET1_DD (len.rwu.r, str.mqu.r)
	JSB	ret-status.wlc = LIB$SGET1_DD_R6 (len.rwu.v, str.mqu.v)
OTS$SGET1_DD		Allocate One Dynamic String ret-status.wlc = OTS$SGET1_DD (len.rwu.r, str.mqu.r)
	JSB	ret-status.wlc = OTS$SGET1_DD_R6 (len.rwu.v, str.mqu.v)
STR$GET1_DX		Allocate One Dynamic String ret-status.wlc = STR$GET1_DX (len.rwu.r, str.mqu.r)
	JSB	ret-status.wlc = STR$GET1_DX_R4 (len.rwu.v, str.mqu.v)
LIB$SFREE1_DD		Deallocate One Dynamic String ret-status.wlc = LIB$SFREE1_DD (dsc-adr.mqu.r)
	JSB	ret-status.wlc = LIB$SFREE1_DD6 (dsc-adr.mqu.v)
OTS$SFREE1_DD		Deallocate One Dynamic String ret-status.wlc = OTS$SFREE1_DD (dsc-adr.mqu.r)
	JSB	ret-status.wlc = OTS$SFREE1_DD6 (dsc-adr.mqu.v)
STR$FREE1_DX		Deallocate One Dynamic String ret-status.wlc = STR$FREE1_DX (dsc-adr.mqu.r)
	JSB	ret-status.wlc = STR$FREE1_DX_R4 (dsc-adr.mqu.v)
LIB$SFREEN_DD		Deallocate n Dynamic Strings ret-status.wlc = LIB$SFREEN_DD (dsc-num.rlu.r, first-dsc-adr.mqu.r)
	JSB	ret-status.wlc = LIB$SFREEN_DD6 (dsc-num.rlu.v, first-dsc-adr.mqu.v)
OTS$SFREEN_DD		Deallocate n Dynamic Strings ret-status.wlc = OTS$SFREEN_DD (dsc-num.rlu.r, first-dsc-adr.mqu.r)
	JSB	ret-status.wlc = OTS$SFREEN_DD6 (dsc-num.rlu.v, first-dsc-adr.mqu.v)

A.5 Signaling and Condition Handling Procedures

A.5.1 Establishing a Condition Handler

LIB$ESTABLISH	Establish a Condition Handler for FORTRAN old-handler.flc.rp = LIB$ESTABLISH (new-handler.flc.rp)
LIB$REVERT	Delete Condition Handler for FORTRAN old-handler.wa.v = LIB$REVERT ( )

A.5.2 Enable/Disable Hardware Conditions

LIB$DEC_OVER	Enable/Disable Decimal Overflow old-setting.wlu.v = LIB$DEC_OVER (new-setting.rbu.r)
LIB$FLT_UNDER	Enable/Disable Floating Underflow old-setting.wlu.v = LIB$FLT_UNDER (new-setting.rbu.r)
LIB$INT_OVER	Enable/Disable Integer Overflow old-setting.wlu.v = LIB$INT_OVER (new-setting.rbu.r)

A.5.3 Signal Generators

LIB$SIGNAL	Signals Exception Condition CALL LIB$SIGNAL (condition-value.rlc.v [, parameters.rl.v, ...])
LIB$STOP	Stop Execution via Signaling CALL LIB$STOP (condition-value.rlc.v [, parameters.rl.v, ...])

A.5.4 Signal Handlers

LIB$MATCH_COND	Match Condition Value index.wlu.v = LIB$MATCH_COND (cond-val.rlc.r, cond-val-i.rlc.r, ...)
LIB$FIXUP_FLT	Fix Up Floating Reserved Operand ret-status.wlc = LIB$FIXUP_FLT (sig-args-adr.rl.ra, mch-args-adr.rl.ra [, new-operand.rf.r])
LIB$SIG_TO_RET	Convert any Signal to Return Status ret-status.wlc = LIB$SIG_TO_RET (sig-args-adr.rl.ra, mch-args-adr.rl.ra)

A.6 Syntax Analysis Procedures

LIB$TPARSE	Table-Driven Finite-State Parser ret-status.wlc = LIB$TPARSE (param-blk.mz.r,state-table.rz.r, key-table.rz.r)
LIB$LOOKUP_KEY	Scan Keyword Table ret-status.wlc = LIB$LOOKUP_KEY (string-descr-adr.rt.dx, key-table-adr.rlu.ra [,key-value-adr.wlu.r [,full-descr-adr.wt.dx [,outlen.ww.r]]])

A.7 Cross-Reference Procedures

LIB$CRF_INS_KEY	Place Symbol Value in Cross-Reference Table ret-status.wlc = LIB$CRF_INS_KEY (output-format-table.rl.r, key.rl.r, value.rl.r, flags.rl.r)
LIB$CRF_INS_REF	Place Symbol Name in Cross-Reference Table ret-status.wlc = LIB$CRF_INS_REF (output-format-table.rl.r, key.rl.r, ref-ind.rl.r, ref-flags.rl.r, def-ind.rl.r)
LIB$CRF_OUTPUT	Output Cross-Reference Table ret-status.wlc = LIB$CRF_OUTPUT (output-format-table.rl.r, line-width.rl.r, pag1-lines.rl.r, pagn-lines.rl.r, prt-ind.rl.r, sav-ind.rl.r)

Appendix B
Run-Time Library Error Messages

B.1 Introduction

Condition value symbols are returned to signal successful procedure completion or to show that an error occurred during procedure execution. Each condition value symbol is a unique, system-wide global symbol with a 32-bit condition value.

The first two or three letters of a condition value symbol indicate the facility detecting the error as follows:

LIB$__	General Procedures
MTH$__	Mathematics Procedures
OTS$__	Language-Independent Support Procedures
STR$__	String Procedures
SS$__	VAX/VMS Operating System

The remaining letters in the symbol are made up of the first three letters of each of the first three words in the message (not counting articles and prepositions). Two-letter words are filled out with an underline character.

Many errors also have language-specific error numbers.

B.2 The Error Signaling Sequence

The system establishes a number of default handlers before the main program is called. When an error condition is signaled, the process stack is scanned from the last item on the stack to the first item on the stack, and each

condition handler established is called in turn. One of the system default handlers then prints the error message and proceeds with one of the following actions depending upon the severity of the error:

Error Severity	Action
INFO	Continues image at point of condition
SUCCESS	Continues image at point of condition
WARNING	Continues image at point of condition
ERROR	Continues image at point of condition
SEVERE	Exits the image

Most errors are signaled as SEVERE. Thus, the default action for most errors is to exit the image. Independent of error severity, procedures that encounter these errors are either "continuable" or "noncontinuable." If the error messages that follow specify "continuable," the procedure can continue execution when the error occurs by calling LIB$SIGNAL, which will signal an exception condition. If the error messages specify "noncontinuable," execution halts as the Run-Time Library calls LIB$STOP.

User-written condition handlers are called before any system default handlers. Thus, a user-written handler can override or alter the affect of a default handler. If a user-written handler changes the severity of an error in a continuable procedure to ERROR or WARNING and resignals the image to continue, the image will continue to execute after the default handler prints the message. If a user-written handler returns SS$CONTINUE on an error in a continuable procedure, the image continues execution at the point of the exception with no further stack scan and no error message printed.

A user-written handler cannot alter the affect of a system default handler on an error in a noncontinuable procedure. The only way a user-written handler can avoid image exit in this case is by an appropriate stack unwind that will continue the image at a point other than at the point of the exception. (See Chapter 6 for a more complete description of user control of error handling.)

B.3 Exceptions

Although most signaled errors are SEVERE with the procedure being noncontinuable, the following errors are SEVERE with the procedure being continuable:

Condition Value Symbol	Message
MTH$__abcdefghi	All Mathematics Procedure errors except MTH$__WRONUMARG and MTH$__INVARGMAT
SS$__DECOVF	Decimal Overflow
SS$__FLTDIV	Arithmetic trap, floating divide by zero
SS$__FLTDIV__F	Arithmetic fault, floating divide by zero

(continued on next page)

Condition Value Symbol	Message
SS$_FLTOVF	Arithmetic trap, floating overflow
SS$_FLTOVF_F	Arithmetic fault, floating overflow
SS$_FLTUND	Arithmetic trap, floating underflow
SS$_FLTUND_F	Arithmetic fault, floating underflow
SS$_INTDIV	Integer Zero Divide
SS$_INTOVF	Integer Overflow
SS$_SUBRNG	Subscript Out of Range

The following error has a severity of ERROR and the procedure is continuable:

FOR$_OUTCONERR Output Conversion Error

B.4 Error Message Descriptions

The following error descriptions are grouped by facility and arranged alphabetically by condition value symbol. The description in uppercase text next to the condition value symbol is the actual message printed. The next line in the error description shows the severity of the error, and whether or not execution can be continued at the point where the error was detected. The paragraph following each message explains the error condition and suggests what recovery action the user can take. This same paragraph is also available interactively using the system command:

HELP ERROR	Prints the error message format and lists the facility names for which there is additional information
HELP ERROR facility	Prints brief description of the facility and lists the error codes for which there is additional information
HELP ERROR facility code	Prints the actual error message, an explanation of the error condition, and may suggest a recovery action the user can take

Figure B-1 is a sample dialogue showing how to use the HELP command.

Figure B-1: Sample Dialogue of the HELP ERROR Command

```
$ HELP ERRORS

  ERRORS

  Errors are displayed in the format:
     %facility-l-code, text
```

(continued on next page)

```
  where:
     "facility" is the name of the facility which produced the
     error (e.g. FOR for FORTRAN).
     "l" is a one letter code indicating the severity of the error.
     The severities are:
          I - Information
          S - Success
          W - Warning
          E - Error
          F - Severe Error
     "code" is an abbreviation for the message text.

  Further help for some of the more common errors can be found by
  typing:   HELP ERROR facility code
  For more information, see the VAX/VMS System Messages and Recovery
  Procedures Manual.

  Additional information available:

  FOR         LIB         MTH         OTS         SYSTEM
$ HELP ERROR SYSTEM

  ERRORS

  SYSTEM

    VAX/VMS and hardware generated messages

    Additional information available:

    ACCVIO      FLTDIV      FLTDIV_F    FLTOVF      FLTOVF_F
    FLTUND      FLTUND_F    INTDIV      INTOVF      SUBRNG
$ HELP ERROR SYSTEM FLTDIV_F

  ERRORS

    SYSTEM

      FLTDIV_F

       arithmetic fault, floating divide by zero

       During a floating-point arithmetic operation an attempt was
       made to divide by zero.

$
```

B.5 General Library Return Status Condition Values

The Run-Time Library does not signal the following symbolic condition values. Rather, these values are returned as 32-bit VAX-11 procedures return status condition values. User programs can signal them by calling LIB$SIGNAL or LIB$STOP, in which case the following messages in uppercase will appear.

LIB$__AMBKEY — xxx IS AN AMBIGUOUS KEYWORD

The keyword does not contain sufficient characters to obtain a unique match in the keyword table passed as a parameter.

LIB$__ATTCONSTO ATTEMPT TO CONTINUE FROM STOP

A condition handling procedure attempted to continue from a call to LIB$STOP; that is, it attempted to continue after an error in a noncontinuable procedure.

LIB$__BADBLOADR BAD BLOCK ADDRESS

LIB$FREE__VM has been called with an address of an invalid block of storage. Either the address is not in the range previously allocated by LIB$GET__VM or the low bits are not clear for the assigned alignment.

LIB$__BADBLOSIZ BAD BLOCK SIZE

LIB$GET__VM has been called with zero or too large a block size.

LIB$__BADSTA BAD STACK

An improper format encountered on the process stack was inaccessible during scanning. The user program has probably written on the stack. Recompiling FORTRAN procedures with /CHECK:BOUNDS qualifier may find an array reference out of bounds.

LIB$__EF__ALRFRE EVENT FLAG ALREADY FREE

The event flag specified by LIB$FREE__EF is already free.

LIB$__EF__ALRRES EVENT FLAG ALREADY RESERVED

The event flag specified by LIB$RESERVE__EF is already reserved.

LIB$__EF__RESSYS EVENT FLAG RESERVED TO SYSTEM

The event flag specified by LIB$FREE__EF or LIB$RESERVE__EF is outside the ranges of 1–23 and 32–63.

LIB$__FATERRLIB FATAL ERROR IN LIBRARY

An internal consistency check has failed in the Run-Time Library. This usually indicates a programming error in the Run-Time Library and should be reported to DIGITAL.

LIB$__INPSTRTRU INPUT STRING TRUNCATED

An input string accepted by LIB$GET__INPUT has been truncated in order to fit the string descriptor passed to it.

LIB$__INSEF	INSUFFICIENT EVENT FLAGS There are no event flags available for allocation.
LIB$__INSLUN	INSUFFICIENT LOGICAL UNIT NUMBERS There are no logical unit numbers available for allocation.
LIB$__INSVIRMEM	INSUFFICIENT VIRTUAL MEMORY A call to LIB$GET__VM has failed because the user program has exceeded the image quota for virtual memory. This quota can be increased by a suitably privileged command.
LIB$__INTLOGERR	INTERNAL LOGIC ERROR A general library procedure has detected an internal logic error. Such a condition should be reported to DIGITAL.
LIB$__INVARG	INVALID ARGUMENTS(S) A calling program has passed one or more invalid arguments to a general library procedure. Consult the description of the procedure for the proper argument format.
LIB$__INVSTRDES	INVALID STRING DESCRIPTOR A string descriptor passed to a general library procedure did not contain a valid DSC$B__CLASS field.
LIB$__INVSCRPOS	INVALID SCREEN POSITION VALUES Line-number or Column-number was equal to zero.
LIB$__INVTYPE	INVALID LIB$TPARSE STATE TABLE ENTRY The state table passed to the LIB$TPARSE procedure was not valid and was unable to be processed.
LIB$__LUNALRFRE	LOGICAL UNIT NUMBER ALREADY FREE The logical unit number that is specified by LIB$FREE__LUN is already free.
LIB$__LUNRESSYS	LOGICAL UNIT NUMBER RESERVED TO SYSTEM The logical unit number that is specified by LIB$FREE__LUN is outside the range of 100 to 119.
LIB$__NOTFOU	NOT FOUND LIB$FFS or LIB$FFC did not find set or clear bit

LIB$__PUSSTAOVE PUSHDOWN STACK OVERFLOW

The image pushdown stack has overflowed. Relink program specifying a larger stack.

LIB$__SIGNO__ARG SIGNAL WITH NO ARGUMENTS

LIB$SIGNAL or LIB$STOP has been called with no arguments. This condition is signaled.

LIB$__SYNTAXERR STRING SYNTAX ERROR DETECTED BY LIB$TPARSE

The string passed to the LI$TPARSE procedure was unable to be parsed due to syntax error.

LIB$__UNRKEY xxx IS AN UNRECOGNIZED KEYWORD

The keyword is not contained in the keyword table passed as a parameter.

LIB$__USEFLORES USE OF FLOATING RESERVED OPERAND

The executing image has accessed a reserved floating-point operand.

B.6 Mathematical Procedures Runtime Errors

The following messages result from incorrect calls to mathematical procedures. A user-supplied handler can set the reserved operand result by modifying the image of R0 or R0/R1 in the signal mechanism vector, (CHF$L__MCH__SAVR0, CHF$L__MCH__SAVR1). See Chapter 6 for a detailed description of condition handling.

MTH$__FLOOVEMAT FLOATING OVERFLOW IN MATH LIBRARY
SEVERE continuable

An overflow condition was detected during execution of a mathematical procedure. The result returned is the reserved operand: minus zero, if execution is continued by a condition handling procedure. If the result is used in a subsequent operation, error SS$__ROPRAND occurs.

MTH$__FLOUNDMAT FLOATING UNDERFLOW IN MATH LIBRARY
SEVERE continuable

An underflow condition was detected during execution of a Mathematical Library procedure and the caller was enabled for floating underflow traps. (See description of LIB$FLT__UNDER procedure.) The result returned is zero, if execution is continued by a condition handling procedure.

MTH$__INVARGMAT INVALID ARGUMENT TO MATH LIBRARY
SEVERE noncontinuable

One of the mathematical procedures has been called with an invalid argument.

MTH$__LOGZERNEG LOGARITHM OF ZERO OR NEGATIVE VALUE
SEVERE continuable

An attempt was made to take the logarithm of zero or a negative number. The result returned is the reserved operand: minus zero if execution is continued by a condition handling procedure. If the result is used in a subsequent operation, error SS$__ROPRAND occurs.

MTH$__SIGLOSMAT SIGNIFICANCE LOST IN MATH LIBRARY
SEVERE continuable

Occurs if the magnitude of the argument is so large that significance is lost from the result. The permitted argument ranges are:

SIN, COS	-2**30 < X < 2**30
DSIN, DCOS	-2**31 < X < 2**31
GSIN, GCOS	-2**31 < X < 2**31
HSIN, HCOS	-2**31 < X < 2**31

MTH$__SQUROONEG SQUARE ROOT OF NEGATIVE VALUE
SEVERE continuable

An attempt was made to evaluate the square root of a negative value. The result returned is the reserved operand: minus zero if execution is continued by a condition handling procedure. If the result is used in a subsequent operation, SS$__ROPRAND occurs.

MTH$__UNDEXP UNDEFINED EXPONENTIATION
SEVERE continuable

An attempt was made to perform an exponentiation which is mathematically undefined; that is, 0.**0. The result returned is the reserved operand: minus zero for floating-point operations, and 0 for integer operations if execution is continued by a condition handling procedure. If the reserved operand result is used in a subsequent operation, error SS$__ROPRAND occurs.

MTH$__WRONUMARG WRONG NUMBER OF ARGUMENTS
SEVERE noncontinuable

An attempt was made to call a library procedure with an improper number of arguments.

B.7 Language-Independent Errors

The following errors are language-independent. The fifth argument returned by ERRSNS is the indicated 32-bit VAX-11 condition value:

OTS$__FATINTERR FATAL INTERNAL ERROR IN RUN-TIME LIBRARY
SEVERE noncontinuable

An explicit or implicit call to the Run-Time Library has resulted in the failure of an internal consistency check. This usually indicates a programming error in the Run-Time Library and should be reported to DIGITAL.

OTS$__INPCONERR INPUT CONVERSION ERROR
SEVERE noncontinuable

Either an invalid character or overflow occurred.

OTS$__INTDATCOR INTERNAL DATA CORRUPTED IN RUN-TIME LIBRARY
SEVERE noncontinuable

On a call to the Run-Time Library, a data base consistency check failed. A user program can cause this by referencing outside of a dimensioned array or requesting input to an address outside the program. Try recompiling with the /CHECK:BOUNDS qualifier to check all array references.

OTS$__INVSTRDES INVALID STRING DESCRIPTOR
SEVERE noncontinuable

A string descriptor passed to a language support procedure did not contain a valid DSC$B__CLASS field.

OTS$__IO__CONCLO I/O CONTINUED TO CLOSED FILE
SEVERE noncontinuable

I/O transfer attempted to a closed file. The I/O was initiated while the file was open.

OTS$__OUTCONERR OUTPUT CONVERSION ERROR
SEVERE noncontinuable

Output Conversion error. The output string is of zero length.

OTS$__USEFLORES USE OF FLOATING RESERVED OPERAND
WARNING continuable

The executing image has accessed a reserved floating operand.

B.8 String Procedures Run-Time Errors

The following messages result from invalid calls to the STR$ facility:

STR$__DIVBY__ZER DIVISION BY ZERO
SEVERE noncontinuable

The string arithmetic routines attempted to take the reciprocal of a string whose numeric value was 0.

STR$__FATINTERR FATAL INTERNAL ERROR
SEVERE noncontinuable

An internal consistency check has failed. This usually indicates an internal error in the Run-Time Library and should be reported to DIGITAL.

STR$__ILLSTRCLA ILLEGAL STRING CLASS
SEVERE noncontinuable

The class code found in the class field of a descriptor is not a string class code supported by the VAX/VMS Procedure Calling and Condition Handling Standard.

STR$__ILLSTRPOS ILLEGAL STRING POSITION
SUCCESS continuable

Successfully completed except one of the character-position parameters to a string routine pointed to a character-position before the beginning of the input string (was less than 1 but 1 was used) or after the end of the input string (was greater than the length of the input string but the length of the input string was used).

STR$__ILLSTRSPE ILLEGAL STRING SPECIFICATION
SUCCESS continuable

Successfully completed except the character-position parameters specifying a substring of a string parameter were inconsistent because the ending character-position was less than the starting character-position, a null string was used.

STR$__INSVIRMEM INSUFFICIENT VIRTUAL MEMORY
SEVERE noncontinuable

An attempt to allocate heap storage for use as dynamic strings or string temporaries failed.

STR$__NEGSTRLEN NEGATIVE STRING LENGTH
SUCCESS continuable

Successfully completed except that a length parameter to a string routine had a negative value, lengths of strings must always be positive or 0.0 was used.

STR$__STRIS__INT STRING IS INTERLOCKED
SEVERE noncontinuable

Code being executed at AST level attempted writing into a string that was being written into or whose length was being used for length computation immediately before the interrupt.

