digital

VAX-11 Guide to Creating Modular Library Procedures

Order No. AA-H500B-TE

VAX11

April 1980

This document describes how to design and code procedures for insertion in an object module library or a sharable image. It includes the modular programming standard and recommendations for modular programming in any language.

VAX-11 Guide to Creating Modular Library Procedures

Order No. AA-H500B-TE

OPERATING SYSTEM AND VERSION:	VAX/VMS	V2.0
SOFTWARE VERSION:	VAX/VMS	V2.0

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, February 1979
Revised, April 1980

Contents

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

Preface

Document Objectives

A procedure is modular if it follows rules and principles that permit it to be successfully linked with other procedures following the same rules and principles. The *VAX-11 Guide to Creating Modular Library Procedures* amasses these rules and principles into the modular programming standard. Following this standard will lead to more reliable programs, and reduce checkout time and maintenance effort.

This manual is a tutorial guide to designing and coding modular procedures written in VAX-11 MACRO, BLISS-32, VAX-11 BASIC, VAX-11 FORTRAN or VAX-11 PASCAL. You can use these procedures for general programming or for inclusion in a procedure library. The libraries include the system default object library, user-created object libraries, and user-created sharable images.

The guide includes required and optional modular programming techniques, recommended style, and a description of how to install modular procedures in DIGITAL-supplied and user-created libraries.

Intended Audience

This manual is intended for advanced system and applications programmers who are already familiar with VAX/VMS system concepts. Readers should be familiar with the VAX/VMS operating system and proficient in a language supported by VAX/VMS.

Document Structure

All chapters in this manual are tutorial.

- Chapter 1 is an overview of modular programming and of libraries. It explains the options you have in creating your own procedures and libraries and how to determine the type of library you should create.
- Chapter 2 explains how to design and document the interface between a modular procedure and its calling program.

- Chapter 3 describes how procedures use storage and how to maintain modularity while using different types of storage.
- Chapter 4 describes specific modular coding techniques in VAX-11 MACRO, BLISS-32, VAX-11 BASIC, VAX-11 FORTRAN and VAX-11 PASCAL. This includes required and optional parts of the standard for initialization, resource allocation, passing strings, use of system services and invoking user action routines.
- Chapter 5 describes how to signal and return error conditions from modular procedures.
- Chapter 6 describes programming techniques that allow asynchronous system traps (ASTs) to occur without conflicting with executing modular procedures.
- Chapter 7 describes how to insert or replace a procedure in the system default object library, and how to create and link with either a user-created object library or a user-created sharable image.

The appendixes provide useful background information:

- Appendix A is the VAX-11 Modular Programming Standard, consisting of required, optional, and recommended rules and principles. Required rules must be followed. Optional rules must be followed or documented in the procedure documentation as not being followed. Recommendations are suggestions for programming style, but are not necessary for procedures to be modular.
- Appendix B presents the notation for describing procedure parameters.
- Appendix C details the VAX/VMS naming conventions.

Associated Documents

The following documents are associated with this manual:

- *VAX-11 Run-Time Library Reference Manual*
- *VAX/VMS System Services Reference Manual*
- *VAX-11 Linker Reference Manual*
- *VAX-11 BASIC User's Guide*
- *VAX-11 BASIC 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-11 MACRO User's Guide*
- *VAX-11 MACRO Language Reference Manual*
- *VAX-11 BLISS-32 User's Guide*
- *VAX-11 BLISS-32 Language Guide*

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 decimal.

Unless otherwise specified, all commands end with a carriage return.

Lowercase characters indicate variable information; uppercase characters indicate literal information, which you must enter exactly as shown.

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

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
- 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

In diagrams, the following conventions are used:

————————► control path

– – – – – – ► data path

• ——— • ———► interface

Summary of Technical Changes

This manual documents the *VAX-11 Guide to Creating Modular Procedures* Version 2.0. This section summarizes the technical changes from Version 1.5.

The following languages have been added:

- VAX-11 BASIC
- VAX-11 PASCAL

The following facilities have been added:

- BAS$ - VAX-11 BASIC specific support procedures
- PAS$ - VAX-11 PASCAL specific support procedures
- STR$ - String support procedures

The following have been added to the list of acceptable datatypes:

• bpv	Bound Procedure Value
• cit	COBOL intermediate temporary
• dc	D–floating complex
• dsc	Descriptor (used by descriptors)
• g	Double Precision G–floating
• gc	G–floating complex
• h	Quadruple precision H–floating
• hc	H–floating complex
• o	Octaword integer (signed)
• ou	Octaword logical (unsigned)

The following have been added to the list of acceptable parameter forms:

• nca	Non-contiguous array
• sd	scalar decimal descriptor

In addition, the VAX-11 Modular Programming Standard, Appendix A, underwent a major revision. Most of the original rules and principles have been revised to make them more easily understood. There are now 62 rules or recommendations in the standard.

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

Chapter 1 Introduction

A procedure is a set of related instructions that performs a task. Typically, a procedure is invoked by executing a VAX–11 CALLS or CALLG instruction. Each language defines a procedure differently; for example, in:

- MACRO, a procedure begins with .ENTRY and ends with RET
- BLISS, a procedure is declared as a ROUTINE with the default linkage
- BASIC, a procedure is a main program, subprogram, or function
- FORTRAN, a procedure is a main program, subroutine, or function
- PASCAL, a procedure is declared as a procedure or function

A procedure is modular if it follows rules and principles that permit it to be successfully linked together in arbitrary ways with other procedures that follow the same rules and principles. You can use modular procedures for general programming or for inclusion in procedure libraries. Libraries are merely a way of collecting procedures for ease of access by calling programs.

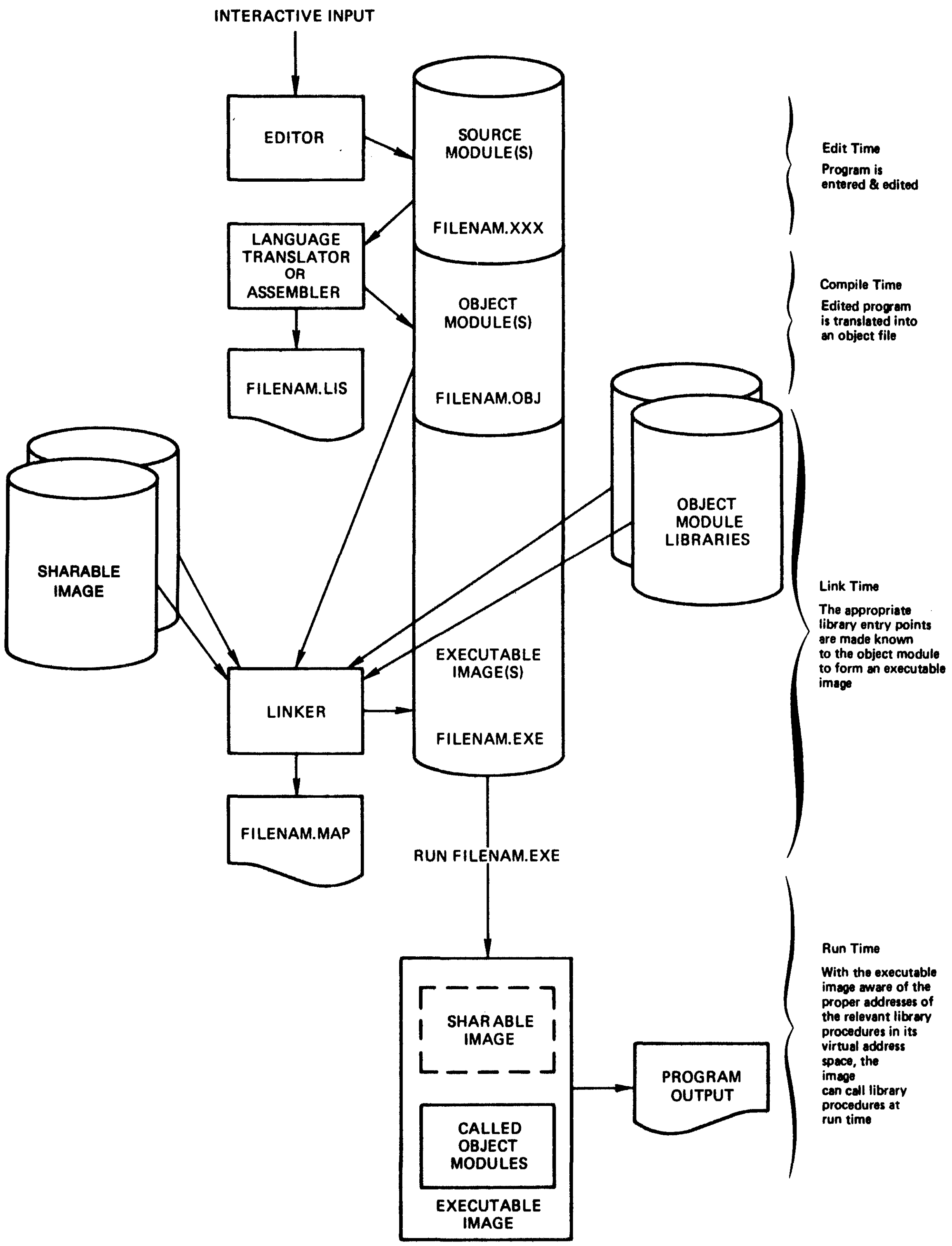
NOTE

Much of this manual refers to libraries of procedures. However, the discussion also applies to general programming.

The VAX–11 Linker resolves references to external procedure names by searching any user library specified in the LINK command followed by the default system libraries. A program can then call library procedures at run time.

Figure 1–1 shows the development of a program that calls one or more procedures in a library. Depending on the options you select when writing modular procedures, you can control linker access to your procedures and, subsequently, the way procedures appear at run time. For example, procedures within a sharable image save memory and disk space, because all user processes access a single copy.

Figure 1-1: Developing a Program that Calls Library Procedures


INTERACTIVE INPUT
EDITOR
SOURCE MODULE(S)
FILENAM.XXX
LANGUAGE TRANSLATOR OR ASSEMBLER
OBJECT MODULE(S)
FILENAM.OBJ
FILENAM.LIS
SHARABLE IMAGE
OBJECT MODULE LIBRARIES
EXECUTABLE IMAGE(S)
FILENAM.EXE
LINKER
FILENAM.MAP
RUN FILENAM.EXE
SHARABLE IMAGE
CALLED OBJECT MODULES
EXECUTABLE IMAGE
PROGRAM OUTPUT
Edit Time
Program is entered & edited
Compile Time
Edited program is translated into an object file
Link Time
The appropriate library entry points are made known to the object module to form an executable image
Run Time
With the executable image aware of the proper addresses of the relevant library procedures in its virtual address space, the image can call library procedures at run time

1.1 Using Libraries with VAX/VMS

Procedures can be grouped in libraries in one of two ways:

- As an object module library. A call to a procedure in an object module library causes the linker to copy and link the module containing the procedure into the calling program's object file. The module and the program then become a single executable image.
- As a sharable image. A call to a procedure in a sharable image causes the linker to reserve space for the entire contents of the sharable image in the program's executable image. The sharable image is not (usually) copied into the executable image. Instead, it is mapped into the process address space at run time.

Sections 1.1.1 through 1.1.4 describe the default system libraries and recommendations for creating object module libraries and sharable images.

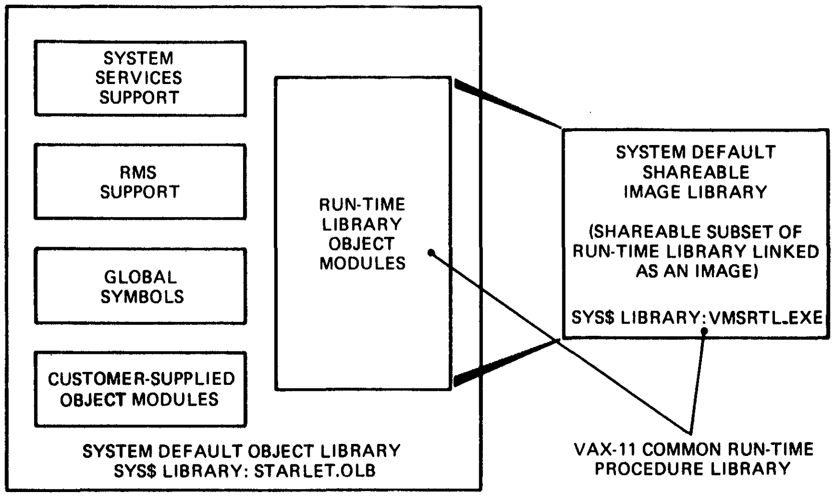
1.1.1 DIGITAL-Supplied Libraries

The VAX-11 Common Run-Time Procedure Library, supplied by DIGITAL, consists of modular procedures that provide support for components of the VAX/VMS system. The Run-Time Library includes procedures that support the language compilers, as well as those that are generally useful to programs. Procedures from the Run-Time Library exist in two forms:

- The default system object module library, STARLET.OLB, contains all procedures.
- The default system sharable image, VMSRTL.EXE, contains a subset of the VAX-11 Common Run-Time Procedure Library made sharable to save memory.

Figure 1-2 shows the VAX/VMS libraries, including the default system object module library, STARLET.OLB, and the sharable image, VMSRTL.EXE.

Figure 1-2: DIGITAL-Supplied Libraries



SYSTEM SERVICES SUPPORT
RMS SUPPORT
GLOBAL SYMBOLS
CUSTOMER-SUPPLIED OBJECT MODULES
RUN-TIME LIBRARY OBJECT MODULES
SYSTEM DEFAULT OBJECT LIBRARY SYS$ LIBRARY: STARLET.OLB
SYSTEM DEFAULT SHAREABLE IMAGE LIBRARY
(SHAREABLE SUBSET OF RUN-TIME LIBRARY LINKED AS AN IMAGE)
SYS$ LIBRARY:VMSRTL.EXE
VAX-11 COMMON RUN-TIME PROCEDURE LIBRARY

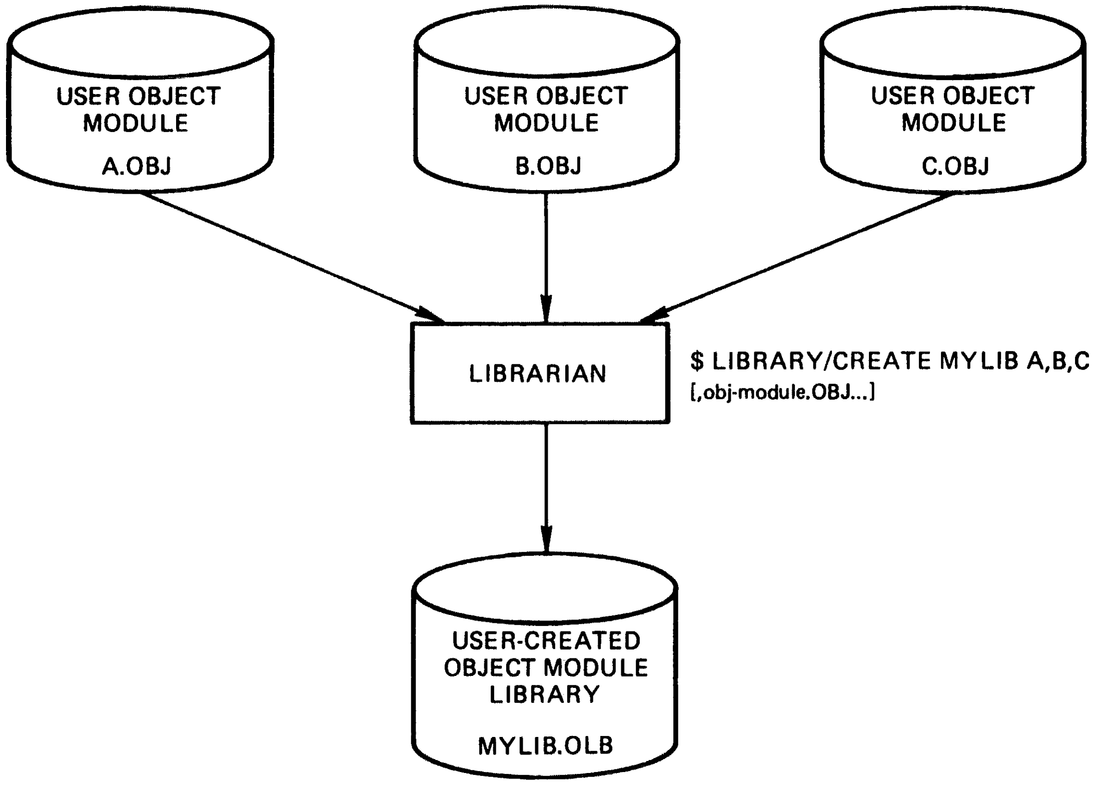
The linker automatically searches both of these libraries for unresolved references to global symbols as a result of the LINK command. First, the linker searches VMSRTL.EXE, a sharable subset of STARLET.OLB. If the linker resolves a reference with this sharable image, it reserves space for (as opposed to copying) the entire sharable image at the end of the executable program image being created.

After searching VMSRTL.EXE, the linker searches the default object library, STARLET.OLB, for any remaining unresolved references. If the linker finds one, it copies the pertinent module into the executable image. At run-time the sharable image is mapped into the end of the process address space if the linker had reserved space for it.

1.1.2 User-Created Object Module Libraries

A user-created object module library consists of procedures you can write in any programming language. You can create an object library from object files using the LIBRARY command (see the *VAX/VMS Command Language User's Guide*). The default file type for object module library files is OLB. For object files, it is OBJ. Figure 1-3 shows the development of a user-created object library.

Figure 1-3: Creating an Object Module Library



You can either explicitly or implicitly include library modules in the program being created:

- Implicit inclusion occurs when a file specified in the LINK command contains an object module that refers to a global symbol defined in the library that the linker searches.
- Explicit inclusion occurs when you name a module with the /INCLUDE qualifier after the library name in the LINK command.

The linker follows these conventions in using object libraries:

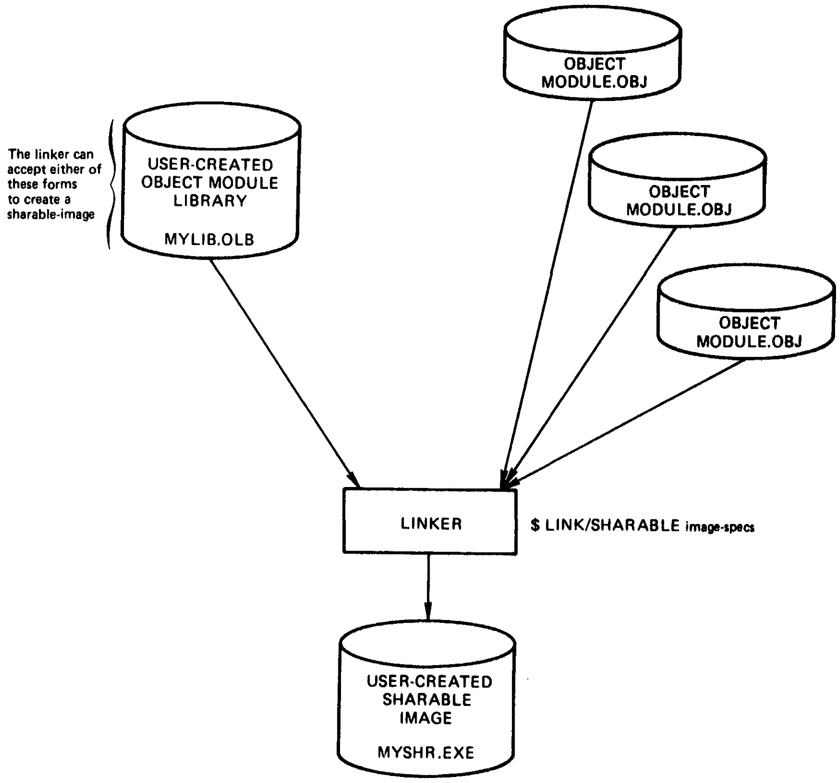
- The linker processes all input files, including libraries, in the sequence in which you name them.
- If you specify both the /LIBRARY and /INCLUDE qualifiers after a library file specification, the linker includes the named module first and then, if necessary, searches the library.
- The linker searches the default system library for unresolved references after it has processed all named input files, including user libraries.

The *VAX-11 Linker Reference Manual* provides more information on the linker's use of libraries.

1.1.3 User-Created Sharable Images

Figure 1-4 shows the development of a user-created sharable image.

Figure 1-4: Creating a Sharable Image

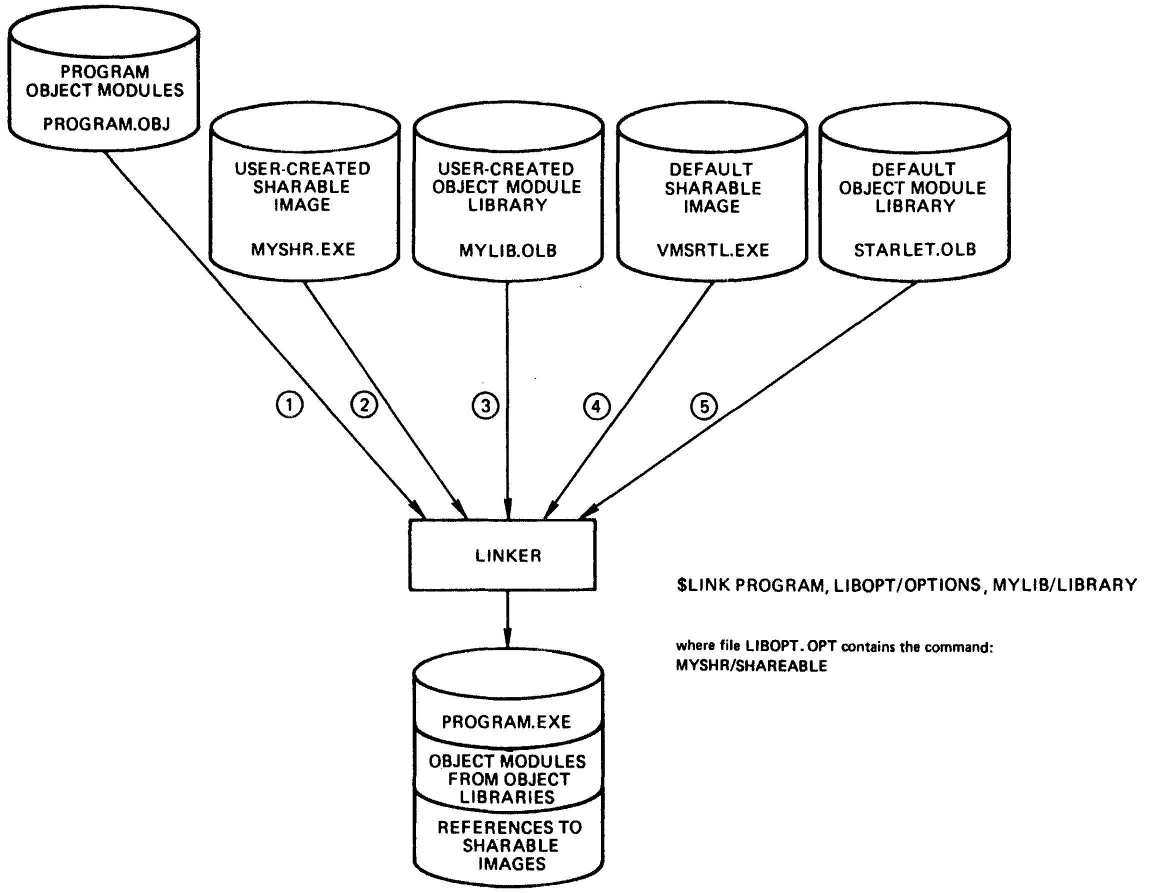

OBJECT MODULE.OBJ
The linker can accept either of these forms to create a sharable-image
USER-CREATED OBJECT MODULE LIBRARY
MYLIB.OLB
OBJECT MODULE.OBJ
OBJECT MODULE.OBJ
LINKER
$ LINK/SHARABLE image-specs
USER-CREATED SHARABLE IMAGE
MYSHR.EXE

A user-created sharable image can consist of a subset of a user-created object library. It contains modular procedures usually written in position-independent code; the procedures should be used frequently enough to warrant being shared among processes. You can specify the user-created sharable image as input to the linker by using the /OPTIONS qualifier after the name of the options file in the LINK command. The *VAX-11 Linker Reference Manual* details the benefits and uses of sharable images.

1.1.4 Linking Programs to Run-Time Libraries

Figure 1-5 shows how each type of Run-Time Library is linked to a program object module to form an executable image.

Figure 1-5: Linking Programs to Run-Time Libraries


PROGRAM OBJECT MODULES
PROGRAM.OBJ
USER-CREATED SHARABLE IMAGE
MYSHR.EXE
USER-CREATED OBJECT MODULE LIBRARY
MYLIB.OLB
DEFAULT SHARABLE IMAGE
VMSRTL.EXE
DEFAULT OBJECT MODULE LIBRARY
STARLET.OLB
1
2
3
4
5
LINKER
$LINK PROGRAM, LIBOPT/OPTIONS, MYLIB/LIBRARY
where file LIBOPT.OPT contains the command:
MYSHR/SHAREABLE
PROGRAM.EXE
OBJECT MODULES FROM OBJECT LIBRARIES
REFERENCES TO SHARABLE IMAGES

When the link command shown is given, these events occur:

1. PROGRAM.OBJ is linked into the image.
2. MYSHR.EXE, the user-created sharable image specified indirectly with the options file LIBOPT.OPT, is unconditionally included. References (if any) are resolved and address space is allocated.
3. MYLIB.OLB, the user-created object library specified in the LINK command, is searched. If references are resolved, the VAX-11 Linker includes a copy of the modules resolving those references in the image.