STR$__STRTOOLON STRING IS TOO LONG (GREATER THAN 65535)
FATAL noncontinuable

An attempt was made to create a string that was longer than allowed by the String Facility or the descriptors in the VAX/VMS Procedure Calling and Condition Handling Standard. The maximum length string supported is 65,535.

STR$__TRU TRUNCATION
WARNING continuable

An attempt was made to place more characters into a string than it could contain. The value was truncated on the right to fit.

STR$__WRONUMARG WRONG NUMBER OF ARGUMENTS
SEVERE noncontinuable

A String facility entry was called without the correct number of arguments.

B.9 Hardware Trap Conditions

The following messages result from arithmetic overflow and underflow conditions:

SS$__DECOVF DECIMAL OVERFLOW
SEVERE continuable

During an arithmetic operation, a decimal value has exceeded the largest representable decimal number. The result of the operation is set to the correctly signed least significant digit. This does not occur in FORTRAN.

SS$__FLTDIV ARITHMETIC TRAP, FLOATING/DECIMAL DIVIDE BY ZERO
SEVERE continuable

During a floating-point arithmetic operation, an attempt was made to divide by zero. The result of the operation is set to minus zero which is a reserved operand and the PC is advanced to the next instruction. If the result is used in a subsequent operation error

SS$__ROPRAND occurs. During a decimal string operation, the divisor was 0. The result is set to UNPREDICTABLE.

SS$__FLTDIV__F — ARITHMETIC FAULT, FLOATING DIVIDE BY ZERO
SEVERE continuable

During a floating-point arithmetic operation, an attempt was made to divide by zero. This condition is a fault which means that the PC is pointing to the instruction that faulted. Attempting to continue without changing either the input operands or the PC will result in the same exception.

SS$__FLTOVF — ARITHMETIC TRAP, FLOATING OVERFLOW
SEVERE continuable

During an arithmetic operation, a floating-point value has exceeded the largest representable floating-point number. The result of the operation is set to minus zero which is a reserved operand and the PC is advanced to the next instruction. If the result is used in a subsequent operation error code SS$__ROPRAND occurs. The result is also set to minus zero.

SS$__FLTOVF__F — ARITHMETIC FAULT, FLOATING OVERFLOW
SEVERE continuable

During an arithmetic operation, a floating-point value has exceeded the largest representable floating-point number. This condition is a fault which means that the PC is pointing to the instruction that faulted. Attempting to continue without changing either the input operands or the PC will result in the same exception.

SS$__FLTUND — ARITHMMETIC TRAP, FLOATING UNDERFLOW
SEVERE continuable

During an arithmetic operation, a floating-point value has become less than the smallest representable floating-point number, and has been replaced with a value of zero and the PC is advanced to the next instruction. (Note: usually this trap is disabled and so does not generate an exception condition. It can be enabled and disabled at run-time for the duration of a single program unit by calling LIB$FLT__UNDER.)

SS$__FLTUND__F — ARITHMETIC FAULT, FLOATING UNDERFLOW
SEVERE continuable

During an arithmetic operation, a floating-point value has become less than the smallest representable

floating-point number, and has been replaced with a value of zero. This condition is a fault which means that the PC is pointing to the instruction that faulted. Attempting to continue without changing either the input operands or the PC will result in the same exception.

SS$_INTOVF INTEGER OVERFLOW

SEVERE continuable

During an arithmetic operation an integer's value has exceeded byte, word or longword range. The result of the operation is the correct low-order part. Note that by default this trap is enabled. It can be enabled or disabled at run time for the duration of a single program unit by calling LIB$INT_OVER. It can be disabled at compile time by using the qualifier /CHECK:NOOVERFLOW.

SS$_INTDIV INTEGER ZERO DIVIDE

SEVERE continuable

During an integer mode arithmetic operation an attempt was made to divide by zero. The result is set to the dividend which is equivalent to division by one.

SS$_SUBRNG SUBSCRIPT OUT OF RANGE

SEVERE continuable

An array reference has been detected which is outside the array as described by the array declarator. Execution continues. (This checking is performed only for program units compiled with the qualifier /CHECK:BOUNDS in effect.)

Appendix C
VAX–11 Procedure Calling and Condition Handling Standard

8 Feb 80 - Version 7.0

This appendix is the VAX–11 Procedure Calling Standard used with the VAX–11 hardware procedure call mechanism. This standard applies to:

1. All externally callable interfaces in DIGITAL–supported, standard system software
2. All intermodule CALLs to major VAX–11 components
3. All external procedure CALLs generated by standard DIGITAL language processors

This standard does not apply to calls to internal (or local) routines, or language support routines. Within a single module, the language processor or programmer can use a variety of other linkage and argument-passing techniques.

The standard defines and supports passing arguments by immediate value, by reference and by descriptor. However, the immediate value mechanism is only intended for use by VAX/VMS system services and within programs written in BLISS or MACRO.

The procedure CALL mechanism depends on agreement between the calling and called procedures to interpret the argument list. The argument list does not fully describe itself.

This standard specifies the following attributes of the interfaces between modules:

- Calling sequence — the instructions at the call site and at the entry point

- Argument (or parameter) list — the structure of the list describing the arguments to the called procedure
- Function value return — the form and conventions for the return of the function value as a value or as a condition value to indicate success or failure
- Register usage — which registers are preserved and who is responsible for preserving them
- Stack usage — rules governing the use of the stack
- Argument data types — the data types of arguments that can be passed
- Argument (or parameter) descriptor formats — how descriptors are passed for the more complex arguments
- Condition handling — how exception conditions are signaled and how they can be handled in a modular fashion
- Stack unwinding — how the current thread of execution can be aborted cleanly

The goals in developing the VAX-11 Procedure Calling Standard were:

- The standard must be applicable to all inter-module callable interfaces in the VAX-11 software system. Specifically, the standard must consider the requirements of MACRO, BLISS, BASIC, FORTRAN, PASCAL, COBOL and CALLs to the operating system and library procedures. The needs of other languages that DIGITAL may support in the future must be met by the standard or by compatible revision to it.
- The standard should not include capabilities for lower-level components (such as BLISS, MACRO, operating system) that cannot be invoked from the higher-level languages.
- The calling program and called procedure can be written in different languages. The standard attempts to reduce the need for use of language extensions for mixed language programs.
- The procedure mechanism must be sufficiently economical in both space and time to be used and usable as the only calling mechanism within VAX-11.
- The standard should contribute to the writing of error-free, modular, and maintainable software. Effective sharing and re-use of VAX-11 software modules are significant goals.
- The standard must allow the called procedure a variety of techniques for argument handling. The called procedure can:

  1. Reference arguments indirectly through the argument list
  2. Copy atomic data types, strings and array
  3. Copy addresses of atomic data types, strings and arrays

- The standard should provide the programmer with some control over fixing, reporting, and flow of control on hardware and software exceptions.
- The standard should provide subsystem and application writers with the ability to override system messages to provide a more suitable application oriented interface.
- The standard should add no space or time overhead to procedure calls and returns that do not establish handlers and should minimize time overhead for establishing handlers at the cost of increased time overhead when exceptions occur.

Some possible attributes of a procedure-calling mechanism were considered and rejected. Specific non-goals for the VAX–11 procedure CALL mechanism include:

- It is not necessary for the procedure mechanism to provide complete checking of argument data types, data structures, and parameter access. The VAX–11 protection and memory-management system is not dependent upon "correct" interactions between user-level calling and called procedures. Such extended checking may be desirable in some circumstances, but system integrity is not dependent upon it.
- The VAX–11 procedure mechanism need not provide complete information for an interpretive DEBUG facility. The definition of the DEBUG facility includes a DEBUG symbol table which contains the required descriptive information.

The following definitions apply to this standard:

- A procedure is a closed sequence of instructions that is entered from and returns control to the calling program.
- A function is a procedure that returns a single value according to the standard conventions for value returning. If additional values are returned, they are returned via the argument list.
- A subroutine is a procedure that does not return a known value according to the standard conventions for value returning. If values are returned, they are returned via the argument list.
- An address is a 32–bit VAX–11 address positioned in a longword item.
- Immediate value is a mechanism for passing input parameters in which the actual value is provided in the longword argument list entry by the calling program.
- Reference is a mechanism for passing parameters in which the address of the parameter is provided in the longword argument list by the calling program.
- Descriptor is a mechanism for passing parameters in which the address of a descriptor is provided in the longword argument list entry. The descriptor contains the address of the parameter, the data type, size and additional information needed to describe fully the data passed.

- An exception condition is a hardware or software detected event that alters the normal flow of instruction execution. It usually indicates a failure.
- A condition value is a 32-bit value used to identify an exception condition uniquely. A condition value may be returned to a calling program as a function value or signaled using the VAX-11 Signaling mechanism.
- Language support procedures are called implicitly to implement higher level language constructs. They are not intended to be called explicitly from user programs.
- Library procedures are called explicitly using the equivalent of a CALL statement or function reference.

C.1 Calling Sequence

At the option of the calling program, the called procedure is invoked using either the CALLG or CALLS instruction:

```
CALLG   arglst, proc
CALLS   argcnt, proc
```

CALLS pushes the argument count argcnt onto the stack as a longword and sets the argument pointer (AP) to the top of the stack. The complete sequence using CALLS is:

```
push    argn
...
push    arg1
CALLS   #n,proc
```

If the called procedure returns control to the calling program, control must return to the instruction immediately following the CALLG or CALLS instruction. Skip returns and GOTO returns are only allowed during stack unwind operations.

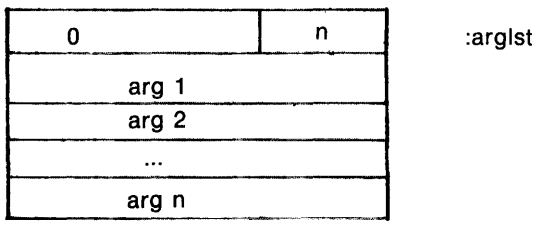
The called procedure returns control to the calling program by executing the return instruction, RET.

C.2 Argument List

The argument list is the primary means of passing information to and receiving results from a procedure.

C.2.1 Argument List Format

The argument list is a sequence of longwords:



The first longword contains the argument count as an unsigned integer in the low byte. The 24 high-order bits are reserved to DIGITAL and must be zero. To access the argument count, the called procedure must ignore the reserved bits and access the count with a MOVZBL, TSTB, or equivalent instruction.

The remaining longwords can be:

1. An uninterpreted 32-bit value (immediate value mechanism) if the called procedure expects less than 32 bits, it accesses the low-order bits and ignores the unwanted high-order bits.
2. An address (reference mechanism) typically a pointer to a scalar data item, an array, or a procedure.
3. An address of a descriptor (descriptor mechanism). See Section C.8 for descriptor formats.

The standard permits immediate value, reference, descriptor, or combinations of these mechanisms. Interpretation of each argument list entry depends on agreement between the calling and called procedures. High-level languages use the reference or descriptor mechanism for passing input parameters. VAX/VMS System Services and MACRO or BLISS programs can use all three mechanisms.

A procedure with no arguments is called with a list consisting of a 0 argument count longword. This is accomplished as follows:

```
CALLS    #0, proc
```

A missing or null argument, for example CALL SUB(A,,B), is represented by an argument list entry consisting of a longword 0. Some procedures allow trailing null arguments to be omitted, others require all arguments. See each procedure's specification for details.

The argument list must be treated as read-only data by the called procedure.

C.2.2 Argument Lists and High-Level Languages

High-level language functional notations for procedure calls are mapped into VAX-11 argument lists according to the following rules:

1. Arguments are mapped from left to right to increasing argument list offsets. The left-most (first) argument has an address of Arglst+4, the next has an address of Arglst+8, ...
2. Each argument position corresponds to a single VAX-11 argument list entry.

C.2.2.1 Order of Argument Evaluation — Since most high-level languages do not specify the order of evaluation (with respect to side effects) of arguments, those language processors can evaluate arguments in any convenient order.

In constructing an argument list on the stack, a language processor can evaluate arguments from right to left and push their values on the stack. If

call-by-reference semantics are used, argument expressions can be evaluated from left to right, with pointers to the expression values or descriptors being pushed from right to left.

The choice of argument evaluation order and code generation strategy is constrained only by the definition of the particular language. Programs should not be written that depend on the order of evaluation of arguments.

C.2.2.2 Language Extensions for Argument Transmission — The VAX-11 procedure standard permits arguments to be passed by immediate value, by reference, or by descriptor. All language processors, except MACRO and BLISS, pass arguments by reference or descriptor.

Language extensions are needed to reconcile the different argument passing mechanisms. Each language processor gives the user explicit control of argument passing mechanism in the calling program. For example, FORTRAN provides the following intrinsic compile-time functions:

%VAL(arg)	Immediate Value Mechanism - Corresponding argument list entry is the 32-bit value of the argument, arg, as defined in the language.
%REF(arg)	Reference Mechanism - Corresponding argument list entry contains the address of the value of the argument, arg, as defined in the language.
%DESCR(arg)	Descriptor Mechanism - Corresponding argument list entry contains the address of a VAX-11 descriptor of the argument, arg, as defined in this appendix and the language.

These intrinsic functions can be used in the syntax of a procedure call to control generation of the argument list. For example:

```
CALL SUB1(%VAL(123), %REF(X), %DESCR(A))
```

In other languages the same effect might be achieved by appropriate attributes of the declaration of SUB1 made in the calling program. Thus, the user might write:

```
CALL SUB1(123,X,A)
```

after making the external declaration for SUB1.

C.3 Function Value Return

A function value is returned in register R0 if its data type is representable in 32 bits or registers R0 and R1 if representable in 64 bits. Two separate 32-bit entities cannot be returned in R0 and R1 because high level languages cannot process them.

If the function value needs more than 64 bits, the actual-argument list and the formal-argument list are shifted one entry. The new, first entry is reserved

for the function value. In this case one of the following mechanisms is used to return the function value:

1. If the maximum length of the function value is known (for example, octaword integer, H_floating, or fixed-length string), the calling program can allocate the required storage and pass the address of the storage or a descriptor for the storage as the first argument.
2. The calling program can allocate a dynamic string descriptor. The called procedure then allocates storage for the function value and updates the contents of the dynamic string descriptor using VAX-11 Run-Time Library procedures.

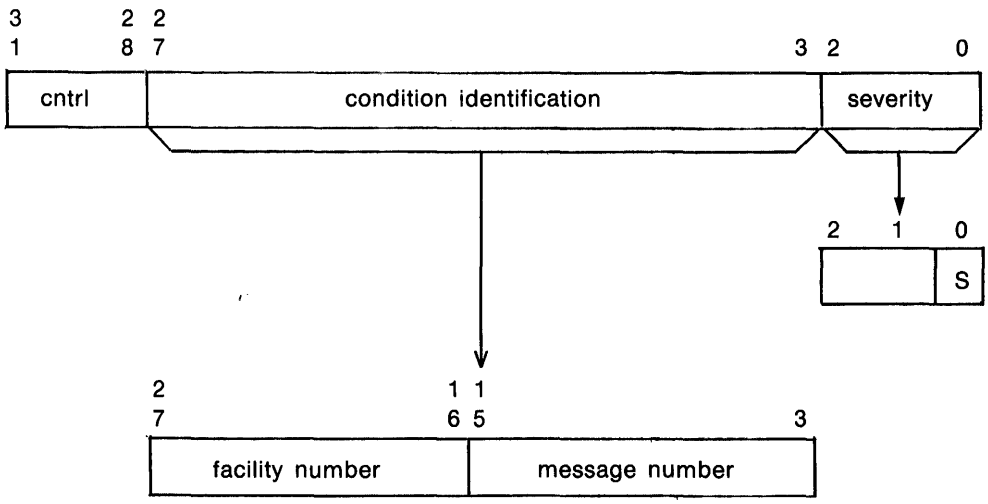
Some procedures, such as operating system calls and many library procedures, return a success/fail value as a longword function value in R0. Bit 0 of the value is set (Boolean true) for a success and clear (Boolean false) for a failure. The particular success or failure status is encoded in the remaining 31 bits, as described in the next section.

C.4 Condition Value

VAX-11 uses condition values for the following:

- To report the success or failure of a called procedure
- To describe an exception condition when it occurs
- To identify system messages
- To report program success or failure for command language testing

A condition value is a longword that includes fields to describe the software component generating the value, the reason the value was generated and the error severity status. The format of the condition value is:



condition identification
Identifies the condition uniquely on a system-wide basis.

facility
Identifies the software component generating the condition value. Bit 27 is set for customer facilities and clear for DIGITAL facilities.

message number
A status identification that is, a description of the hardware exception that occurred or a software-defined value. Message numbers with bit 15 set are specific to a single facility. Message numbers with bit 15 clear are system wide status codes.

severity
The severity code bit 0 is set for success (logical true) and clear for failure (logical false), bits 1 and 2 distinguishes degrees of success or failure. The three bits, 0 through 2, taken as an unsigned integer, are interpreted as follows:

STS$K_WARNING	0 = warning
STS$K_SUCCESS	1 = success
STS$K_ERROR	2 = error
STS$K_INFO	3 = information
STS$K_SEVERE	4 = severe_error
	5, 6, 7 reserved for DIGITAL

Section C.4.1 describes the severity code more fully.

cntrl
Four control bits. Bit 28 inhibits the message associated with the condition value from being printed by the $EXIT system service. This bit is set by the system default handler after it has output an error message using the $PUTMSG system service. It should also be set in the condition value returned by a procedure as a function value, if the procedure has also signaled the condition (so that the condition has been either printed or suppressed). Bits 29 through 31 must be zero; they are reserved for future use by DIGITAL.

Software symbols are defined for these fields as follows:

Mnemonic	Value	Meaning	Field
STS$V_COND_ID	3	position of 27:3	-condition identification
STS$S_COND_ID	25	size of 27:3	
STS$M_COND_ID	mask	mask for 27:3	
STS$V_INHIB_MSG	1@28	position for 28	-inhibit message on image exit
STS$S_INHIB_MSG	1	size for 28	
STS$M_INHIB_MSG	mask	mask for 28	
STS$V_FAC_NO	16	position of 27:16	-facility number
STS$S_FAC_NO	12	size of 27:16	
STS$M_FAC_NO	mask	mask for 27:16	

Mnemonic	Value	Meaning	Field
STS$V_CUST_DEF	27	position for 27	-customer facility
STS$S_CUST_DEF	1	size for 27	
STS$M_CUST_DEF	1@27	mask for 27	
STS$V_MSG_NO	3	position of 15:3	-message number
STS$S_MSG_NO	13	size of 15:3	
STS$M_MSG_NO	mask	mask for 15:3	
STS$V_FAC_SP	15	position of 15	-facility specific
STS$S_FAC_SP	1	size for 15	
STS$M_FAC_SP	1@15	mask for 15	
STS$V_CODE	3	position of 14:3	-message code
STS$S_CODE	12	size of 14:3	
STS$M_CODE	mask	mask for 14:3	
STS$V_SEVERITY	0	position of 2:0	-severity
STS$S_SEVERITY	3	size of 2:0	
STS$M_SEVERITY	7	mask for 2:0	
STS$V_SUCCESS	0	position of 0	-success
STS$S_SUCCESS	1	size of 0	
STS$M_SUCCESS	1	mask for 0	

C.4.1 Interpretation of Severity Codes

A severity code of 0 indicates a warning. This code is used whenever a procedure produces output, but the output might not be what the user expected; for example, a compiler modification of a source program.

A severity code of 1 indicates that the procedure generating the condition value completed successfully, that is, as expected.

A severity code of 2 indicates that an error has occurred, but that the procedure did produce output. Execution can continue but the results produced by the component generating the condition value are not all correct.

A severity code of 3 indicates that the procedure generating the condition value was successfully completed, but has some parenthetical information to be included in a message if the condition was signaled.

A severity code of 4 indicates that a severe_error occurred and the component generating the condition value was unable to produce output.

When designing a procedure the choice of severity code for its condition values should be based on the following default interpretations. The calling program typically performs a low bit test, so it treats warnings, errors, and severe_errors as failures, and success and information as successes. If the condition value is signaled (see Section C.10.3), the default handler treats severe_errors as reason to terminate and all the others as the basis for attempting to continue. When the program image exits, the command interpreter by default treats errors and severe_errors as the basis for stopping the job, and warnings, information, and successes as the basis for continuing.

The following table summarizes the default interpretation of condition values:

Severity	Routine	Signal	Default at Program Exit
success	normal	continue	continue
information	normal	continue	continue
warning	failure	continue	continue
error	failure	continue	stop job
severe_error	failure	exit	stop job

The default for signaled messages is to output a message to file SYS$OUTPUT. In addition, for severities other than success (STS$K_SUCCESS) a copy of the message is made on file SYS$ERROR. At program exit, success and information completion values do not generate messages, while warning, error, and severe_error condition values generate messages to both files SYS$OUTPUT and SYS$ERROR, unless bit 28 (STS$V_INHIB_MSG) is set.

Unless there is a good basis for another choice, a procedure should use either success or severe_error as its severity for each condition value.

C.4.2 Use of Condition Values

VAX-11 software components return condition values when they complete execution. When a severity code of warning, error, or severe_error is generated, the status code describes the nature of the problem. This value can be tested to change the flow of control of a procedure and/or be used to generate a message. User procedures can also generate condition values to be examined by other procedures and by the command interpreter. User-generated values should set bit 27 and bit 15 so these condition values will not conflict with values generated by DIGITAL.

C.5 Register Usage

The following registers have defined uses:

Register	Use
PC	Program counter.
SP	Stack pointer.
FP	Current stack frame pointer. Must always point at current frame. No modification is permitted within a procedure body.
AP	Argument pointer. When a call occurs, AP must point to a valid argument list. A procedure without parameters points to an argument list consisting of a single longword containing the value 0.

(continued on next page)

Register	Use
R1	Environment value. When a call to a procedure that needs an environment value occurs, the calling program must set R1 to the environment value. See bound procedure value in Section C.7.3.
R0,R1	Function value return registers. These registers are not to be preserved by any called procedure. They are available to any called procedure as temporary registers.

Registers R2 through R11 are to be preserved across procedure calls. The called procedure can use registers R2 through R11 provided it saves and restores them using the procedure entry mask mechanism. The entry mask mechanism must be used so that any stack unwinding done by the condition handling mechanism will correctly restore all registers. In addition, PC, SP, FP, and AP are always preserved by the CALL instructions and restored by the RET instruction. However, AP can be used as a temporary register by a called procedure.

C.6 Stack Usage

The stack frame created by the CALLG/CALLS instructions for the called procedure is:

```
condition handler (0)    :(SP):(FP)
mask/PSW
AP
FP
PC
R2      (optional)
...
R11     (optional)
```

FP always points at the condition handler longword of the stack frame, (see Section C.9). Other use of FP within a procedure is prohibited.

The contents of the stack located at addresses higher than the mask/PSW longword belong to the calling program they should not be read or written by the called procedure, except as specified in the argument list. The contents of the stack located at addresses lower than SP belong to interrupt and exception routines, they are continually and unpredictably modified.

The called procedure allocates local storage by subtracting the required number of bytes from the SP provided on entry. This local storage is automatically freed by the RET instruction.

Bit 28 of the mask/PSW longword is reserved to DIGITAL for future extensions to the stack frame.

C.7 Argument Data Types

Each data type implemented for a higher level language uses one of the following VAX data types for procedure parameters and elements of file records. When existing data types are not sufficient to satisfy the semantics of a language, new data types will be added to this standard.

This section also indicates the spelling and punctuation that is used for the name of each data type. In running text, the data type names are not capitalized, except as shown. Also, they are not normally indicated in bold face, italics, or underlined.

Data types fall into three categories: atomic, string, and miscellaneous. These data types can generally be passed by immediate value (if 32 bits or less), by reference or by descriptor. The encoding given in this section is used whenever it is necessary to identify data types, such as in a descriptor. Unless explicitly stated otherwise, all data types represent signed quantities.

NOTE

The unsigned quantities throughout this standard do not allocate space for the sign. All bit or character positions are used for significant data.

C.7.1 Atomic Data Types

The following atomic data types are defined and have the following encoding:

Symbol	Code	Name/Description
DSC$K_DTYPE_Z	0	*unspecified* The calling program has specified no data type. The called procedure should assume the argument is of the correct type.
DSC$K_DTYPE_V	1	*bit* Ordinarily a bit string (see Section C.8, Argument Descriptor Formats).
DSC$K_DTYPE_BU	2	*byte logical* 8-bit unsigned quantity.
DSC$K_DTYPE_WU	3	*word logical* 16-bit unsigned quantity.
DSC$K_DTYPE_LU	4	*longword logical* 32-bit unsigned quantity.
DSC$K_DTYPE_QU	5	*quadword logical* 64-bit unsigned quantity.
DSC$K_DTYPE_OU	25	*octaword logical* 128-bit unsigned quantity.
DSC$K_DTYPE_B	6	*byte integer* 8-bit signed 2's-complement integer.

(continued on next page)

Symbol	Code	Name/Description
DSC$K_DTYPE_W	7	*word integer* 16-bit signed 2's-complement integer.
DSC$K_DTYPE_L	8	*longword integer* 32-bit signed 2's-complement integer.
DSC$K_DTYPE_Q	9	*quadword integer* 64-bit signed 2's-complement integer.
DSC$K_DTYPE_O	26	*octaword integer* 128-bit signed 2's-complement integer.
DSC$K_DTYPE_F	10	*F_floating* 32-bit F_floating quantity representing a single-precision number.
DSC$K_DTYPE_D	11	*D_floating* 64-bit D_floating quantity representing a double-precision number.
DSC$K_DTYPE_G	27	*G_floating* 64-bit G_floating quantity representing a double-precision number.
DSC$K_DTYPE_H	28	*H_floating* 128-bit H_floating quantity representing a quadruple-precision number.
DSC$K_DTYPE_FC	12	*F_floating complex* Ordered pair of F_floating quantities, representing a single-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_DC	13	*D_floating complex* Ordered pair of D_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_GC	29	*G_floating complex* Ordered pair of G_floating quantities, representing a double-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_HC	30	*H_floating complex* Ordered pair of H_floating quantities, representing a quadruple-precision complex number. The lower addressed quantity is the real part, the higher addressed quantity is the imaginary part.
DSC$K_DTYPE_CIT	31	*COBOL Intermediate Temporary* A floating-point datum with an 18-digit normalized decimal fraction and a 2-decimal-digit exponent. The fraction is a packed decimal string. The exponent is a 16-bit 2's-complement integer (See section C.7.4 for more detail).

C.7.2 String Data Types

The following string types are ordinarily described by a string descriptor. The data type codes that follow occur in those descriptors:

Symbol	Code	Name/Description
DSC$K_DTYPE_T	14	*ASCII text* A sequence of 8-bit ASCII characters.
DSC$K_DTYPE_NU	15	*numeric string, unsigned*
DSC$K_DTYPE_NL	16	*numeric string, left separate sign*
DSC$K_DTYPE_NLO	17	*numeric string, left overpunched sign*
DSC$K_DTYPE_NR	18	*numeric string, right separate sign*
DSC$K_DTYPE_NRO	19	*numeric string, right overpunched sign*
DSC$K_DTYPE_NZ	20	*numeric string, zoned sign*
DSC$K_DTYPE_P	21	*packed decimal string*

C.7.3 Miscellaneous Data Types

The following miscellaneous data types are defined and have the following encoding:

Symbol	Code	Name/Description
DSC$K_DTYPE_ZI	22	*sequence of instructions*
DSC$K_DTYPE_ZEM	23	*procedure entry mask*
DSC$K_DTYPE_DSC	24	*descriptor* This data type allows a descriptor to be a data type, thus, levels of descriptors are allowed.
DSC$K_DTYPE_BPV	32	*bound procedure value* A two longword entity in which the first longword contains the address of a procedure entry mask and the second longword is the environment value. The environment value is determined in a language specific manner when the original bound procedure value is generated. When the bound procedure is called, the calling program loads the second longword into R1. When the environment value is not needed, this data type can be passed using the immediate value mechanism. In this case, the argument list entry contains the address of the procedure entry mask and the second longword is omitted.

The type codes 33 through 191 are reserved to DIGITAL. Codes 192 through 255 are reserved for DIGITAL's Computer Special Systems Group and for customers for their own use.

C.7.4 COBOL Intermediate Temporary Data Type

A COBOL intermediate temporary datum is 12 contiguous bytes starting on an arbitrary byte boundary. It is specified by its address, A.

1 1 1 1 5 4 3 2	1 1 1 0 9 8	7 6 5 4	3 2 1 0	
exponent				:A
f<16>	f<15>	0	f<17>	:A+2
f<12>	f<11>	f<14>	f<13>	:A+4
f<8>	f<7>	f<10>	f<9>	:A+6
f<4>	f<3>	f<6>	f<5>	:A+8
f<0>	sign	f<2>	f<1>	:A+10

A COBOL intermediate temporary datum represents a floating-point datum with a normalized 18-digit packed decimal fraction and a 16-bit 2's-complement integer exponent. Bytes 0 and 1 are the exponent. Bytes 2 through 11 contain the normalized packed decimal fraction. The sign of the datum is the sign of the fraction. If the fraction is zero, the value of the datum is zero.

If the exponent is from -99 to +99, operations can be performed on this datum. If the exponent is outside this range, a reserved operand condition is signalled (see section C.9). If a calculated datum has an exponent greater than +99, the exact result with the low-order 15 bits of the true exponent is stored in the result datum and an overflow condition is signalled.

If a calculated datum has an exponent less than -99, the exact result with the low-order 15 bits of the true exponent is stored in the result datum and an underflow condition is signalled. The condition handler can take the appropriate action. Condition mnemonics have a COB$__ prefix and are documented with the COBOL part of the Run-Time Library. An exponent value of -32768 is taken as reserved and should be used to encode reserved operands such as uninitialized datum, indeterminate value, etc. By convention, if the fraction of a result is 0, the exponent is set to 0. Fractions are generated with preferred sign codes and avoid -0.

C.8 Argument Descriptor Formats

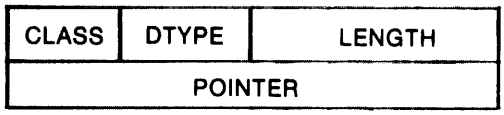
A uniform descriptor mechanism is defined for use by all procedures that conform to the VAX-11 Procedure Calling Standard. Descriptors are uniformly typed and the mechanism is extensible. When existing descriptors are not sufficient to satisfy the semantics of a language, new descriptors will be added to this standard.

NOTE

Unless explicitly stated otherwise, all fields in descriptors represent unsigned quantities and are read-only from the point of view of the called procedure.

C.8.1 Descriptor Prototype

Each class of descriptor consists of at least 2 longwords in the following format:



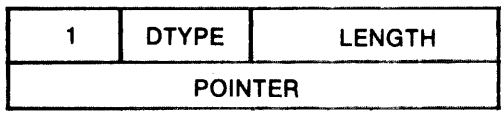
:Descriptor

Symbol	Description
DSC$W_LENGTH <0,15:0>	A one-word field specific to the descriptor class typically a 16-bit (unsigned) length.
DSC$B_DTYPE <0,23:16>	A one-byte data type code (see C.7).
DSC$B_CLASS <0,31:24>	A none-byte descriptor class code (see C.8.2 through C.8.11).
DSC$A_POINTER <1,31:0>	A longword containing the address of the first byte of the data element described.

Note that the descriptor can be placed in a pair of registers with a MOVQ instruction and then the length and address can be used directly. This gives a word length, so the class and type are placed as bytes in the rest of that longword. When the class field is zero, no more than the above information can be assumed.

C.8.2 Scalar, String Descriptor (DSC$K_CLASS_S)

A single descriptor form is used for scalar data and fixed length strings. Any VAX data type can be used with this description.



:Descriptor

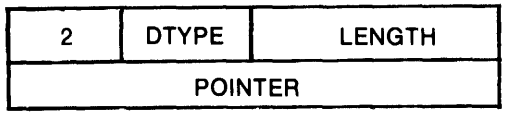
Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one-byte data type code (see Section C.7).
DSC$B_CLASS	1 = DSC$K_CLASS_S.
DSC$A_POINTER	Address of first byte of data storage.

If the string must be extended in a string comparison or is being copied to a fixed length string containing a greater length, the ASCII space character (hex 20) is used as the fill character.

C.8.3 Dynamic String Descriptor (DSC$K_CLASS_D)

A single descriptor form is used for dynamically allocated strings. When a string is written, either or both the length field and the pointer field can be

changed. The VAX-11 Run-Time Library provides procedures for changing fields. As an input parameter this format is interchangeable with class 1 (DSC$K_CLASS_S).



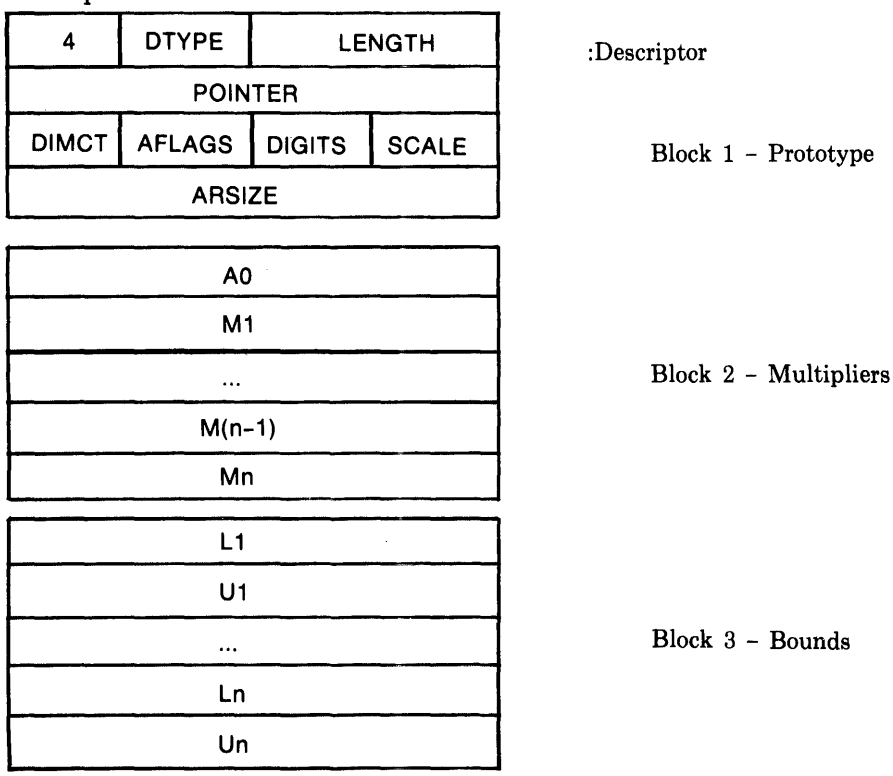
Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one-byte data type code (see Section C.7).
DSC$B_CLASS	2 = DSC$K_CLASS_D.
DSC$A_POINTER	Address of first byte of data storage.

C.8.4 Varying String Descriptor

Reserved for use by DIGITAL.

C.8.5 Array Descriptor (DSC$K_CLASS_A)

The array descriptor is used to describe contiguous arrays of atomic data types or contiguous arrays of fixed length strings. An array descriptor consists of three contiguous blocks. The first block contains the descriptor prototype information and is part of every array descriptor. The second and third blocks are optional. If the third block is present, so is the second. A complete array descriptor has the form:



Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one byte data type code (see Section C.7).
DSC$B_CLASS	4 = DSC$K_CLASS_A.
DSC$A_POINTER	Address of first actual byte of data storage.
DSC$B_SCALE	Signed power of ten multiplier to convert the internal form to external form. For example, if internal number is 123 and scale is +1, then the external number is 1230.
DSC$B_DIGITS	If non-zero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.
DSC$B_AFLAGS <2,23:16>	Array flag bits:
Reserved <2,19:16>	Must be zero.
DSC$V_FL_REDIM <2,20>	If set, the array can be redimensioned, that is, DSCA_A0, DSCL_Mi, DSC$L_Li, and DSC$L_Ui may be changed. The redimensioned array cannot exceed the size allocated to the array (DSC$L_ARSIZE).
DSC$V_FL_COLUMN <2,21>	If set, the nelements of the array are stored by columns (FORTRAN). That is, the leftmost subscript (first dimension) is varied most rapidly, and the rightmost subscript (nth dimension) is varied least rapidly. If not set, the elements are stored by rows (most other languages). That is, the rightmost subscript is varied most rapidly and the leftmost subscript is varied least rapidly.
DSC$V_FL_COEFF <2,22>	If set, the multiplicative coefficients in Block 2 are present. Must be set if DSC$V_FL_BOUNDS is set.
DSC$V_FL_BOUNDS <2,23>	If set, the bounds information in Block 3 is present and requires that DSC$V_FL_COEFF be set.
DSC$B_DIMCT <2,31:24>	Number of dimensions, n.
DSC$L_ARSIZE <3,31:0>	Total size of array (in bytes unless the DSC$B_TYPE field contains the value 1 or 21, see description for DSC$W_LENGTH). A redimensioned array may use less than the total size allocated.
DSC$A_A0 <4,31:0>	Address of element A(0,0,...,0). This need not be within the actual array. It is the same as DSC$A_POINTER for zero-origin arrays.
DSC$L_Mi <4+i,31:0>	Addressing coefficients. (Mi = Ui-Li+1)
DSC$L_Li <3+n+2*i,31:0>	Lower bound (signed) of ith dimension.
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of ith dimension.

The following formulas specify the effective address, E, of an array element. Modification is required if DSC$B_DTYPE contains a 1 or 21 because DSC$W_LENGTH is given in bits or 4-bit nibbles rather than bytes.

The effective address, E, for element A(I):

E = A0 + I*LENGTH
  = POINTER + [I - L1]*LENGTH

The effective address, E, for element A(I1,I2) with DSC$V_FL_COLUMN clear:

E = A0 + [I1*M2 + I2]*LENGTH
  = POINTER + [[I1-L1]*M2 + I2 - L2]*LENGTH

The effective address, E, for element A(I1,I2) with DSC$V_FL_COLUMN set:

E = A0 + [I2*M1 + I1]*LENGTH
  = POINTER + [[I2-L2]*M1 + I1 - L1]*LENGTH

The effective address, E, for element A(I1, . . . ,In) with DSC$V_FL_COLUMN clear:

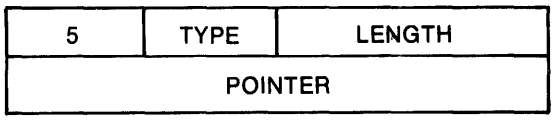
E = A0 + [[[[...[I1]*M2 + ...]*M(n-2) + I(n-2)]*M(n-1)
  + I(n-1)]*Mn + In]*LENGTH
  = POINTER + [[[[...[I1 - L1]*M2 + ...]*M(n-2) + I(n-2)
  - L(n-2)]*M(n-1) + I(n-1) - L(n-1)]*Mn + In - Ln]*LENGTH

The effective address, E, for element A(I1, . . . ,In) with DSC$V_FL_COLUMN set:

E = A0 + [[[[...[In]*M(n-1) + ...]*M3 + I3]*M2 + I2]*M1 + I1]*LENGTH
  = POINTER + [[[[...[In - Ln]*M(n-1) + ...]*M3 + I3 - L3]*M2
  + I2 - L2]*M1 + I1 - L1]*LENGTH

C.8.6 Procedure Descriptor (DSC$K_CLASS_P)

The descriptor for a procedure specifies its entry address and function value data type, if any. A procedure descriptor has the form:



Symbol	Description
DSC$W_LENGTH	Length associated with the function value.
DSC$B_DTYPE	Function value data type code (see Section C.7).
DSC$B_CLASS	5 = DSK$K_CLASS_P.
DSC$A_POINTER	Address of entry mask to routine.

Procedures return a function value in R0, R1/R0, or using the first argument list entry depending on the size of the data type (see Section C.3).

C.8.7 Procedure Incarnation Descriptor (DSC$K_CLASS_PI)

Obsolete.

C.8.8 Label Descriptor (DSC$K_CLASS_J)

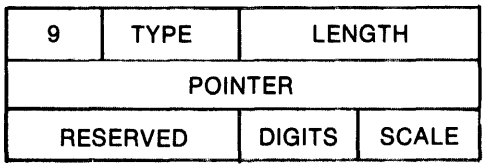
Reserved for use by the VAX-11 Debugger.

C.8.9 Label Incarnation Descriptor (DSC$K_CLASS_JI)

Obsolete.

C.8.10 Decimal Scalar String Descriptor (DSC$K_CLASS_SD)

Decimal size and scaling information for scalar data and simple strings is given in a single descriptor form as follows:



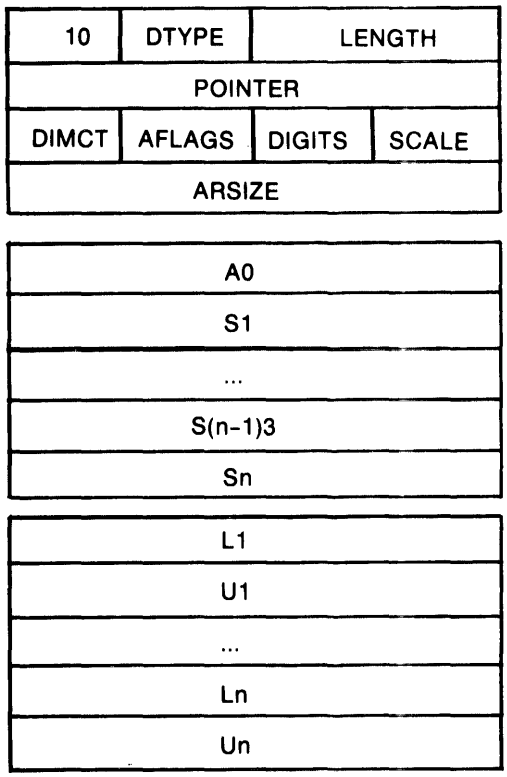
Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one byte data type code (see Section C.7).
DSC$B_CLASS	9 = DSC$K_CLASS_SD.
DSC$A_POINTER	Address of first byte of data storage.
DSC$B_SCALE	Signed power of ten multiplier to convert the internal form to external form. For example, if internal number is 123 and scale is +1, then the external number is 1230.
DSC$B_DIGITS	If non-zero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.
Reserved <2,31:16>	Reserved for future use. Must be zero.