4. VMSRTL.EXE, the default sharable image, is automatically included if and only if it resolves any remaining unresolved references.

NOTE

> You can use the /NOSYSSHR qualifier to request the linker to omit the search of the default sharable image.

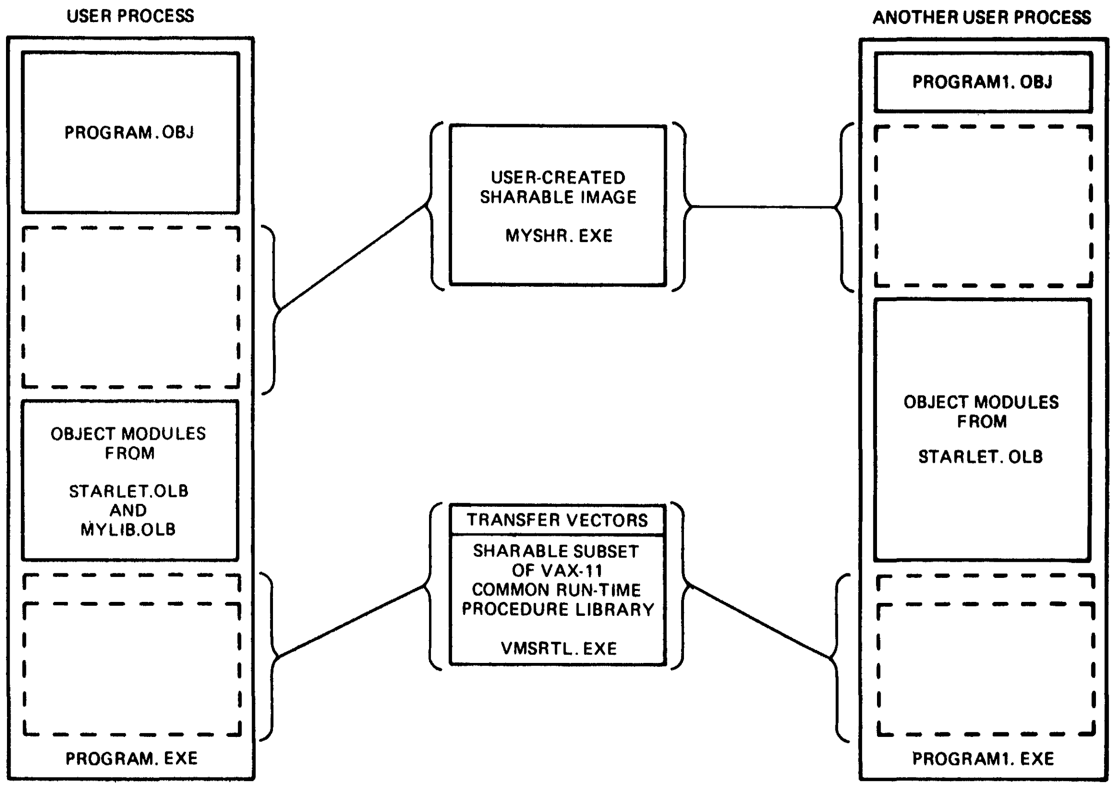
5. STARLET.OLB, the default object library, is automatically searched if any unresolved references remain. If references are resolved, the VAX-11 Linker includes a copy of the modules in the image that resolve those references.

NOTE

> You can use the /NOSYSLIB qualifier to request the VAX-11 Linker to omit the search of both the default sharable image and the default object module library.

The resulting executable image can be executed in a user process by using the RUN command. This is shown in Figure 1-6.

Figure 1-6: Executing an Image that Calls Library Procedures



USER PROCESS
ANOTHER USER PROCESS
PROGRAM.OBJ
OBJECT MODULES FROM STARLET.OLB AND MYLIB.OLB
PROGRAM. EXE
USER-CREATED SHARABLE IMAGE
MYSHR. EXE
TRANSFER VECTORS
SHARABLE SUBSET OF VAX-11 COMMON RUN-TIME PROCEDURE LIBRARY
VMSRTL. EXE
PROGRAM1. OBJ
OBJECT MODULES FROM STARLET. OLB
PROGRAM1. EXE

Note that copies of the modules taken from STARLET.OLB and MYLIB.OLB are bound with each image that links with the object libraries; in contrast, the sharable modules in VMSRTL.EXE and MYSHR.EXE reside in individual image files shared between processes at run-time.

1.2 Designing and Coding Modular Procedures

To ensure that your procedures are compatible with all other procedures and programs executing on VAX/VMS, you should follow the programming techniques and recommendations described in this manual.

The modular programming standard specifies:

- A subset of the VAX-11 Procedure Calling Standard to be used, and
- Additional techniques and recommendations for modular programming

The standard is used by DIGITAL to develop VAX/VMS library software.

Any modular procedure can be placed in an object library, a sharable image, or both.

Any procedure placed in a sharable image should also be placed in an object module library. Then, if you want, a particular module can be extracted. Also, if a very large program is close to your system's virtual memory limit, you can include only called modules from the object library rather than allocate virtual memory for the entire sharable image.

1.2.1 Advantages of Modular Programming

The modular programming "standard" described in this manual offers several advantages over writing a complex program as a single source module. However, there is much more to modular programming than breaking a program into procedures. Each procedure must observe a number of rules and principles to be modular, that is, to be able to be combined in arbitrary ways with other procedures following the same rules and principles to form programs. Thess following the same rules and principles to form programs. These rules and principles are collected in the form of a standard in Appendix A.

If you follow all the required elements of the standard, you will gain the following advantages:

- You can use any modular procedure in any program.
- You can add a modular procedure to a library at any time.
- You need not rewrite common algorithms every time a new program needs them.
- You can divide a complex program into simpler procedures to reduce development time, reduce complexity, and increase reliability.
- You can replace a procedure without modifying the calling program.
- You can control process-wide resource allocation.
- You can use different programming languages to write different procedures for a program.

Following the optional elements of the standard specified in this manual yields these additional advantages:

- Sharable library procedures can save memory and link time.
- AST-reentrant procedures can be called by AST-level procedures.
- Modular procedures that conform to all coding recommendations are similar in format and therefore are easier to use and maintain.
- Structured programming recommendations let your procedures work together in a logical pattern.

1.2.2 Modular Programming Standard Parts

Appendix A lists the elements of the modular programming standard explained in this manual. There are three parts to the standard:

- Required elements.
- Optional elements that you must follow or document your intent not to follow.
- Recommendations that make it easier for your modules to be used by others. However, not following the recommendations does not affect modularity.

The following sections describe major aspects of the modular programming standard.

1.2.3 Storage

Most procedures use some type of storage to retain information either during a single procedure activation or between successive activations. The types of storage are:

- Static
- Stack
- Heap

While any modular procedure can use any of the three types, there are certain principles that should be followed. These principles are explained in Chapter 3.

1.2.4 Naming Rules and Recommendations

This manual describes naming rules for procedures, modules, and program sections (PSECTS). It also explains the naming recommendations for file names. (See Sections 2.2 and 4.2.)

1.2.5 Process-Wide Resource Allocation

Process-wide resources can be allocated as needed to any procedure in a process. They include:

- Virtual memory
- Logical unit numbers
- Event flags
- Dynamic strings

Moreover, you can create additional resources. Modular procedures follow the standard of allocating resources by calling a resource-allocating procedure (they do not allocate the resource directly themselves). This prevents conflicts that could occur if two procedures were to allocate the same resource. The available resources and allocation methods are described in Section 4.4.

1.2.6 Use of System Services

Modular procedures can use system services that conform to the modular programming standard. Section 4.6 lists all system services and indicates those that modular procedures can use.

1.2.7 Signaling and Condition Handling

Modular procedures adhere to rules to indicate errors. For example, modular procedures either return a condition value or call system-signaling procedures to output error messages. Chapter 5 describes the programming rules for signaling and condition handling and techniques of signaling between related procedures.

1.2.8 AST-Reentrant Procedures

VAX/VMS provides a mechanism to interrupt image execution in response to an external asynchronous event. When the event occurs, a user-supplied asynchronous system trap (AST) routine is called.

An AST-reentrant procedure can be interrupted and reexecuted before resuming execution at the point of the interrupt. Thus, it can be called from AST-level and/or non-AST-level routines. Most modular procedures are AST-reentrant. Chapter 6 describes how to write AST-reentrant modular procedures.

1.2.9 Position-Independent Code

Position-independent code executes correctly no matter where it is placed in the virtual address space after it is linked. To be modular, a sharable image must contain position-independent code. However, it can have non-position-independent data.

The *VAX-11 Linker Reference Manual* discusses position-independent code in detail.

1.2.10 Transfer Vectors

Transfer vectors eliminate the need to relink images that call procedures in a sharable image every time a new version of the sharable image is installed. You can add transfer vectors to procedures in a sharable image at any time, as Chapter 7 explains.

1.3 Creating and Modifying Libraries

You can add procedures to the default system object library (STARLET.OLB). You can modify procedures in your own object library or your own sharable image. (See Chapter 7 for more detail.)

1.3.1 Creating and Updating Object Libraries

You can create or add modules to an object module library, including the default system object library (STARLET.OLB), with the LIBRARY command.

This command can also replace a module in an object library.

1.3.2 Creating Sharable Images

You create a sharable image to optimize storage space and access time. These images contain code that many users can share. Specific advantages are:

- Conservation of disk storage space
- Reduction of paging I/O
- Conservation of memory at run time
- Reduction in link time, since a shared library is pre-linked

If your sharable image is written in position-independent code and you have provided transfer vectors, you can eliminate the need to relink all images that called the old version when you install a new version.

Observe these rules-of-thumb when deciding whether to create a sharable image:

- The combined code of all procedures in the planned sharable image should be at least 10K bytes.
- The number of potential simultaneous users for these procedures should be three or more.

1.3.3 Updating Sharable Images

To add or modify code in a sharable image, you must reinstall the entire image. You can update any user-created sharable image.

You cannot add or modify the system default sharable image VMSRTL.EXE. However, you can make a user-created sharable image that contains VMSRTL.EXE, and substitute it for VMSRTL.EXE. However, all user and system programs must then be relinked.

Chapter 2
Designing Modular Procedure Interfaces

If the interface between each procedure and its caller is modular, then any procedure can fit together with any other group of procedures in a program. If you follow the design techniques described in this chapter, your procedures will operate successfully with other modular procedures.

The following design aspects are discussed:

- Checklist of design and coding steps
- Explicit parameter types and passing mechanisms
- Implicit parameters
- Documentation of procedure functions
- Control of human readable output
- Timer and resource allocation procedures

This chapter contains required rules and principles that must be followed to ensure modularity, optional rules and principles that require documentation if not followed, and recommendations that are suggested to ensure uniformity and ease of use.

2.1 Checklist of Design and Coding Steps

The following checklist should help you:

- Design the interface between the procedure and its caller
- Design modular procedures
- Code procedures

The section numbers indicate where you can find detailed information.

1. Select procedure name(s) and facility name (see Section 2.2).
2. Define a procedure's explicit parameters (see Section 2.3).

   Choose these characteristics for each explicit parameter (see Sections 2.3.1 and 2.3.2):

   - Access Type
   - Data Type
   - Passing Mechanism
   - Parameter Form

   Place the parameters in the calling sequence in the correct order (see Section 2.3.6).
3. Decide whether and how the procedure will retain information from one activation to another (see Section 2.4).
4. Determine how procedures will indicate error and success conditions (see Section 2.3.7 and Chapter 5).
5. Provide optional action routines if your procedure produces human readable output to a character imaging device (see Section 2.6).
6. Provide statistic and status entry points for any resource allocation procedure (see Section 2.7).
7. Write documentation for procedures and modules (see Section 2.8):

   - Write module descriptions
   - Write procedure descriptions

8. Decide how each procedure will utilize storage. Determine the type of storage to be used and steps required to maintain modularity (see Chapter 3).
9. Consider structured programming recommendations (see Section 4.1).

   Decide:

   - The number of procedures involved
   - How they interact with each other
   - How they are arranged in modules
   - Whether they are potentially sharable

10. Check Appendix A for the complete list of required, optional and recommended elements of the modular programming standard before coding procedures.

11. Determine what resources your procedure will need. If a resource allocation procedure does not exist for the resources you need, write one and add it to STARLET.OLB (see Section 4.4).
12. Code each procedure to handle error conditions (see Chapter 5).
13. Decide whether to make procedures AST-reentrant (see Chapter 6).
14. Follow coding rules and recommendations while writing code (see Section 4.1). Be sure to follow the standard in these areas:
    - Initialization (if needed) (see Section 4.3)
    - Use of system services (if needed) (see Section 4.6)
    - Passing string parameters (if needed) (see Section 4.5)
15. While fixing bugs in your procedures, be sure to maintain modularity.
16. Add debugged procedures to an object module library and/or install as a sharable image, as required (see Chapter 7).

2.2 Procedure Names

Entry point naming standards follow the VAX–11 global symbol-naming standards. A global symbol takes the general form:

```
fac$symbol    (DIGITAL-supplied)
fac_symbol     (user-created)
```

where:

fac
: is, typically, a 3–character facility name.

symbol
: is a 1 to n-character symbol, such that the entire global symbol does not exceed 15 characters.

NOTE

Until all DIGITAL-supported VAX languages provide 31–character symbols, library procedures should limit global names to 15 characters.

A symbol generally consists of a verb followed by the object. Together, verb and object describe the procedure's action, such as LIB$GET_VM (Get Virtual Memory). The facility name and the character symbol are separated by a single dollar sign if the procedure is DIGITAL-supplied, and by an underscore if the procedure is user-created. This convention avoids conflict between DIGITAL and user procedure names.

Some global procedures are not intended to be part of the modular interface and are only internally available within a set of procedures. These procedures' names are differentiated by a double dollar sign if they are DIGITAL-supplied, and by a triple underscore if they are user-created. Note that three underscores are used to differentiate user-created internal global entry point names from user-created condition value symbols with two underscores.

Entry point names of procedures called only from inside the same module need not include the facility name, provided the entry point names are not global symbols.

2.2.1 Facility Names

The DIGITAL-defined facility names are registered in a DIGITAL-maintained, system-wide registry. These facility names are used in the VAX-11 Run-Time Library:

LIB	General purpose
MTH	Mathematics
STR	Strings
OTS	Language-independent support
BAS	BASIC Support
FOR	FORTRAN Support
PAS	PASCAL Support

For language support, the facility name is generally the same as the default file type for the language. Appendix B contains other available facility names.

You can also create your own facility names if none of the DIGITAL-defined ones are appropriate.

2.2.2 Condition Value Symbols

Condition value symbols symbolically define unique, system-wide, 32-bit condition values. These values are used in return status codes, signal argument lists, and as message identifiers. Condition value symbols have the general form:

```
fac$_symbol    (DIGITAL-supplied)
fac__symbol    (user-created)
```

A unique, 12-bit facility number is assigned to each facility name for the facility number field in a condition value.

2.2.3 Creating Your Own Facilities

You can create your own facilities by defining a unique facility name and facility number. Bit 27 (STS$V_CUST_DEF) of a condition value indicates whether that value is user- or DIGITAL-supplied. This bit must be 1 if the facility number is user-created.

2.3 Explicit Parameters

Explicit parameters are a procedure's primary interface with the outside world. Therefore, rules for parameter types and passing mechanisms must be carefully followed to maintain a modular interface.

2.3.1 Parameter Characteristics

Every parameter has these characteristics:

Characteristic	Example
Access type	read, write, modify, ...
Data type	longword, floating, ASCII text, ...
Passing mechanism	by immediate value, by reference, by descriptor
Parameter form	scalar, array, ...

Table 2–1 lists the possible alternatives for each characteristic. Appendix C describes each alternative in detail. This list is complete for all characteristics allowed by the VAX–11 Procedure Calling Standard.

The letter abbreviations to the left of each characteristic are shorthand notations used in program documentation to record the characteristics of each parameter. The complete parameter description is:

<name>.<access type><datatype>.<passing mechanism><parameter form>

For example, the calling sequence of LIB$GET_INPUT is:

ret-status.wlc.v = LIB$GET_INPUT (get-str.wt.dx [,prompt-str.rt.dx [,outlen.ww.r]])

Table 2-1: Procedure Parameter 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

<parameter form>	
-	Scalar
a	Array reference or descriptor
d	Dynamic string descriptor
nca	Non-contiguous array descriptor
p	Procedure reference or descriptor
s	Fixed length string descriptor
sd	scalar decimal descriptor
x	Class type in descriptor

<passing mechanism>	
d	By descriptor
r	By reference
v	By immediate value

<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	Double precision D-floating
dc	D-floating complex
dsc	Descriptor (used by descriptors)
f	Single precision F-floating
fc	F-Floating complex
g	Double precision G-floating
gc	G-floating complex
h	Quadruple precision 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 integer (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).

2.3.2 Library Facility Passing Mechanisms

Library facilities usually have a distinct interface style for passing mechanisms and data forms. If you use one of the facilities that has already established a technique, you should follow their guidelines. For example, the calling program passes all input scalars to LIB facility procedures by reference. Table 2–2 summarizes the passing mechanisms used with each data form for the library facilities shown.

Table 2–2: Parameter Passing Mechanisms Used by Library Facilities

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

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

2.3.3 Parameter Passing Mechanisms

The procedures designed to be called explicitly from higher level languages should minimize the need for language extensions in the calling program. Therefore, LIB, MTH, and STR accept atomic data types by reference rather than by immediate value, and strings by descriptor rather than by reference. Passing atomic data types differs from calling techniques for VMS System Services: System Services accept atomic data type input parameters by immediate value, not by reference.

2.3.4 String Descriptors

The calling program passes all strings by descriptor to every library facility. The descriptor for the string(s) must have a length and pointer (see Section C.8 in the *VAX-11 Run-Time Procedure Library Reference Manual* for a complete description). Table 2-3 lists the string-passing techniques used for the library facilities shown (See Section 4.5 for passing strings as output parameters).

Table 2-3: String-Passing Techniques Used by Library Facilities

String Type	**String Descriptor Fields**			**Facility**
	Class	**Length**	**Pointer**	
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.

2.3.5 Optional Parameters

An optional parameter is one that the calling program can omit. The calling program indicates the omission by passing argument list entries containing zero. For a trailing optional parameter, the calling program can pass a shortened list or a zero argument list entry.

NOTE

VMS System Services, unlike the Run-Time Library, cannot accept a shortened argument list. Omitted arguments must always be indicated with a zero argument list entry.

For parameters passed by immediate value, there is no distinction between passing a zero value and passing a zero argument list entry.

2.3.6 Order of Parameters

Procedures in the VAX-11 Run-Time Procedure Library follow a consistent pattern for positioning parameters. Procedure designers should group parameters in this left-to-right order:

1. Required input parameters (read access)
2. Required input-output parameters (modify access)
3. Required output parameters (write access)
4. Optional input parameters (read access)
5. Optional input-output parameters (modify access)
6. Optional output parameters (write access)

Note that optional parameters follow required parameters. Omitting optional parameters shortens the parameter list.

Required parameters are accessed in a left-to-right order. The only exceptions are functions in which the function value exceeds 64 bits, such as strings or quadruple precision H-floating, and so cannot be returned in R0/R1. In this case, the calling program uses the first parameter to specify where the function value is to be stored and the other parameters are shifted right one position. (See the VAX-11 Procedure Calling Standard.)

2.3.7 Error and Condition Values

A procedure can indicate errors to its caller by either returning a condition value as a completion code or signaling the error. It is recommended that, whenever possible, modular procedures return a completion code as a function value. Then, when an error occurs, the completion code indicates the error to the caller of the procedure. At that point, the caller can choose a recovery path. For a description of signaling, see Chapter 5 in this manual and Chapter 6 of the *VAX-11 Run-Time Library Reference Manual.* It is harder for the calling program to recover from an error that is signaled than from one returned as a function status.

Procedures in the following facilities handle errors in specific ways:

LIB	Always returns completion code.
MTH	Always signals errors (function value is the mathematical value returned).
OTS STR BAS FOR PAS	Returns completion code when a check of the code will not impose an excessive speed or space penalty on the caller; otherwise, it signals the error. **NOTE** BAS procedures always signal I/O errors.

2.4 Implicit Parameters

In addition to explicit parameters, there can be parameters that are not specified in the parameter list. These implicit parameters provide additional information to your procedure from static storage locations. There are two types:

- Implicit parameters allocated by the calling program
- Implicit parameters allocated by your procedure

When deciding whether your procedure will have implicit parameters, you should consider the advantages and disadvantages discussed in the following sections. It is easier to maintain modularity by not using them. If your procedure must retain information from previous activations and you want to avoid using implicit parameters, read Section 2.5. If you must use implicit parameters, read the rest of this section and the discussion of static storage in Chapter 3.

2.4.1 Implicit Parameters Allocated by the Calling Program

There are two types of implicit parameters the calling program can allocate:

- Statically allocated variables in a named PSECT (for example, COM and MAP in BASIC, COMMON in FORTRAN or variables declared in the outer block of a procedure or program in PASCAL)
- Statically allocated global variables (for example, symbols defined with a double colon :: in MACRO, and GLOBAL variables in BLISS)

There are several disadvantages to using implicit inputs allocated by the calling program:

- Two programmers can use the same PSECT name or global variable for different quantities. This error will be undetected.
- The calling program is no longer independent of the called procedure, as a change in one could inadvertently affect the other.
- In BASIC and FORTRAN, the calling program has to declare all of COMMON regardless of the number of implicit inputs actually needed.
- If your procedures are put in a sharable image, they cannot be called from outside the shared image, since the shared image could not reference implicit parameters that are outside of the image.

Because of these disadvantages, using implicit parameters allocated by the calling program violates the modular programming standard.

2.4.2 Implicit Parameters Allocated by the Called Procedure

There is one type of implicit parameter allocated by the called procedure: local static storage. The procedure declares static storage using .BYTE through .QUAD in MACRO, OWN in BLISS, and all variables in FORTRAN. BASIC and PASCAL do not have the notion of static storage that satisfies modular programming requirements.

Implicit inputs retained in local static storage usually keep track of resources (by resource allocating procedures) and shorten the explicit parameter list. However, the use of implicit inputs by nonresource-allocating procedures can lead to unexpected results. Assume that procedure A is to leave information for a companion procedure B. Thus, B has both explicit inputs (from its caller) and implicit inputs (from A's storage). Next, consider that a calling program calls A, then calls procedure X, and finally calls B. For the calling program to get correct results from B, it must know that X (and any procedures that X calls) did not make a call to A (such a call would change the implicit inputs A leaves for B).

The following FORTRAN example of LIB_GET_STRING reads a string from the terminal; LIB_GET_STR_LEN, a companion procedure, returns the length of the string last read by LIB_GET_STRING.