C.8.11 Non-Contiguous Array Descriptor (DSC$K_CLASS_NCA)

The non-contiguous array descriptor describes an array where the storage of the array elements is allocated with a fixed, non-zero number of bytes separating elements. The difference between the addresses of two adjacent elements is termed the stride.

The DSC$K_CLASS_A array descriptor is the preferred array descriptor for passing arrays between separately compiled modules. If an array is contiguous, that is, the stride of the most rapidly varying dimension is equal to the data element size, the array descriptor with DSC$B_CLASS equal to 4 is to be used (see Section C.8.5).

The non-contiguous array descriptor consists of 3 contiguous blocks. The first block contains the descriptor prototype information. A complete non-contiguous array descriptor has the form:



Symbol	Description
DSC$W_LENGTH	Length of data item in bytes, unless the DSC$B_DTYPE field contains the value 1 (bit) or 21 (packed decimal). Length of data item is in bits for bit string. Length of data item is the number of 4-bit digits (not including the sign) for packed decimal string.
DSC$B_DTYPE	A one byte data type code (see Section C.7).
DSC$B_CLASS	10 = DSC$K_CLASS_NCA.
DSC$A_POINTER	Address of first actual byte of data storage.
DSC$B_SCALE	Signed power of ten multiplier to convert the internal form to the external form. For example, if the internal number is 123 and scale is +1, then the external number is 1230.
DSC$B_DIGITS	If non-zero, unsigned number of decimal digits in the internal representation. If zero, the number of digits can be computed based on DSC$W_LENGTH.

Symbol	Description
DSC$B_AFLAGS <2,23:16>	Array flag bits.
Reserved <2,19:16>	Must be zero.
DSC$V_FL_REDIM <2,20>	Must be zero.
DSC$V_FL_COLUMN <2,21>	If set, the elements of the array are stored by columns (FORTRAN). That is, the leftmost subscript (first dimension) is varied most rapidly, and the rightmost subscript (nth dimension) is varied least rapidly. If not set, the elements are stored by rows (most other languages). That is, the rightmost subscript is varied most rapidly and the leftmost subscript is varied least rapidly.
DSC$V_FL_COEFF <2,22>	Always set, the strides in Block 2 must be present.
DSC$V_FL_BOUNDS <2,23>	Always set, the bounds in Block 3 must be present.
DSC$B_DIMCT <2,31:24>	Number of dimensions, n.
DSC$L_ARSIZE <3,31:0>	Must be zero. (Reserved for future standardization by DIGITAL)
DSC$A_A0 <4,31:0>	Address of element A(0,0,...,0). This need not be within the actual array. It is the same as DSC$A_POINTER for zero-origin arrays. DSC$A_A0 = POINTER - (S1*L1 + S2*L2 + ...+Sn*Ln)
DSC$L_Si <4+i,31:0>	Stride of the ith dimension. The difference between the addresses of successive elements of the ith dimension.
DSC$L_L1 <3+n+2*i,31:0>	Lower bound (signed) of the ith dimension.
DSC$L_Ui <4+n+2*i,31:0>	Upper bound (signed) of the ith dimension.

The following formulas specify the effective address, E, of an array element. Modification is required if DSC$B_DTYPE equals 1 or 21 because DSC$W_LENGTH is given in bits or 4-bit nibbles rather than bytes.

The effective address, E, of A(I):

$$E = A0 + S1*I$$
$$= POINTER + S1*[I - L1]$$

The effective address, E, of A(I1,I2):

$$E = A0 + S1*I1 + S2*I2$$
$$= POINTER + S1*[I1 - L1] + S2*[I2 - L2]$$

The effective address, E, of A(I1, . . . ,In):

$$E = A0 + S1*I1 + \ldots + Sn*In$$
$$= POINTER + S1*[I1 - L1] + \ldots + Sn*[In - Ln]$$

C.8.12 Reserved Descriptors

Descriptor classes 11 through 191 are reserved for DIGITAL. Classes 192 through 255 are reserved for DIGITAL's Computer Special System group and customers.

C.9 VAX-11 Conditions

A condition is either:

- A hardware-generated synchronous exception
- A software event that is to be processed in a manner analogous to a hardware exception.

Floating-point overflow trap, memory access violation exception, and reserved operation exception are examples of hardware-generated conditions. An output conversion error, an end-of-file, or the filling of an output buffer are examples of software events that might be treated as conditions.

Depending on the condition and on the program, four types of action can be taken when a condition occurs:

1. *Ignore the condition.* For example, if an underflow occurs in a floating-point operation, continuing from the point of the exception with a zero result may be satisfactory.
2. *Take some special action and then continue from the point at which the condition occurred.* For example, if the end of a buffer is reached while a series of data items are being written, the special action is to start a new buffer.
3. *End the operation and branch from the sequential flow of control.* For example, if the end of an input file is reached, the branch exits from a loop that is processing the input data.
4. *Treat the condition as an unrecoverable error.* For example, when the floating divide by zero exception condition occurs, the program exits, after writing (optionally) an appropriate error message.

When an unusual event orcondition value to the caller indicating what has happened (see Section C.4). The caller tests the condition value and takes the appropriate action.

When an exception is generated by the hardware, a branch out of the program's flow of control occurs automatically. In this case, and for certain software generated events, it is more convenient to handle the condition as soon as it is detected rather than to program explicit tests.

C.9.1 Condition Handlers

To handle hardware- or software-detected exceptions, the VAX-11 Condition Handling Facility allows the programmer to specify a condition handler procedure to be called when an exception condition occurs. This same handler procedure may also be used to handle software-detected conditions.

An active procedure can establish a condition handler to be associated with it. The presence of a condition handler is indicated by a nonzero address in the first longword of the procedure's stack frame. When an event occurs that is to be treated using the condition handling facility, the procedure detecting the event signals the event by calling the facility and passing a condition value describing the condition that occurred. This condition value has the format and interpretation as described in Section C.4. All hardware exceptions are signaled.

When a condition is signaled, the condition handling facility looks for a condition handler in the current procedure's stack frame. If a handler is found, it is entered. If no handler is associated with the current procedure, the immediately preceding stack frame is examined. Again, if a handler is found it is entered. If a handler is not found, the search of previous stack frames continues until the default condition handler established by the system is reached or the stack runs out.

The default condition handler prints messages indicated by the signal argument list by calling the Put Message (SYS$PUTMSG) system service, followed by an optional symbolic stack traceback. Success conditions with STS$K_SUCCESS result in messages to file SYS$OUTPUT only. All other conditions, including informational messages (STS$K_INFO), produce messages on files SYS$OUTPUT and SYS$ERROR.

For example, if a procedure needs to keep track of the occurrence of the floating-point underflow exception, it can establish a condition handler to examine the condition value passed when the handler is invoked. Then when the floating-point underflow exception occurs, the condition handler will be entered and will log the condition. The handler will return to the instruction immediately following the instruction causing the underflow.

If floating-point operations occur in many procedures of a program, the condition handler can be associated with the program's main procedure. When the condition is signaled, successive stack frames are searched until the stack frame for the main procedure is found, at which time the handler will be entered. If a user program has not associated a condition handler with any of the procedures that are active at the time of the signal, successive stack frames will be searched until the frame for the system program invoking the user program is reached. A default condition handler that prints an error message will then be entered.

C.9.2 Condition Handler Options

Each procedure activation potentially has a single condition handler associated with it. This condition handler will be entered whenever any condition is signaled within that procedure. (It can also be entered as a result of signals within active procedures called by the procedure.) Each signal includes a condition value (see Section C.4), which describes the condition causing the signal. When the condition handler is entered, the condition value should be

examined to determine the cause of the signal. After the handler has processed the condition or chosen to ignore it, it can:

- Return to the instruction immediately following the signal. Note that it is not always possible to make such a return.
- Resignal the same or a modified condition value. A new search for a condition handler will begin with the immediately preceding stack frame.
- Signal a different condition.
- Unwind the stack.

C.10 Operations Involving Condition Handlers

The functions provided by the VAX-11 Condition Handling Facility are to:

1. *Establish a condition handler.* A condition handler is associated with the current procedure by placing the handler's address in the current procedure's activation stack frame.
2. *Revert to the caller's handling.* If a condition handler has been established, it can be removed by clearing its address in the current procedure activation's stack frame.
3. *Enable or disable certain arithmetic exceptions.* The following hardware exceptions can be enabled or disabled by software: floating-point underflow, integer overflow, and decimal overflow. No signal occurs when the exception is disabled.
4. *Signal a condition.* Signaling a condition initiates the search for an established condition handler.
5. *Unwind the stack.* Upon exit from a condition handler it is possible to remove one or more frames occurring before the signal from the stack. During the unwinding operation, the stack is scanned and if a condition handler is associated with a frame, that handler is entered before the frame is removed. Unwinding the stack allows a procedure to perform application specific cleanup operations before exiting.

C.10.1 Establish a Condition Handler

Each procedure activation has a condition handler potentially associated with it using longword 0 in its stack frame. Initially, longword 0 contains 0, indicating no handler. A handler is established by moving the address of the handler's procedure entry point mask to the establisher's stack frame.

In addition, VAX/VMS provides three statically allocated exception vectors for each access mode of a process. These vectors are available to declare condition handlers that take precedence over any handlers declared at the procedure level. These are used, for example, to allow a debugger to monitor all exceptions and for the system to establish a last chance handler. Since these handlers do not obey the procedure nesting rules, they should not be used by modular code. Instead the stack based declaration should be used.

The code to establish a condition handler is:

```
MOVAB handler_entry_point,0(FP)
```

C.10.2 Revert to the Caller's Handling

Reverting to the caller's handling deletes the condition handler associated with the current procedure activation. This is done by clearing the handler address in the stack frame.

The code to revert to the caller's handling is:

```
CLRL 0(FP)
```

C.10.3 Signal a Condition

The signal operation is the method used for indicating the occurrence of an exception condition. To issue a message and be able to continue execution after handling the condition, a program calls the LIB$SIGNAL procedure as follows:

```
CALL LIB$SIGNAL (condition_value, arg_list...)
```

To issue a message, but not continue execution, a program calls LIB$STOP, as follows:

```
CALL LIB$STOP (condition_value, arg_list...)
```

In both cases, condition_value indicates the condition that is signaled. However, LIB$STOP sets the severity of the condition_value to be a severe_error. The remaining arguments describe the details of the exception. These are the same arguments used to issue a system message.

Note that unlike most calls, LIB$SIGNAL and LIB$STOP preserve R0 and R1 as well as the other registers. Therefore, a debugger can insert a call to LIB$SIGNAL to display the entire state of the process at the time of the exception. It also allows signals to be coded in MACRO without changing the register usage. This feature of preserving R0 and R1 is is useful for debugging checks and gathering statistics. Hardware and system service exceptions behave like calls to LIB$SIGNAL.

The signal procedure examines the two exception vectors, and then up to 64K previous stack frames, and finally the last-chance exception vector, if necessary. The current and previous stack frames are found by using FP and chaining back through the stack frames using the saved FP in each frame. The exception vectors have three address locations per access mode.

As a part of image start-up, the system declares a default last-chance handler. This handler is used as a last resort when the normal handlers are not performing correctly. The debugger can replace the default system last-chance handler with its own.

In some frame before the call to the main program, the system establishes a default catch-all condition handler that issues system messages. In a

subsequent frame before the call to the main program, the system usually establishes a traceback handler. These system-supplied condition handlers use condition_value to get the message and then use the remainder of the argument list to format and output the message through the system service, SYS$PUTMSG.

If the severity field of the condition_value (bits 0 through 2) does not indicate a severe_error (that is, a value of 4) these default condition handlers return with SS$_CONTINUE. If the severity is severe_error, these default handlers exit the program image with the condition value as the final image status.

The stack search ends when the old FP is 0 or is not accessible, or when 64K frames have been examined. If no condition handler is found, or all handlers returned with a SS$_RESIGNAL, then the vectored last-chance handler is called.

If a handler returns SS$_CONTINUE, and LIB$STOP was not called, control returns to the signaler. Otherwise LIB$STOP issues a message that an attempt was made to continue from a noncontinuable exception and exits with the condition value as the final image status.

Table C-1 lists all combinations of interaction between condition handler actions, the default condition handlers, the type of signals, and the call to signal or stop. In the table, "cannot continue" indicates an error which results in the message: ATTEMPT TO CONTINUE FROM STOP.

Table C-1: Interaction between Handlers and Default Handlers

Call to:	Signaled Condition Severity <2:0>	Default Handler Gets Control	Handler Specifies Continue	Handler Specifies UNWIND	No Handler Is Found (stack bad)
LIB$SIGNAL or hardware exception	<4	condition message RET	RET	UNWIND	Call last chance handler EXIT
	=4	condition message EXIT	RET	UNWIND	Call last chance handler EXIT
LIB$STOP	force (=4)	condition message EXIT	"cannot continue" EXIT	UNWIND	Call last chance handler EXIT

C.11 Properties of Condition Handlers

C.11.1 Condition Handler Parameters and Invocation

If a condition handler is found on a software detected exception, the handler is called with an argument list consisting of:

continue = handler (signal_args, mechanism_args)

Each argument is a reference to a longword vector. The first longword of each vector is the number of remaining longwords in the vector. The symbols CHF$L_SIGARGLST (=4) and CHF$L_MCHARGLST (=8) can be used to access the condition handler arguments relative to AP.

Signal_args is the condition argument list from the call to LIB$SIGNAL or LIB$STOP expanded to include the PC and PSL of the next instruction to execute on a continue. In particular, the second longword is the condition_value being signaled.

Because bits 0 through 2 of the condition_value indicate severity and do not indicate which condition is being signaled, the handler should examine only the condition identification, that is, condition value (bits 3 through 27). The setting of bits 0 through 2 varies depending upon the environment. In fact, some handlers may simply change the severity of a condition and resignal. The symbols CHF$L_SIG_ARGS (=0) and CHF$L_SIG_NAME (=4) can be used to refer to the elements of the signal vectors.

Mechanism_args is a five-longword vector:

4	CHF$L_MCH_ARGS
frame	CHF$L_MCH_FRAME
depth	CHF$L_MCH_DEPTH
R0	CHF$L_MCH_SAVR0
R1	CHF$L_MCH_SAVR1

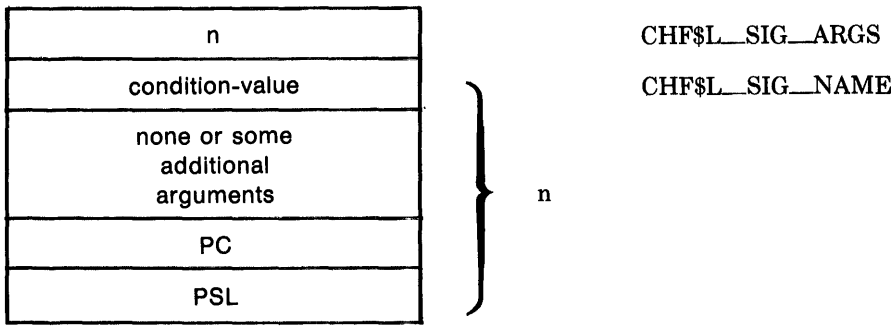
The frame is the contents of the FP in the establisher's context. This can be used as a base to access the local storage of the establisher if the restrictions described in Section C.11.2 are met.

The depth is a positive count of the number of procedure activation stack frames between the frame in which the exception occurred and the frame depth that established the handler being called. Depth has the value 0 for an exception handled by the procedure activation invoking the exception (that is, containing the instruction causing the hardware exception or calling LIB$SIGNAL). Depth has positive values for procedure activations calling the one having the exception (1 for the immediate caller, etc.).

If a system service gives an exception, the immediate caller of the service is notified at depth = 1. Depth has value -2 when the condition handler is established by the primary exception vector, -1 when it is established by the secondary vector, and -3 when it is established by the last-chance vector.

The contents of R0 and R1 are the same as at the time of the call to LIB$SIGNAL or LIB$STOP.

For hardware detected exceptions, the condition-value indicates which exception vector was taken and the next 0 or several longwords are additional parameters. The remaining two longwords are the PC and PSL:



If one of the default condition handlers established by the system is entered, it calls the system service, SYS$PUTMSG, to interpret the signal argument list and output the indicated information or error message. See the description of SYS$PUTMSG in the *VAX/VMS Systems Services Reference Manual* for the format of the signal argument list.

C.11.2 Use of Memory

A condition handler and procedures it calls are restricted to referring to explicitly passed arguments only. Handlers cannot refer to common or other external storage, and they cannot reference local storage in the procedure that established the handler. The existence of handlers does not affect compiler optimization. Compilers that do not follow this rule must ensure that any variables referred to by the handler are always in memory.

C.11.3 Returning from a Condition Handler

Condition handlers are invoked by the VAX-11 Condition Handling Facility. Therefore, the return from the condition handler is to the condition handling facility.

To continue from the instruction following the signal, the handler must return with the function value SS$_CONTINUE ("true," that is, with bit 0 set). If, however, the condition was signaled with a call to LIB$STOP, the image will exit. To resignal the condition, the condition handler returns with the function value SS$_RESIGNAL ("false," that is, with bit 0 clear). To alter the severity of the signal, the handler modifies the low-order three bits of the condition-value longword in the signal-args vector and resignals. If the condition handler wants to alter the defined control bits of the signal, the handler modifies bits 31:28 of condition-value and resignals. To unwind, the handler calls SYS$UNWIND and then returns. In this case, the handler function value is ignored.

C.11.4 Request to Unwind

To unwind, the handler or any procedure it calls can perform:

```
success = SYS$UNWIND
                ( [depadr =  handler depth  + 1],
                  [new_PC =  return PC ] )
```

The argument depadr specifies the address of a longword that contains the number of presignal frames (depth) to be removed. If that number is less than or equal to 0 then nothing is to be unwound. The default (address = 0) is to return to the caller of the procedure that established the handler that issued the $UNWIND service. To unwind to the establisher, the depth from the call to the handler should be specified. When the handler is at depth 0, it can achieve the equivalent of an unwind operation to an arbitrary place in its establisher by altering the PC in its signal-args vector and returning with SS$_CONTINUE instead of performing an unwind.

The argument new_PC specifies the location to receive control when the unwinding operation is complete. The default is to continue at the instruction following the call to the last procedure activation removed from the stack.

The function value SUCCESS is a standard success code (SS$_NORMAL), or indicates failure with one of the following return status condition values:

- No signal active (SS$_NOSIGNAL)
- Already unwinding (SS$_UNWINDING)
- Insufficient frames for depth (SS$_INSFRAME)

The unwinding operation occurs when the handler returns to the condition handling facility. Unwinding is done by scanning back through the stack and calling each handler that has been associated with a frame. The handler is called with exception SS$_UNWIND to perform any application specific cleanup. In particular, if the depth specified includes unwinding the establisher's frame, the current handler will be recalled with this unwind exception.

The call to the handler takes the same form as previously described, with the following values:

```
signal_args
        1
        condition_value = SS$_UNWIND

mechanism_args
        4
        frame   establisher's frame
        depth   0 (that is, unwinding self)
        R0      R0 that unwind will restore
        R1      R1 that unwind will restore
```

After each handler is called, the stack is cut back to the previous frame.

Note that the exception vectors are not checked because they are not being removed. Any function value from the handler is ignored. To specify the value of the top level "function" being unwound, the handler should modify R0 and R1 in the mechanism__args vector. They will be restored from the mechanism__args vector at the end of the unwind. Depending on the arguments to SYS$UNWIND, the unwinding operation will be terminated as follows:

- SYS$UNWIND(0,0) — unwind to the establisher's caller with the establisher function value restored from R0 and R1 in the mechanism-args vector.
- SYS$UNWIND(depth,0) — unwind to the establisher at the point of the call that resulted in the exception. The contents of R0 and R1 are restored from R0 and R1 in the mechanism__args vector.
- SYS$UNWIND(depth,location) — unwind to the specified procedure activation and transfer to a specified location with the contents of R0 and R1 from R0 and R1 in the mechanism__args vector.

SYS$UNWIND can be called whether the condition was a software exception signaled by calling LIB$SIGNAL or LIB$STOP, or was a hardware exception. Calling SYS$UNWIND is the only way to continue execution after a call to LIB$STOP.

C.11.5 Signaler's Registers

Because the handler is called, and can in turn call routines, the actual values of the registers that were in use at the time of the signal or exception can be scattered on the stack. To find the registers R2 through FP, a scan of stack frames must be performed starting with the current frame and ending with the call to the handler. During the scan, the last frame found to save a register contains that register's contents at the time of the exception. If no frame saved the register, the register is still active in the current procedure. The frame of the call to the handler can be identified by the return address of SYS$CALL__HANDL+4. Thus, the registers are:

R0, R1	in mechanism__args
R2..R11	last frame saving it
AP	old AP of SYS$CALL__HANDL+4 frame
FP	old FP of SYS$CALL__HANDL+4 frame
SP	equal to end of signal-args vector+4
PC, PSL	at end of signal-args vector

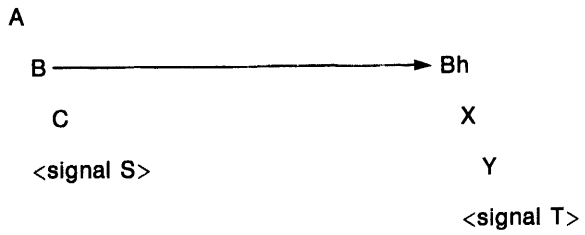
C.12 Multiple Active Signals

A signal is said to be active until the signaler gets control again or is unwound. A signal can occur while a condition handler or a procedure it has called is executing in response to a previous signal. For example, procedure (A, B, C, ...) establishes a condition handler (Ah, Bh, Ch, ...). If A calls B and B calls C

which signals S and Ch resignals, then Bh gets control. If Bh calls procedure X and X calls procedure Y and Y signals T the stack is:

```
<signal T>
    Y
    X
    Bh
<signal S>
    C
    B
    A
```

which was programmed:



The handlers are searched for in the order: Yh, Xh, Bhh, Ah. Note that Ch is not called because it is a structural descendant of B. Bh is not called again because that would require it to be recursive. Recursive handlers could not be coded in nonrecursive languages such as FORTRAN. Instead, Bh can establish itself or another procedure as its handler (Bhh).

The following algorithm is used on the second and subsequent signals which occur before the handler for the original signal returns to the condition handling facility. The primary and secondary exception vectors are checked. Then, however, the search backward in the process stack is modified. In effect, the stack frames traversed in the first search are skipped over in the second search. Thus, the stack frame preceding the first condition handler up to and including the frame of the procedure that has established the handler is skipped. Despite this skipping, depth is not incremented. The stack frames traversed in the first and second search are skipped over in a third search, etc. Note that if a condition handler signals, it will not automatically be invoked recursively. However, if a handler itself establishes a handler, this second handler will be invoked. Thus, a recursive condition handler should start by establishing itself. Any procedures invoked by the handler are treated in the normal way that is, exception signaling follows the stack up to the condition handler.

If an unwinding operation is requested while multiple signals are active, all the intermediate handlers are called for the operation. For example, in the above diagram, if Ah specifies unwinding to A, the following handlers will be called for the unwind: Yh, Xh, Bhh, Ch, and Bh.

For proper hierarchical operation, an exception that occurs during execution of a condition handler established in an exception vector should be handled by that handler rather than propagating up the activation stack. To prevent such propagation, the vectored condition handler should establish a handler in its stack frame to handle all exceptions.

C.13 Change History

The following changes have been made to the VAX-11 Procedure Calling and Condition Handling Standard from revision 4 in the *VAX-11 Run-Time Library Reference Manual* (AA-D036A-TE, August 1978).

Rev 6 to Rev 7:

1. Editorial changes.
2. Clarified the names of the data types, including spelling.
3. Indicated that the Procedure Incarnation Description (Class = 6) and Label Incarnation Description (Class = 8) are obsolete since they are not used by any currently planned languages.
4. Indicated that the Label Descriptor (Class = 7) is reserved for use by the VAX-11 Debugger.
5. Changed ARSIZE field in non-contiguous array descriptor to 'must be zero' since it is not meaningful for arrays that are actually non-contiguous.

Rev 5 to Rev 6:

1. Added REDIM flag to array descriptor.
2. Added descriptor data type.
3. Added non-contiguous array.
4. Added COBOL intermediate temporary data type.
5. Indicated that the by-value mechanism is for calling the operating system or for use in programs written in MACRO or BLISS. In order to reduce confusion with languages implementing by-value semantics (which now must use the reference mechanism), the term by-value mechanism has been changed to immediate value mechanism.
6. Removed the references to specific languages, except examples and historical references.
7. Clarified that the use of CALLG vs CALLS is an option of the calling program.
8. Indicated that language extensions to force mechanism could be achieved in external declaration.
9. Added bound procedure value data type.
10. Added protocol for calling a procedure requiring an environment value in R1. (bound procedure value)
11. Clarified that AP is a free temporary.
12. Divided data types into 3 categories: atomic, string, and miscellaneous.

Rev 4 to Rev 5:

1. Added octaword, G_floating, H_floating data types.
2. Added Decimal Scalar String Descriptor.

Appendix D
Algorithms for Mathematics Procedures

This appendix presents the algorithms of the mathematics procedures described in Chapter 4.

D.1 Floating Mathematical Functions

D.1.1 Arc Cosine

ACOS(X) is computed as:

If X = 0, then ACOS(X) = PI/2
If X = 1, then ACOS(X) = 0
If X = -1, then ACOS(X) = PI
If 0 < X < 1, then ACOS(X) = ATAN(SQRT(1-X**2)/X)
If -1 < X <0, then ACOS(X) = ATAN(SQRT(1-X**2)/X) + PI
If 1 < |X|, error

DACOS(X) is computed as:

If X = 0, then DACOS(X) = PI/2
If X = 1, then DACOS(X) = 0
If X = -1, then DACOS(X) = PI
If 0 < X < 1, then DACOS(X) = DATAN(DSQRT(1-X**2)/X)
If -1 < X < 0, then DACOS(X) = DATAN(DQSRT(1-X**2)/X) + PI
If 1 < |X|, error

GACOS(X) is computed as:

If X = 0, then GACOS(X) = PI/2
If X = 1, then GACOS(X) = 0
If X = -1, then GACOS(X) = PI
If 0 < X < 1, then GACOS(X) = GATAN(GSQRT(1-X**2)/X)
If -1 < X < 0, then GACOS(X) = GATAN(GSQRT(1-X**2/X) + PI
If 1 < |X|, error

HACOS(X) is computed as:

If X = 0, then HACOS(X) = PI/2
If X = 1, then HACOS(X) = 0
If X = -1, then HACOS(X) = PI
If 0 < X < 1, then HACOS(X) = HATAN(HSQRT(1-X**2)/X)
If -1 < X < 0, then HACOS(X) = HATAN(HSQRT(1-X**2)/X) + PI
If 1 < |X|, error

D.1.2 Arc Sine

ASIN(X) is computed as:

If X = 0, then ASIN(X) = 0
If X = 1, then ASIN(X) = PI/2
If X = -1, then ASIN(X) = -PI/2
If 0 < |X| < 1, then ASIN(X) = ATAN(X/SQRT(1-X**2))
If 1 < |X|, error

DASIN(X) is computed as:

If X = 0, then DASIN(X) = 0
If X = 1, then DASIN(X) = PI/2
If X = -1, then DASIN(X) = -PI/2
If 0 < |X| < 1, then DASIN(X) = DATAN(X/DSQRT(1-X**2))
If 1 < |X|, error

GASIN(X) is computed as:

If X = 0, then GASIN(X) = 0
If X = 1, then GASIN(X) = PI/2
If X = -1, then GASIN(X) = -PI/2
If 0 < |X| < 1, then GASIN(X) = GATAN(X/GSQRT(1-X**2))
If 1 < |X|, error

HASIN(X) is computed as:

If X = 0, then HASIN(X) = 0
If X = 1, then HASIN(X) = PI/2
If X = -1, then HASIN(X) = -PI/2
If 0 < |X| < 1, then HASIN(X) = HATAN(X/HSQRT(1-X**2))
If 1 < |X|, error

D.1.3 Arc Tangent

ATAN(X) is computed as:

1. If X < 0, then
   Begin
   Perform Steps 2, 3, and 4 with arg = |X|
   Negate the result since ATAN(X) = -ATAN(-X)
   Return
   End

2. If X > 1, then
   Begin
   Perform Steps 3 and 4 with Arg = 1/|X|
   Negate result and add a bias of PI/2 since
   ATAN(|X|) = PI/2 - ATAN(1/|X|)
   Return
   End
3. At this point the argument is 1 >= X >= 0
   If |X| > TAN(PI/12), then:
   Begin
   Perform Step 4 with arg = (X * SQRT(3) - 1)/
   (SQRT(3) + X)
   Add PI/6 to the result
   Return
   End
   Note: (X * SQRT(3) -1)/(X + SQRT(3)) >= TAN(PI/12) for
   |X| >= TAN(PI/12)
4. Finally, the argument is |X| >= TAN(PI/12)
   Begin
   ATAN(X) = X * SUM(C[i] * X**(2*i)), i = 0:4
   Return
   End
   The coefficients C[i] are drawn from Hart #4941.

DATAN(X) is computed as:

1. If X < 0, then
   Begin
   Perform Steps 2, 3, and 4 with Arg = |X|
   Negate the result since DATAN(X) = -DATAN(-X)
   Return
   End
2. At this point the argument is positive or has been made positive.
   If X > 1, then:
   Begin
   Perform Steps 3 and 4 with arg = 1/|X|.
   Negate result and add a bias of PI/2 since
   DATAN(|X|) = PI/2 - DATAN(1/|X|)
   Return
   End
3. At this point the argument is 1 >= X >= 0
   If |X| > DTAN(PI/12 then:
   Begin
   Perform Step 4 with Arg = (X*DSQRT(3) - 1)/
   (DSQRT(3) + X)
   Add PI/6 to the result
   Return
   End

Note: (X*DSQRT(3) -1)/(X + DSQRT(3)) >= DTAN(PI/12 for
|X| >= DTAN(PI/12)

4. Finally, the argument is |X| >= DTAN(PI/12): Begin
   DATAN(X) = X * SUM(C[i] * X**(2*i)), i = 0:8
   Return
   End

   The coefficient C[i]'s are drawn from Hart #4941.

GATAN(X) is computed as:

1. If X < 0, then
   Begin
   Perform Steps 2, 3, and 4 with Arg = |X|
   Negate the result since GATAN(X) = -GATAN(-X)
   Return
   End

2. At this point the argument is positive or has been made positive.
   If X > 1, then:
   Begin
   Perform Steps 3 and 4 with arg = 1/|X|.
   Negate result and add a bias of PI/2 since
   GATAN(|X|) = PI/2 - GATAN(1/|X|)
   Return
   End

3. At this point the argument is 1 >= X >= 0
   If |X| > GTAN(PI/12 then:
   Begin
   Perform Step 4 with Arg = (X*GSQRT(3) - 1)/
   (GSQRT(3) + X)
   Add PI/6 to the result
   Return
   End

   Note: (X*GSQRT(3) -1)/(X + GSQRT(3)) >= GTAN(PI/12) for
   |X| >= GTAN(PI/12)

4. Finally, the argument is |X| >= GTAN(PI/12):
   Begin
   GATAN(X) = X * SUM(C[i] * X**(2*i)), i = 0:8
   Return
   End

   The coefficient C[i]'s are drawn from Hart #4941.

HATAN(X) is computed as:

1. If X = 0 then return HATAN(0) = 0
   If X < 0 then
   Begin

Perform steps 2 and 3 with arg = |X|
Negate the result since HATAN(X) = -HATAN(-X)
Return
End

2. At this point the argument is positive or has been made positive.
If X = 1 then return HATAN(1) = PI/4
If X > 1 then
Begin
Perform step 3 with arg = 1/|X|
Negate result and add a bias of PI/2 since HATAN(|X|) =
PI/2 - HATAN(1/|X|)
Return
End

3. At this point the argument is 0 <= X < 1.
Compute HATAN(X) = HATAN(XHI) + HATAN(XLO/(1 + X**XHI))
where:
XHI = INT(X*16)/16
XLO = X - XHI

HATAN(XLO/(1 + X*XHI)) = polynomial approximation of degree 11

D.1.4 Arc Tangent with Two Parameters

ATAN2(X,Y) is computed as:

If Y = 0 or X/Y > 2**25, ATAN(X,Y) = PI/2 * (sign X)
If Y > 0 and X/Y =< 2**25, ATAN2(X,Y) = ATAN(X/Y)
If Y < 0 and X/Y =< 2**25, ATAN2(X,Y) = PI * (sign X)
+ ATAN(X/Y)

DATAN2(X,Y) is computed as:

If Y = 0 or X/Y > 2**57, DATAN2(X,Y) = PI/2 * (Sign X)
If Y > 0 and X/Y =< 2**57, DATAN2(X,Y) = DATAN(X/Y)
If Y < 0 and X/Y =< 2**57, DATAN2(X,Y) = PI * (Sign X)
+ DATAN(X/Y)

GATAN2(X,Y) is computed as:

If Y = 0 or X/Y > 2**57, GATAN2(X,Y) = PI/2 * (Sign X)
If Y > 0 and X/Y =< 2**57, GATAN2(X,Y) = GATAN(X/Y)
If Y < 0 and X/Y =< 2**57, GATAN2(X,Y) = PI * (Sign X)
+ GATAN(X/Y)

HATAN2(X,Y) is computed as:

If Y = 0 or X/Y > 2**114, HATAN2(X,Y) = PI/2 * (Sign X)
If Y > 0 and X/Y =< 2**114, HATAN2(X,Y) = HATAN(X/Y)
If Y < 0 and X/Y =< 2**114, HATAN2(X,Y) = PI * (Sign X)
+ HATAN(X/Y)

D.1.5 Common Logarithm

ALOG10(X) is computed as:

ALOG10(E) * ALOG(X)

DLOG10(X) is computed as:

DLOG10(E) * DLOG(X)

GLOG10(X) is computed as:

GLOG10(E) * GLOG(X)

HLOG10(X) is computed as:

HLOG10(E) * HLOG(X)

where:

E = 2.718, the base of the natural log system.

See the description of Natural Logarithm (Section D.1.11) for the complete algorithm.

D.1.6 Cosine

COS(X) is computed as:

SIN(X + PI/2)

See the description of SIN(X) (Section D.1.12) for the complete algorithm.

DCOS(X) is computed as:

DSIN(X+PI/2)

See the description of DSIN(X) (Section D.1.12) for the complete algorithm.

GCOS(X) is computed as:

GSIN(X+PI/2)

See the description of GSIN(X) (Section D.1.12) for the complete algorithm.

HCOS(X) is computed as:

HSIN(X+PI/2)

See the description of HSIN(X) (Section D.1.12) for the complete algorithm.

D.1.7 Exponential

EXP(X) is computed as:

If X > 88.028, overflow occurs
If X <= -89.416, EXP(X) = 0
If |X| < 2**-28, EXP(X) = 1

Otherwise:

EXP(X) = 2**Y * 2**Z * 2**W

where:

Y = INTEGER(X*LOG2(E)
V = FRAC(X*LOG2(E)) * 16
Z = INTEGER(V)/16
W = FRAC(V)/16

2**W = polynomial approximation of degree 4

DEXP(X) is computed as:

If X > 88.028, overflow occurs
If X <= -89.416, DEXP(X) = 0
If |X| < 2**-28, DEXP(X) = 1

Otherwise:

DEXP(X) = 2**Y * 2**Z * 2**W

where:

Y = INTEGER(X*LOG2(E)
V = FRAC(X*LOG2(E)) * 16
Z = INTEGER(V)/16
W = FRAC(V)/16

2**W = polynomial approximation of degree 8

GEXP(X) is computed as:

If X > 709.08, overflow occurs
If X <= -709.79, GEXP(X) = 0
If |X| < 2**-28, GEXP(X) = 1

Otherwise:

GEXP(X) = 2**Y * 2**Z * 2**W

where:

Y = INTEGER(X*LOG2(E)
V = FRAC(X*LOG2(E)) * 16
Z = INTEGER(V)/16
W = FRAC(V)/16

2**W = polynomial approximation of degree 8

HEXP(X) is computed as:

If X > 11355.83, overflow occurs
If X <= -11356.52, HEXP(X) = 0
If |X| < 2**-114, HEXP(X) = 1

Otherwise:

HEXP(X) = 2**Y * 2**Z * 2**W

where:

Y = INTEGER(X*HLOG2(E))
V = FRAC(X*HLOG2(E)) * 16
Z = INTEGER(V)/16
W = FRAC(V)/16

2**W = polynomial approximation of degree 14

D.1.8 Hyperbolic Cosine

COSH(X) is computed as:

If |X| < 2**-11, COSH(X) = 1

If 2**-11 =< |X| < 0.25,
COSH(X) = polynomial approximation of degree 3

If 0.25 =< |X| =< 87.0,
COSH(X) = (EXP(X) + EXP(-X))/2

If 87.0 < |X| and |X| - LOG(2) < 87,
COSH(X) = EXP(|X| - LOG(2))

If 87.0 < |X| and |X| - LOG(2) >= 87, then overflow

DCOSH(X) is computed as:

If |X| < 2**-27, DCOSH(X) = 1

If 2**-27 =< |X| < 0.25,
DCOSH(X) = polynomial approximation of degree 5

If 0.25 =< |X| =< 87.0,
DCOSH(X) = (DEXP(X) + DEXP(-X))/2

If 87.0 < |X| and |X| - LOG(2) < 87,
DCOSH(X) = DEXP(|X| - LOG(2))

If 87.0 < |X| and |X| - LOG(2) >= 87, then overflow

GCOSH(X) is computed as:

If |X| < 2**-27, GCOSH(X) = 1

If 2**-27 =< |X| < 0.25,
GCOSH(X) = polynomial approximation of degree 5

If 0.25 =< |X| =< 709.0,
GCOSH(X) = (GEXP(X) + GEXP(-X))/2

If 709.0 < |X| and |X| - LOG(2) < 709,
GCOSH(X) = GEXP(|X| - LOG(2))

If 709.0 < |X| and |X| - LOG(2) >= 709, then overflow

HCOSH(X) is computed as:

If |X| < 2**-56, HCOSH(X) = 1

If 2**-56 < |X| < 0.25,
HCOSH(X) = polynomial approximation of degree 13

If 0.25 =< |X| <= 11355.0,
HCOSH(X) = (HEXP(X) + HEXP(-X))/2

If 11355.0 < |X| and |X| - HLOG(2) < 11355.0
HCOSH(X) = HEXP(|X| - HLOG(2))

If 11355.0 < |X| and |X| - HLOG(2) >= 11355.0, then overflow

D.1.9 Hyperbolic Sine

SINH(X) is computed as:

If |X| < 2**-11, SINH(X) = X

If 2**-11 =< |X| < 0.25,
SINH(X) = polynomial approximation of degree 3

If 0.25 =< |X| =< 87.0,
SINH(X) = (EXP(X) - EXP(-X))/2

If 87.0 < |X| and |X| - LOG(2) < 87,
SINH(X) = sign(X) * EXP(|X| - LOG(2))

If 87.0 < |X| and |X| - LOG(2) >= 87, then overflow

DSINH(X) is computed as:

If |X| < 2**-27, DSINH(X) = X

If 2**-27 =< |X| < 0.25,
DSINH(X) = polynomial approximation of degree 5

If 0.25 =< |X| =< 87.0,
DSINH(X) = (DEXP(X) - DEXP(-X))/2

If 87.0 < |X| and |X| - LOG(2) < 87,
DSINH(X) = sign(X) * DEXP(|X| - LOG(2))

If 87.0 < |X| and |X| - LOG(2) >= 87, then overflow

GSINH(X) is computed as:

If |X| < 2**-27, GSINH(X) = X

If 2**-27 =< |X| < 0.25,
GSINH(X) = polynomial approximation of degree 5

If 0.25 =< |X| =< 709.0,
GSINH(X) = (GEXP(X) - GEXP(-X))/2

If 709.0 < |X| and |X| - LOG(2) < 709,
GSINH(X) = sign(X) * GEXP(|X| - LOG(2))

If 709.0 < |X| and |X| - LOG(2) >= 709, then overflow

HSINH(X) is computed as:

If |X| < 2**-56, HSINH(X) = X

If 2**-56 <= |X| < 0.25,
HSINH(X) = polynomial approximation of degree 12

If 0.25 <= |X| <= 11355.0,
HSINH(X) = (HEXP(X) - HEXP(-X))/2

If 11355.0 < |X| and |X| - HLOG(2) < 11355.0,
HSINH(X) = sign(X) * HEXP(|X| - HLOG(2))

If 11355.0 < |X| and |X| - HLOG(2) >= 11355.0, then overflow

D.1.10 Hyperbolic Tangent

TANH(X) is computed as:

If |X| =< 2**-14, then TANH(X) = X

If 2**-14 < |X| =< 0.25, then TANH(X) = SINH(X) / COSH(X)

If 0.25 < |X| < 16.0, then
TANH(X) = (EXP(2*X) - 1)/(EXP(2*X) + 1)

If 16.0 =< |X|, then TANH(X) = sign(X) * 1

DTANH(X) is computed as:

If |X| =< 2**-14, then DTANH(X) = X

If 2**-14 < |X| =< 0.25, then DTANH(X) = DSINH(X)/DCOSH(X)

If 0.25 < |X| < 16.0, then
DTANH(X) = (DEXP(2*X) - 1)/(DEXP(2*X) + 1)

If 16.0 =< |X|, then DTANH(X) = sign(X) * 1

GTANH(X) is computed as:

If |X| =< 2**-14, then GTANH(X) = X

If 2**-14 < |X| =< 0.25, then GTANH(X) = GSINH(X)/GCOSH(X)

If 0.25 < |X| < 16.0, then
GTANH(X) = (GEXP(2*X) - 1)/(GEXP(2*X) + 1)

If 16.0 =< |X|, then GTANH(X) = sign(X) * 1

HTANH(X) is computed as:

If |X| <= 2**-59, then HTANH(X) = X

If 2**-59 < |X| <= 0.25, then HTANH(X) = HSINH(X)/HCOSH(X)

If 0.25 < |X| < 16.0, then
HTANH(X) = (HEXP(2*X) - 1)/(HEXP(2*X) + 1)

If 16.0 <= |X|, then HTANH(X) = sign(X) * 1

D.1.11 Natural Logarithm

ALOG(X) is computed as:

If X =< 0, an error is signaled

Therefore, let X = Y * (2**A)
where
1/2 =< Y < 1.