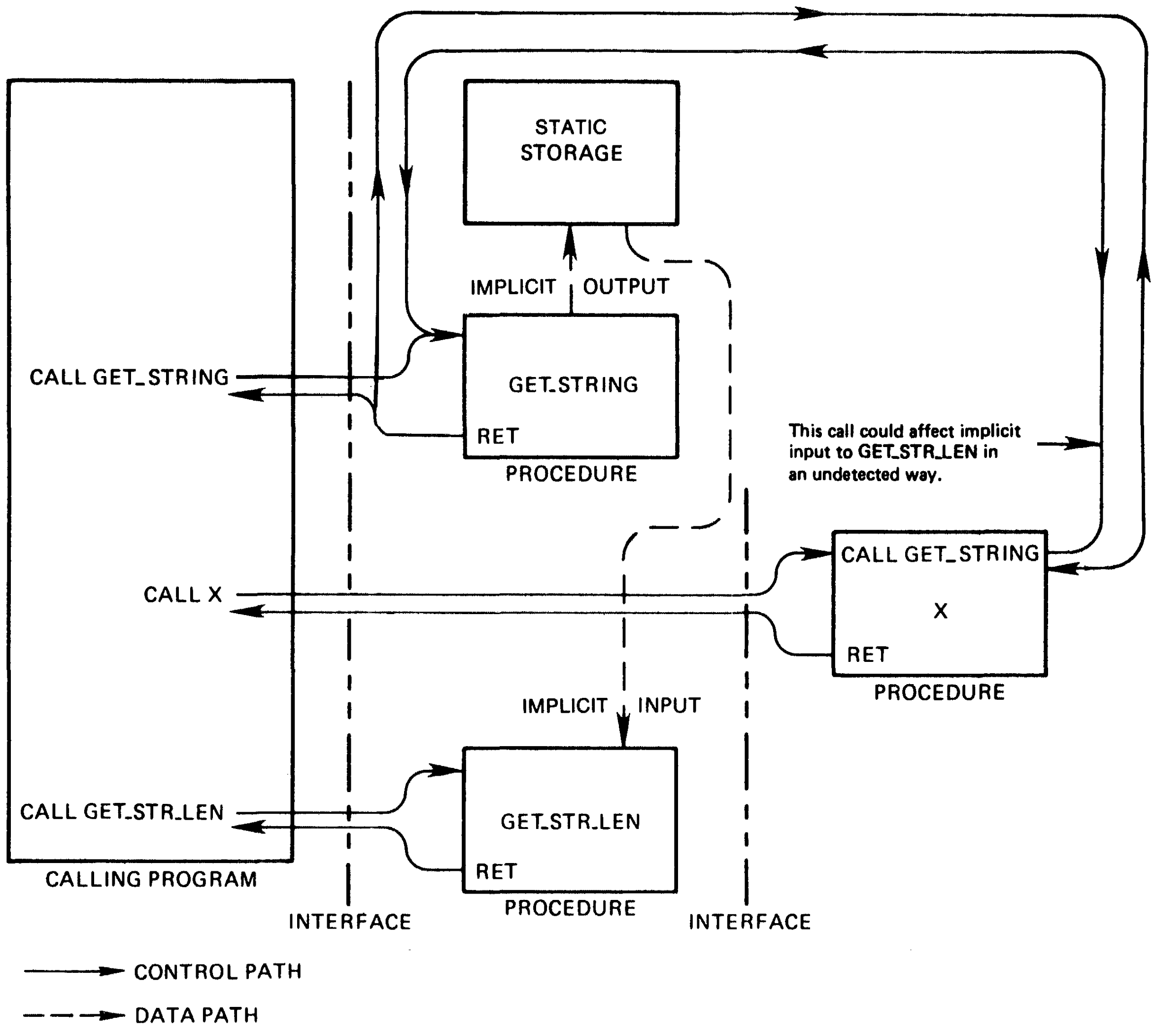
```
C       Procedure to Read String from Terminal
        FUNCTION LIB_GET_STRING (LEN)
        INTEGER*4 LENGTH         ! Place to remember length
        CHARACTER*(*) LIB_GET_STRING
        READ 100, LENGTH, LIB_GET_STRING
100     FORMAT (Q, A)            ! Set LENGTH to length of line input
        RETURN

C       Procedure to Return Length of String Last Read
        ENTRY LIB_GET_STR_LEN
        LEN = LENGTH             ! LENGTH is implicit input parameter
        RETURN
        END
```

This calling program could get unexpected results if procedure X also happens to call LIB_GET_STRING. Instead of getting the length of the string read in statement 1000, statement 2000 uses the length of the string read in procedure X.

```
        CHARACTER*60 NAME
1000    NAME = LIB_GET_STRING ()
        CALL X (...)
2000    ... = NAME (1:LIB_GET_STR_LEN())
```

Figure 2-1 illustrates this situation pictorially.

Figure 2-1: How Implicit Inputs Can Violate Modularity



The use of such implicit parameters violates the modular programming standard since one of the objectives of modular programming is to permit procedures to be combined arbitrarily without the need to understand the internal workings of each. The aforementioned problem will also occur if X is rewritten to include a call to A. A calling program that assumes the old version of X thus would not get correct results on its call to B.

Furthermore, the same problems can occur with any nonresource-allocating procedure that leaves results for itself as future implicit parameters.

The following section describes how to avoid the problems of implicit input parameters allocated by the calling program or the called procedure.

2.5 How to Avoid Implicit Parameters

There are four ways to write nonresource-allocating procedures that avoid the implicit parameter problems described in Section 2.4:

- When one procedure obtains results from another, combine the two procedures into a single call (see Section 2.5.1).
- Provide a single call that calls an action routine supplied by the calling program part way through your procedure's execution (see Section 2.5.2).
- Designate responsibility for retaining information from a procedure activation to the calling program. This is done with an explicit parameter (see Section 2.5.2).
- Specify your interface to consist of a sequence of calls to different procedures. The first call should save the contents of any still active implicit parameters on a push down stack in heap storage. The last call should restore the old implicit parameters. Thus, static storage is made available to your sequence of procedures for implicit inputs to be passed between them (see Section 3.3.1).

2.5.1 Combining Procedures

Often nonresource-allocating procedures that leave results for one another can be combined into a single procedure that returns all information explicitly in a single call. Consider the Section 2.4.2 example of the companion procedures LIB_GET_STRING and LIB_GET_STR_LEN.

Changing LIB_GET_STRING and LIB_GET_STR_LEN to a single procedure, Procedure X can no longer modify LIB_GET_STRING's storage before the length can be returned.

```
C       Procedure to Read String from Terminal
        FUNCTION LIB_GET_STRING (LEN)
        CHARACTER*(*) LIB_GET_STRING
        READ 100, LEN, LIB_GET_STRING
100     FORMAT (Q, A)          !Set LENGTH to length of line input
        RETURN
        END
```

This calling program obtains both the string and its length in a single call, thereby preventing procedure X from causing unexpected side effects.

```
     CHARACTER*60 NAME
1000 NAME = LIB_GET_STRING (NAME_LEN)
                                    !set NAME_LEN to length of NAME
     CALL X (...)
2000 ... = NAME (1:NAME_LEN)
```

2.5.2 User Action Routine

Instead of providing two procedures, another way to combine several procedures into one call is to let the calling program to gain control at a critical point in your procedure's execution. To do this, you must specify an action routine parameter in your procedure that will be called in the middle of your procedure's execution. Thus, your procedure can execute twice: before and after the action routine with no implicit inputs. BASIC and FORTRAN OPEN statements use this technique by permitting the user to supply a USEROPEN action routine.

To keep the calling program from having to provide implicit inputs for its action routine, your procedure should also provide another parameter that is passed along to the action routine. The calling sequence to your procedure is thus:

```
CALL my-proc (... ,action-routine.flc.rp ,user-arg.xy.z)
```

The calling sequence for the action routine is:

```
CALL action-routine (... ,user-arg.xy.z)
```

See Section 4.7 for an example of the code to invoke a user action routine.

NOTE

> The argument data type and argument passing mechanism provided by the calling program are immaterial to your procedure. Your procedure copies the 32-bit argument list entry, as is, to the action routine.

One problem with the action routine mechanism is that FORTRAN and PASCAL pass procedures as parameters using different mechanisms. In FORTRAN, the arg list entry contains the address of the entry mask of the action routine. In PASCAL, the arg list entry contains the address of the entry mask of the action routine and an environmental pointer.

The VAX-11 Procedure Calling Standard defines Entry Mask (ZEM) and Bound Procedure Value (BPV) data types. However, PASCAL has a language extension to permit the BPV data type to be passed by immediate value, making it identical to ZEM.

A second problem is that your procedure cannot be written in BASIC. However, the calling program can pass a reference to an external symbol that could be an action routine thus achieving the effect of the ZEM data type.

Often it is convenient for your caller's program to specify your procedure such that the action routine parameter and/or user-arg parameter is optional. However, testing for optional parameters can only be done in MACRO or BLISS.

2.5.3 Designating Responsibility to the Calling Program

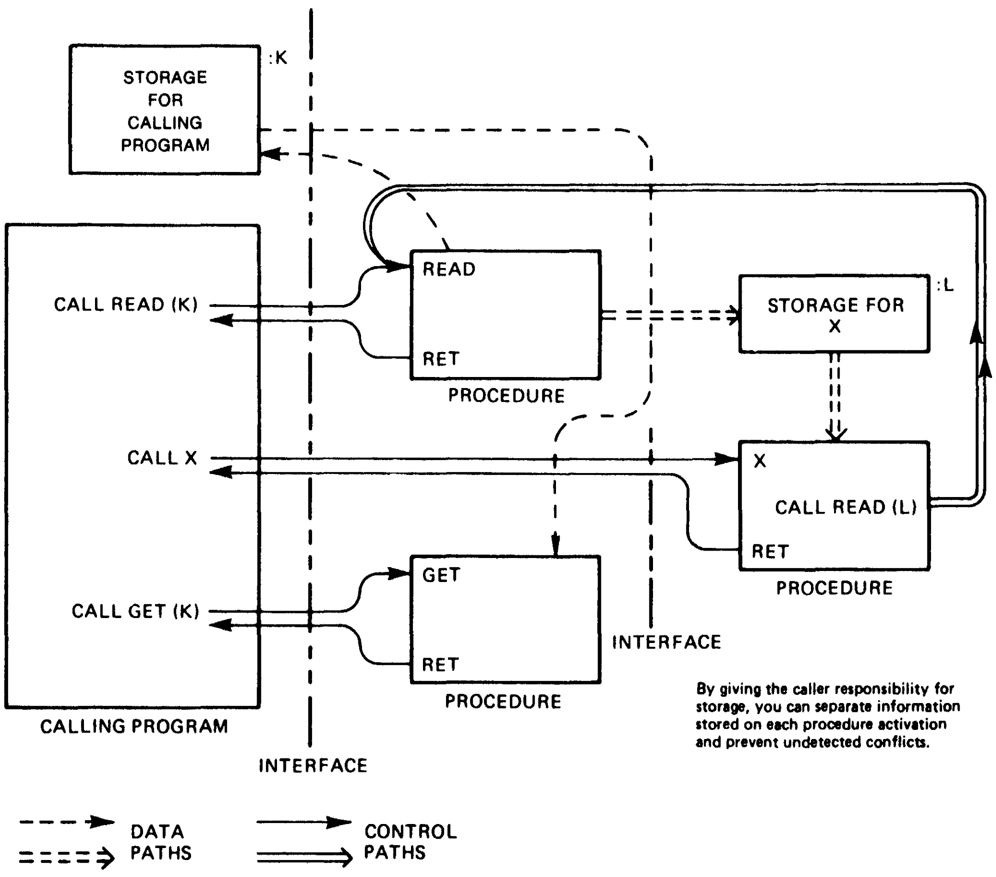
You can give the calling program responsibility for retaining information from one procedure activation to another. You do this in three ways:

- Require the calling program to allocate the necessary storage needed by your procedure. Then have it pass the storage address as an explicit parameter on all calls to your procedure (see Section 2.5.3.1).

- Require the calling program to allocate a longword and pass its address to your procedure as an explicit parameter. On the first call, your procedure will dynamically allocate storage (by calling LIB$GET_VM) and store its address in the caller's longword. On subsequent calls, your procedure will use information left in the storage area from previous calls (see Section 2.5.3.2).
- Require the calling program to pass a process-wide identifying value to all calls to your procedure. The process-wide identifier indicates which information from previous procedure activations is to be used as implicit inputs (see Section 2.5.3.3).

Figure 2-2 shows a calling program with responsibility for explicitly indicating the storage to be used by the called procedure. The following sections show the three ways to do this.

Figure 2-2: Designating Storage Responsibility to the Caller


STORAGE FOR CALLING PROGRAM
:K
CALL READ (K)
CALL X
CALL GET (K)
CALLING PROGRAM
INTERFACE
READ
RET
PROCEDURE
GET
RET
PROCEDURE
STORAGE FOR X
:L
X
CALL READ (L)
RET
PROCEDURE
INTERFACE
By giving the caller responsibility for storage, you can separate information stored on each procedure activation and prevent undetected conflicts.
DATA PATHS
CONTROL PATHS

2.5.3.1 Calling Program Allocates Procedure Storage — In this method, the calling program allocates all storage needed and passes its address as an explicit parameter on each call. For example, the library procedure MTH$RANDOM requires that the calling program allocate storage for the longword seed and to pass its address on each call. The calling sequence is:

```
value.wf.v = MTH$RANDOM (seed.ml.r)
```

MTH$RANDOM takes the seed as input and computes the next random number sequence from the current seed value. MTH$RANDOM returns a random number between 0 and 1 and updates the longword seed passed by the calling program to generate a different value on the next call (the code is shown in Section 3.3.2).

The disadvantage of this method is that you cannot increase the amount of storage needed by your procedure without requiring all calling programs to be rewritten. Thus, you should use this method only when you are confident that your procedure will not be revised in the future to use additional storage.

The next two sections describe interface techniques that permit storage size to change without affecting the interface with the calling program.

2.5.3.2 Calling Program Passes Pointer — In this method, the calling program allocates only a longword pointer for the dynamic heap storage to be allocated by your procedure; it passes the address of the longword as an explicit parameter. There are two interface techniques to indicate that storage is to be initialized:

- Provide a single entry point. A zero value in the longword instructs your procedure to allocate and initialize dynamic heap storage.
- Provide an alternate entry point that stores the address of the allocated storage in the longword. On subsequent calls, the nonzero value instructs your procedure to use that value as the storage address of information from previous calls.

Regardless of the method used to indicate storage allocation and initialization, you must also provide a way to indicate storage deallocation. You can do this with either a separate parameter or separate entry point.

For example, the procedure LIB$INIT_TIMER, which gets times and counts from the operating system, uses a parameter to determine where these values are to be stored. The calling sequence is:

```
ret-status.wlc.v = LIB$INIT_TIMER ([handle.ml.r])
```

handle

Optional address of a longword whose contents specify where the values of times and counts are stored.

If missing, they are stored in static storage, thereby making this call not AST-reentrant.

If zero, a block of dynamic heap storage is allocated by a call to LIB$GET_VM; the values are placed in that block, and the address of the block returned in "handle".

If nonzero, it is considered to be the address of a storage block previously allocated by a call to LIB$INIT_TIMER; the block is reused, and fresh times and counts are stored in it.

Entry point LIB$FREE_TIMER deallocates the block of dynamic heap storage allocated by a previous call to LIB$INIT_TIMER. The calling sequence is:

```
ret-status.wlc.v = LIB$FREE_TIMER (handle.ml.r)
```

handle
: The address of a longword whose contents specify a block of dynamic heap storage where times and counts have been stored. That storage is returned to free storage by calling LIB$FREE_VM.

The LIB$ and STR$ string procedures that handle dynamic strings also use this method. They require that the calling program allocate space for the dynamic string descriptor (two longwords) and pass its address on each call. The second longword contains a pointer to heap storage.

2.5.3.3 Calling Program Passes a Process-Wide Identifier — In this method, the calling program passes a process-wide identifying value to identify implicit results produced on previous calls which will be implicit inputs on this call. Any calling program can use the process-wide identifier. Examples of process-wide identifiers include BASIC/FORTRAN logical unit numbers and VMS System Services I/O channel numbers.

Process-wide identifiers are a resource. Modular programming techniques require that all resources allocated by a procedure be allocated by calling a resource-allocating procedure. This prevents conflicts, since a single procedure can keep track of multiple allocations to more than one procedure or procedure activation. Therefore, if you use this method, you will also have to write a resource-allocating procedure to control the resource. You should add this procedure to the default system object library STARLET.OLB so that all programmers can use it.

The library procedures LIB$GET_LUN and LIB$FREE_LUN allocate and deallocate BASIC/FORTRAN logical unit numbers outside the range normally specified in user programs, that is, outside the range 0 to 99. An example of a resource-allocating procedure that allocates identifying numbers is given in Section 4.4.2.

2.6 Control of Human Readable Output

A modular procedure allows its caller to control human readable output sent to the terminal, queued to a line printer, or written to a file. You do this by providing an optional parameter that the calling program can use to specify an action routine.

If the calling program specifies an action routine, your procedure calls the action routine with each record (line) of output information (instead of outputting it directly to a file or device). The action routine is repeatedly called with the address of a string descriptor for each record. Each record should not exceed 80 characters so it can be printed on most terminals. It begins with a space (FORTRAN convention) and contains no ASCII carriage return (CR) or line feed (LF) characters. Thus, the line can be written into a file having CR, FTN, or PRN record attributes.

The user-supplied action routine can output each record to any output device and return a failure or success status to your procedure. If an error status is returned, your procedure stops calling the action routine and returns the same error status to the original calling program.

To help your caller write a single, multi-purpose action routine, your procedure should also provide an additional optional parameter which, if present, is passed to the action routine as a second argument. Then the calling program can pass information to the action routine that applies to each call.

For example, you could create a procedure LIB_SNAP_SHOT that outputs a memory dump to the output device LPA0 unless the calling program supplied an action routine. The calling sequence is:

ret-status = LIB_SNAP_SHOT (low-adr, high-adr [,user-act-rout [,user-arg]])

LIB_SNAP_SHOT can be called from FORTRAN as follows to output the information on file DUMPOUT.DAT instead of device LPA0:

```
     EXTERNAL PROC
     OPEN (UNIT=10, FILE = 'DUMPOUT')
     .
     .
     .
     IF (.NOT. LIB_SNAP_SHOT (A, B(100), PROC)) GO TO 9999
     .
     .
     .
     END
     FUNCTION PROC (RECORD)
     CHARACTER*(*) RECORD
     INTEGER*4 PROC
     PROC = 0                     ! Assume Error
     WRITE (10, *, ERR=100) RECORD
     PROC = 1                     ! Success
100  RETURN
     END
```

or as follows to output the dump on the controlling output device instead of LPA0:

```
IF (.NOT.LIB_SNAP_SHOT (A, B(100), LIB$PUT_OUTPUT))
```

See Section 4.7 for an example of the code to invoke a user action routine. See also Section 2.5.2 for a discussion of the considerations of using action routines in BASIC, FORTRAN, and PASCAL.

2.7 Timer and Resource Allocation Procedures

All timer and resource allocation procedures should make statistics available for performance evaluation and debugging. These procedures are coded with two additional entry points:

LIB$SHOW_name or LIB_SHOW_name
LIB$STAT_name or LIB_STAT_name

2.7.1 SHOW Entry Point

A SHOW entry point provides formatted strings containing the information you want. It should follow the conventions for providing human readable output (see Section 2.6). The calling sequence is:

ret-status.wlc.v = LIB$SHOW_name ([code.rl.r [,action-routine.flc.rp [,user-arg.xx.x]]])

where:

code
: is an optional code (of the form LIB$K_code) designating the statistic you want. A separate code is defined for each statistic available; the codes are the same for the SHOW and STAT entry points. Codes start at one for each procedure. The specification of each procedure should list the codes used. If omitted or zero, all statistics are provided.

action-routine
: is an optional address of an action routine. If omitted, statistics are output to SYS$OUTPUT.

user-arg
: is an optional user parameter to be passed to the action routine. If omitted, a shortened list is passed to the action routine. The user-arg, if present, is copied to the parameter list passed to the action routine. That is, the 32-bit arg list entry passed by the calling program is copied to the arglst entry passed to the action routine. Thus, the access type, data type, parameter form, and passing mechanism can be arbitrary, as agreed between the calling program and the action routine.

The optional action routine should have the form:

status.wlc.v = ACTION-ROUTINE (string.rt.dx [,user-arg.xx.x])

See Section 4.7 for an example of the code to invoke a user action routine.

2.7.2 STAT Entry Point

A STAT entry point returns the information you want as binary results. The calling sequence is as follows:

ret-status.wlc.v = LIB$STAT_name (code.rl.r, value.wl.r)

where:

code
: is a code designating the statistic you want. A separate code is defined for each statistic available; the codes are the same for the SHOW and STAT entry points. Codes start at 1.

value
: is the value of the returned statistic.

2.8 Documentation of Procedures and Modules

You must document your procedures so you and others can be sure of their objective. Typically, each module contains only one procedure.

2.8.1 How to Write a Module Description

You should place a description containing the following information at the front of each module:

Title:

Gives the module name followed by a one-line functional description.

Version:

Gives the version and a three-digit edit number. Generally 1–001 is the original version.

Facility:

Gives a description of the library facility, such as general utility library (LIB).

Abstract:

Gives a short, three to six-line functional description of the module.

Environment:

Lists any special environmental assumptions that the module can make. These include assumptions made at both compilation and execution time that could affect either the hardware or software environments.

For execution time, describe situations that the module assumes or optional elements of the VAX–11 modular programming standard that your module does not follow. Usually, you should write:

Runs at any access mode – AST reentrant.

Author:

Include your name and the creation date of the module.

Modified by:

Include the modification number, name of modifying programmer, modification date, and a list of the modifications.

This concludes the preface. End with a page delimiter.

Figure 2-3 shows a sample MACRO module description.

Figure 2-3: MACRO Module Description Template

```
.TITLE  LIB$TEMPLATE - Sample module
.IDENT  /1-001/          ; File: LIBTEMPLA.MAR
                         ; Edit: AAA1001

;++
; FACILITY: General Utility Library
;
; ABSTRACT:
;
;       This is a sample module. It is used as a template for
;       coding MACRO modules for the VAX-11 Run-Time Library.
;
; ENVIRONMENT: Runs at any access mode, AST Reentrant
;
; AUTHOR: Ada A. Augusta, CREATION DATE: 01-JAN-1980
;
; MODIFIED BY:
;
; 1-001  - Original. AAA 01-JAN-1980
;--

        .SBTTL DECLARATIONS
;
; LIBRARY MACRO CALLS:
;
        $SSDEF                     ; SS$_ symbols
;
; EXTERNAL DECLARATIONS:
;
        .DSABL  GBL      ; Force all external symbols to be declared
        .EXTRN  LIB$_INVARG        ; Invalid argument
        .EXTRN  LIB$SIG_TO_RET     ; Convert signals to return status
;
; MACROS:
;
;       NONE
;
; EQUATED SYMBOLS:
;
;       NONE
;
; OWN STORAGE:
;
;       .PSECT _LIB$DATA PIC, USR, CON, REL, LCL, NOSHR, -
;                       NOEXE, RD, WRT, LONG
;
;       NONE
;
; PSECT DECLARATIONS:
;
        .PSECT _LIB$CODE PIC, USR, CON, REL, LCL, SHR, -
                        EXE, RD, NOWRT, LONG
```

Figure 2-4 shows a sample BLISS module description.

Figure 2-4: BLISS Module Description Template

```
%TITLE 'LIB$TEMPLATE - Sample module'
MODULE LIB$TEMPLATE (                   ! Sample module
         IDENT = '1-001'                ! File: LIBTEMPLA.B32
         ) =                            ! Edit: AAA1001
BEGIN
!
!++
! FACILITY:      General Utility Library
!
! ABSTRACT:
!
!        This is a sample module. It is used as a template for
!        coding BLISS modules for the VAX-11 Run-Time Library.
!
! ENVIRONMENT:    Runs at any access mode - AST reentrant
!
! AUTHOR: Ada A. Augusta, CREATION DATE: 01-Jan-1980
!
! MODIFIED BY:
!
! 1-001   - Original. AAA 01-Jan-1980
!--

%SBTTL 'Declarations'

!
! SWITCHES:
!

SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL,
         NONEXTERNAL = WORD_RELATIVE);

!
! LINKAGES:
!
!        NONE
!
! TABLE OF CONTENTS:
!

FORWARD ROUTINE
    LIB$TEMPLATE;                       ! Sample routine

!
! INCLUDE FILES:
!

LIBRARY 'RTLSTARLE';                  ! System symbols, typically from
                                      ! SYS$LIBRARY:STARLET.L32
REQUIRE 'RTLIN:RTLPSECT';             ! Define PSECT declarations macros
```

(continued on next page)

Figure 2-4: BLISS Module Description Template (Cont.)

```
!
! MACROS:
!
!       NONE
!
! EQUATED SYMBOLS:
!
!       NONE
!
! FIELDS:
!
!       NONE
!
! PSECTS:
!
DECLARE_PSECTS (LIB);          ! Declare PSECTs for LIB$ facility
!
! OWN STORAGE:
!
!       NONE
!
! EXTERNAL REFERENCES:
!

EXTERNAL ROUTINE
LIB$SIG_TO_RET;                ! Convert signals to return status

EXTERNAL LITERAL               ! Condition value symbols
    LIB$_INVARG;               ! Invalid argument
```

2.8.2 How to Write a Procedure Description

You should place a procedure description at the beginning of each procedure in a module.

Always list each of the following topics regardless of their actual presence. For example, if a procedure has no implicit inputs, write:

```
!
!  Implicit Inputs:
!
!       NONE
!
```

Functional Description:

The functional description describes a module's purpose and completely documents its interfaces.

The description should include the basis for any critical algorithms used, including literature references, where applicable. It should also explain why a particular algorithm was chosen.

Calling Sequence:

A calling sequence to a procedure is described by: (1) a return status, value parameter, or CALL instruction, followed by (2) the procedure name, followed by (3) the parameters used by the procedure.

Parameters should be listed in the order they will be written in a higher-level language. Each parameter characteristic should also be included, using the procedure parameter notation described in Section 2.3.1.

Examples:

ret-status.wlc.v = LIB$GET_INPUT (get-str.wt.dx [,prompt-str.rt.dx [,outlen.ww.r]])

string-len.wlu.v = LIB$LEN (string.rt.dx)

CALL LIB$CRC_TABLE (poly.rlu.r, table.wl.ra)

The calling sequence description includes the instruction for calling the routine and the parameter list, which is typically a list of registers or parameters. In MACRO, each parameter is symbolically defined as the offset relative to the argument pointer, AP.

Formal Parameters:

List any explicit input, input-output, or output parameters including a qualifying description. You should list the parameters in calling sequence order.

Implicit Inputs:

List any inputs from storage, internal or external to the module, that are not specified in the parameter list. Note: usually NONE. See Section 2.4.

Implicit Outputs:

List any outputs to internal or external storage that are not specified in the parameter list.

Completion Status: (or Routine Value:)

List the success or failure condition value symbols that could be returned as completion codes in R0. If your procedure returns a function value other than a condition value in R0, change the heading to Routine Value.

Side Effects:

Describe any functional side effects not evident from a procedure's calling sequence. This includes changes in storage allocation, process status, file operations, and conditions that are signaled. In general, document anything out of the ordinary that the procedure does to the environment. If a side effect modifies local or global storage locations, document it in the implicit output description instead.

Figure 2-5 shows a sample MACRO procedure description.

Figure 2-5: MACRO Procedure Description Template

```
        .SBTTL  LIB$TEMPLATE - Sample routine
;++
; FUNCTIONAL DESCRIPTION:
;
;       This routine is an example for coding procedures
;       in MACRO. It has no computational function.
;
; CALLING SEQUENCE:
;
;       ret_status.wlc.v = LIB$TEMPLATE (parameter.rl.r)
;
; FORMAL PARAMETERS:
;
;       PARAMETER       Longword input parameter
;
; IMPLICIT INPUTS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION STATUS: (or ROUTINE VALUE:)
;
;       SS$_NORMAL      Normal successful completion
;       LIB$_INVARG     Invalid argument
;
; SIDE EFFECTS:
;
;       NONE
;
;--

        .ENTRY LIB$TEMPLATE, ^M<IV>     ; Entry point
        MOVAB   G^LIB$SIG_TO_RET, (FP)  ; Return signals
        TSTB    (AP)                    ; Argument present?
        BGTRU   1$                      ; Maybe.
        MOVL    #LIB$_INVARG, R0        ; No, invalid argument.
        RET                             ; Return
;+
; Come here if the argument count is at least 1.
;-
1$:     MOVL    #SS$_NORMAL, R0         ; Indicate success
        RET                             ; End of routine

        .END                            ; End of module
```

Figure 2-6 shows a sample BLISS procedure description.

Figure 2-6: BLISS Procedure Description Template

```
%SBTTL 'LIB$TEMPLATE - Sample routine'
ROUTINE LIB$TEMPLATE (               ! Sample routine
        PARAMETER                    ! Sample parameter
) =

!++
! FUNCTIONAL DESCRIPTION:
!
!       This routine is an example for coding procedures
!       in BLISS. It has no computational function.
!
! CALLING SEQUENCE:
!
!       ret_status.wlc.v = LIB$TEMPLATE (parameter.rl.r)
!
! FORMAL PARAMETERS:
!
!       PARAMETER       Longword input parameter
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION STATUS: (or ROUTINE VALUE:)
!
!       SS$_NORMAL      Normal successful completion
!       LIB$_INVARG     Invalid argument
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN

        BUILTIN
             ACTUALCOUNT;               ! Sample builtin

        MAP
             PARAMETER : BLOCK [4, BYTE];  ! Sample MAP

        LOCAL
        LOCALVARIABLE;               ! Sample local declaration

        ENABLE
        LIB$SIG_TO_RET;              ! Sample enable declaration

        IF (ACTUALCOUNT () LSSU 1) THEN RETURN (LIB$INVARG);

        RETURN (SS$NORMAL);
        END;                         ! End of routine LIB$TEMPLATE

        END                          ! End of module LIB$TEMPLATE

ELUDOM
```

Chapter 3
Using Storage

3.1 Types of Storage

There are three types of storage: static, stack, and heap. The three forms of storage differ in the method and duration of allocation.

3.1.1 Static Storage

Statically allocated storage is allocated by the linker in one place for the duration of the program's execution. The storage's initial contents are specified in the source program. On calls to a procedure after initialization, the static storage will have the same allocation and the contents left from the previous call.

The following forms of static storage are available in the indicated languages:

- **MACRO**

  These statements (1) allocate or (2) allocate and initialize the static storage amount indicated:

Allocate	**Amount**	**Allocate and initialize (to 10)**
.BLKB	1 Byte	.BYTE 10
.BLKW	1 Word	.WORD 10
.BLKL	1 Longword	.LONG 10
.BLKQ	1 Quadword	.QUAD 10
.BLKO	1 Octaword	.OCTA 10

- **BLISS**

  OWN Storage
  GLOBAL Storage

  In the following BLISS example, A is initialized to 0 and B to 10.

```
OWN
A: LONG,
B: LONG INITIAL(10);
```

- **BASIC**

  All COMMON and MAP data storage is statically allocated.

- **FORTRAN**

  All FORTRAN data storage is statically allocated. It is declared as local variables or arrays or as COMMON. Static storage can be initialized using the DATA statement.

  In the following FORTRAN procedure, variables A, B, C, FUNC, array D, and string E are all statically allocated. Furthermore, variable A is initialized to 10 at compile time, while other variables are initialized to 0. X, Y, and Z are not statically allocated:

```
FUNCTION FUNC(X,Y,Z)
INTEGER*4 A,B,D(100)
DATA A/10/
CHARACTER*10 E
CHARACTER*(*) X
.
.
.
FUNC = C
.
.
.
RETURN
END
```

  Note that variable A is not reinitialized to 10 on subsequent calls to FUNC. Instead, the value of A and all other statically allocated variables retain the values left from the previous call. The SAVE statement must be used to ensure value retention across calls.

- **PASCAL**

  All program or module level storage is statically allocated.

3.1.2 Stack Storage

Dynamically allocated stack storage is allocated by a procedure as needed on the process stack at run time. It is automatically deallocated when the procedure returns control to its caller.

- **MACRO**

  Stack storage is allocated by decrementing the stack pointer (SP) by the number of bytes (n) of storage required:

```
SUBL2    #<n+3>/4,SP
```

- **BLISS**

  Stack storage can be allocated as follows:

```
STACK LOCAL A: LONG;
```

- **BASIC**

  Local variables declared in a BASIC program (except virtual arrays and variables in COM or MAP) are allocated on the stack.

- **FORTRAN**

  Stack storage cannot be allocated by FORTRAN users.

- **PASCAL**

  Stack storage is allocated automatically for all PROCEDURE and FUNCTION local variables.

3.1.3 Heap Storage

Dynamically allocated heap storage is allocated at run time to a procedure activation as needed from a process-wide pool. Heap storage is allocated by calling LIB$GET_VM or the System Service $EXPREG. Heap storage is deallocated — that is, returned to the process-wide pool — by calling LIB$FREE_VM. The system service $CNTREG cannot be used to deallocate heap storage. (See Section 4.6.)

- **MACRO**

  Heap storage can be allocated with a call to LIB$GET_VM. (See Section 5.1 in the *VAX-11 Run-Time Library Reference Manual.*)

- **BLISS**

  Heap storage can be allocated as follows:

  ```
  EXTERNAL   PROCEDURE   LIB$GET_VM:      ADDRESSING_MODE
  (GENERAL);
  IF LIB$GET_VM (UPLIT (100), ADR)
  THEN
          success, ADR set to address allocated
  ```

- **BASIC**

  Heap storage can be allocated, but it must then be passed BY REF to another procedure as an array parameter. BASIC dynamic strings automatically use heap storage.

- **FORTRAN**

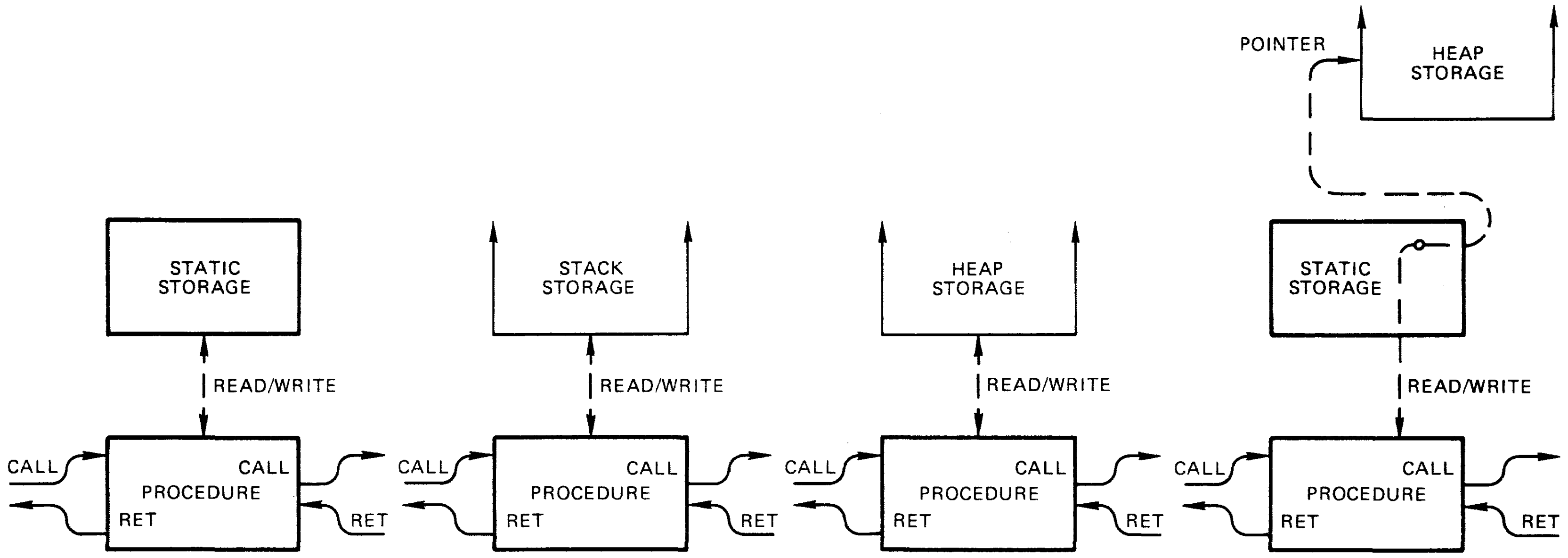
  Heap storage can be allocated, but it must then be passed to another procedure as an array parameter. (See Section 5.1 in the *VAX-11 Run-Time Library Reference Manual.*)

- **PASCAL**

  Heap storage can be allocated by calling NEW and deallocated by calling DISPOSE.

Figure 3-1 shows how the different types of storage are used.

Figure 3-1: Use of Storage Types



Static storage is used when a result must be retained for a future procedure activation.

Stack storage is used when results are needed only for the current procedure activation.

It is deallocated when the procedure returns to its caller.

Heap storage is used when the amount of storage varies from call to call.
Storage is deallocated before control returns to the caller (by calling LIB$FREE_VM.)

Heap storage is also used when the amount of storage needed varies and when results must be retained for a future procedure activation.

It is deallocated by calling LIB$FREE_VM.

3.1.4 Storage Use Summary

Table 3–1 summarizes storage available to the programmer in various language procedures.

Table 3–1: Summary of Storage Use by Language

Language	Storage Type		
	Static	**Stack**	**Heap**
MACRO	Block Storage	Decrementing Stack Pointer	By Calling LIB$GET_VM
BLISS	OWN and GLOBAL	STACK LOCAL	By Calling LIB$GET_VM
BASIC	All COMMON and MAP Data Storage	Local Variables	Dynamic Strings
FORTRAN	All Data Storage	Not Applicable	By calling LIB$GET_VM
PASCAL	All Program or Module Level	PROCEDURE and FUNCTION Local	By Calling NEW

3.2 Choosing a Storage Type

A procedure activation combines procedure-implementing instructions and the stack frame storage allocated when the procedure is called and deallocated when the procedure returns. Two procedure activations can exist at the same time if:

- The procedure is called by an AST-level routine while it is executing (AST-reentrant)
- The procedure is called by a condition handler while it is executing (re-entrant)
- The procedure is called by itself or by another procedure that it has called (recursive)

If a procedure's results must be saved for a subsequent activation, the procedure must use static storage or provide a mechanism for the caller to retain storage to access those results.

If no procedure-activation's results need be retained for subsequent activations, the procedure can use static, stack or heap storage.

Stack storage is always recommended. It is easily allocated, and it performs well in a paging system such as VMS.

Heap storage allocation is slower than stack storage. Heap storage requires explicit deallocation. It is recommended for use instead of stack storage when a variable or large amounts of information must be retained after your procedure returns to its caller.

Avoid static storage wherever possible. It can cause unwanted side effects if used for implicit parameters (see Section 2.4.2), and when used, it is difficult to make your procedure re-entrant or AST-reentrant (see Chapter 6).

3.3 Using Static Storage

There are three classes of procedures that use static storage:

- Process-wide resource-allocating procedures (see Section 4.4)
- Nonresource-allocating procedures that retain information from previous activations in order to shorten the explicit parameter list (see Section 3.3.1 through 3.3.3)
- Procedures that do not make use of retained information from previous activations (see Section 3.3.4)

When you cannot avoid using static storage, you can maintain modularity by using one of the following four techniques.

3.3.1 Allocating Process-Wide Identifiers

Your procedure allocates heap storage and returns the address of the allocated storage as a process-wide identifier to the calling program.

Each set of related calls uses the same identifier, while each set of unrelated calls uses different identifiers.

You must make sure that the modular procedure rather than its caller allocates and deallocates the identifier values. To avoid using static storage, this identifier can be the actual address of heap storage.

Format

ret-status.wlc.v = LIB$INIT_BLOCK (handle.wl.r)

Example (BLISS)

```
ROUTINE LIB$INIT_BLOCK(HANDLE)=
     BEGIN
     EXTERNAL ROUTINE LIB$GET_VM;
     LITERAL BLOCKSIZE = 16;     ! Block size in bytes
     RETURN (LIB$GET_VM (%REF (BLOCKSIZE),.HANDLE));
     END;
```

Example (FORTRAN)

```
FUNCTION LIB$INIT_BLOCK (HANDLE)
INTEGER*4 LIB$INIT_BLOCK, HANDLE, LIB$GET_VM
LIB$INIT_BLOCK = LIB$GET_VM (16, HANDLE)
RETURN
END
```

3.3.2 Caller Passes the Address of Storage

Allow the caller of the procedure to allocate and pass the address of the (static or dynamic) storage area to be used. In the following example, the Math library random number generator (MTH$RANDOM) uses this method to produce the seed.

Format

ran_num.wf.v = MTH$RANDOM (seed.mlu.r)

Example (MACRO)

```
     SEED = 4                       ; formal arg list offset
     .ENTRY  MTH$RANDOM, 0          ; no registers saved, clear IV

;+
; If this were to be placed as an inline expansion,
; EMUL SEED, #69069,#1,R0 should replace the next two
; instructions: this would prevent possible
; integer overflow trapping.
;-

     MULL2    #69069, @SEED(AP)     ; Update seed with multiplier
     INCL     @SEED(AP)             ; Increment seed to protect
                                    ; against strange seeds

;+
; The next instructions convert the seed from unsigned integer
; to floating point in the range 0.0 to 1.0 exclusive.
;-

     EXTZV    #8, #24, @SEED(AP), R0
; Get the most significant bit
                                    ; of the seed in the range
                                    ; 0 to (2**24)-1.
     CVTLF    R0,R0                 ; Convert to floating without
                                    ; rounding. The result is
                                    ; positive and in the range
                                    ; 0.0 to (2.0**24)-1.0

;+
; If this were to be placed as an inline expansion, then
; MULF #^X00003480,R0 could replace the next two instructions.
;-

     BEQL     10$                   ; If zero, already correct.
     SUBW     #24@7, R0             ; DIVF #^F2.0**24
                                    ; The result is now in the
                                    ; range 0.0 to 1.0 exclusive.

10$: RET

     .END
```

The following FORTRAN calling program allocates the needed static storage and passes it to MTH$RANDOM:

```
INTEGER*4   RAN_SEED
REAL*4      RAN_VAL, MTH$RANDOM
RAN_VAL =   MTH$RANDOM (RAN_SEED)
```

RAN_VAL is set to a random floating point value.

3.3.3 Pushing Down the Contents of Static Storage

Specify the interface with the calling program to consist of a sequence of calls. The first call should include saving the contents of any still active implicit parameters on a push down stack in heap storage, not on the process stack; the last call should include restoring the old implicit parameters. Thus, static storage is made available to your sequence of procedures for implicit inputs to be passed between them.

To use this technique, write an initialization procedure in the module that, among other things, automatically pushes the information stored in static storage onto a simulated software stack maintained in heap storage; it will remain there during current and future procedure activations. Then, write a termination procedure that automatically pops information back into static storage.

When using this method, the calling program is to call the initialization and termination procedures at the beginning and ending of the sequence of calls to your module. Additionally, the calling program must establish a condition handler that will call the termination procedure (so the data will be popped back into static storage) in case a stack unwind occurs.

For example, BASIC and FORTRAN language-support procedures push down the contents of static storage for the current I/O statement whenever an I/O statement is initiated. Thus, I/O statements consist of a sequence of calls of the form:

1. I/O statement initialization procedure

   This procedure sets up the I/O system by initializing its static storage for the specific I/O requested, and flags the logical unit to be active. If the specified unit has not already been explicitly opened, a default open can be performed, with buffers and control blocks dynamically allocated. If an I/O statement is already being processed, the static storage used by that I/O statement is "pushed down."

2. Data element transmission procedure(s)

   Each data element transmission procedure copies one data element from/to the user program to/from the I/O buffer for the logical unit. The logical unit is an implicit input.

3. I/O statement termination procedure

   This procedure completes the current I/O statement. The logical unit number is an implicit input. If an I/O statement had been pushed down, it is now popped back into static storage, this restores it as the current I/O statement.

For example, the BASIC statement:

```
PRINT #2, I%,IFUNC(J%),B
```

is compiled as:

```
MOVL #2, -(SP)                  ; Unit Number
CALLS #1, BAS$PRINT             ; Initialize PRINT
                                ; sequential unformatted
PUSHL I%(R11)                   ; Value of I%
CALLS #1, BAS$OUT_I_V_C         ; Transmit integer
PUSHAL J%(R11)                  ; Address of J%
CALLS #1, IFUNC                 ; Call function IFUNC
PUSHL R0                        ; Push function value
CALLS #1, BAS$OUT_I_V_C         ; Transmit by immediate value
MOVF B(R11), -(SP)              ; Value of B
CALLS #1, BAS$OUT_F_V_B         ; Transmit floating
CALLS #0, BAS$IO_END            ; End of the I/O list
```

If function IFUNC performed I/O, the PRINT statement would be pushed down and popped back before control returns from IFUNC.

3.3.4 In Procedures not Needing to Retain Results

You can maintain modularity in procedures that use static storage and do not need to retain values after control is returned to its caller. To do this write each variable before reading it. In FORTRAN, this is done by assigning an expression to each variable before using that variable in another expression. For example, the following FORTRAN code is modular even though static storage is used exclusively:

```
FUNCTION (A)
INTEGER D
D=A
G=D+A
```

In this example, the static variables D and G are initialized to expressions consisting solely of variables passed as explicit input parameters.

3.4 Using Stack Storage

You can use stack storage to maintain modularity and avoid the special considerations necessary for static storage. If your procedures are written in

MACRO, BLISS, BASIC or PASCAL, you should use stack storage exclusively when your procedure does not need to retain values from its previous activations. Note that stack storage is not available in FORTRAN.

Specific advantages of using stack storage are:

- Data is automatically hidden from source code outside the procedure.
- Program performance is improved since the same pages of memory are used by many different procedures.
- Procedures are automatically AST-reentrant.
- Unintended interaction between successive activations of the same procedure is avoided.
- Stack storage is automatically deallocated on procedure return.

3.4.1 Using Stack Storage in MACRO

In MACRO, allocate stack storage by subtracting the number of bytes required from the stack pointer (SP) provided on entry. For efficient operation, you should allocate stack space in multiples of four bytes to keep the stack aligned on a longword boundary. The CALLS and CALLG instructions automatically align the stack at procedure entry time.

The following MACRO procedure concatenates two source strings and returns the result as a single fixed-length string. No restrictions are placed on the overlapping of source and destination strings; therefore, a temporary stack storage technique is used.

In the example, these steps occur:

1. Add the source lengths to the stack pointer SP.
2. Copy first string to stack.
3. Copy second string to stack.
4. Copy stack to result.

The calling sequence is:

CALL LIB_CONC (result.wt.ds, src1.rt.dx, src2.rt.dx)

```
RESULT   = 4              ; Arg list offset for result
SRC1     = 8              ; Arg list offset for source1
SRC2     = 12             ; Arg list offset for source2

.ENTRY   LIB_CONC, ^M<R2,R3,R4,R5,R6>
MOVZWL   @SRC1(AP), R6    ; R6 = length of source1 in bytes
MOVZWL   @SRC2(AP), R0    ; R0 = length of source2 in bytes
ADDL     R0, R6           ; R6 = total length
SUBL     R6, SP           ; Allocate space for SRC1 and SRC2
MOVQ     @SRC1(AP), R0    ; R0 <15:0> = len, R1 = adr of SRC1
MOVC3    R0, (R1), (SP)   ; Move SRC1 to stack
MOVQ     @SRC2(AP), R0    ; R0 <15:0> = len, R1 = adr of SRC2
MOVC3    R0, (R1), (R3)   ; Move SRC2 to stack
MOVQ     @RESULT(AP), R0  ; R0 = len of result, R1 = adr of result
MOVC5    R6, (SP), ^A' ', R0, (R1); copy temporary back to result
RET                       ; Return, deallocating stack storage
```

3.4.2 Using Stack Storage in BLISS

When using stack storage in BLISS, define each variable in the innermost nested block. This keeps the amount of code that affects the variable to a minimum, making it easier to understand and maintain the procedure.

The following BLISS example computes the area of a rectangle, using stack storage to hold the result:

```
ROUTINE COMPUTE_AREA (HEIGHT, WIDTH)=
BEGIN
     STACK LOCAL AREA;
     AREA = .HEIGHT * .WIDTH;
     RETURN .AREA;
     END;
```

3.4.3 Using Stack Storage in BASIC

All variables, strings, and arrays are allocated in the stack in BASIC, except for COMMON (COM or MAP). Thus, it is very easy to use stack storage in BASIC.

The following BASIC example computes the area of a rectangle, using stack storage to hold the result:

```
100 FUNCTION INTEGER COMPUTE_AREA (HEIGHT%, WIDTH%)
200     AREA% = HEIGHT% * WIDTH%
300     COMPUTE_AREA = AREA%
400 FUNCTIONEND
```

3.4.4 Using Stack Storage in PASCAL

All local variables and arrays are allocated in the stack in PASCAL. Thus, it is very easy to use stack storage in PASCAL.

The following PASCAL example computes the area of a rectangle, using stack storage to hold the result:

```
FUNCTION COMPUTE_AREA (HEIGHT : INTEGER;
                        WIDTH : INTEGER) : INTEGER;
VAR AREA : INTEGER;

     BEGIN
     AREA := HEIGHT * WIDTH;
     COMPUTE_AREA := AREA;
     END;
```

3.5 Using Heap Storage

You can use heap storage to dynamically allocate arbitrary amounts of storage. Heap storage is useful for retaining variable amounts of information from one procedure activation to another.

If your procedure does not explicitly deallocate the heap storage (by calling LIB$FREE_VM) before returning to its caller, your procedure must either:

- Retain the address of the heap storage in static storage so that it can be deallocated later, or
- Return the address (and also the responsibility for deallocation) to the caller

This lets you use or deallocate the storage on a later activation. (See Section 2.5.)

3.5.1 Allocate Heap Storage in BLISS

This example allocates a buffer from heap storage:

```
!+
! STRING_PTR is OWN storage which holds a pointer to
! a dynamically allocated buffer of 80 bytes.
!-

OWN STRING_PTR;
        LIB$GET_VM (%REF(80), STRING_PTR);
.
.
.
```

The following BLISS example illustrates the use of heap storage to pass information between calls without using static storage. Instead, the responsibility for deallocation of heap storage belongs to the calling program.