Then, LOG(X) = A * LOG(2) + LOG (Y)

If |X-1| =< 0.25, let W = (X-1)/(X+1)

Then, LOG (X) = W * SUM(C[i] * W**(2*i))

Otherwise, let W = (Y-SQRT(2)/2)/(Y+SQRT(2)/2)

Then, LOG(X) = A * LOG(2) - 1/2 * LOG(2) +
W * SUM C[i] * W**2(2*i)

The coefficients are drawn from Hart #2662.
The polynomial approximation used is of degree 3.

DLOG(X) is computed as:

If X =< 0, an error is signaled

Therefore, let X = Y * (2**A)
where:
1/2 =< Y < 1

Then, DLOG(X) = A * DLOG(2) + DLOG(Y)

If |X-1| <= 0.25, then let W = (X-1)/(X+1)

Then DLOG(X) = W * SUM (C[i] * W**(2*i))

Otherwise, let W = (Y - DSQRT(2)/2)/(Y + DSQRT(2)/2)

Then DLOG(X) = A * DLOG(2) - 1/2 * DLOG(2) +

W * SUM(C[i] * W**(2*i)

The coefficients are drawn from Hart #2662.
The polynomial approximation used is of degree 6.

GLOG(X) is computed as:

If X =< 0, an error is signaled

Therefore, let X = Y * (2**A)
where:
1/2 =< Y < 1

Then, GLOG(X) = A * GLOG(2) + GLOG(Y)

If |X-1| <= 0.25, then let W = (X-1)/(X+1)

Then GLOG(X) = W * SUM (C[i] * W**(2*i))

Otherwise, let W = (Y - GSQRT(2)/2)/(Y + GSQRT(2)/2)

Then GLOG(X) = A * GLOG(2) - 1/2 * GLOG(2) +
W * SUM(C[i] * W**(2*i)

The coefficients are drawn from Hart #2662.
The polynomial approximation used is of degree 6.

HLOG(X) is computed as:

If X <= 0, an error is signaled

Therefore, let X = Y * (2**A)
where:
1/2 <= Y < 1

Then, HLOG(X) = A * HLOG(2) + HLOG(Y)

= A * HLOG(2) + HLOG(YHI + YLO)

where:
YHI = INTEGER(Y*32)/32
YLO = Y - YHI

= A * HLOG(2) + HLOG(YHI * (1+YLO/YHI))

= A * HLOG(2) + HLOG(YHI + HLOG(1 + YLO/YHI)

where:
HLOG(1 + YLO/YHI) = polynomial approximation of degree 22

D.1.12 Sine

SIN(X) is computed as:

If |X| < 2**-14, return X for SIN, 1 for COS

If 2**-14 < |X| <1/2, calculate SIN/COS from series approximations listed below

If 1/2 < |X| < 2**23, let I = INTEGER (|X| / (PI/4))
Y = FRAC (|X| /(PI/4))

The low three bits of I determine the octant in which the reduced argument lies. An eight-way branch indexed by these bits is made for evaluation of the desired function by special series approximations. There are three separate ways in which the branch is made: one for COS, one for SIN with a positive argument, and one for SIN with a negative argument.

All evaluations are carried out so that the last step is an addition of two numbers, with an alignment shift of at least four bits. The larger number is either exactly machine representable, or available with at least four quad bits so that the error bound for the final rounded result is just slightly greater than 1/2 the least significant bit.

Approximations			
Octant bits	**SIN(pos. arg.)**	**COS(pos. arg.)**	**SIN(neg. arg.)**
000	SIN (y * PI/4)	COS (y * PI/4)	-SIN (y * PI/4)
001	COS (1-y) * PI/4	SIN (1-y) * PI/4	-COS (1-y) * PI/4
010	COS (y * PI/4)	-SIN (y * PI/4)	-COS (y * PI/4)
011	SIN (1-y) * PI/4	-COS (1-y) * PI/4	-SIN (1-y) * PI/4
100	-SIN (y * PI/4)	-COS (y * PI/4)	SIN (y * PI/4)
101	-COS (1-y) * PI/4	-SIN (1-y) * PI/4	COS (1-y) * PI/4
110	-COS (y * PI/4)	SIN (y * PI/4)	COS (y * PI/4)
111	-SIN (1-y) * PI/4	COS (1-y) * PI/4	SIN (1-y) * PI/4

If |X| >= 2**30, the error message is:
"MTH$_SIGLOSMAT — SIGNIFICANCE LOST IN MATH LIBRARY."

DSIN(X) is computed as:

Let Q = INTEGER(|X| /(PI/2))
where:

Q = 0 for first quadrant
Q = 1 for second quadrant
Q = 2 for third quadrant
Q = 3 for fourth quadrant

Let Y = FRACTION(|X| /(PI/2))

If |Y| < 2**-28, the sine is computed as:

DSIN(X) = S * (PI/2)

S = Y if Q = 0
S = 1-Y if Q = 1
S = -Y if Q = 2
S = Y-1 if Q = 3

For all other cases:

DSIN(X) = P(Y*PI/2) if Q = 0
DSIN(X) = P((1-Y)*PI/2) if Q = 1
DSIN(X) = P(-Y*PI/2) if Q = 2
DSIN(X) = P((Y-1)*PI/2) if Q = 3

where:

P(Y) = Y*SUM(C[i]*(Y**(2*i))) for i = 0:8

The polynomial approximation used is of degree 8.

The relative error is less than or equal to 10**-18.6. The result is guaranteed to be within the closed interval -1.0 to +1.0.

No loss or precision occurs if |X| < 2 * PI *256.

If |X| >= 2**31, the message: MTH$__SIGLOSMAT — "SIGNIFICANCE LOST IN MATH LIBRARY" is printed.

GSIN(X) is computed as:

Let Q = INTEGER(|X| /(PI/2))
where:

Q = 0 for first quadrant
Q = 1 for second quadrant
Q = 2 for third quadrant
Q = 3 for fourth quadrant

Let Y = FRACTION(|X| /(PI/2))

If |Y| < 2**-28, the sine is computed as:

GSIN(X) = S * (PI/2)

S = Y	if Q = 0
S = 1-Y	if Q = 1
S = -Y	if Q = 2
S = Y-1	if Q = 3

For all other cases:

GSIN(X) = P(Y*PI/2)	if Q = 0
GSIN(X) = P((1-Y)*PI/2)	if Q = 1
GSIN(X) = P(-Y*PI/2)	if Q = 2
GSIN(X) = P((Y-1)*PI/2)	if Q = 3

where:

P(Y) = Y*SUM(C[i]*(Y**(2*i))) for i = 0:8

The polynomial approximation used is of degree 8.

The relative error is less than or equal to 10**-18.6. The result is guaranteed to be within the closed interval -1.0 to +1.0.

No loss of precision occurs if |X| < 2 * PI *256.

If |X| >= 2**31, the message: MTH$__SIGLOSMAT — "SIGNIFICANCE LOST IN MATH LIBRARY" is printed.

HSIN(X) is computed as:

Let Q = INTEGER(|X|/(PI/2))
where:

MOD(Q,4) = 0 for first quadrant
MOD(Q,4) = 1 for second quadrant
MOD(Q,4) = 2 for third quadrant
MOD(Q,4) = 3 for fourth quadrant

Let Y = FRACTION(|X|/(PI/2))

If |Y| < 2**-56, the sine is computed as:

HSIN(X) = S * (PI/2)
where:

S = Y	if MOD(Q,4) = 0
S = 1-Y	if MOD(Q,4) = 1
S = -Y	if MOD(Q,4) = 2
S = Y-1	if MOD(Q,4) = 3

Otherwise:

HSIN(X) = P(Y*PI/2)	if MOD(Q,4) = 0
HSIN(X) = P((1-Y)*PI/2)	if MOD(Q,4) = 1
HSIN(X) = P(-Y*PI/2)	if MOD(Q,4) = 2
HSIN(X) = P((Y-1)*PI/2)	if MOD(Q,4) = 3

where:

P(Y) = Y*SUM(C[i]*(Y**(2*i))) for i = 0:14

The result is guaranteed to be in the closed interval -1.0 to +1.0

If |X| >= 2**31, the message: MTH$__SIGLOSMAT — "SIGNIFICANCE LOST IN MATH LIBRARY" is printed.

D.1.13 Square Root

SQRT(X) is computed as:

If X < 0, an error is signaled.

Let X = 2**K * F
where:

K is the exponential part of the floating-point data
F is the fractional part of the floating-point data

If K is even:

X = 2**(2*P) * F,
SQRT(X) = 2**P * SQRT (F),

1/2 =< F < 1

where:

P = K/2

If K is odd,

X = 2**(2*P+1) * F = 2**(2*P+2) * (F/2),
SQRT(X) = 2**(P+1) * SQRT(F/2),
1/4 =< F/2 < 1/2

Let F' = A*F + B, when K is even:

A = 0.453730314 (octal)
B = 0.327226214 (octal)

Let F' = A*(F/2) + B, when K is odd:

A = 0.650117146 (octal)
B = 0.230170444 (octal)

Let K' = P, when K is even
Let K' = P+1, when K is odd

Let Y[0] = 2**K' * F' be a straight line approximation within the given interval using coefficients A and B which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], two Newton-Raphson iterations are performed:

Y[n+1] = 1/2 * (Y[n] + X/Y[n])

The relative error is < 10**-8.

DSQRT(X) is computed as:

If X < 0, an error is signaled.

Let X = 2**K * F where:

K is the exponential part of the floating-point data
F is the fractional part of the floating-point data

If K is even:

X = 2**(2*P) * F,
DSQRT(X) = 2**P * DSQRT (F),
1/2 =< F < 1

If K is odd:

X = 2**(2*P+1) * F = 2**(2*P+2) * (F/2),
DSQRT(X) = 2**(P+1) * DSQRT(F/2),
1/4 =< F/2 < 1/2

Let F' = A*F + B, when K is even:
A = 0.453730314 (octal)
B = 0.327226214 (octal)

Let F' = A*(F/2) + B, when K is odd:

A = 0.650117146 (octal)
B = 0.230170444 (octal)

Let K' = P, when K is even.
Let K' = P+1, when K is odd.

Let Y[0] = 2**K' * F' be a straight line approximation within the given interval using coefficients A and B which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], three Newton-Raphson iterations are performed:

Y[n+1] = 1/2 * (Y[n] + X/Y[n])

The relative error is < 10**-17.

GSQRT(X) is computed as:

If X < 0, an error is signaled.

Let X = 2**K * F where:

K is the exponential part of the floating-point data
F is the fractional part of the floating-point data

If K is even:

X = 2**(2*P) * F,
GSQRT(X) = 2**P * GSQRT (F),
1/2 =< F < 1

If K is odd:

X = 2**(2*P+1) * F = 2**(2*P+2) * (F/2),
GSQRT(X) = 2**(P+1) * GSQRT(F/2),
1/4 =< F/2 < 1/2

Let F' = A*F + B, when K is even:

A = 0.453730314 (octal)
B = 0.327226214 (octal)

Let F' = A*(F/2) + B, when K is odd:

A = 0.650117146 (octal)
B = 0.230170444 (octal)

Let K' = P, when K is even.
Let K' = P+1, when K is odd.

Let Y[0] = 2**K' * F' be a straight line approximation within the given interval using coefficients A and B which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], three Newton–Raphson iterations are performed:

Y[n+1] = 1/2 * (Y[n] + X/Y[n])

The relative error is < 10**–17.

HSQRT(X) is computed as:

If X < 0, signal error.
If X = 0, return HSQRT(X) = 0.

Let X = 2**K * F where:

K is the exponential part of the floating-point data
F is the fractional part of the floating-point data

If K is even:

X = 2**(2*P) * F,
HSQRT(X) = 2**P * HSQRT(F),
1/2 <= F < 1

If K is odd:

X = 2**(2*P+1) * F = 2**(2*P+2) * (F/2),
HSQRT(X) = 2**(P+1) * HSQRT(F/2),
1/4 <= F/2 < 1/2

Let F' = A*F + B, when K is even:

A = 0.453730314 (octal)
B = 0.327226214 (octal)

Let F' = A*(F/2) + B, when K is odd:

A = 0.650117146 (octal)
B = 0.230170444 (octal)

Let K' = P, when K is even.
Let K' = P+1, when K is odd.

Let Y[0] = 2**K' * F' be a straight line approximation within the given interval using coefficients A and B which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], five Newton-Raphson iterations are performed:

Y[n+1] = (1/2) * (Y[n] + X/Y[n])

D.1.14 Tangent

TAN(X) is computed as:

1) Calculate SIN

   If error from SIN, then return with reserved operand.
   If SIN (X) = 0, then return TAN (X) = 0.

2) Calculate Cosine

   No need to check for reserved operand, as error would be caught in Step 1.
   No need to check for zero, as it would be caught in Step 3.

3) Calculate SIN/COS

   Hardware trap occurs if divide-by-zero or overflow error occurs.

DTAN(X) is computed as:

1) Calculate DSIN

   If error from DSIN, then return with reserved operand.
   If DSIN (X) = 0, then return DTAN (X) = 0.

2) Calculate DCOS

   No need to check for reserved operand, as error would be caught in Step 1.
   No need to check for zero, as it would be caught in Step 3 as a hardware trap.

3) Calculate DSIN/DCOS

Hardware trap is signaled if divide by zero or overflow error occurs.

GTAN(X) is computed as:

1) Calculate GSIN

If error from GSIN, then return with reserved operand.
If GSIN (X) = 0, then return GTAN (X) = 0.

2) Calculate GCOS

No need to check for reserved operand, as error would be caught in Step 1.
No need to check for zero, as it would be caught in Step 3 as a hardware trap.

3) Calculate GSIN/GCOS

Hardware trap is signaled if divide by zero or overflow error occurs.

HTAN(X) is computed as:

1) Calculate HSIN

If error from HSIN, then return with reserved operand.
If HSIN(X) = 0, then return HTAN(X) = 0.

2) Calculate HCOS

No need to check for reserved operand, as error would be caught in Step 1.
No need to check for zero, as it would be caught in Step 3 as a hardware trap.

3) Calculate HSIN/HCOS

Hardware trap is signaled if divide by zero or overflow error occurs.

D.2 Exponentiation Functions

D.2.1 Floating Base to Floating Power

OTS$POWDD is computed as:

The D_floating result for this function is given by:

Base	Exponent	Result
=0	>0	0.0
=0	=0	Undefined Exponentiation
=0	<0	Undefined Exponentiation
<0	Any	Undefined Exponentiation

(continued on next page)

Base	Exponent	Result
>0	>0	DEXP(exponent * DLOG(base))
>0	=0	1.0
>0	<0	DEXP(exponent * DLOG(base))

Floating-point overflow can occur.

Undefined exponentiation occurs if the base is 0 and the exponent is 0 or negative, or if the base is negative.

OTS$POWDR is computed as:

Convert the F_floating exponent to D_floating and then calculate the D_floating result using the same code as OTS$POWDD.

OTS$POWRD is computed as:

Convert the F_floating base to D_floating and then calculate the D_floating result using the same code as OTS$POWDD.

OTS$POWGG is computed as:

The G_floating result for this function is given by:

Base	Exponent	Result
=0	>0	0.0
=0	=0	Undefined Exponentiation
=0	<0	Undefined Exponentiation
<0	Any	Undefined Exponentiation
>0	>0	GEXP(exponent * GLOG(base))
>0	=0	1.0
>0	<0	GEXP(exponent * GLOG(base))

Floating-point overflow can occur.

Undefined exponentiation occurs if the base is 0 and the exponent is 0 or negative, or if the base is negative.

OTS$POWHH is computed as:

The H_floating result for this function is given by the same algorithm as OTS$POWGG except HEXP and HLOG are substituted for GEXP and GLOG.

OTS$POWRR is computed as:

The F_floating result for this function is given by:

Base	Exponent	Result
=0	>0	0.0
=0	=0	Undefined Exponentiation
=0	<0	Undefined Exponentiation
<0	Any	Undefined Exponentiation
>0	>0	EXP(exponent * ALOG(base))
>0	=0	1.0
>0	<0	EXP(exponent * ALOG(base))

Floating-point overflow can occur.

Undefined exponentiation occurs if the base is 0 and the exponent is 0 or negative, or if the base is negative.

D.2.2 Floating Base to Integer Power

OTS$POWDJ
OTS$POWGJ
OTS$POWHJ_R3
OTS$POWRJ

All of the above functions use the same basic algorithm. However, the internal calculations and the floating-point result are computed at the same precision level as the base value.

The floating-point result is given by:

Base	Exponent	Result
Any	>0	Product (base * 2**i) where i is each nonzero bit position in |exponent|
>0	=0	1.0
=0	=0	Undefined exponentiation
<0	=0	1.0
>0	<0	1.0 / product (base * 2**i) where i is each nonzero bit position in |exponent|
=0	<0	Undefined exponentiation
<0	<0	1.0 / Product (base * 2**i) where i is each nonzero bit position in |exponent|

Floating-point overflow can occur.

Undefined exponentiation occurs if the base is 0 and the exponent is 0 or negative.

D.2.3 Integer Base to Integer Power

OTS$POWII
OTS$POWJJ

All of the above functions use the same basic algorithm. However, the internal calculations and the signed integer result are computed at the same precision level as the base value.

The signed integer result is given by:

Base	Exponent	Result
Any	>0	Product (base * 2**i) where i is each non-zero bit position in exponent
>0	=0	1
=0	=0	Undefined exponentiation
<0	=0	1
>1	<0	0
=1	<0	1
=0	<0	Undefined Exponentiation
=−1	<0 and even	1
=−1	<0 and odd	−1
<−1	<0	1

Integer overflow can occur.

Undefined exponentiation occurs if the base is 0 and the exponent is 0 or negative.

Appendix E
Image Initialization and Termination

Normally, both user and library procedures are written so that they are self-initializing. This means that they can process information with no special action required by the calling program. Initialization is automatic because either: 1) the procedure's statically allocated data storage is initialized at compile or link time, or 2) a statically allocated flag is tested and set on each call so that initialization occurs only on the first call.

Any special initialization — such as a call to other procedures or to system services — can be performed on the first call before the main program is initialized. For example, you can establish a new environment to alter the way errors are handled or messages are printed.

Such special initialization is required only rarely; however, it need not be done by requiring the caller of the procedure to make an explicit initialization call. The Run-Time Library provides a system declaration mechanism that performs all such initialization calls before the main program is called. Special initialization is thus invisible to subsequent callers of the procedure.

This Appendix describes the system declaration mechanism, including LIB$INITIALIZE, which performs calls to any initialization procedure declared by the user. This mechanism is also available to Run-Time Library so that user procedures that require special initialization can be added to the library. However, use of LIB$INITIALIZE is discouraged and should be used only when no other method is suitable. One major problem with the LIB$INITIALIZE mechanism is that it cannot be used by procedures in a sharable image.

E.1 Image Initialization

Before the main program or main procedure is called, a number of system initialization procedures are called as specified by a one, two, or three longword initialization list set up by the linker. This list consists of the addresses

of the debugger (if present) the LIB$INITIALIZE procedure (if present) and the entry point of the main program or main procedure, in that order. The following initialization steps take place:

1. The image activator maps the user program image into the address space of the process and sets up useful information such as the program name. Then it starts up the command interpreter.

2. The command interpreter sets up an argument list (see Section E.2) and calls the next procedure in the initialization list (debugger, LIB$INITIALIZE, main program, or main procedure).

3. The debugger, if present, initializes itself and calls the next procedure in the initialization list (LIB$INITIALIZE, main program, or main procedure).

4. LIB$INITIALIZE, if present, is a library procedure that calls each library and user initialization procedure declared using the system LIB$INITIALIZE mechanism (see Section E.4). Then it calls the main program or main procedure.

5. The main program or main procedure executes, and at the user's discretion, accesses its argument list to scan the command or obtain information about the image. The main program or main procedure can then call other procedures.