Figure 3-2: Allocating Heap Storage in BLISS

```
ROUTINE RANSUB (SEED, DATA, NUM_VALS) =

!++
! FUNCTIONAL DESCRIPTION:
!
!       Compute a random number by using a congruential generator
!       but reordering its outputs randomly to avoid correlation
!       between successive results.
!
! FORMAL PARAMETERS:
!
!       SEED.ml.r       The address of a longword containing the
!                       seed. If the seed is 0, then the data block
!                       pointed to by DATA is assumed to be dynamic
!                       and is deallocated (by calling LIB$FREE_VM).
!
!       DATA.ml.r       The address of a longword that contains a
!                       pointer to the address of the data block
!                       needed for reordering the outputs. If the
!                       pointer is zero, the block is allocated.
!
!       NUM_VALS.rl.r   The number of values over which to
!                       reorder the outputs of the basic generator.
!
! IMPLICIT INPUTS:
!
!       None
!
! IMPLICIT OUTPUTS:
!
!       None
!
! ROUTINE VALUE:
!
!       A random number from 0.0 up to but not including 1.0.
!       In the "final" call, the value 1.0 is returned.
!
! COMPLETION CODES:
!
!       None
!
! SIDE EFFECTS:
!
!       Can allocate or deallocate virtual memory.
!
!--

BEGIN

LOCAL

            RAN1,           !Interim random number

            RETURN_VALUE;   !Random number returned

BUILTIN

            CVTLF;          !Convert integer (long) to floating

IF (...DATA EQL 0)

THEN

!+
!You must set up the data block that remembers old values for
```

(continued on next page)

Figure 3-2: Allocating Heap Storage in BLISS (Cont.)

```
!scattering purposes. The data block is formatted as follows:
!
! 0       Length, for LIB$_FREE_VM
! 4       Current seed for the main random number generator
! 8       Current seed for the auxiliary generator, which
!          scatters the outputs of the main generator
! 12-end Numbers produced recently by the main generator,
!          for scattering purposes.
!-
         BEGIN
!+
! If you cannot get enough virtual memory,
! use the following algorithm.
!-

         IF (NOT (LIB$GET_VM (%REF((..NUM_VALS + 3)*4), .DATA)))
         THEN RETURN (MTH$RANDOM (.SEED));

!+
! If you get the memory, you must initialize it.
!-

         IF (..SEED EQL 0)
         THEN .SEED = 1; !Don't be confused by funny seed

         ..DATA = ..NUM_VALS;    !Amount to free
         ..DATA + 4 = (..SEED);    !Seed for main generator
         ..DATA + 8 = (..SEED)*(..SEED);    !Seed for scattering function
!+
! Store values from the main generator in the remainder of
! the data block.
!-

         INCR COUNTER FROM 3 TO ..NUM_VALS + 3 DO
         (..DATA) + (.COUNTER*4) = MTH$RANDOM (..DATA + 4);

         END;               !of initialization

         IF (..SEED EQL 0)
         THEN
!+
! This is the "final" call to the random number generator.
! Return the data block to free storage and return to the caller with
! value 1.0, which is invalid under all other circumstances.
!-
         BEGIN
!+
! Give the user back the latest seed so he can run this procedure
! again without getting the same sequence of random numbers.
!-
         .SEED = .(..DATA + 4);
!+
! Return the data block to free storage.
!-
         LIB$FREE_VM (%REF (((...DATA) + 3)*4), .DATA);
!+
! Set the data's pointer to zero, so another call can initialize
! the data block again.
!-
         .DATA = 0;
!+
! Return the value 1.0.
!-

         CVTLF (%REF (1), RETURN_VALUE);
         RETURN (.RETURN_VALUE);
         END;
```

(continued on next page)

Figure 3-2: Allocating Heap Storage in BLISS (Cont.)

```
          CVTLF (%REF (1), RETURN_VALUE);
          RETURN (.RETURN_VALUE);
          END;

!+
! Compute a random number from 0.0 to 1.0, using scattering, and
! return it.
!
! First compute a random, 24-bit integer to index into
! the random number table. Use the same algorithm as the
! main generator, but with a different (usually) seed.
!-
          ..DATA + 8 = .(..DATA + 8) * 6909;
          ..DATA + 8 = .(..DATA +8) + 1;
          RAN1 = (.(..DATA + 8)<8, 24>);
!+
! Reduce the 24-bit random number modulo the table size
! and add the offset for the random numbers.
!-
          RAN1 = (.RAN1 MOD ...DATA) + 3;
!+
! Get a value from the table and replace it with a new value.
!-
          RETURN_VALUE = .((..DATA) + (.RAN1 * 4));
          (..DATA) + (.RAN1 * 4) = MTH$RANDOM ((..DATA) + 4);
!+
! Return to the caller of the random number generator
! the value from the table.
!-
          RETURN (.RETURN_VALUE);
          END;            !of RANSUB
```

Chapter 4
Coding Modular Procedures

This chapter describes how to code modular procedures and make existing procedures modular in MACRO, BLISS, BASIC, FORTRAN and PASCAL. These areas are discussed:

- Structured programming recommendations
- Coding rules and recommendations
- Procedure initialization
- Resource allocation
- Use of system services
- Invoking optional user action routines

Chapter 5 explains signaling and condition handling.

If you want your procedure to be AST reentrant, refer to Chapter 6 for additional coding techniques.

4.1 Structured Programming

Constructing good software depends on how well you organize the software project into conceptual layers. Program writing should start with the outermost abstract form and move inward toward successively greater detail.

Before coding individual procedures, consider how they might be grouped into modules. If you have a number of procedures that access common data or control blocks, try to organize them into separate levels, with each level having responsibility for different parts of the data structure.

4.1.1 Levels of Abstraction

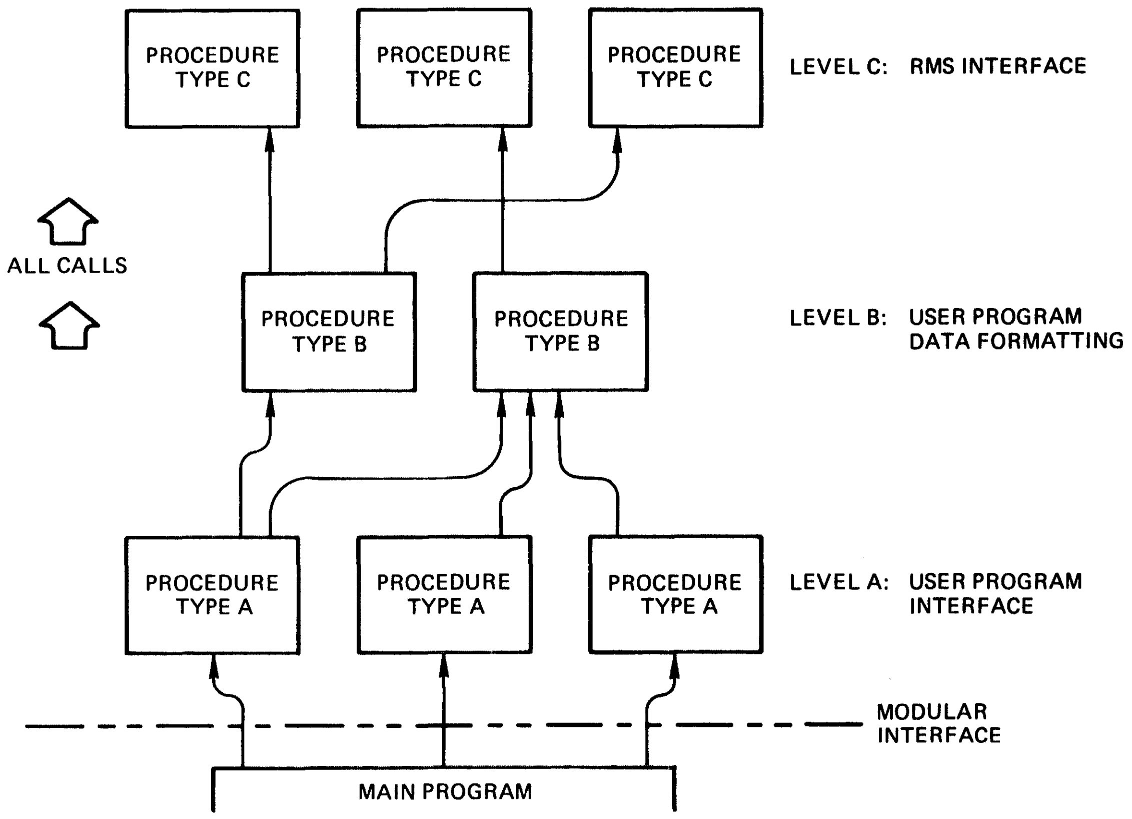
If you are writing a large number of related procedures that call one another or access common data blocks, try to achieve an understandable relationship among them. You can do this by organizing procedures to minimize interaction with each other and with the data structure:

- Organize procedures into levels of abstraction.
- Make sure each level needs to make calls only to the next level.
- Restrict read/write access at each level to nonoverlapping subsets of the data.

For example, Figure 4–1 shows the BASIC and FORTRAN record I/O statement processing procedures. These are implemented in the following three levels:

- User program interface (UPI)
- User program data formatting (UDF)
- Record processing and VAX–11 RMS interface (REC)

Figure 4–1: Levels of Abstraction



All calls are made in one direction: to the next innermost level. Procedures at different levels should also be in different modules.

4.1.2 Grouping Procedures

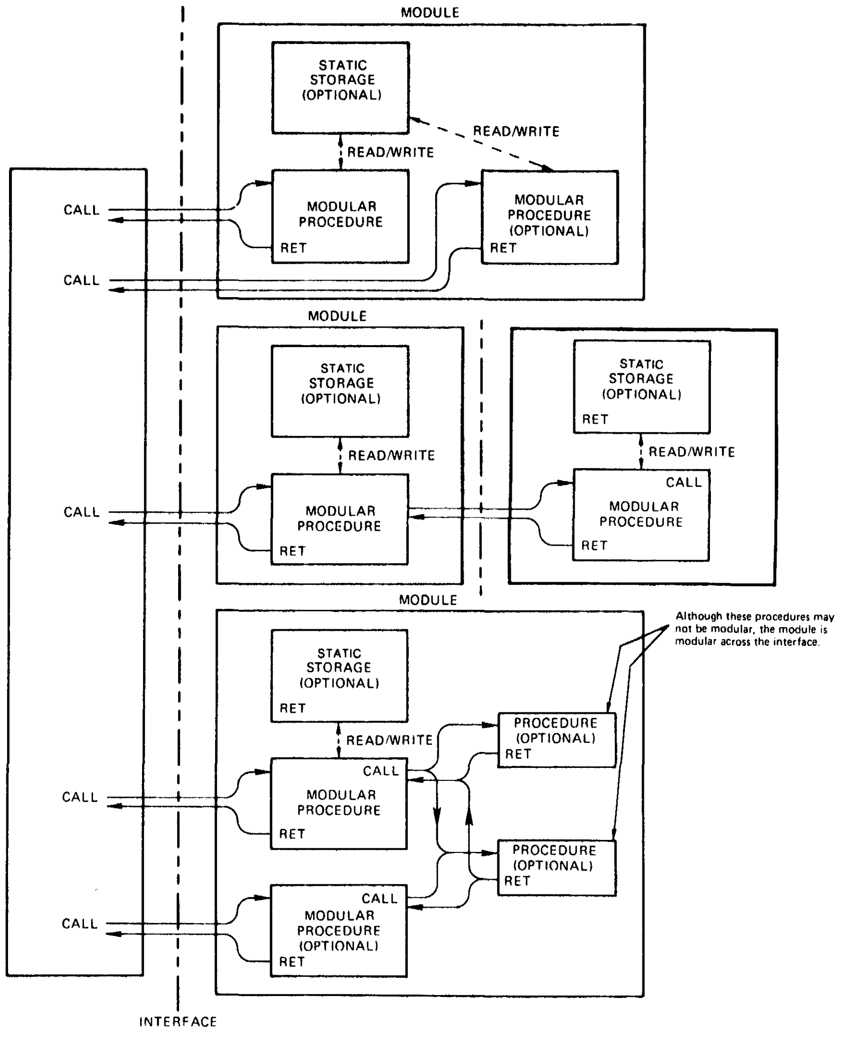
Each module should contain a single procedure. Occasionally, you may find it convenient to place more than one procedure in a single module if a procedure is called only by other procedures in that module. Grouping procedures in a module is also recommended if two or more procedures:

- Share the same static storage or
- Have similar calling sequences, perform similar functions, and share a significant amount of common code

The VAX-11 Linker always brings the entire module containing a called procedure into the image if any of its entry points are referenced. Thus, placing each procedure in a separate module reduces image size. It also increases the flexibility afforded a user of a procedures library because you can supply your own version of one procedure while using other procedures from the library. If many procedures have been grouped in a single module, the linker must link all of them or none.

Figure 4-2 shows possible groupings of procedures.

Figure 4-2: Possible Procedure Groupings



4.2 Coding Rules and Recommendations

Coding rules and recommendations help maintain modularity and produce consistent, readable software. You should choose simple rules. DIGITAL uses the following coding rules and recommendations for all modular procedures. You must follow the sections marked "standard". You may choose to follow sections marked "recommended" for procedures to be uniform and, therefore, easier to learn and remember how to use.

4.2.1 Relocatable Modules (Standard)

Most modules are, by default, relocatable during linking. The compiler or translator makes it appear to the linker that each module starts at location 0. The linker relocates each module to make it fit with the other modules being linked to form an executable image. A nonrelocatable module is a module with absolute storage allocation. It does not adhere to the modular standard since each absolute assignment might conflict with a similar assignment in another module.

4.2.2 File Names (Recommended) and Module Names (Standard)

File names are derived from procedure names. If a module contains a single procedure, the file name consists of the first nine characters of the procedure name without the dollar signs and underscores. If the module contains more than one procedure, a more general file name is used, composed of the facility prefix and the first noun common to all procedure names in the module. File name extensions are the standard default extensions for the source language.

Module names are identical to file names except for the dollar sign (DIGITAL-supplied) or underscore (user-supplied) inserted after the facility code. Module names do not have extensions.

For example, the MTH$EXP procedure is contained in module MTH$EXP and the file MTHEXP.MAR. The LIB$GET_VM and LIB$FREE_VM procedures are contained in the module LIB$VM and the file LIBVM.B32.

4.2.3 PSECT Names (Standard)

The code and data sections of a customer library procedure have two separate PSECTs, named _fac_CODE and _fac_DATA, where fac is the facility name. DIGITAL uses _fac$CODE and _fac$DATA PSECT names.

Position-independent constant data is in the _fac_CODE PSECT (_fac$CODE for DIGITAL) to shorten the references. For example, _LIB$CODE and _LIB$DATA are the only two PSECT names used by LIB$ procedures. The collating sequence for leading underscores causes the linker to place all library procedures after the user program in the executable image. Therefore, a library procedure will not be placed between two user modules. This prevents it from adversely affecting byte or word displacement addressing that the user program contains. The declarations are:

- **MACRO**

```
.PSECT _fac_CODE PIC,USR,CON,REL,LCL,SHR,EXE,RD,NOWRT
.PSECT _fac_DATA PIC,USR,CON,REL,LCL,NOSHR,NOEXE,RD,WRT
```

- **BLISS**

```
PSECT
      CODE = _fac_CODE (READ, NOWRITE, EXECUTE, SHARE, PIC,
         CONCATENATE, ADDRESSING_MODE (WORD_RELATIVE)),
      PLIT = _fac_CODE (READ, NOWRITE, EXECUTE, SHARE, PIC,
         CONCATENATE, ADDRESSING_MODE (WORD_RELATIVE)),
      OWN = _fac_DATA (READ, WRITE, NOEXECUTE, NOSHARE, PIC,
         CONCATENATE, ADDRESSING_MODE (LONG_RELATIVE)),
      GLOBAL = _fac_DATA (READ, WRITE, NOEXECUTE, NOSHARE, PIC,
         CONCATENATE, ADDRESSING_MODE (LONG_RELATIVE));
```

- **BASIC and FORTRAN**

  You do not have control over PSECT names, except named program COMMON. Note, however, that program COMMON replaces the PSECT attribute CONCATENATE with OVERLAY. Therefore, storage that you allocate using COMMON might overlay that allocated by a procedure written by someone else. Such a conflict between the two modules would be possible and would go undetected. Therefore, use of COMMON violates the modular programming standard.

- **PASCAL**

  You do not have control over PSECT names.

4.2.4 Parameter Definition Files (Recommended)

In some programs, it may be necessary to make identical parameter declarations in several modules. In MACRO, BLISS, BASIC, FORTRAN and PASCAL, such declarations are centralized in one place.

- **MACRO**

  An auxiliary source file or macro library can be specified in the command line.

- **BLISS**

  Your source program can contain a REQUIRE or LIBRARY declaration that specifies a file to be included at the point of the declaration.

- **BASIC**

  An auxiliary source file can be APPENDed to your main source program prior to compilation.

- **FORTRAN**

  Your source program can contain an INCLUDE statement that specifies a file to be included at the point of the statement.

- **PASCAL**

  Your source program can contain an %INCLUDE directive that specifies a file to be included at the point of the statement.

You should use this technique to declare the symbolic offsets in a control block accessed from several modules.

4.2.5 Symbols versus Numbers (Recommended)

Symbols, not numbers, should be used as much as possible. This improves understanding and provides more information for cross-reference listings.

- **MACRO**

  The use of local numeric labels is encouraged within a logical unit of code that fits on a single listing page.

- **BLISS**

  The defined transportable symbols are used for hardware defined quantities. For example, the size of a general value is %BPVAL (bits per value) instead of 32, and the length of a general value is %UPVAL (addressable units per value) instead of 4.

4.2.6 Line Length (Recommended)

The maximum line length for source code in each language follows. Line lengths are shown for actual source code (not including sequence numbers).

Language	Maximum Line Length
MACRO	80
BLISS	124
BASIC	512
FORTRAN	72
PASCAL	(no limit)

4.2.7 Uppercase and Lowercase (Recommended)

Uppercase should be used for all source code except comments. Upper- and lowercase should be used for all comments. Comments that are complete sentences should start with a capital letter and end with a period.

4.2.8 Optional Spaces (Recommended)

A single space must always follow a comma and precede and follow an equal sign (=). A single space must precede a left parentheses or a left square bracket (except in MACRO), but not a left angle bracket. A space also follows an exclamation mark or semicolon to separate a comment from the source code. Plus and minus symbols (+ and –) are surrounded by spaces in expressions.

4.2.9 Block Comments (Recommended)

You can comment on blocks of statements by writing one or more lines preceding the block. Comments start in column 1, independent of the indentation of the code. The first comment line contains a single plus sign (+); the last comment line contains a single minus sign (-). Block comments do not need to be set of by additional blank lines since the two flag lines (starting with + and -) serve that purpose. Comment delimiters are followed by one space, except when followed by the first + and the last -, as shown in this MACRO example:

```
        MOVL    R0, TABLE               ; Store current char. adr. in
                                        ; Code table
;+
; This is a block comment in MACRO.
;-
10$:    MOVL    TABLE, R0               ; R0 = current character
                                        ; address
```

4.2.10 Branch and Jump Instructions (Recommended)

- **MACRO**

  You should arrange code so branch and jump instructions refer to labels located forward in the program listing (except for loops and first-time initialization). This greatly improves the readability and understandability of the program as you read forward through the listing. When you encounter a label, you would have already read all of the code that could transfer to the label.

  Initialization of dynamically allocated stack and heap data only involves writing the data after each allocation before reading it.

4.3 Initializing Modular Procedures

Some modular procedures must initialize themselves before they can execute correctly. Examples of initialization are:

- Storing in static storage a value that can only be determined at run time
- Declaring an exit handler using the $DCLEXH system service
- Allocating a process-wide resource once
- Opening a process-permanent file the first time,in case a logical name was used and the file was not already opened

- **Other languages**

  GO TO instructions should follow similar conventions.

You must perform initialization carefully to avoid violating modularity principles:

- You must perform initialization so it does not affect the calling program. Therefore, you cannot perform initialization by providing an entry point that must be called before any other entry point is called: this would force the calling program to provide an initialization entry point to its caller, and so forth. Also you would not be able to replace a module that does not have an initialization call with one that does without rewriting your calling programs.

- If your procedure uses LIB$INITIALIZE, you must preserve a modular environment that does not conflict with the environment set by any other procedure using LIB$INITIALIZE.

Figure 4-3 shows five ways your procedures can perform initialization. The use of each method is explained in the following sections.

Figure 4-3: Methods of Initializing

Initialization Needed	Method				
	Initialize at Compile/Link Time	Call LIB$INITIALIZE Before Main Program (At Run Time)	Set a First Time Flag (At Run Time)	Initialize Each Time it is Allocated (At Run Time)	Initialize Each Time Procedure Is Called (At Run Time)
Of Static Storage:	●	●	●		
Of Stack Storage:					●
Of Heap Storage:				●	
To Allocate Resources:		●	●		
To Set Up $EXIT Handler:		●	●		
To Open a Process-Permanent File:		●	●		
To Set Up a Handler Before the Main Program:		●			

4.3.1 Initializing Storage Areas

For a procedure to produce predictable results, all statically and dynamically allocated areas must be initialized to known values before they are read.

The initialization of static storage need happen only one time in each image activation. Thus, the known values can be specified:

- At compile time by using a data initializing statement
- At link time by using a data allocation statement
- At run time on the first call to the procedure

4.3.2 Initializing Static Storage

If your procedure has static storage, you usually initialize it to zero. You can do this: (1) explicitly with a data initialization statement or (2) implicitly with the linker.

To save disk space, the linker does not include data pages initialized to zero in the .EXE file. In addition, I/O is eliminated since data pages are allocated upon your first access after the image is activated.

The following examples show initialization of a longword, DAT, in static storage at compile or link time.

STATEMENT	INITIALIZED VALUE
• **MACRO**	
DAT: .BLKB 1	0
DAT: .LONG 0	0
DAT: .LONG 100	100
• **BLISS**	
OWN DAT;	0
OWN DAT INITIAL(0);	0
OWN DAT INITIAL(100);	100
• **FORTRAN**	
INTEGER*4 DAT	0
DATA DAT /0/	0
DATA DAT /100/	100

NOTE

BASIC does not have static storage within a module, only common static storage.

PASCAL has static storage at Module level. However, you cannot initialize it at compile time.

4.3.3 Testing and Setting First-Time Flag

Occasionally, your procedure requires initialization that can be performed only at runtime, the first time your procedure is called. Examples are:

- Initializing static storage to a value that can only be determined at run time.
- Establishing an EXIT handler
- Allocating a resource
- Opening a process permanent file

To do first-time initialization, your procedure tests and then sets to one a statically allocated first time flag each time it is called. This flag is initialized to zero at compile or link time. Setting and testing the flag with the VAX instruction BBSS (Branch on Bit Set and Set) insures that initialization is executed exactly once.

However, if the procedure is to be AST-reentrant, it must follow these steps:

1. Test the first-time flag.
2. If it is set, initialization is complete.
3. Otherwise, disable ASTs, remember previous state of AST enable, and retest the flag.
4. If the flag is now set, initialization was performed by an AST that went off between the first test and the AST disable; enable ASTs if remembered state of ASTs was enable – initialization is complete.
5. Otherwise, perform the initialization.
6. Set the flag.
7. Enable ASTs if remembered state of ASTs was enable – initialization is complete.

NOTE

ASTs should only be enabled (Step 4 or Step 7) if they were enabled before Step 3. The $SETAST system service, used to disable ASTs, indicates whether ASTs were enabled when the procedure was called.

For example, your procedure can use the VAX instructions INSQUE and REMQUE to maintain a set of queues whose headers are in static storage. However, to maintain a position-independent data region, the address in the queue header can be initialized only at run time. The STR$COPY procedures use this technique to initialize dynamic string storage to a set of empty queues. Each allocation of dynamic string storage is performed by first trying to remove a pre-allocated block from the appropriate queue.

The following BLISS fragment shows the code for opening a process permanent file on the first call. The code could be part of a GET routine.

```
     LOCAL
          RET_STATUS,           ! RMS status
          FAB : $FAB_DECL,      ! FAB
          RAB : $RAB_DECL;      ! RAB

     MAP
          GET_STRING : REF BLOCK [8, BYTE], ! String desc.
          PROMPT_STRING : REF BLOCK [8, BYTE], ! String desc.
          GET_ISI : REF VECTOR [1, WORD, UNSIGNED]; ! Place in
                                ! static storage to remember ISI
!+
! Enable a handler to return string signals as error
! codes to the caller.
!-
     ENABLE
          LIB$SIG_TO_RET;
          IF (.GET_ISI [0] EQL 0)
          THEN
!+
! First call, initialize FAB
!-
          BEGIN
          $FAB_INIT (FAB = FAB, FAC = GET,    ! File access: GET
            FNA = .DEVICE_NAME,               ! File name: DEVICE_NAME
            FNS = .DEVICE_NAME_LEN);          ! File name size
!+
! Open DEVICE_NAME, remember RMS internal stream identifier
!-
          RET_STATUS = $OPEN (FAB = FAB);     ! Fab adr = FAB
!+
! If the OPEN fails, return the RMS status code.
!-
     IF ( NOT .RET_STATUS) THEN RETURN (.RET_STATUS);

          $RAB_INIT (FAB = FAB, RAB = RAB);
          RET_STATUS = $CONNECT (RAB = RAB); ! Connect RAB to file
!+
! If the CONNECT fails, return the RMS status code.
!-
          IF ( NOT .RET_STATUS) THEN RETURN (.RET_STATUS);

          GET_ISI [0] = .RAB [RAB$W_ISI];     ! Remember ISI
          END
!+
! File already open, just initialize RAB
! including ISI returned from first $OPEN
!-
     ELSE
          BEGIN
          $RAB_INIT (FAB = FAB, RAB = RAB);
          RAB [RAB$W_ISI] = .GET_ISI [0];
          END;
!+
! Continue setup and get string
!-
```

Another example of performing first-time initialization transparent to the caller is to establish an EXIT handler to perform once some cleanup operation when the image exits. Again, this is done by testing and then setting a first-time flag. If the flag is clear, the Declare EXIT Handler system service ($DCLEXH) is called to establish the exit handler.

4.3.4 Adding a Dispatch Address to PSECT LIB$INITIALIZE

To use this method, your module generates one or more longwords that contain one or more addresses of procedures to be called by the system before the main program is called. Your module must declare these longwords to be part of the overlayed PSECT LIB$INITIALIZE. (Examples of this method are shown in Appendix E of the *VAX-11 Run-Time Library Reference Manual.*)

Note that a module in a sharable image cannot use this method because the PSECT contribution would be to the shared image, not to the user program image. Furthermore, if this method were used, a modular procedure could not establish a condition handler before a main program (using LIB$INITIALIZE) to alter how signaled errors are handled, when the handling conflicts with a condition handler that might be established by another procedure before the main program.