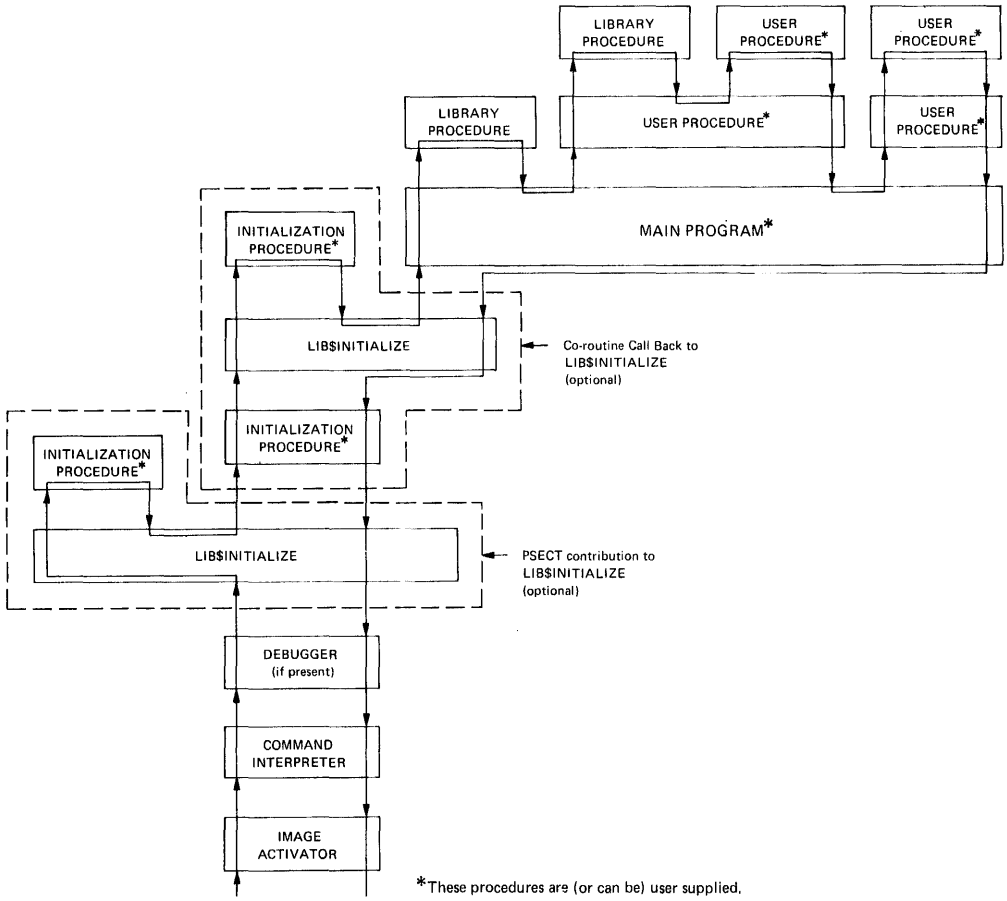
6. Eventually, the main program or main procedure terminates by executing a return instruction (RET) with R0 set to a standard completion code to indicate success or failure, where bit 0 equals 1 for success or 0 for failure. See Chapter 6 and Appendix C for a description of condition values.

7. The completion code is returned to LIB$INITIALIZE (if present), the debugger (if present) and finally to the command interpreter which issues a $EXIT system service with the completion status as a parameter. Any declared exit handlers are called at this point. (See Declare Exit Handler $DCLEXH System Service – in the *VAX/VMS System Services Reference Manual*.)

Main programs should not call the $EXIT system service directly. If they do, other programmers cannot reuse them as callable procedures.

Figure E–1 illustrates the sequence of calls and returns in a typical image initialization. Each box is a procedure activation as represented on the image stack. The top of the stack is at the top of the figure. Each upward arrow represents the result of a CALLS or CALLG instruction which creates a procedure activation on the stack to which control is being transferred. Each downward arrow represents the result of a RET (return) instruction. A RET instruction removes the procedure activation from the stack and causes control to be transferred downward to the next box.

Figure E-1: Sequence of Events during Image Initialization



A user program can alter the image initialization sequence by making a PSECT contribution to PSECT LIB$INITIALIZE and declaring EXTERNAL LIB$INITIALIZE. This adds the optional initialization steps shown in Figure E-1 termed: PSECT contribution to LIB$INITIALIZE. (A PSECT is a portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section. See Section E.4.) If the initialization procedure also performs a co-routine call back to LIB$INITIALIZE, the optional steps termed: Co-routine call back to LIB$INITIALIZE shown in Figure E-1 are added to the image initialization sequence (see Section E.5).

E.2 Initialization Argument List

The parameter list passed from the command interpreter, the debugger, and/or LIB$INITIALIZE to the main program is the same for each procedure activation, and consists of:

(start-adr-adr, cli-co-rout, ...)

start-adr-adr
: the absolute virtual address of the entry in the initialization vector which used to perform the call (modify access, passed by immediate value).

cli-co-rout
: the address of the command interpreter coroutine to obtain command arguments (function call access, passed by reference).

...
: Useful image information such as the program name (See the *VAX/VMS Operator's Guide*).

The debugger and/or LIB$INITIALIZE can call the next procedure in the initialization chain using the following coding sequence:

```
   .
   .
   .
ADDL    #4, 4(AP)       ; step to next initialization list entry
MOVL    @4(AP), R0      ; R0=next address to call
CALLG   (AP), (R0)      ; call next initialization procedure
   .
   .
   .
```

This coding sequence violates the VAX-11 Procedure Calling Standard (see Appendix C) by modifying the contents of an argument list entry. However, the argument list can be expanded in the future without requiring any change to either the debugger or to LIB$INITIALIZE.

E.3 Declaring Initialization Procedures

Any library or user program module can declare an initialization procedure. This procedure will be called when the image is started. The declaration is made by making a contribution to PSECT LIB$INITIALIZE which contains a list of procedure entry point addresses to be called before the main program or main procedure is called. The following MACRO example declares an initialization procedure by placing the procedure entry address INIT_PROC in the list:

```
.EXTRN  LIB$INITIALIZE  ; cause library initialization
                        ; dispatcher to be loaded
.PSECT  LIB$INITIALIZE, NOPIC, USR, CON, REL, GBL, NOSHR,-
                        NOEXE, RD, NOWRT, LONG
.LONG   INIT_PROC       ; contribute entry point address of
                        ; initialization routine.
.PSECT  ...
```

The .EXTRN declaration links the initialization procedure dispatcher, LIB$INITIALIZE into your program's image. The reference contains a definition of the special global symbol LIB$INITIALIZE which is the procedure

entry point address of the dispatcher. The linker stores the value of this special global symbol in the initialization list along with the starting address of the debugger and main program. The GBL specification ensures that the PSECT LIB$INITIALIZE contribution will not be affected by any clustering performed by the linker.

E.4 Dispatching to Initialization Procedures

The LIB$INITIALIZE dispatcher calls each initialization procedure in the list with the following argument list:

```
CALL init-proc
(init-co-routine, cli-co-rout, ...)
```

init-co-routine
: Address of library coroutine to be called to effect a coroutine linkage with LIB$INITIALIZE (function call access, passed by reference).

cli-co-rout
: Address of the command interpreter coroutine used to obtain command language arguments (function call access, passed by reference).

...
: The rest of the argument list to be passed to the main program (see Section E.3).

Note that this argument list is the same as the one passed to the main program except for the first parameter.

E.5 Initialization Procedure Options

An initialization procedure has a number of options:

1. It can set up an exit handler by calling the Declare Exit Handler $DCLEXH system service, although this is generally done with a statically allocated first-time flag.
2. It can initialize statically allocated storage, although this is preferably done at image activation time using compile-time and link-time data initialization declarations, or using a first-time call flag in its statically allocated storage.
3. It can call the initialization dispatcher (instead of returning to it) by calling init-co-routine. This achieves a coroutine linkage. Control will return to the initialization procedure when the main program returns control. Then, the initialization procedure should also return control to pass back the completion code returned by the main program (to the debugger and/or command interpreter).
4. It can establish a condition handler in the current frame before performing the previous step above. This will leave the initialization procedure condition handler on the image stack for the duration of the image execution.

Since this will occur after the command interpreter has set up the catch-all stack frame handler, and after the debugger has set up its stack frame handler, the initialization procedure handler can override either of these handlers since it will receive signals before they do. (See Chapter 6 for a description of condition handlers.)

For example, the following MACRO code fragment shows an initialization procedure establishing a handler, calling the init-co-routine procedure (to effect a coroutine call), getting control after the main program returns, and returning to the normal exit processing:

```
INIT_PROC::
        .WORD   ^M<>                    ; no registers used
        MOVAL   HANDLER, (FP)           ; establish handler
        ..                              ; perform any other initialization

        CALLG   (AP), @INIT_CO_ROUTINE(AP)
                                        ; continue initialization which
10$:                                    ; then calls main program or
                                        ; procedure.
        ...                             ; Return here when main program
                                        ; returns with R0 = completion
        RET                             ; status return to normal exit
                                        ; processing with R0 = completion
                                        ; status

HANDLER:                                ; condition handler
        .WORD   ^M<...>                 ; register mask
        ...                             ; handle condition
                                        ; could unwind to 10$
        MOVL    ..., R0                 ; Set completion status with a
                                        ; condition value
        RET                             ; resignal or continue depending
                                        ; on R0 being SS$_RESIGNAL or
                                        ; SS$_CONTINUE.
```

The FORTRAN language support procedures use the same mechanism to declare the initialization procedure, COM__STARTUP, if the PDP-11 FORTRAN IV-PLUS compatibility routines, ERRSET or ERRTST, are called by the user program. COM__STARTUP establishes a condition handler, COM__HANDLER, and makes a coroutine call to LIB$INITIALIZE. To isolate and modularize this technique, the ERRSET and ERRTST procedures are contained in a single module which also contains COM__STARTUP and COM__HANDLER as local rather than global procedures.

E.6 Image Termination

Main programs and main procedures terminate by executing a return instruction (RET). This returns control to the caller, which may have been LIB$INITIALIZE, the debugger, or the command interpreter. The completion code, SS$__NORMAL, which has the value 1, should be used to indicate normal successful completion.

Any other condition value can be used to indicate success or failure completion. This condition value is used as the parameter to the exit ($EXIT)

system service by the command interpreter. If the severity field (STS$V_SEVERITY) is SEVERE or ERROR, the continuation of a batch job or command procedure is affected. See Appendix C for a description of the format and interpretation of condition values in various contexts.

Main programs are discouraged from calling the $EXIT system service directly. This allows them to be more like ordinary modular procedures and hence usable by other programmers as callable procedures.

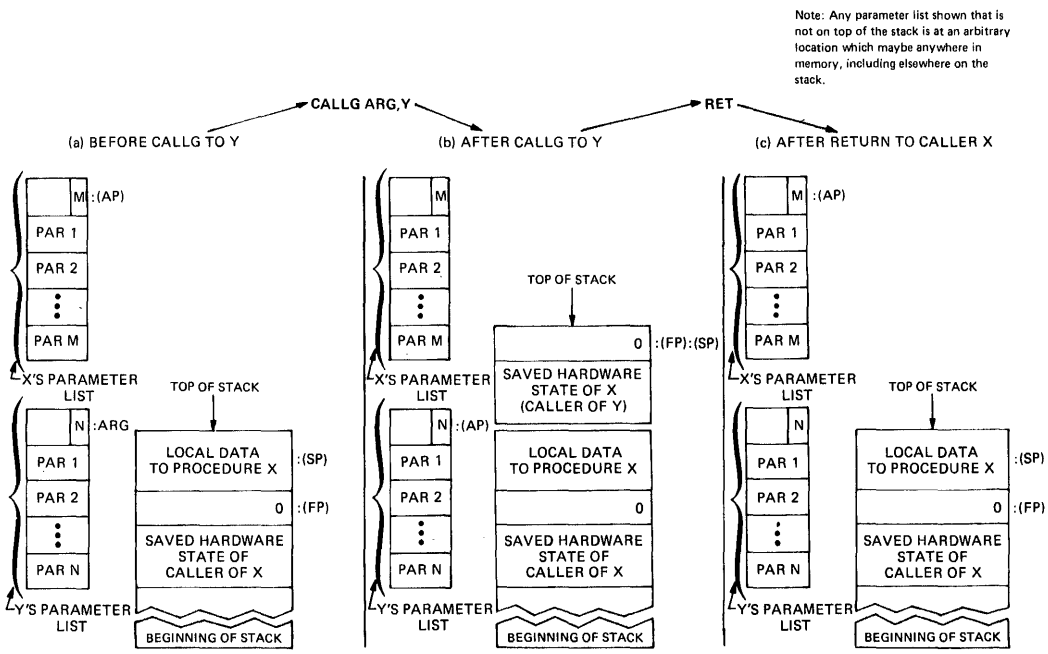
Appendix F
CALLG, CALLS Instructions

A CALLG or a CALLS MACRO instruction can be used to call procedures written in any language. In the CALLG instruction, the argument list can be allocated anywhere in memory. In the CALLS instruction, the argument list is allocated on top of the stack. The called procedure is unaware of which instruction is actually used.

F.1 CALLG Instruction

Figure F–1 illustrates a CALLG instruction. In the example shown, Procedure X calls Procedure Y. The stack and parameter lists are shown in three states: (a) before X calls to Y, (b) after X calls to Y, and (c) after Y returns.

Figure F–1: CALLG Instruction Sequence



Part (a) shows the stack before X calls Y. The argument pointer (AP) points to X's parameter list, and the hardware state of the caller of X is on the stack. (The hardware state contains the processor status word, argument pointer, frame pointer, and stack pointer at the time when the CALLG is executed.) The CALLG instruction in this example is: CALLG ARG,Y where ARG points to the parameter list to be passed to Y, and names Y as the procedure to be called. When the CALLG instruction is executed in Part (b) the hardware state of Procedure X (Procedure Y's caller) is pushed onto the stack and the contents of the argument pointer are changed from the address of the X parameter list to that of the Y parameter list. Procedure Y can then access its parameter list using the AP general register. After procedure Y executes, a return to its caller is made in Part (c). As this occurs, all of Procedure Y's related hardware states are popped off the stack and the argument pointer (AP) is restored to the address of X's parameter list. (The procedure lists remain in the same arbitrary locations throughout, which may be elsewhere on the stack or anywhere else in memory.)

The following object code shows how LIB$INSV (PAR1,PAR2,PAR3,PAR4) is invoked by way of a CALLG instruction. Note that the first item in ARGLST, the parameter list count, contains the total number of parameters in the list (and that Procedure Y used in the preceeding description is equivalent to LIB$INSV in this example).

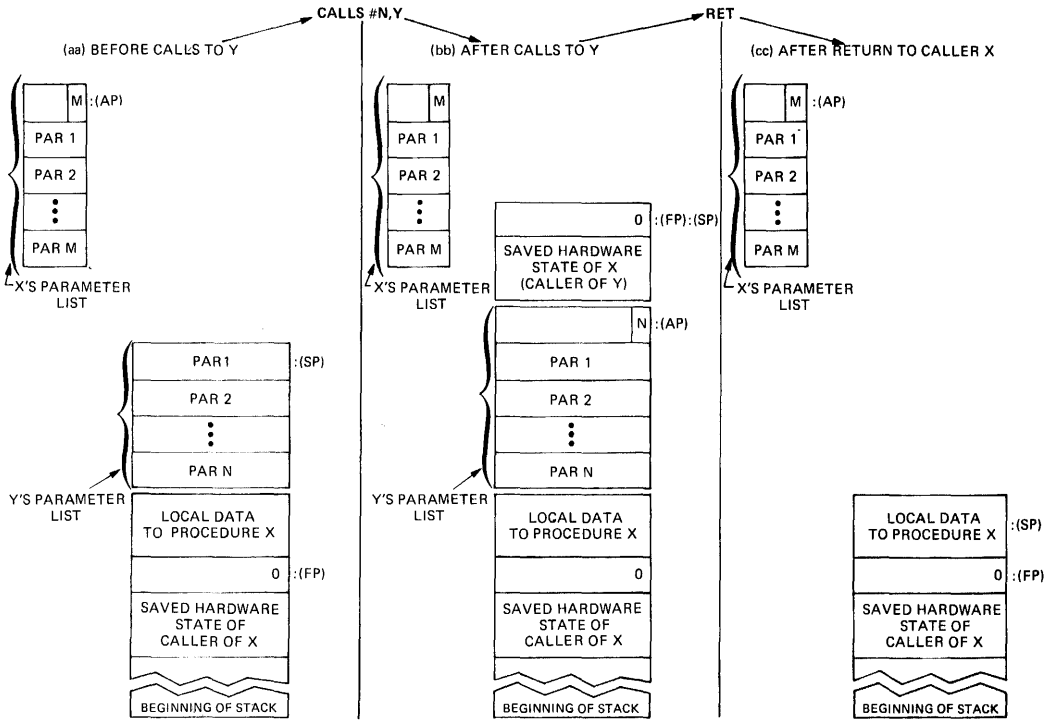
```
ARGLST:  .LONG          4           ; Argument list count
         .ADDRESS       PAR1        ; Address of field
         .ADDRESS       PAR2        ; Address of position
         .ADDRESS       PAR3        ; Address of size
         .ADDRESS       PAR4        ; Address of longword
          .                          ; to receive field
          .
          .
         CALLG          ARGLST, LIB$INSV ; Call LIB$INSV
```

F.2 CALLS Instruction

Figure F-2 illustrates a CALLS instruction. In the example shown, Procedure X calls Procedure Y. The stack and parameter lists are shown in the same three states that are shown for the CALLG instruction in the preceding Figure F-1.

Part (aa) of the CALLS instruction shows that Procedure X has pushed all of the parameter list entries onto the stack except for the parameter list count N. (Since N appears in the CALLS instruction, it is not put on the stack yet.) Note that the parameter list is pushed on in reverse order, that is, Parameter N is pushed on first and Parameter 1 is pushed on last. The CALLS instruction is: CALLS N,Y. When the CALLS instruction is executed, in Part (bb), the parameter list count N is pushed onto the stack followed by the saved hardware state of Procedure X (the caller of Y). The contents of the argument pointer (AP) are set to the address of Procedure Y's parameter list. Procedure Y can then read the parameter list passed to it. After the return to Procedure X in Part (cc), both the Y parameter list and the associated hardware state are removed from the stack, and the argument pointer (AP) is restored to the address of Procedure X's parameter list.

Figure F-2: CALLS Instruction Sequence



The following object code shows how LIB$INSV is invoked by way of a CALLS instruction. Note that the parameter list count (4 in this example) appears in the CALLS instruction. The hardware automatically places this value on the top of the stack:

```
PUSHAL    PAR4              ; Push address of longword
                            ; to receive field
PUSHAL    PAR3              ; Push address of size
PUSHAL    PAR2              ; Push address of position
PUSHAL    PAR1              ; Push address of field
CALLS     #4, LIB$INSV      ; Call LIB$INSV
```

Appendix G
Sample Programs Using LIB$TPARSE

This appendix contains two sample programs using LIB$TPARSE.

G.1 Sample MACRO Program Using LIB$TPARSE

```
        .TITLE  CREATE_DIR - Create Directory File
        .IDENT  "X0000"
;+
;
; This is a sample program that accepts and parses the command line
; of the CREATE/DIRECTORY command. This program contains the VAX/VMS
; call to acquire the command line from the command interpreter
; and parse it with TPARSE, leaving the necessary information in
; its global data base. The command line has the following format:
;
;       CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
;                  /OWNER_UIC=[2437,25]
;                  /ENTRIES=100
;                  /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
;
; The three qualifiers are optional. Alternatively, the command
; may take the form
;
;       CREATE/DIR DEVICE:[202,31]
;
; using any of the optional qualifiers.
;
;-

;+
;
; Global data, control blocks, etc.
;
;-
        .PSECT IMPURE,WRT,NOEXE
;
; Define control block offsets
;
        $CLIDEF
        $TPADEF
;
```

```
; Define parser flag bits for flags longword
;
UIC_FLAG        = 1                     ; /UIC seen
ENTRIES_FLAG    = 2                     ; /ENTRIES seen
PROT_FLAG       = 4                     ; /PROTECTION seen
;
; command interpreter request descriptor block to get the line
;
REQ_COMMAND:
        $CLIREQDESC,-
                RQTYPE = CLI$K_GETCMD
;
; TPARSE parameter block
;
TPARSE_BLOCK:
        .LONG   TPA$K_COUNTO            ; Longword count
        .LONG   TPA$M_ABBREV!-          ; Allow abbreviation
                TPA$M_BLANKS            ; Process spaces explicitly
        .BLKL   TPA$K_LENGTH0-8         ; Remainder set at run time
;
; Parser global data
;
PARSER_FLAGS:   .BLKL   1               ; Keyword flags
DEVICE_STRING:  .BLKL   2               ; Device string descriptor
ENTRY_COUNT:    .BLKL   1               ; Space to preallocate
FILE_PROTECT:   .BLKL   1               ; Directory file protection
UIC_GROUP:      .BLKL   1               ; Temp for UIC group
UIC_MEMBER:     .BLKL   1               ; Temp for UIC member
FILE_OWNER:     .BLKL   1               ; Actual file owner UIC
NAME_COUNT:     .BLKL   1               ; Number of directory names
DIRNAME1:       .BLKL   2               ; Name descriptor 1
DIRNAME2:       .BLKL   2               ; Name descriptor 2
DIRNAME3:       .BLKL   2               ; Name descriptor 3
DIRNAME4:       .BLKL   2               ; Name descriptor 4
    "       "                   "
    "       "                   "
    "       "                   "
DIRNAME8:       .BLKL   2               ; Name descriptor 8

        .SBTTL Main Program
;+
;
; This is the main program of the CREATE/DIRECTORY utility. It gets
; the command line from the command interpreter and parses it.
;
;-
        .PSECT CODE,EXE,NOWRT
CREATE_DIR::
        .WORD   ^M<R2,R3,R4,R5>         ; Save registers
;
; Call the command interpreter to obtain the command line.
;
        CLRQ    -(SP)                   ; 2 zero args
        PUSHAL  REQ_COMMAND             ; and request block
        CALLS   #3,@CLI$A_UTILSERV(AP)  ; Call back the interpreter
;
; Copy the input string descriptor into the TPARSE control block
; and call LIB$TPARSE. Note that impure storage is assumed to be zero.
;
        MOVZWL  REQ_COMMAND+CLI$W_RQSIZE,-
                TPARSE_BLOCK+TPA$L_STRINGCNT
        MOVL    REQ_COMMAND+CLI$L_RQADDR,-
                TPARSE_BLOCK+TPA$L_STRINGPTR
        PUSHAL  UFD_KEY
        PUSHAL  UFD_STATE
```

```
        PUSHAL  TPARSE_BLOCK
        CALLS   #3,LIB$TPARSE
        BLBC    R0,SYNTAX_ERR
;
; Parsing is complete.
;
            .           ;
            .
            .           ; (Process the command)
            .
            .
            .           ;
            .
        MOVL    #1,R0   ; Return success
        RET


        .SBTTL  Parser State Table
        $INIT_STATE     UFD_STATE,UFD_KEY
;
; Read over the command name (to the first blank in the command).
;
        $STATE  START
        $TRAN   TPA$_BLANK,,BLANKS_OFF
        $TRAN   TPA$_ANY,START
;
; Read device name string and trailing colon.
;
        $STATE
        $TRAN   TPA$_SYMBOL,,,,DEVICE_STRING
        $STATE
        $TRAN   ':'
;
; Read directory string, which is either a UIC string or a general
; directory string.
;
        $STATE
        $TRAN   !UIC,,MAKE_UIC
        $TRAN   !NAME
;
; Scan for options until end of line is reached
;
        $STATE  OPTIONS
        $TRAN   '/'
        $TRAN   TPA$_EOS,TPA$_EXIT
        $STATE
        $TRAN   'OWNER_UIC',PARSE_UIC,,UIC_FLAG,PARSER_FLAGS
        $TRAN   'ENTRIES',PARSE_ENTRIES,,ENTRIES_FLAG,PARSER_FLAGS
        $TRAN   'PROTECTION',PARSE_PROT,,PROT_FLAGS,PARSER_FLAGS
;
; Get file owner UIC
;
        $STATE  PARSE_UIC
        $TRAN   ':'
        $TRAN   '='
        $STATE
        $TRAN   !UIC,OPTIONS
;
; Get number of directory entries
;
        $STATE  PARSE_ENTRIES
        $TRAN   ':'
        $TRAN   '='
        $STATE
        $TRAN   TPA$_DECIMAL,OPTIONS,,,ENTRY_COUNT
;
```

```
; Get directory file protection. Note that the bit masks generate the
; protection in complement form. It will be uncomplemented by the main
; program.
;
        $STATE    PARSE_PROT
        $TRAN     ':'
        $TRAN     '='
        $STATE
        $TRAN     '('

        $STATE    NEXT_PRO
        $TRAN     'SYSTEM', SYPR
        $TRAN     'OWNER', OWPR
        $TRAN     'GROUP', GRPR
        $TRAN     'WORLD', WOPR
        $STATE    SYPR
        $TRAN     ':'
        $TRAN     '='

        $STATE    SYPRO
        $TRAN     'R',SYPRO,,^X0001,FILE_PROTECT
        $TRAN     'W',SYPRO,,^X0002,FILE_PROTECT
        $TRAN     'E',SYPRO,,^X0004,FILE_PROTECT
        $TRAN     'D',SYPRO,,^X0008,FILE_PROTECT
        $TRAN     TPA$_LAMBDA,ENDPRO

        $STATE    OWPR
        $TRAN     ':'
        $TRAN     '='

        $STATE    OWPRO
        $TRAN     'R',OWPRO,,^X0010,FILE_PROTECT
        $TRAN     'W',OWPRO,,^X0020,FILE_PROTECT
        $TRAN     'E',OWPRO,,^X0040,FILE_PROTECT
        $TRAN     'D',OWPRO,,^X0080,FILE_PROTECT
        $TRAN     TPA$_LAMBDA,ENDPRO

        $STATE    GRPR
        $TRAN     ':'
        $TRAN     '='

        $STATE    GRPRO
        $TRAN     'R',GRPRO,,^X0100,FILE_PROTECT
        $TRAN     'W',GRPRO,,^X0200,FILE_PROTECT
        $TRAN     'E',GRPRO,,^X0400,FILE_PROTECT
        $TRAN     'D',GRPRO,,^X0800,FILE_PROTECT
        $TRAN     TPA$_LAMBDA,ENDPRO

        $STATE    WOPR
        $TRAN     ':'
        $TRAN     '='

        $STATE    WOPRO
        $TRAN     'R',WOPRO,,^X1000,FILE_PROTECT
        $TRAN     'W',WOPRO,,^X2000,FILE_PROTECT
        $TRAN     'E',WOPRO,,^X4000,FILE_PROTECT
        $TRAN     'D',WOPRO,,^X8000,FILE_PROTECT
        $TRAN     TPA$_LAMBDA,ENDPRO

        $STATE    ENDPRO
        $TRAN     ',',NEXT_PRO
        $TRAN     ')',OPTIONS
;
; Subexpression to parse a UIC string.
;
```

```
        $STATE    UIC
        $TRAN     '['
        $STATE
        $TRAN     TPA$_OCTAL,,,,UIC_GROUP
        $STATE
        $TRAN     ','
        $STATE
        $TRAN     TPA$_OCTAL,,,,UIC_MEMBER
        $STATE
        $TRAN     ']',TPA$_EXIT,CHECK_UIC
;
; Subexpression to parse a general directory string
;
        $STATE    NAME
        $TRAN     '['
        $STATE    NAME0
        $TRAN     TPA$_STRING,,STORE_NAME
        $STATE
        $TRAN     ',',NAME0
        $TRAN     ']',TPA$_EXIT
        $END_STATE

        .SBTTL    Parser Action Routines
        .PSECT    CODE,EXE,NOWRT
;
; Shut off explicit blank processing after passing the command name.
;
BLANKS_OFF:
        .WORD     0                         ; No registers saved (or used)
        BBCC      #TPA$V_BLANKS,TPA$L_OPTIONS(AP),10$
10$:    RET
;
; Check the UIC for legal value range.
;
CHECK_UIC:
        .WORD     0                         ; No registers saved (or used)
        TSTW      UIC_GROUP+2               ; UIC components are 16 bits
        BNEQ      10$
        TSTW      UIC_MEMBER+2
        BNEQ      10$
        MOVW      UIC_GROUP,FILE_OWNER+2    ; Store actual UIC
        MOVW      UIC_MEMBER,FILE_OWNER     ; After checking
        RET
10$:    CLRL      R0                        ; Value out of range - fail
        RET                                 ; The transition
;
; Store a directory name component.
;
STORE_NAME:
        .WORD     0                         ; No registers saved (or used)
        MOVL      NAME_COUNT,R1             ; Get count of names so far
        CMPL      R1,#8                     ; Maximum of 8 permitted
        BGEQU     10$
        INCL      NAME_COUNT                ; Count this name
        MOVAQ     DIRNAME1[R1],R1           ; Address of next descriptor
        MOVQ      TPA$L_TOKENCNT(AP),(R1)   ; Store the descriptor
        CMPL      (R1),#9                   ; Check the length of the name
        BGTRU     10$                       ; Maximum is 9
        RET
10$:    CLRL      R0                        ; Error in directory name
        RET
;
; Convert a UIC into its equivalent directory file name.
;
MAKE_UIC:
        .WORD     0                         ; No registers saved (or used)
```

```
        TSTB    UIC_GROUP+1             ; Check UIC for byte values,
        BNEQ    10$                     ; Since UIC type directories
        TSTB    UIC_MEMBER+1            ; Are restricted to this form
        BNEQ    10$
        MOVL    #6,DIRNAME1             ; Directory name is 6 bytes
        MOVAL   UIC_STRING,DIRNAME1+4   ; Point to string buffer
        $FAOL   CTRSTR=FAO_STRING,-     ; Convert UIC to octal string
                OUTBUF=DIRNAME1,-
                PRMLST=UIC_GROUP
        RET
10$:    CLRL    R0                      ; Range error - fail it
        RET

FAO_STRING:     .LONG   STRING_END-STRING_START
STRING_START:   .ASCII  '!OB!OB'
STRING_END:

        .END    CREATE_DIR
```

G.2 Sample BLISS Program Using LIB$TPARSE

```
MODULE CREATE_DIR (                     ! Create directory file
                IDENT = 'X0000',
                MAIN = CREATEDIR) =
BEGIN
!+
!
! This is a sample program that accepts and parses the command line
! of the CREATE/DIRECTORY command. Note that this is not in fact
! from the VAX CREATE/DIRECTORY utility!  It is a hypothetical
! example program. This program contains the operating system call
! to acquire the command line from the CLI and parse it with
! TPARSE, leaving the necessary information in its global data
! base. The command line is of the following format:
!
!       CREATE/DIR DEVICE:[MARANTZ.ACCOUNT.OLD]
!                  /UIC=[2437,25]
!                  /ENTRIES=100
!                  /PROTECTION=(SYSTEM:R,OWNER:RWED,GROUP:R,WORLD:R)
!
! The three qualifiers are optional. Alternatively, the command
! may take the form
!
!        CREATE/DR DEVICE:[202,31]
!
! using any of the optional qualifiers.
!
!-

!+
!
! Global data, control blocks, etc.
!
!-

LIBRARY 'SYS$LIBRARY:STARLET';
LIBRARY 'SYS$LIBRARY:TPAMAC';

!
! Macro to make the TPARSE control block addressable as a block
! through the argument pointer.
!
```

```
MACRO
                TPARSE_ARGS =
                        BUILTIN AP;
                        MAP AP : REF BLOCK [,BYTE];
                        %;
!
! Declare routines in this module.
!
FORWARD ROUTINE
        CREATE_DIR,                             ! Mail program
        BLANKS_OFF,                             ! No explicit blank processing
        CHECK_UIC,                              ! Validate and assemble UIC
        STORE_NAME,                             ! Store next directory name
        MAKE_UIC;                               ! Make UIC into directory name
!
! Define parser flag bits for flags longword
!
LITERAL
        UIC_FLAG        = 0,                    ! /UIC seen
        ENTRIES_FLAG    = 1,                    ! /ENTRIES seen
        PROT_FLAG       = 2;                    ! /PROTECTION seen
!
! CLI request descriptor block to get the command line
!
OWN
        REQ_COMMAND = $CLIREQDESC (
                RQTYPE = CLI$K_GETCMD
                                                );
!
! TPARSE parameter block
!
OWN
        TPARSE_BLOCK    : BLOCK [TPA$K_LENGTH0, BYTE]
                INITIAL (TPA$K_COUNT0,          ! Longword count
                        TPA$M_ABBREV            ! Allow abbreviation
                        OR TPA$M_BLANKS);       ! Process spaces explicitly
!
! Parser global data
!
OWN
       PARSER_FLAGS     : BITVECTOR [32],       ! Keyword flags
       DEVICE_STRING    : VECTOR [2],           ! Device string descriptor
       ENTRY_COUNT,                             ! Space to preallocate
       FILE_PROTECT,                            ! Directory file protection
       UIC_GROUP,                               ! Temp for UIC group
       UIC_MEMBER,                              ! Temp for UIC member
       FILE_OWNER,                              ! Actual file owner UIC
       NAME_COUNT,                              ! Number of directory names
       UIC_STRING       : VECTOR [6, BYTE],     ! Buffer for string
       NAME_VECTOR      : BLOCKVECTOR [0, 2];   ! Vector of descriptors
       DIRNAME1         : VECTOR [2],           ! Name descriptor 1
       DIRNAME2         : VECTOR [2],           ! Name descriptor 2
       DIRNAME3         : VECTOR [2],           ! Name descriptor 3
       DIRNAME4         : VECTOR [2],           ! Name descriptor 4
       DIRNAME5         : VECTOR [2],           ! Name descriptor 5
       DIRNAME6         : VECTOR [2],           ! Name descriptor 6
       DIRNAME7         : VECTOR [2],           ! Name descriptor 7
       DIRNAME8         : VECTOR [2];           ! Name descriptor 8
!
! Structure macro to reference the descriptor fields in the vector of
! descriptors.
!
MACRO
       STRING_COUNT     = 0, 0, 32, 0%,         ! Count field
       STRING_ADDR      = 1, 0, 32, 0%;         ! Address field
```

```
!+
!
! TPARSE state table to parse the command line
!
!-

$INIT_STATE     (UFD_STATE, UFD_KEY);
!
! Read over the command name (to the first blank in the command).
!
$STATE (START,
        (TPA$_BLANK,, BLANKS_OFF),
        (TPA$_ANY, START)
        );
!
! Read device name string and trailing colon.
!
$STATE (,
        (TPA$_SYMBOL,,,, DEVICE_STRING)
        );

$STATE (,
        (':')
        );
!
! Read directory string, which is either a UIC string or a general
! directory string.
!
$STATE (,
        ((UIC),, MAKE_UIC),
        ((NAME))
        );
!
! Scan for options until end of line is reached
!
$STATE (OPTIONS,
        ('/'),
        (TPA$_EOS, TPA$_EXIT)
        );

        ('UIC', PARSE_UIC,, 1^UIC_FLAG, PARSER_FLAGS),
        ('ENTRIES', PARSE_ENTRIES,, 1^ENTRIES_FLAG, PARSER_FLAGS),
        ('PROTECTION', PARSE_PROT,, 1^PROT_FLAGS, PARSER_FLAGS)
        );

!
! Get file owner UIC
!
$STATE (PARSE_UIC,
        (':'),
        ('=')
        );

$STATE (,
        ((UIC), OPTIONS)
        );
!
! Get number of directory entries
!
$STATE (PARSE_ENTRIES,
        (':'),
        ('=')
        );
```

```
$STATE (,
        (TPA$_DECIMAL, OPTIONS,,, ENTRY_COUNT)
        );
!
! Get directory file protection. Note that the bit masks generate the
! protection in complement form. It will be uncomplemented by the main
! program.
!
$STATE (PARSE_PROT,
        (':'),
        ('=')
        );

$STATE (,
        ('(')
        );

$STATE (NEXT_PRO,
        ('SYSTEM', SYPR),
        ('OWNER', OWPR),
        ('GROUP', GRPR),
        ('WORLD', WOPR)
        );

$STATE (SYPR,
        (':'),
        ('=')
        );

$STATE (SYPRO,
        ('R', SYPRO,, %X'0001', FILE_PROTECT),
        ('W', SYPRO,, %X'0002', FILE_PROTECT),
        ('E', SYPRO,, %X'0004', FILE_PROTECT),
        ('D', SYPRO,, %X'0008', FILE_PROTECT),
        (TPA$_LAMBDA, ENDPRO)
        );

$STATE (OWPR,
        (':'),
        ('=')
        );

$STATE (OWPRO,
        ('R', OWPRO,, %X'0010', FILE_PROTECT),
        ('W', OWPRO,, %X'0020', FILE_PROTECT),
        ('E', OWPRO,, %X'0040', FILE_PROTECT),
        ('D', OWPRO,, %X'0080', FILE_PROTECT),
        (TPA$_LAMBDA, ENDPRO)
        );

$STATE (GRPR,
        (':'),
        ('=')
        );

$STATE (GRPRO,
        ('R', GRPRO,, %X'0100', FILE_PROTECT),
        ('W', GRPRO,, %X'0200', FILE_PROTECT),
        ('E', GRPRO,, %X'0400', FILE_PROTECT),
        ('D', GRPRO,, %X'0800', FILE_PROTECT),
        (TPA$_LAMBDA, ENDPRO)
        );
```

```
$STATE (WOPR,
        (':'),
        ('=')
        );

$STATE (WOPRO,
        ('R', WOPRO,, %X'1000', FILE_PROTECT),
        ('W', WOPRO,, %X'2000', FILE_PROTECT),
        ('E', WOPRO,, %X'4000', FILE_PROTECT),
        ('D', WOPRO,, %X'8000', FILE_PROTECT),
        (TPA$_LAMBDA, ENDPRO)
        );

$STATE (ENDPRO,
        (', ', NEXT_PRO),
        (')', OPTIONS)
        );
!
! Subexpression to parse a UIC string.
!
$STATE (UIC,
        ('[')
        );

$STATE (,
        (TPA$_OCTAL,,,, UIC_GROUP)
        );

$STATE (,
        (', ')
        );

$STATE (,
        (TPA$_OCTAL,,,, UIC_MEMBER)
        );

$STATE (,
        (']', TPA$_EXIT, CHECK_UIC)
        );
!
! Subexpression to parse a general directory string
!
$STATE (NAME,
        ('[')
        );

$STATE (NAME0,
        (TPA$_STRING,, STORE_NAME)
        );

$STATE (,
        (',', NAME0),
        (']', TPA$_EXIT)
        );

PSECT  OWN = $OWN$;
PSECT  GLOBAL = $GLOBAL$;
GLOBAL ROUTINE CREATE_DIR (START_ADDR, CLI_CALLBACK) =

!+
!
! This is the main program of the CREATE/DIRECTORY utility. It gets
! the command line from the CLI and parses it with TPARSE.
!
!-
```

```
LOCAL
        STATUS;                             ! Status from LIB$TPARSE

EXTERNAL ROUTINE
       LIB$TPARSE       : ADDRESSING_MODE (GENERAL);
!
! Call the CLI to obtain the command line.
!
(.CLI_CALLBACK) (REQ_COMMAND, 0, 0);

!
! Copy the input string descriptor into the TPARSE control block
! and call TPARSE. Note that impure storage is assumed to be zero.
!
TPARSE_BLOCK[TPA$L_STRINGCNT] = .REQ_COMMAND[CLI$W_RQSIZE];
TPARSE_BLOCK[TPA$L_STRINGPTR] = .REQ_COMMAND[CLI$L_RQADDR];
STATUS = LIB$TPARSE (TPARSE_BLOCK, UFD_STATE, UFD_KEY);
IF NOT .STATUS
THEN
                .
                .
       (Handle syntax error)
                .
                .
!
! Parsing is complete. The utility may now go about its business.
!
                .
                .
                .
                .
                .
                .
                .
RETURN 1;
END;                        ! End of routine CREATE_DIR

!+
!
! Parser action routines
!
!-


!
! Shut off explicit blank processing after passing the command name.
!
ROUTINE BLANKS_OFF =
    BEGIN
    TPARSE_ARGS;

    AP[TPA$V_BLANKS] = 0;
    1
    END;

!
! Check the UIC for legal value range.
!
ROUTINE CHECK_UIC =
    BEGIN
    TPARSE_ARGS;

    IF .UIC_GROUP<16,16> NEQ 0
    OR .UIC_MEMBER<16,16> NEQ 0
    THEN RETURN 0;
```

```
    FILE_OWNER<0,16> = .UIC_MEMBER;
    FILE_OWNER<16,16> = .UIC_GROUP;
    1
    END;

!
! Store a directory name component.
!
ROUTINE STORE_NAME =
    BEGIN
    TPARSE_ARGS;

    IF .NAME_COUNT GEQU 8
    OR .AP[TPA$L_TOKENCNT] GTRU 9
    THEN RETURN 0;
    NAME_COUNT = .NAME_COUNT + 1;
    NAME_VECTOR [.NAME_COUNT, STRING_COUNT] = .AP[TPA$L_TOKENCNT];
    NAME_VECTOR [.NAME_COUNT, STRING_ADDR] = .AP[TPA$L_TOKENPTR];
    1
    END;

!
! Convert a UIC into its equivalent directory file name.
!
ROUTINE MAKE_UIC =
    BEGIN
    TPARSE_ARGS;

    IF .UIC_GROUP<8,8> NEQ 0
    OR .UIC_MEMBER<8,8> NEQ 0
    THEN RETURN 0;
    DIRNAME1[0] = 0;
    DIRNAME1[1] = UIC_STRING;
    $FAOL (CTRSTR = UPLIT (6, UPLIT BYTE ('!OB!OB')),
           OUTBUF = DIRNAME1,
           PRMLST = UIC_GROUP
           );
    1
    END;

END
ELUDOM                  ! End of module CREATE_DIR
```

Index

D

E

F

G

H

I

M

N

Q

R

S

T

U

V

VAX-11 Run-Time
Reference Manual
AA-D036B-TE

Reader's Comments

Note: This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

— — —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

ATTN: Commercial Engineering Publications MK1-2/ H3
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK N.H. 03054

— — — Do Not Tear - Fold Here and Tape —