4.4 Allocating Resources

A resource is a part of the hardware or software system that can be allocated and deallocated. It is therefore either in use or free for use. For reliable operation, each instance of a resource must be allocated to only one owner at a time. All potential owners must agree beforehand on the technique for allocating each resource.

There are process-wide resources and system-wide resources. System-wide resources, such as disk memory and physical memory, are allocated on behalf of a process by the operating system. The following discussion is limited to process-wide resources.

Process-wide resources are allocated on behalf of a procedure activation executing within a single process by one of two methods:

- A single allocator is used by all procedures in the image to allocate (and/or deallocate) the resource.
- A standard discipline is agreed on so many allocators can make nonconflicting allocations.

Examples of the single allocator approach are when:

- The linker allocates relocatable virtual addresses among competing procedures in an image.
- The $ASSIGN system service assigns I/O channel numbers to competing procedures in a single process for each procedure that needs a separate channel.
- The library procedures LIB$GET_VM and LIB$FREE_VM allocate and deallocate virtual memory to requesting procedures in an image (see Chapter 5 of the *VAX-11 Run-Time Library Reference Manual* for the process-wide resource allocating procedures provided by the system).

Examples of the multiple-allocator approach are when:

- Each procedure allocates and deallocates its own stack storage using registers FP and SP to maintain discipline

- Each procedure allocates registers from the pool of process registers (R2 to R11) after saving the contents of these registers on the process stack using the entry mask mechanism

4.4.1 Using Storage with Resource-Allocating Procedures

A resource-allocating procedure must use some static storage to keep track of allocated and deallocated resources. Therefore, all resource-allocating procedures should follow the special considerations needed by AST-reentrant procedures with static storage (see Chapter 6).

You cannot use BASIC and PASCAL to write modular resource allocating procedures because:

- BASIC does not use static storage in a module.

- PASCAL uses static storage at the module level, but the module-level declarations must match those in the main program.

4.4.2 Allocating Identification Numbers

The following MACRO procedure LIB_GET_INUM allocates identifying numbers that can be used to identify a resource:

```
TAB:    .WORD     0                          ; Bitmap for flags
        .ENTRY    LIB_GET_INUM, ^M<>
        FFC       #1, #10, TAB, R0           ; Find first free id. no.
        BEQ       20$                        ; Branch if none free
        BBSS      R0, TAB, 10$               ; Indicate id. no. in use
10$:    MOVL      R0, @4(AP)                 ; Return id. no. found
        MOVL      #1, R0                     ; Indicate success
        RET

20$:    CLRL      @4(AP)                     ; Return 0
        CLRL      R0                         ; Indicate failure
        RET
        .END
```

To make this procedure AST-reentrant, move the label 10$ from the MOVL instruction to the FFC instruction. (See Section 6.3.2.)

The equivalent FORTRAN module contains procedures to allocate and deallocate identifying numbers:

```
      FUNCTION LIB_GET_INUM (INUM)
      INTEGER*4 INUM, INUM_TABLE(100)
      LIB_GET_INUM = 1                    ! Assume success
      DO 10 I=1,100
        IF(INUM_TABLE(I) .EQ. 0) THEN
          INUM_TABLE (I) = 1              ! Flag unit as in use
          INUM = I-1
          RETURN                          ! return id no.
        ENDIF
10    CONTINUE
      LIB_GET_INUM = 0                    ! Indicate failure
      RETURN

C     Deallocate identifying number

      ENTRY LIB_FREE_INUM

      IF (INUM_TABLE(INUM+1) .EQ. 1 THEN
        INUM_TABLE(INUM+1) = 0            ! Flag unit as free
        LIB_FREE_INUM = 1                 ! Indicate success
      ELSE IF
        LIB_FREE_INUM = 0                 ! Indicate already free
      ENDIF
      RETURN
      END
```

LIB_GET_INUM can be called from a FORTRAN program in the following way:

```
.
.
.
IF .NOT. (LIB_GET_INUM(I)) THEN GO TO error
... = I
.
.
.
```

4.4.3 Process-Wide Resources

Table 4–2 shows process-wide resources and their single allocator or discipline for multiple allocators.

VAX/VMS does not provide resource allocation procedures or allocation discipline for the following resources:

- Logical Names
- Process Names
- Event flag cluster numbers 2 and 3

However, if a library resource allocation procedure does not exist, you can write your own, as indicated by the examples in Section 4.4.2.

Table 4-1: Methods of Allocating Resources

Resource	Allocation Method
R0, R1	Not a shared resource.
R2:R15	Preserved using stack frame discipline. (See Appendix C.)
PSL	Preserved using stack frame discipline.
Virtual memory	Allocated statically by linker.
	Allocated dynamically by either $EXPPRG or LIB$GET_VM.
	Deallocated dynamically only by LIB$FREE_VM.
Static storage for nonresource allocation procedures	Procedure to push old contents onto a stack in heap storage and another to pop old contents back.
	Caller allocates storage.
Process-wide identifiers for static storage	Procedure to assign process-wide identifiers.
Dynamic string memory	Written only by calling LIB$, OTS$, or STR$ string procedures. (See Chapter 5 of the *VAX-11 Run-Time Library Reference Manual.*)
VMS event flags	Process local event flags 32-63 allocated by calling LIB$GET_EF. Process local event flags 1-23 and 32-63 can be reserved by calling LIB$RESERVE_EF. All can be freed by calling LIB$FREE_EF.
BASIC/FORTRAN logical unit numbers (channel numbers)	Process logical unit numbers 100-119 allocated by calling LIB$GET_LUN and freed by calling LIB$FREE_LUN.
Condition codes (message IDs)	Bits 27:16 contain the facility number. Bit 27 is 0 for those signed out by DIGITAL, and 1 for those signed out by customers. Each allocator must ensure uniqueness in bits 15:3. Also, the symbols for the completion status codes and signaled conditions are contained in a separate source file for each facility.
Global Symbols	DIGITAL-assigned symbols available to users contain a single "$". Within DIGITAL, a facility prefix identifies a person responsible for allocating unique symbols. Global symbols not available to users contain two dollar signs. User-defined symbols should contain a _ instead of a $ to avoid conflict with DIGITAL symbols.

4.5 Passing Strings as Parameters

This section describes the techniques your procedures can use to accept and return fixed-length and dynamic string parameters.

For both string types, the calling program either: (1) allocates the string's descriptor and passes its address or (2) passes the address of a descriptor that had been passed to it (by its caller).

The descriptor contains:

- A 16-bit string length in bytes (DSC$W_LENGTH)
- An 8-bit data type code (DSC$B_DTYPE)
- An 8-bit descriptor class code (DSC$B_CLASS) and
- A 32-bit address of the first byte of the string (DSC$A_POINTER)

The calling program indicates the descriptor class in the DSC$B_CLASS field. A fixed-length descriptor cannot be modified by the called procedure. However, the called procedure (using dynamic string-allocating library procedures) can modify the length and address field of a dynamic string descriptor. The following section describes input and output parameters in detail.

4.5.1 Accepting Input String Parameters

Procedures accept both fixed-length and dynamic string descriptors as input parameters in the same way: the string length, string address, and data type fields are in the same place in both classes of descriptor. Thus, a procedure can accept either class of string. Modular procedures can read strings by any of the following methods:

- Accessing the length and address field indirectly through the parameter list
- Copying the address of the string descriptor
- Copying the contents of the string descriptor (see Section 4.5.3 about setting the class field in the copied descriptor, if it is to be passed to other procedures)

4.5.2 Returning Output String Parameters

This section describes the semantics of returning fixed-length or dynamic strings as output parameters or as a function value.

The semantics for returning a fixed-length string are:

- The called procedure does not modify the string descriptor passed by the calling program.
- The called procedure writes the string starting at the address specified in the descriptor (DSC$A_POINTER). If the actual string length indicated in the descriptor (DSC$W_LENGTH) is not the correct size, the called procedure fills the string with trailing ASCII spaces or truncates it on the right.

- If truncation occurs, the called procedure can return either: (1) the success condition value STR$_TRU or LIB$STRTRU or (2) an error condition value as a completion status in R0 depending upon the application.

The semantics for returning a dynamic string are:

- The called procedure can modify the string descriptor passed by the calling program only:
  - if the descriptor class code is dynamic (DSC$K_CLASS_D) and
  - by calling the dynamic string allocation procedures (STR$GET1_DX, STR$COPY_DX, or STR$COPY_R).
- Using the dynamic descriptor passed by the calling program, the called procedure can use either of these methods:
  - Create the entire string to be returned and pass it to STR$COPY_DX, or STR$COPY_R to be copied using the dynamic descriptor as the destination, or
  - Allocate the total amount of string space needed (by calling STR$GET1_DX using the descriptor passed by the calling program) and fill the dynamically allocated area piece-by-piece using the modified contents of the descriptor.
- If the resource-allocating string procedure exhausts the virtual memory for your process, your procedure should also indicate the error to the calling program by either: (1) returning the error condition value in R0 (LIB$ convention) or (2) signaling the error condition (STR$ convention).
- The called procedure cannot make a copy of the dynamic string descriptor since its contents can change whenever the string is written. Therefore, each dynamic string must have one and only one dynamic string descriptor pointing to it.

The calling program can always pass either: (1) a fixed-length string or (2) a dynamic string, as indicated in the DSC$B_CLASS field in the descriptor (fixed length is DSC$K_CLASS_S = 1 or DSC$K_CLASS_Z = 0; dynamic is DSC$K_CLASS_D = 2).

Your procedure interface specification can indicate that your procedure will return an output string parameter (or function value) by using either: (1) the semantics indicated by the calling program in the descriptor (preferred) or (2) fixed-length string semantics.

A modular procedure cannot expect or require a calling program to pass a dynamic string. However, if you are using:

- The preceding method 1, your procedure can always call the library procedure, since it performs the semantics indicated in the descriptor.

- The preceding method 2, before calling STR$GET1_DX, your procedure must check the class code in the string descriptor (DSC$B_CLASS) and perform fixed-length semantics explicitly if the class code is DSC$K_CLASS_S = 1 or DSC$K_CLASS_Z = 0.

The following table shows the action your procedure takes for all combinations of interface specification and descriptor class passed by the calling program:

Table 4-2: Procedure's Action for Strings Passed by Calling Program

	Interface Specification for Output String	
String passed by calling program	Fixed-length semantics (-.wt.ds) (ignore DSC$B_CLASS)	Semantics specified by calling program (-.wt.dx) (observe DSC$B_CLASS)
Fixed-length (DSC$B_CLASS=0,1)	Space fill or truncate using DSC$W_LENGTH and DSC$A_POINTER	Space fill or truncate using DSC$W_LENGTH and DSC$A_POINTER
Dynamic (DSC$B_CLASS=2)	Space fill or truncate using DSC$W_LENGTH and DSC$A_POINTER	Use library dynamic string procedures

4.5.3 Passing String Parameters to Other Procedures

The following restrictions apply to string parameters passed from the calling program to your procedure, and then from your procedure to another procedure:

- If you have specified that your procedure (and any it calls) only accesses the string as an input parameter, your procedure can pass the address of either (1) the original descriptor (preferred) or (2) a copy of the descriptor.
- If you have specified that your procedure (and any it calls) accesses the parameter as an output parameter using fixed-length semantics (wt.ds), your procedure can pass the address of either:
  - The original descriptor (to any procedure accessing it), or
  - A copy of the descriptor in which the class code field has been forced to fixed-length (DSC$K_CLASS_S = 1) to any procedure accessing it as output using the semantics specified by the calling program (wt.dx).
- If you have specified that your procedure (or any it calls) accesses the parameter as an output parameter using the semantics specified by the calling program, your procedure must pass the address of the original descriptor: a dynamic string must have one and only one descriptor pointing to it.

- If you do not know the semantics used by a procedure that your procedure calls, you should assume the most general case and pass the address of the original descriptor, not a copy.

4.6 Using VAX/VMS System Services

The following sections list the VAX/VMS system services by categories. Procedures that call nonmodular system services are nonmodular themselves. Procedures using nonmodular system services should list them in the SIDE EFFECTS section of the procedure description.

NOTE

The first column in each section indicates whether the service is modular; the numbers in parentheses refer to explanatory notes in Section 4.6.13.

4.6.1 Event Flag Services

no(16)	$ASCEFC	Associate Common Event Flag Cluster
no(16)	$DACEFC	Disassociate Common Event Flag Cluster
no(16)	$DLCEFC	Delete Common Event Flag Cluster
yes(1)	$SETEF	Set Event Flag
yes(1)	$CLREF	Clear Event Flag
yes	$READEF	Read Event Flag
yes(1)	$WAITFR	Wait For Single Event Flag
yes(1)	$WFLOR	Wait For Logical OR of Event Flag
yes(1)	$WFLAND	Wait For Logical AND of Event Flag

4.6.2 Asynchronous System Trap (AST) Services

yes(15)	$SETAST	Set AST Enable
yes	$DCLAST	Declare AST
yes	$SETPRA	Set Power Recovery AST
no(5,2)	$CLRAST	Clear AST Enable

4.6.3 Logical Name System Services

VMS stores logical names in process-wide storage. Therefore, they cause the same modularity problems as other static storage.

no(2,13)	$CRELOG	Create Logical Name
no(3,13)	$DELLOG	Delete Logical Name
yes	$TRNLOG	Translate Logical Name

4.6.4 I/O System Services

yes	$ASSIGN	Assign I/O Channel
yes(3)	$DASSGN	Deassign I/O Channel

yes(1)	$QIO	Queue I/O Request
yes(1)	$QIOW	Queue I/O Request and Wait For Event Flag
yes(1)	$INPUT	Queue Input Request and Wait For Event Flag
yes(1)	$OUTPUT	Queue Output Request and Wait For Event Fl
yes	$ALLOC	Allocate Device
yes(3)	$DALLOC	Deallocate Device
yes	$GETCHN	Get I/O Channel Interface
yes	$GETDEV	Get I/O Device Information
yes	$GETCHN	Get I/O Channel Information
no(3)	$CANCEL	Cancel I/O on Channel
no(2,13)	$CREMBX	Create Mailbox and Assign Channel
yes(3)	$DELMBX	Delete Mailbox
no	$BRDCST	Send Message to All Terminals
yes	$SNDACC	Send Message to Accounting Manager
yes	$SNDSMB	Send Message to Symbiont Manager
yes	$SNDERR	Send Message to Error Logger
yes	$SNDOPR	Send Message to Operator

NOTE

The first column in each section indicates whether the service is modular; the numbers in parentheses refer to explanatory notes in Section 4.6.13.

4.6.5 Process Control Services

When using the process control services, you must specify the process name parameter as zero; otherwise, a resource allocation procedure is needed to assign different values.

yes(4)	$CREPRC	Create Process
yes(3,4)	$DELPRC	Delete Process
yes(3,4)	$SUSPND	Suspend Process
yes(3,4)	$RESUME	Resume Process
yes	$HIBER	Hibernate
yes(3)	$WAKE	Wakeup
yes(3,4)	$SCHDWK	Schedule Wakeup
yes(3,4)	$CANWAK	Cancel Wakeup
no(5)	$EXIT	Exit
yes(3)	$FORCEX	Force Exit
yes	$DCLEXH	Declare Exit Handler
yes	$CANEXH	Cancel Exit Handler
no(3,4)	$SETPRN	Set Process Name
yes(3)	$SETPRI	Set Priority
no(2)	$SETRWM	Set Resource Wait Mode
yes(4)	$GETJPI	Get Job/Process Information
yes(3)	$SETPRV	Set Privileges

4.6.6 Timer and Time Conversion System Services

yes	$GETTIM	Get Time
yes	$NUMTIM	Convert Binary Time to Numeric Time
yes	$ASCTIM	Convert Binary Time to ASCII String
yes	$BINTIM	Convert ASCII String to Binary time
yes(1)	$SETIMR	Set Timer
yes(1)	$CANTIM	Cancel Timer Request
no(2)	$SETIME	Set System Time

4.6.7 Condition Handling System Services

no(2)	$SETEXV	Set Exception Vector
no(2)	$SETSFM	Set System Service Failure Exception Mode
yes	$UNWIND	Unwind Call Stack
no(8)	$DCLCMH	Declare Change Mode or Compatibility Mode Handler

NOTE

The first column in each section indicates whether the service is modular; the numbers in parentheses refer to explanatory notes in Section 4.6.13.

4.6.8 Memory Management System Services

yes(11)	$EXPREG	Expand Program/Control Region
no(6)	$CNTREG	Contract Program/Control Region
yes(17)	$CRETVA	Create Virtual Address Space
yes(17)	$DELTVA	Delete Virtual Address Space
yes(7,18)	$CRMPSC	Create and Map Global Section
no(7)	$UPDSEC	Update Global Section File on Disk
yes(7,18)	$MGBLSC	Map Global Section
yes(3)	$DGBLSC	Delete Global Section
no(5)	$LKWSET	Lock Pages in Working Set
no(5)	$ULWSET	Unlock Pages from Working Set
yes	$PURGWS	Purge Working Set
no(8)	$LCKPAG	Lock Page in Memory
no(8)	$ULKPAG	Unlock Page from Memory
no(8)	$ADJWSL	Adjust Working Set Limit
yes(17)	$SETPRT	Set Protection on Pages
no(5)	$SETSWM	Set Process Swap Mode

4.6.9 Change Mode System Services

no(8)	$CMEXEC	Change Mode to Executive Mode
no(8)	$CMKRNL	Change Mode to Kernel Mode
no(8)	$ADJSTK	Adjust Outer Mode Stack Pointer

4.6.10 Error Messages

The error message identification (32-bit condition code) has an allocation discipline in which bits 26:16 are assigned by DIGITAL as facility codes. Each facility is administered by someone who ensures uniqueness of bits 15:3. However, for correct modularity, all modular procedures must use LIB$SIGNAL (or LIB$STOP) error handling rather than outputting an error message themselves. Only the catch-all handler can use the following system services.

yes	$GETMSG	Get Message
yes(10,12)	$PUTMSG	Put Message

NOTE

The first column in each section indicates whether the service is modular; the numbers in parentheses refer to explanatory notes in Section 4.6.13.

4.6.11 Formatted ASCII Output

yes	$FAO	Formatted ASCII Output
yes	$FAOL	Formatted ASCII Output with List Parameter

4.6.12 RMS System Services

In the following calls, the file name is passed as an explicit parameter or is derived from an explicit parameter passed to a modular procedure from a nonmodular procedure or from a user. Otherwise, the file name can conflict with one that already exists. Do not use the RMS optional success and error action routines; they depend on AST interrupts being enabled even for synchronous I/O. This dependency is not appropriate for modular procedures.

NOTE

In principle, a modular procedure could: (1) save, and enable ASTs using $SETAST (see note 15 in Section 4.6.13), (2) do synchronous RMS I/O with action routines and (3) restore the AST enables. However, the extra overhead would probably not be worth the trouble.

yes(3)	$CLOSE	CLOSE file
yes	$CONNECT	CONNECT I/O stream
yes(9)	$CREATE	CREATE file
yes(3)	$DELETE	DELETE record
yes(3)	$DISCONNECT	DISCONNECT I/O stream
yes	$DISPLAY	DISPLAY information
yes(3)	$ERASE	ERASE file
yes	$EXTEND	EXTEND file
yes	$FIND	FIND record
yes(3)	$FLUSH	Write out all modified I/O Buffers
yes(3)	$FREE	Unlock all previously locked records

yes(14)	$GET	GET record
yes	$NXTVOL	Magnetic tape processing continues to next volume
yes(9)	$OPEN	Open File
yes(14)	$PUT	Write a new record to a file
yes	$READ	Retrieve a specified number of bytes from a file
yes(3)	$RELEASE	Unlock a record pointed to by RFA field
yes	$REWIND	Position first record of a file
yes	$SPACE	Space forward or backward in a file
yes	$TRUNCATE	Truncate a sequential file
yes	$UPDATE	Update an Existing Record
yes(3)	$WAIT	Determine completion of asynchronous operation
yes	$WRITE	Write specified number of bytes to a file

4.6.13 Modular Procedure Notes

1. This service has a process-wide resource (event flag) as an input parameter. Process local event flags must be allocated by calling the library procedures LIB$GET_EF and LIB$FREE_EF to ensure allocation of a unique event flag.

2. This service changes process-wide static storage of VMS from the default expected by modular procedures. Thus, use by more than one procedure could cause a conflict. Further problems result if an AST interrupt occurs while static storage is in a nondefault state.

3. A module can only deallocate or operate upon items (for example, processes, memory, devices, global sections) that are known to have been allocated by it.

4. No process name can be specified since there would have to be a group-wide allocator to allocate a unique process name within the group.

5. This service could adversely affect the execution of other modular/reentrant procedures in the process.

6. You cannot use $CNTREG to contract the program or control region because you would violate the standard of not relying on a particular value of an implicit input to a procedure. Some other procedure might have expanded the region after you had expanded it.

7. These services need a system-wide, group-wide, or process-wide allocation procedure or discipline.

8. These services could adversely affect the execution of user-written procedures that are not modular/reentrant because they are also using these system services.

9. File names must be passed as explicit arguments or derived from explicit arguments passed to a modular procedure from a nonmodular procedure or from a user via SYS$INPUT.

10. Use LIB$SIGNAL instead. This lets the caller write application-specific error messages.

11. If LIB$FREE_VM deallocates space in the program region, LIB$GET_VM must be called to reuse the deallocated space. (See Chapter 5 of the *VAX-11 Run-Time Library Reference Manual.*)
12. Modular procedures should provide an optional action routine parameter so that the calling program can control human-readable output.
13. This service needs a logical name allocation procedure.
14. To be AST reentrant when using $GET and $PUT, check for record stream active error (RMS$_RSA). If the error is encountered, call $WAIT and try again. (See Section 6.4.)
15. To use $SETAST in a modular procedure, you must save the old setting and restore it before returning to the calling program. You must also establish a condition handler to restore the setting in case of a stack unwind.
16. For modularity, this service requires a resource-allocating procedure to allocate event flag cluster numbers 2 and 3, which are not provided for in the VAX-11 Run-Time Library.
17. This service applies only to pages that were statically or dynamically allocated to your procedure.
18. $CRMPSC and $MGBLSC apply only with the SEC$M_EXPREG flag that creates or maps a global section into the first available space set.

4.7 Invoking Optional User Action Routines

An optional user action routine is a useful way to let the calling program gain control at a critical point in your procedure's algorithm.

There are two VAX-11 data types used to represent a procedure to be passed as a parameter. The first, and simplest, is used by FORTRAN and is expected by the Run-Time Library, RMS, and VMS System Services. It is called Entry Mask (ZEM). The second is used by PASCAL and other languages where a particular procedure activation must be specified: the procedure might do up-level addressing of a variable defined in a syntactically outer block and hence, allocated in another frame. It is called Bound Procedure Value (BPV).

For ZEM passed by reference, the argument list entry contains the address of the procedure entry mask to be called. For BPV passed by reference, the argument list entry contains the address of two longwords. The first longword contains the address of the procedure and the second contains the environment pointer to be loaded into R1 before the procedure is called. The VAX-11 Procedure Calling Standard explicitly permits a BPV data type to be passed by immediate value, in which case the second longword is omitted entirely, and the first longword (address of entry mask) is placed in the argument list entry making it identical to ZEM passed by reference.

To provide a user-action routine interface for your procedure, you must first decide whether to use the ZEM or BPV data type. Since higher-level languages that support BPV by default, must provide the language extension to force immediate value, ZEM is more language-independent. However, ZEM is more awkward for calling programs written in languages like PASCAL.

To make it easy for the calling program to pass information to its action routine your procedure should supply an optional user-arg parameter that the calling program can pass to its action routine. Your procedure merely copies the argument list entry of the user-arg, if present, to the argument list it passes to the action routine. This achieves the same effect as up-level addressing.

Often it is convenient to specify a default action if the optional action routine is not supplied by the calling program.

To provide a user action routine, your procedure should have the following calling sequence:

CALL myproc (...[,action-routine.fzemlc.r[,user-arg.xy.z]])

or

CALL myproc (...[,action-routine.fbpvlc.r[,user-arg.xy.z]])

The user action routine has the calling sequence:

status.wlc.v = action-routine (...[,user-arg.xy.z])

where your procedure copies the 32-bit arg list entry passed by the calling program to the argument list provided to the action routine. Thus, the calling program and its action routine can communicate using any data type, access type, passing mechanism, or arg form.

The following code fragment shows how to test for the presence of an optional user action routine and pass it a line of text and the optional user arg. If no user action routine is supplied, LIB$PUT_OUTPUT is called as the default action routine.

```
    ... = 4                     ; Formal 1 - ...
    ACTION_ROUTINE = 8          ; Formal 2 - action routine
    USER_ARG = 12               ; Formal 3 - user arg

    .ENTRY MYPROC ^M< ... >
    .
    .
    .
    MOVAQ    ..., R0        ; R0 = adr. of string descr. for line
    CMPB     (AP), #<ACTION_ROUTINE/4>
; Test no. of caller parameters
    BLSSU    30$            ; Branch if no action routine specified
    TSTL     ACTION_ROUTINE(AP)
; Test for 0 action routine adr.
    BEQU     30$            ; Branch if no action routine specified
             ; (LIB$ convention)
    CMPB     (AP), #<USER_ARG/4>
; Test no. of caller parameters
    BLSSU    20$            ; Branch if no optional user-arg par.
```
(continued on next page)

```
;+
; Call user action routine (ZEM) with optional user-arg parameter
;-
     PUSHL    USER_ARG(AP)       ; 2nd par = user-arg list entry
     PUSHL    R0                 ; 1st par = adr. of string descr.
     CALLS    #2, @ACTION_ROUTINE(AP)
; Call user action routine
     BRB      40$                ; Join common code
;+
; Call user action routine (ZEM) without optional user-arg paramete
;-
20$: PUSHL    R0                 ; 1st par = adr. of string descr.
     CALLS    #1, @ACTION_ROUTINE(AP)
; Call user action routine
     BRB      40$
;+
; Call LIB$PUT_OUTPUT - caller did not supply user action routine.
;-
30$: PUSHL    R0                 ; 1st par = adr. of string descr.
     CALLS    #1, LIB$PUT_OUTPUT
; Output line to SYS$OUTPUT
40$: BLBC     R0,...             ; Test for error status
```

To call a BPV user action routine replace the two CALLS instructions with the following, where n is 2 and 1, respectively.

```
MOVQ
@ACTION_ROUTINE(AP)
; R0 = adr of procedure,



; R1 = environment value

CALLS
#n,(R0)
; Call user supplied action routine
```

Chapter 5
Signaling and Condition Handling

A modular procedure should not print error or informational messages either: (1) directly on an output device or (2) by calling the $PUTMSG system service.

Instead, a modular procedure uses either of these techniques:

- It returns a condition value as a function value (preferred). (See Section 5.2.)
- It signals a condition value by calling LIB$SIGNAL or LIB$STOP when a failure occurs. The absence of a signal indicates success. (See Section 5.3 of this manual and Section 6.6 of the *VAX-11 Run-Time Library Reference Manual.*)

Otherwise, the calling program is unable to control or change all effects of your procedure, thereby precluding use of it in certain situations. For example, an applications program used by a nonprogramming clerk should output an applications-specific message (such as "Please start over"), not a systems programming-oriented message (such as "MRS, maximum record size invalid").

5.1 Condition Values

A condition value is a 32-bit quantity. Success or failure is indicated in bit 0 as a 1 or 0. Besides indicating success or failure of your program, the condition value can provide this information:

- Severity of the failure
- Error identification
- Associated message text
- Facility detecting the error
- Control of error message printing

5.2 Returning a Condition Value as a Function Value

These structured programming advantages are inherent in returning a condition value as a function value:

- All execution paths are confined to syntactic blocks that have a single entry and a single exit point from the calling program's viewpoint.
- Error contingencies are considered when the calling program is written, thereby increasing program reliability.
- The action of the calling program is clearly indicated when errors occur.

Your procedure can be called as a main program, and the condition value will be returned to the command language interpreter.

The following sections describe how a procedure can return a condition value and how a calling program can check it for success or failure.

5.2.1 Returning and Checking an Error Status

- **MACRO**

  The following example shows the simplest way to return success or failure from a MACRO procedure (see Section 5.2.2 for the preferred way, using condition value symbols that indicate the specific reason for an error):

```
        .ENTRY   PROC, M<...>
.
.
.
;+
; Success return
;-
        MOVL     #1,    R0;    R0 = 1 - success
        RET
;+
; Failure return
;-
        CLRL     R0          ;  R0 = 0 - failure
        RET
```

  This example shows how a MACRO calling program can check for success or failure in a called procedure (whether called procedure returns 0 or 1 or uses a symbolic condition value):

```
.EXTRN PROC
CALLG    ARGLST, PROC           ; call procedure
BLBC     R0, 10$  ; branch on error
```

- **BLISS**

  The following example shows a simple way to return success or failure from a BLISS procedure (see Section 5.2.2 for the preferred way, using condition value symbols that indicate the specific reason for an error):

```
GLOBAL ROUTINE PROC (X,Y,Z) =
BEGIN
     .
     .
     .
IF ... THEN RETURN 1 ELSE RETURN 0 ;
END ;
```

This example shows how a BLISS calling program can check for success or failure in a called procedure (whether called procedure returns 0 or 1 or uses a symbolic condition value):

```
EXTERNAL ROUTINE PROC;
IF PROC (A,B,C)
THEN
     success
ELSE
     failure
```

- **BASIC**

  The following example shows a simple way to return success or failure from a BASIC procedure (see Section 5.2.2 for the preferred way, using condition value symbols that indicate the specific reason for an error):

```
100 FUNCTION INTEGER PROC(X,Y,Z)
     .
     .
     .
200 IF (...) THEN
          PROC = 1%
     ELSE
          PROC = 0%
300 FUNCTIONEND
```

  This example shows how a BASIC calling program can check for a success or failure status (whether the called procedure returns 0 or 1 or uses a symbolic condition value):

```
100 EXTERNAL INTEGER FUNCTION PROC
200 IF (PROC (A,B,C) and 1%) THEN
          success
     ELSE
          failure
```

- **FORTRAN**

  The following example shows a simple way to return success or failure from a FORTRAN procedure (see Section 5.2.2 for the preferred way, using condition value symbols that indicate the specific reason for an error):

```
FUNCTION PROC (X,Y,Z)
INTEGER*4 PROC
     .
     .
     .
```

(continued on next page)

```
IF (...) THEN
    PROC = 1
ELSE
    PROC = 0
ENDIF
RETURN
END
```

This example shows how a FORTRAN calling program can check for a success or failure status (whether the called procedure returns 0 or 1 or uses a symbolic condition value):

```
EXTERNAL PROC
INTEGER*4 PROC
    .
    .
    .
IF (PROC (A,B,C)) THEN
    success
ELSE
    failure
ENDIF
```

- **PASCAL**

  The following example shows a simple way to return success or failure from a PASCAL procedure:

```
FUNCTION PROC (X:..., Y:..., Z:...):INTEGER;
.
.
.
IF (...) THEN
     PROC = 1
ELSE
     PROC = 0
END;
RETURN;
END;
```

  This example shows how a PASCAL calling program can check for success or failure status (whether the called procedure returns 0 or 1 or uses a symbolic condition value):

```
EXTERNAL PROC (M:..., N:..., O:...):INTEGER;
IF ODD PROC (A, B, C) THEN success;
```

5.2.2 Condition Values

The format of the condition value is:
where:

condition identification (STS$V_COND_ID)
  Identifies the condition uniquely on a system-wide basis.

facility (STS$V_FAC_NO)
  Identifies the software component generating the condition value. Bit 27 is set for customer facilities and clear for DIGITAL facilities.

message number (STS$V_MSG_NO)
  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 and hence reserved to DIGITAL.

severity (STS$V_SEVERITY)
  Indicates the severity code: bit 0 is set for success (logical true) and is clear for failure (logical false); bits 1 and 2 distinguish degrees of success or failure. Taken together the bits 0 through 2 define the severity of the error as follows:

STS$K_WARNING	0 = warning
STS$K_SUCCESS	1 = success
STS$K_ERROR	2 = error
STS$K_INF	3 = information
STS$K_SEVERE	4 = severe-error

cntrl
  Four control bits. Bit 28 is set to inhibit printing the message associated with the condition value by the $EXIT system service. It should be set in the condition value returned by a procedure as a function value if the procedure has also signaled the condition. Bits 29 thru 31 must be zero; they are reserved for DIGITAL.

A list of facility numbers and codes is found in Appendix B of this manual. To distinguish your condition values from those used by DIGITAL, you should set bits 15 (STS$V_FAC_SP) 27 (STS$V_CUST_DEF) to 1.

5.2.3 Defining Condition Value Symbols

To make condition value symbols available to calling programs in a convenient manner, you should assign a unique global symbol to each distinct error your procedure detects. The global symbols should have the form:

fac$_error-name (DIGITAL-supplied)
fac__error-name (User-supplied)

You can also define success condition values in order to indicate various forms of success. For example, the system service $SETEF (Set Event Flag) returns SS$_WASCLR or SS$_WASSET to indicate whether the event flag was previously clear or set.

If you place your procedures in a user-created or DIGITAL-supplied library, you can include the global symbol definitions there as well so they are available to any module making an external declaration.

To uniquely define condition value symbols so neither the name nor the value can duplicate those defined by another user or by DIGITAL, you must:

1. Choose an existing facility name or create one. If you create one, you must register it with someone at your installation responsible for the uniqueness of such symbols. This person will assign a unique 12-bit number to be used in the STS$V__FAC__NO field (bit 27 must be set for non-DIGITAL facilities).
2. Place all symbol definitions for a given facility in a single source file.
3. Define values for each symbol such that each value is unique in bits 14 through 3.
4. Make sure that bits 27 and 15 are set to prevent conflict with DIGITAL-supplied software.
5. Set bits 27 through 16 to the correct facility number.

The following examples describe how to define these global condition value symbols:

LIB____NOSUCHFILE	– no such file
LIB____NOSUCHDEV	– no such device
LIB____NOSUCHDIR	– no such directory

- **MACRO**

  Assume the LIB facility has facility number 24, which is placed in a field ending at bit 16. Bits 27 and 15 are set to 1.

```
LIB__FAC = <24@16> + <1@27> + <1@15> ; define facility
SEVERE = 4                           ; severity = severe
LIB__NOSUCHFILE ==  LIB__FAC + SEVERE + 1@3
LIB__NOSUCHDEV  ==  LIB__FAC + SEVERE + 2@3
LIB__NOSUCHDIR  ==  LIB__FAC + SEVERE + 3@3
```

- **BLISS**

```
GLOBAL LITERAL
        LIB__FAC = 24^16 + 1^27 + 1^15,
        SEVERE = 4
        LIB__NOSUCHFILE = LIB__FAC + SEVERE + 1^3,
        LIB__NOSUCHDEV  = LIB__FAC + SEVERE + 2^3,
        LIB__NOSUCHDIR  = LIB__FAC + SEVERE + 3^3;
```

- **BASIC**

  Global symbols can be defined in MACRO for use by a BASIC program or procedure.

- **FORTRAN**

  Global symbols can be defined in MACRO for use by a FORTRAN program or procedure.

- **PASCAL**

  Global symbols can be defined in MACRO for use by a PASCAL program or procedure.

5.2.4 Using Global Condition Values in a Calling Program

A calling program can identify a condition value returned by a procedure and take action for each specific value returned. When identifying a condition value, the calling program should ignore bits 31 through 28 and bits 2 through 0 since they are supplemental to the identification of the error. In some cases, the condition value may have been signaled before being returned as a function value. Therefore, these bits may differ from the values defined symbolically. If the facility-specific bit (bit 15) is 0, then the facility number field (bits 27 through 16) should also be ignored.

The library procedure LIB$MATCH__COND uses this algorithm for matching condition values. This procedure is described in Chapter 6 of the *VAX-11 Run-Time Library Reference Manual.*

The format for LIB$MATCH__COND is:

index = LIB$MATCH__COND (condition-value, cond-value-i ...)

condition-value
: Address of longword containing the condition value to be matched.

cond-value-i
: Address of longword containing the condition value to be compared with condition-value.

index
: 0, if no match is found; i for a match between the first and (i+1)st parameter.

The following sections show examples that use the condition values previously described in a program to branch to different instructions on each different condition value. Examples are in MACRO, BLISS, BASIC, FORTRAN and PASCAL both with (preferred) and without the use of LIB$MATCH__COND.

- **MACRO**

```
.EXTRN    LIB_PROC, LIB__NOSUCHFIL, LIB__NOSUCHDEV,
          LIB__NOSUCHDIR
CALLG     ARGLST, LIB_PROC
BLBS      R0, 10$             ; Branch if success
CMPL      R0, #LIB__NOSUCHFIL
BEQL      20$                 ; Branch if no such file
CMPL      R0, #LIB__NOSUCHDEV
BEQL      30$                 ; Branch if no such device
CMPL      R0, #LIB__NOSUCHDIR
BEQL      40$                 ; Branch if no such directory
                              ; Here if any other error
```

By using LIB$MATCH_COND, the preceding example changes to this:

```
 .EXTRN       LIB_PROC, LIB$MATCH_COND, LIB__NOSUCHFIL,
              LIB__NOSUCHDEV, LIB__NOSUCHDIR
 CALLG        ARGLST, LIB_PROC
 BLBS         R0, 10$            ; Branch if success
 PUSHAL       #LIB__NOSUCHDIR
 PUSHAL       #LIB__NOSUCHDEV
 PUSHAL       #LIB__NOSUCHFIL
 PUSHL        R0
 CALLS        #4, LIB$MATCH_COND

15$:          CASEB   R0, #1, #3
              20$-15$            ; No such file
              30$-15$            ; No such device
              40$-40$            ; No such directory
                                 ; Here if any other error
```

- **BLISS**

  The following BLISS example also branches upon identification of a particular condition value:

```
EXTERNAL ROUTINE LIB_PROC: ADDRESSING_MODE (GENERAL):
EXTERNAL LITERAL LIB__NOSUCHFIL, LIB__NOSUCHDEV,
 LIB__NOSUCHDIR:

ROUTINE ...
     BEGIN
     LOCAL COND_VAL;
     COND_VAL = LIB_PROC (...)
     IF NOT .COND_VAL
     THEN
        SELECTONE    .COND_VAL OF
        SET
            [LIB__NOSUCHFIL]: ... ;
            [LIB__NOSUCHDEV]: ... ;
            [LIB__NOSUCHDIR]: ... ;
            [OTHERWISE]:        ... ;
            TES;
```

  By using LIB$MATCH_COND, the preceding example changes to this:

```
EXTERNAL ROUTINE
   LIB_PROC: ADDRESSING_MODE (GENERAL),
   LIB$MATCH_COND:  ADDRESSING_MODE (GENERAL);
EXTERNAL LITERAL
   LIB__NOSUCHFIL, LIB__NOSUCHDEV, LIB__NOSUCHDIR;


ROUTINE ...
   BEGIN
   LOCAL COND_VAL;
   COND_VAL = LIB_PROC (...)
   IF NOT .COND_VAL
   THEN
```

```
CASE LIB$MATCH_COND (.COND_VAL,
          %REF(LIB__NOSUCHFIL),
          %REF(LIB__NOSUCHDEV),
          %REF(LIB__NOSUCHDIR)) FROM 1 TO 3 OF
   SET
   [1]: ... ;
   [2]: ... ;
   [3]: ... ;
   [OUTRANGE]: ... ;
   TES;
```

- **BASIC**

  The following example illustrates using condition values to branch to several different statements in BASIC:

```
100   EXTERNAL INTEGER FUNCTION LIB_PROC
200   EXTERNAL INTEGER CONSTANT LIB__NOSUCHFIL, LIB__NOSUCHDEV,
           LIB__NOSUCHDIR
300   DECLARE INTEGER COND_VAL
400   COND_VAL = LIB_PROC (...)
500   IF (COND_VAL AND 1%) <> 1% THEN
           IF COND_VAL = LIB__NOSUCHFIL THEN
                ...
           ELSE IF COND_VAL = LIB__NOSUCHDIR THEN
                ...
           ELSE IF COND_VAL = LIB__NOSUCHDEV THEN
                ...
           ELSE
                ...
600   ...
```

  This example does the same thing in BASIC using LIB$MATCH_COND:

```
100   EXTERNAL INTEGER FUNCTION LIB_PROC, LIB$MATCH_COND
200   EXTERNAL INTEGER LIB__NOSUCHFIL, LIB__NOSUCHDEV,
           LIB__NOSUCHDIR
300   DECLARE INTEGER COND_VAL
400   COND_VAL = LIB.PROC(...)
500   IF (CON_VAL AND 1%) <> 1% THEN
           ON LIB$MATCH_COND(LIB__NOSUCHFFIL, LIB__NOSUCHDEV,
               LIB__NOSUCHDIR)
           GOTO 1000, 2000, 3000
1000 ...
2000 ...
3000 ...
```

- **FORTRAN**

  The following example illustrates using condition values to branch to several different statements in FORTRAN:

```
 EXTERNAL LIB_PROC, LIB__NOSUCHFIL, LIB__NOSUCHDEV,
 1LIB__NOSUCHDIR
 INTEGER*4 LIB_PROC, COND_VAL
  .
  .
```

(continued on next page)

```
 COND_VAL = LIB_PROC (...)
 IF (.NOT. COND_VAL) THEN
   IF (COND_VAL .EQ. %LOC(LIB__NOSUCHFIL)) THEN
     ...
   ELSE IF (COND_VAL .EQ. %LOC(LIB__NOSUCHDEV)) THEN
     ...
   ELSE IF (COND_VAL .EQ. %LOC(LIB__NOSUCHDIR)) THEN
     ...
   ELSE
     ...
   ENDIF
```

The following example does the same thing in FORTRAN using LIB$MATCH_COND:

```
 EXTERNAL LIB_PROC, LIB$MATCH_COND
 EXTERNAL LIB__NOSUCHFIL, LIB__SUCHDEV, LIB__NOSUCHDIR
 INTEGER*4 LIB_PROC, LIB$MATCH_COND, COND_VAL
 .
 .
 .
 COND_VAL = LIB_PROC (...)
 IF (.NOT. COND_VAL)
1GOTO 20,30,40 LIB$MATCH_COND(COND_VAL,
2LIB__NOSUCHFIL, LIB__NOSUCHDEV, LIB__NOSUCHDIR)
```

- **PASCAL**

The following example illustrates using condition values to branch to several different statements in PASCAL:

```
CONST
     %INCLUDE 'SYS$LIBRARY : SIGDEF.PAS'
CASE LIB_PROC ( ... ) OF
     LIB__NOSUCHFIL : ...    ;    { No such file}
     LIB__NOSUCHDEV : ...   ;     { No such device}
     LIB__NOSUCHDIR : ...         { No such directory}
     OTHERWISE : ...              { No match}
END;
```

The following example does the same thing in PASCAL using LIB$MATCH_COND:

```
FUNCTION LIB$MATCH_COND(A,B,C,D : INTEGER) : INTEGER;
     EXTERN;
CONST
     %INCLUDE 'SYS$LIBRARY : SIGDEF.PAS'
COND_VAL := LIB_PROC ( ... )
CASE LIB$MATCH_COND (COND_VAL, LIB__NOSUCHFIL,
     LIB__NOSUCHDEV, LIB__NOSUCHDIR) OF
     0 : ... ;                      { No match}
     1 : ... ;                      { No such file}
     2 : ... ;                      { No such device}
     3 : ...                        { No such directory}
     END;
```

5.3 Signaling Error Conditions

5.3.1 Signal Exception Condition

LIB$SIGNAL is called whenever it is necessary to indicate an exception condition or display a message rather than return a status code to the calling program. LIB$SIGNAL scans the stack frame-by-frame starting with the most recent frame calling each established handler.

The format is:

```
CALL LIB$SIGNAL (condition-value [,parameters...])
```

condition-value
: A standard signal name designating a VAX–11 system-wide 32–bit condition value (passed by immediate value).

parameters
: Optional additional FAO (formatted ASCII output) parameters for message (passed by immediate value).See Chapter 7 of the *VAX–11 Run-Time Library Reference Manual* or the description of $PUTMSG system service in the *VMS System Service Reference Manual* for the interpretations of the FAO parameters.

5.3.2 Stop Execution via Signaling

LIB$STOP is called whenever it is necessary to indicate an exception condition or display a message when it is impossible to continue execution or return a status code to the calling program. LIB$STOP scans the stack frame-by-frame, starting with the most recent frame calling each established handler. LIB$STOP guarantees that control does not return to the caller. The format is:

```
CALL LIB$STOP (condition-value [,parameters...])
```

The LIB$STOP parameters are identical to those described in Section 5.3.1 for LIB$SIGNAL.

The *VAX–11 Run-Time Library Reference Manual* discusses LIB$SIGNAL and LIB$STOP in more detail.

5.4 Internal Signaling

Because you can choose to organize procedures in levels of abstraction (see Section 4.1), some procedures might not be available to the calling program across the modular interface. You could use internal signaling between procedures that are at different levels.

To use internal signaling, the procedures that can be called across the modular interface must establish a condition handler. Whenever any of your procedures detect an error, they might call a central error-signaling procedure and pass the error number as a parameter to be used in a 32–bit condition value (bits 14 through 3). This error-signaling procedure would convert the error number to a 32–bit condition value by:

- Shifting the error number left by 3 bits
- Inserting a severity code (usually severe = 4)
- Setting the facility number field (bits 26 to 16)
- Setting bits 27 and 15

The error-signaling procedure then adds any extra arguments and signals the error by calling LIB$SIGNAL or LIB$STOP. For example, the FORTRAN support library procedure FOR$$SIGNAL adds the current logical unit number and file name to the argument list, followed by the VAX–11 RMS condition value and status value from the current FAB or RAB. The $$ indicates that FOR$$SIGNAL is an internal interface and is not part of the interface to user programs.

Your specific condition handler is then entered. It can decide how to proceed from this point. Usually, it unwinds to the caller of the establisher (which is the program calling across the modular interface) of the handler. The signaled condition value is the value returned to the calling program that called the establisher (outermost layer). You can make LIB$SIG_TO_RET the specific error condition handler. LIB$SIG_TO_RET can also be called from your handler.

For example, the condition handler established by the FORTRAN support procedures inserts the program counter (PC) of the calling program into the signal argument list and either: (1) resignals or (2) unwinds to the ERR= address if ERR= is specified by the calling program as an optional argument.

The internal signaling procedure FOR$$SIGNAL does not know the PC of the calling program. However, it is easy for the handler to find it, since the handler is passed the address of the stack frame of the establisher (which contains the PC of the calling program).

5.5 Creating a Procedure Activation Environment

You can use the VAX/VMS error-signaling mechanism to create a special per-procedure activation environment. This is needed to implement most higher-level languages. In such cases, the compiled code for each procedure activation establishes a language-specific condition handler. The address of the handler (stored in longword zero of the stack frame) can also serve as a way to identify in which language the procedure was written. This is useful for language support procedures that need to know the layout of the stack frame.

Such a handler takes appropriate language-specific action on software errors signaled by mathematics (MTH), string (STR) or language support (BLI, BAS, FOR, PAS) procedures, or by hardware errors. By using such a per-activation mechanism, procedures of different languages can call one another, each with its own environment.

Note that the main program is also a procedure and follows the same per-procedure activation technique. Furthermore, the code generated by the main program must not call a language initialization routine, since the main program might call procedures written in any language. Alternately, a subroutine written in a particular language cannot depend on the main program being written in the same language. Hence, the subroutine cannot depend on a particular main program initialization code having been called.

Chapter 6
Coding Modular AST-Reentrant Procedures

This chapter describes coding techniques for modular procedures that use the VAX/VMS AST (asynchronous system trap) interrupt mechanism themselves, or permit calling programs to use it. A procedure is AST-reentrant if it:

- Can be interrupted between any two instructions, permitting it or any related procedure to be called (reentered)
- Executes correctly when continued

This chapter describes:

- How to code AST-reentrant procedures
- How to code I/O that may or may not be at the AST level

All modular procedures should be AST-reentrant so they can be called from any program. If your procedure is not AST-reentrant or calls any procedure that is not, your program documentation should relate this to warn others against using your procedure.

NOTE

Do not confuse the term AST-reentrant with reentrant, which refers to a more restrictive set of conditions encountered when static storage is shared between processes. In such a situation, there can be more than two threads of concurrent execution, and each thread can alternately progress toward an end. The restrictions become even more severe if the processes can be executing simultaneously on several processors.

Since most modular procedures share code (and not data) between processes, not all of the techniques described in this chapter are applicable to reentrant procedures on single or multiprocessor configurations that share data between processes. All of the techniques in this chapter assume that data is statically allocated per-process.

6.1 AST Interrupts in a Process

Some VAX/VMS system services let an event interrupt a process. Since the interrupt occurs out of sequence with respect to process execution, the interrupt mechanism is called an asynchronous system trap (AST). An AST interrupt transfers control to a user-specified routine that services the event. The AST routine can call other procedures, including library procedures. The AST routine and any procedures it calls are said to be executing at AST level.

While at AST level, a process cannot be re-interrupted at the same access mode. The process runs to completion at the AST level before the non-AST level procedure resumes and can execute another instruction. Hence, a process is either executing at AST level or at non-AST level, and thus consists of two "threads of execution," one thread at each level.

NOTE

> The AST level cannot stall or use "busy wait" to avoid being called before the non-AST level is out of a critical section of code.

When the AST routine finishes servicing the event, it returns control to its caller (VMS). This automatically continues execution of the interrupted procedure at the point of interruption.

For example, you could call the Set Timer system service ($SETIMR) that would specify the address of an AST level procedure to be executed when a specified time elapses. When the requested time occurs, the system 'delivers' an AST interrupt by stopping the currently executing procedure and calling the specified AST routine. Another example of an AST event is typing CTRL/C on the terminal.

For information on the implementation of AST interrupts by system services, see the *VAX/VMS System Service Reference Manual.*

6.1.1 Using AST Routines

To use AST interrupts, you must write an AST routine to take control at AST level. An AST routine must follow these guidelines:

- It must be separate from the currently executing procedure.
- It must not modify data or instructions used by the interrupted procedure or its callers.
- It is called with a CALLG instruction.
- If it modifies any registers other than R0 and R1, it must set bits in the entry mask to save the contents of the registers.
- If it calls any other procedures, they must all be AST-reentrant.
- It must return to its caller with a RET instruction.

6.1.2 Interrupting a Non-AST Reentrant Procedure

If an AST interrupt occurs during the execution of a non-AST reentrant procedure, you can get unpredictable results from either the AST level procedure or the interrupted procedure.

BASIC procedures can be made AST-reentrant since local variables are allocated on the stack. Avoid use of static storage by not using COMMON (COM) and built-in functions that have static storage. These include STATUS, CTRLC, RCTRLC, DET, FIELD, NUM, NUM2, ON ERROR GO TO, RESUME, and RECOUNT. You should also avoid the SYS functions ASSIGN, DEASSIGN, message send and receive.

A non-trivial FORTRAN procedure cannot be made AST-reentrant. Hence, FORTRAN procedures can be called only at the AST level or the non-AST level, but not both.

PASCAL procedures can be made AST-reentrant since local variables are allocated on the stack. Avoid use of static storage by not declaring variables at the program or module level.

6.2 Writing AST-Reentrant Modular Procedures

You must observe the following rules when writing AST-reentrant procedures:

- Only AST-reentrant procedures can be called at both the AST and the non-AST level. Since an AST interrupt can arrive at any time, AST-reentrant procedures must be written so that an AST interrupt can occur between any two instructions without interfering with the correct operation at either the AST or non-AST levels. If a single instruction is interruptable, an AST interrupt can also occur within that instruction. (For more information, see the *VAX-11 Architecture Handbook.*)
- An AST-reentrant procedure cannot call any non-AST reentrant procedures.
- If both an AST level and a non-AST level procedure concurrently access a data base in static storage, each procedure must make sure that race condition interference does not occur. (See Section 6.3.)

NOTE

The term race condition refers to a situation where two independently executing threads of execution can access the same data in a conflicting manner. For example, a race condition exists if a single instance of a process-wide resource can be allocated to different procedures at both the AST and non-AST level.

- If I/O at the AST level is performed, you should avoid simultaneous I/O of the same data base from both AST and non-AST level procedures. (See Section 6.4.)

A procedure that has no static storage and calls no other procedures is automatically AST-reentrant. A procedure that has no static storage and only calls other AST-reentrant procedures is also AST-reentrant. Hence, AST-reentrant procedures should use stack and/or heap storage.

A procedure with static storage can be AST-reentrant, although this is difficult to program since you could be changing statically allocated data when the interrupt occurs.

6.3 Eliminating Race Conditions During Concurrent Access

There are a number of ways for your procedure to eliminate race condition interference when accessing and modifying data in its static storage:

- Perform all accessing or modification in a single uninterruptable instruction.
- Detect concurrency of data base access using "test and set" instructions at data base entry and exit.
- Keep a call-in-progress count that is incremented when your procedure is called and decremented when it returns. The count is used as an index into separate allocated areas.
- Disable AST interrupts upon entry and restore the enable state on exit.

The following sections describe these methods.

6.3.1 Performing all Accesses in One Instruction

For some applications, all data modification in static storage can be performed in a single uninterruptable instruction. For example, you can use queue instructions at the beginning and end of your procedure to control resource allocation.

The remove queue instruction removes a control block (containing an instance of a process-wide resource) from the free list of available resources, making the resource available to the program. The insert queue instruction places the control block back in the free list when the program no longer needs the resource.

The queue headers are allocated in static storage. The control blocks themselves can be in static storage (if a specific number of resources are needed) or in dynamic heap storage (if a variable number of resources are needed).

For example, STR$COPY allocates and deallocates string space in heap storage. A fixed number of queue headers are allocated in static storage — one queue for each string length.

The following example illustrates an AST-reentrant procedure that uses queue instructions to control allocation of quadword blocks:

```
        .PSECT   _LIB_DATA PIC,USR,CON,REL,LCL,NOSHR,NOEXE,RD,WRT
FLAG:   .LONG    0                    ; First-time flag
Q_HED   .LONG    0,0
        .PSECT   _LIB_CODE PIC,USR,CON,REL,LCL,SHR,EXE,RD,NOWRT
        .ENTRY   LIB_GET_X,^M<>
        BBC      FLAG, FIRST          ; Branch on 1st call only
TRY:    REMQUE   @Q_HED, R0           ; R0 = address of queue
        BVS      10$                  ; Branch if empty and fill
        RET

10$:    BSBB     FILL                 ; Fill queues
        BRB      TRY                  ; Try again
;+
; Here on first call only
;-
FIRST:  $SETAST  #0                   ; Disable ASTs, R0=old setting
        BBSS     FLAG, 20$            ; Branch if already set
        MOVAL    Q_HED, Q_HED         ; Make queue empty
        MOVAL    Q_HED, Q_HED+4       ; Back pointer too
        BSBB     FILL                 ; Fill queues
20$:    CMPL     #SS$_WASSET, R0      ; Were ASTs enabled before?
        BNEQ     TRY                  ; No, leave disabled, retry
        $SETAST  #1                   ; Yes, enable ASTs
        BRB      TRY                  ; Try again

FILL:   get space for 10 quadwords by calling LIB$GET_VM
        and insert in queue using INSQUE
        RSB
```

NOTE

The above example could be recoded using REMQHI and INSQHI and avoid the need to have a first time flag. This is because an empty queue is represented as zero entries for the interlocked, self-relative, queue instructions.

In some applications, the static storage is divided into two or more sections; each section in a queue.

You can use a single queue instruction at the beginning of your procedure to remove one section, and another can be used at the end to insert the section back in the queue.

While a section is removed from the queue, your procedure can modify data in it. If an AST-interrupt occurs while the section is removed, a different section of data is used instead, thus avoiding conflicts with the interrupted procedure.

6.3.2 Using "Test and Set" Instructions

To detect concurrent access of static storage at both AST and non-AST levels, you should add the following steps to your procedures:

- Place a branch on bit set and then set instruction BBSS immediately before each of your procedures accesses static storage.

- Access and/or modify static storage.
- Place a branch on bit clear and then clear instruction BBCC immediately after each of your procedures has completed access to static storage.

The BBSS instruction detects that concurrency is about to take place before static storage has been accessed. There are two alternate techniques for resolving concurrency conflicts detected by the BBSS and BBCC instructions:

- Use separate, statically allocated areas at the AST and non-AST levels. When the BBSS instruction detects concurrency at the beginning, use the second allocated area. Note that this technique does not work if an exception condition occurs between execution of the BBSS instruction and the BBCC instruction or if your procedure has not established a condition handler. This is because a condition handler established by the calling program might also simultaneously call your procedure.
- Reexecute your procedure if concurrency is detected at the end. When the BBCC instruction detects this concurrency, branch back to the beginning of your procedure and try again.

The following example illustrates the latter technique. This MACRO procedure, LIB_GET_INUM, allocates and deallocates identifying numbers:

```
.TITLE    LIB_GET_INUM  - Allocate and deallocate id. nos. 1 - 10
TAB:      .WORD    0                 ; Bitmap for flags
          .ENTRY   LIB_GET_INUM, ^M<>
10$:      FFC      #1, #10,TAB, R0   ; Find first free id, no.
          BEQ      20$               ; Branch if none free
          BBSS     R0, TAB, 10$      ; Indicate id.  no. in use
          MOVL     R0, @4(AP)        ; Return id. no. found
          MOVL     #1, R0            ; Indicate success
          RET
20$:      CLRL     @4(AP)            ; Return 0
          CLRL     R0                ; Indicate failure
          RET
          .END
```

6.3.3 Keeping a Call-in-Progress Count

If the data base is to be kept separate between each call, you can keep track of when your procedure is called by using a call-in-progress count. Before data base access, the count is incremented and used to index into a table of addresses for the separate data bases. You should check for a count that exceeds the table length. After the data base has been accessed, the count is decremented.

This technique has an advantage over the BBxx technique because it can handle more than two levels of reentrance. However, it is less reliable, since an exception can cause the count never to be decremented, leading to an eventual procedure malfunction. You can avoid this by establishing a condition handler in your procedure.

6.3.4 Disabling AST Interrupts

Sometimes the only way to avoid race conditions is to disable AST interrupts during the access to static storage and restore the state of the AST enable at the end. However, this technique could adversely affect performance of real-time programs using AST interrupts. Therefore, you should avoid it whenever you can use any technique described in Sections 6.3.1 to 6.3.3.

Try to minimize the number of instructions during which the AST interrupts are disabled. Before disabling AST interrupts, establish a condition handler to restore the AST level in case an exception or stack unwind occurs.

The following BLISS example disables ASTs and then restores the state of the enable before returning to its caller:

```
GLOBAL
     STR$$V_INIT : INITIAL (0)


     BEGIN
     LOCAL
          AST_STATUS;
!+
! Disable ASTs so we can test STR$$V_INIT. We must not permit an
! AST after we start to initialize the queues.
!-
     AST_STATUS = $SETAST(ENBFLG = 0)
     IF (NOT .STR$$V_INIT)
     THEN
          BEGIN
!+
! We must do the initialization.
!-
          .
          .
          .
!+
! Mark the initialization as complete.
!-
          BLOCK [STR$$V_INIT,0,0,1,0] = 1;
          END;
!+
! If ASTs were enabled when we entered, re-enable them.
!-
     IF (.AST_STATUS EQL SS$_WASSET) THEN $SETAST (ENBFLG = 1);
     RETURN;
     END;
```

6.4 Performing I/O at the AST Level

If your procedure performs I/O using VAX-11 RMS system services, there are several coding techniques you must observe for your procedure to be AST-reentrant:

- When opening process permanent files such as SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, or SYS$ERROR, check for the VAX-11

RMS error status RMS$_ACT (Active) after each $CREATE or $OPEN service. This error indicates that a record operation had already started for the process permanent file. It does not occur for nonprocess permanent files, and the open service follows the constraints of shared access to the file that may have been imposed by a previous open service. If the error occurs, perform a $WAIT using the same file access block (FAB). When control returns to your procedure, try the $CREATE or $OPEN service again. Repeat this sequence until it succeeds.

- When performing record I/O to any type of file, check for the RMS error status RMS$_RSA (record stream active) after each $GET and $PUT service. This error indicates that a record operation had already been started for the file. If the error occurs, perform a $WAIT using the same record block (RAB). When control returns to your procedure, try the $GET or $PUT service again. Repeat this procedure until it succeeds.

  The BASIC and FORTRAN I/O support procedures use this technique to perform I/O at AST and non-AST level. The VAX/VMS Put Message system service ($PUTMSG) also uses this technique so an error message signaled at AST level will be output on SYS$OUTPUT even though the non-AST level is also calling $PUT.

- Avoid storing data in a record access block (RAB) that VAX-11 RMS could still be accessing. Your procedure can do this in two ways:

  - Allocating the RAB on the stack so AST and non-AST level have separate RABs.

  - Allocating the RAB in static storage along with a busy bit. The busy bit is tested and set using a BBSS instruction before the RAB is accessed. If the RAB is already busy, your procedure executes a $WAIT using that RAB.

For synchronous I/O (that is always completed before returning control to your procedure) you can use either of the techniques previously described. However, the first is more reliable, since it has no static storage and hence cannot behave erroneously if an exception were signaled.

For asynchronous I/O (when control is returned to your procedure before I/O is completed), you must use the second technique.

Chapter 7
Building Modular Procedure Libraries

Modular procedure libraries consist of compiled and assembled object code associated with a calling program at link time. References to procedures in these libraries are resolved when the linker searches the user libraries specified in the LINK command or the default system libraries. The program can then call library procedures at run time.

You can create a modular procedure library by following the guidelines of this chapter. You can place procedures in either an object module library or a sharable image. Before starting, make sure the modular procedures conform to the rules listed in Appendix A.

7.1 Building the Default System Object Library

You can also place procedures in the default system object library STARLET.OLB. However, you must have the privileges of a system manager to do this.

7.1.1 Adding to the System Default Object Library

With the privileges of the system manager, you can use the following command to add procedures to STARLET.OLB. The general form is:

```
$ LIBRARY/REPLACE SYS$LIBRARY:STARLET file-spec[,...]
```

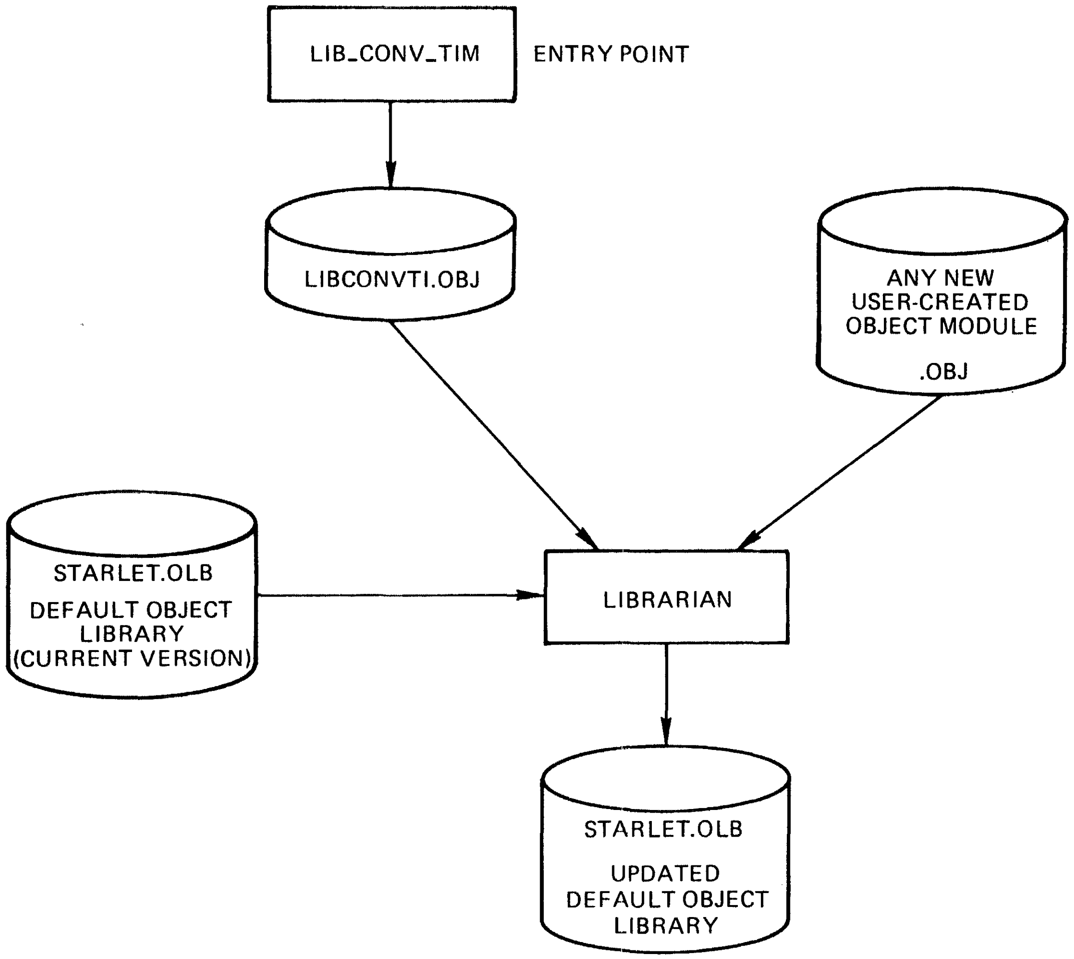
To use any of the LIBRARY command qualifiers with this command, see the *VAX/VMS Command Language User's Guide.*

Figure 7–1 shows the installation of a user-created procedure in STARLET.OLB. In this example, LIB_CONV_TIM is a sample procedure that converts system time to a specific format and is contained in a module LIBCONVTI.OBJ. The following command adds the module to the system default library STARLET.OLB:

```
$ LIBRARY/REPLACE SYS$LIBRARY:STARLET.OLB LIBCONVTI
```

After this command, the updated STARLET.OLB contains the new procedure LIB_CONV_TIM.

Figure 7-1: Adding a User-Created Procedure to the Default Object Library


LIB_CONV_TIM
ENTRY POINT
LIBCONVTI.OBJ
ANY NEW USER-CREATED OBJECT MODULE
.OBJ
STARLET.OLB
DEFAULT OBJECT LIBRARY (CURRENT VERSION)
LIBRARIAN
STARLET.OLB
UPDATED DEFAULT OBJECT LIBRARY

7.1.2 Accessing the Default System Object Library

Accessing procedures in STARLET.OLB requires no special LINK command. STARLET.OLB is automatically searched during any LINK command after the default system sharable image is searched, and if any unresolved strong references remain. If references are found in STARLET.OLB, the linker includes the modules containing the references in the executable image.

The linker can be instructed not to search the system default libraries by using the following qualifiers in the LINK command:

/NOSYSLIB – System will not search STARLET.OLB or VMSRTL.EXE.

/NOSYSSHR – System will not search the sharable subset of STARLET.OLB, VMSRTL.EXE.

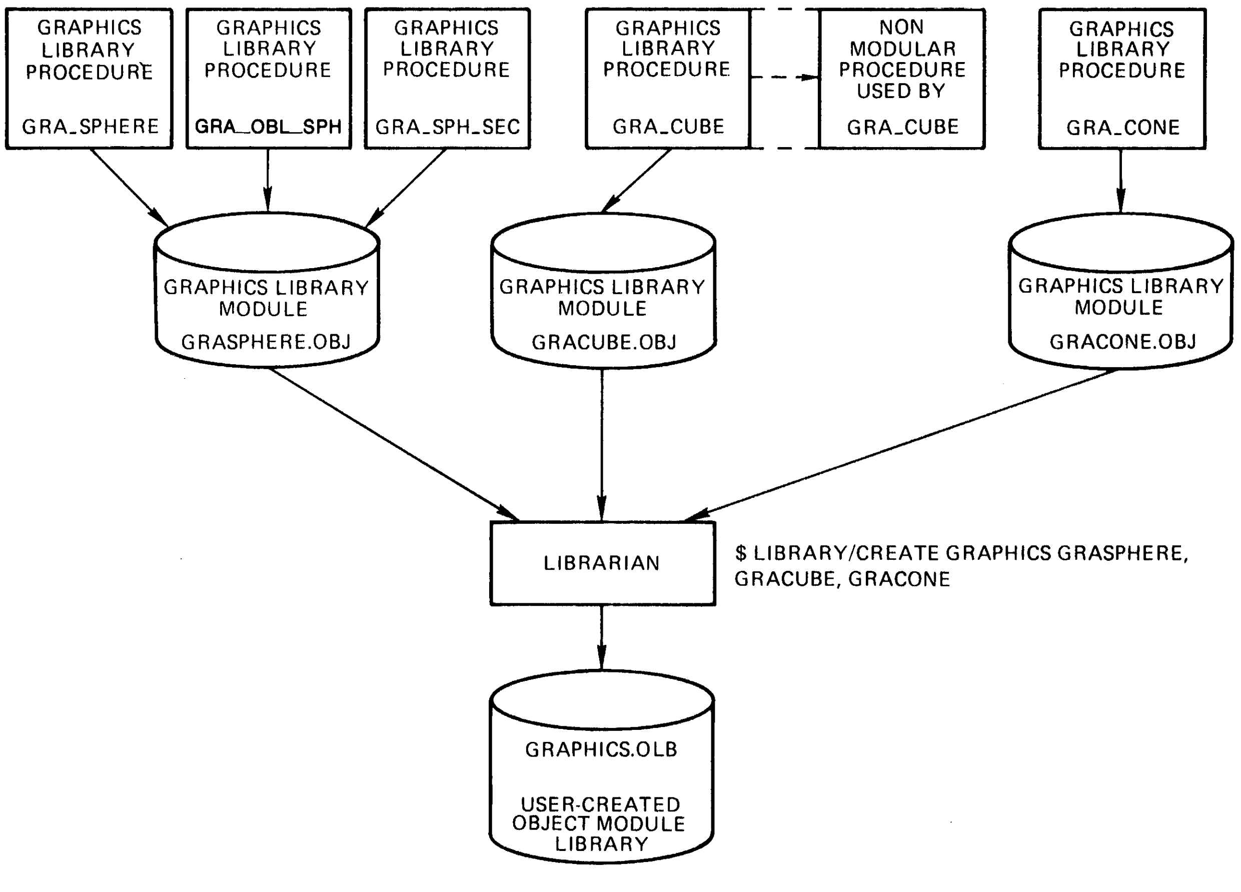
More detailed information on the LIBRARY and LINK commands is in the *VAX/VMS Command Language User's Guide.*

7.2 Building a User-Created Object Module Library

A user-created object module library consists of procedures you write in any VAX-supported programming language.

You can create an object library from object files using the LIBRARY command. Figure 7–2 shows the development of a theoretical user-created library of graphics procedures, called GRAPHICS.

Figure 7–2: Development of a User-Created Object Module Library


GRAPHICS LIBRARY PROCEDURE GRA_SPHERE
GRAPHICS LIBRARY PROCEDURE GRA_OBL_SPH
GRAPHICS LIBRARY PROCEDURE GRA_SPH_SEC
GRAPHICS LIBRARY PROCEDURE GRA_CUBE
NON MODULAR PROCEDURE USED BY GRA_CUBE
GRAPHICS LIBRARY PROCEDURE GRA_CONE
GRAPHICS LIBRARY MODULE GRASPHERE.OBJ
GRAPHICS LIBRARY MODULE GRACUBE.OBJ
GRAPHICS LIBRARY MODULE GRACONE.OBJ
LIBRARIAN
$ LIBRARY/CREATE GRAPHICS GRASPHERE, GRACUBE, GRACONE
GRAPHICS.OLB
USER-CREATED OBJECT MODULE LIBRARY

The library facility code used is GRA. The modular procedures envisioned produce mathematical representations of circles, cylinders, squares, and other geometric shapes. For example, the module GRASPHERE.OBJ might contain several related procedures that create spheres (GRA_SPHERE), oblate spheroids (GRA_OBL_SPH), and spherical sections (GRA_SPH_SEC) grouped because they share similar code. The module GRACUBE.OBJ could contain both a procedure that generates cube shapes and a nonmodular procedure that it calls. (Note, however, that the module GRACUBE.OBJ is still modular.)

The LIBRARY command for building a user-created object library has the following general form:

```
$ LIBRARY/CREATE library-name file-spec[,...]
```

The following example shows the creation of the user-created object library GRAPHICS.OLB that contains the modules GRASPHERE.OBJ and GRACUBE.OBJ. (.OBJ and .OLB are the default file types for object modules and object libraries, respectively, and are included here for clarity only.)

```
$ LIBRARY/CREATE GRAPHICS.OLB GRASPHERE.OBJ,GRACUBE.OBJ
```

After this command is given, GRAPHICS is ready to be linked with an application program.

7.2.1 Accessing a User-Created Object Library

You can include library modules in the calling program's executable image either implicitly or explicitly:

- The /LIBRARY qualifier causes the linker to search the library specified and implicitly includes modules containing definitions of symbols to which there are outstanding references.
- The /INCLUDE qualifier simplifies the linker's search since it instructs the linker to explicitly include a specified module in the image.
- Default User-defined libraries.

Any object library specified in an application program's LINK command is linked to that program if references to procedures in that library are encountered. A simple form of the LINK command is:

```
$ LINK application-program, user-created-library/LIBRARY
```

Any module in an object module library explicitly specified in an application program's LINK command is included in the executable image being created. A simple form is:

```
$LINK application-program, user-library/INCLUDE = (object-module, ...)
```

7.3 Building a User-Created Sharable Image

Placing procedures in a sharable image can reduce memory requirements and improve system performance if a number of application programs share the same set of procedures. However, the entire sharable image must be rebuilt any time a modification is made to it.

A sharable image can be built from either position-independent or nonposition-independent code.

The linker and image activator treat sharable images as follows:

1. The linker allocates position-dependent sharable images.
2. Position-independent sharable images are allocated and at higher addresses.

Therefore, the size of each sharable image section must remain identical from one update to the next unless it is the last sharable image in the P0 address space; in this case, it is free to grow.

7.3.1 Creating Sharable Images

- **BASIC**

  You can create a sharable image containing BASIC procedures. BASIC generates position-independent code, except for arrays in Common (COM) which are passed as parameters to other procedures or built-in functions. An array descriptor so generated is not position-independent.

  - If new versions of the image are larger than old versions, rebuilding sharable images requires relinking all programs bound with the old one.
  - If a user-created sharable image has VMSRTL.EXE linked to it at creation, and you specify that user image last, you can install a new version of VMSRTL.EXE without relinking.
  - Transfer vectors can be used with BASIC-sharable libraries if all images are approximately the same size and padded with extra space between them. Transfer vectors let a sharable image be relinked without relinking all programs that called the old version.

- **FORTRAN**

  You can create a sharable image containing FORTRAN procedures. However, because FORTRAN data PSECTS are not position-independent, FORTRAN procedures are also not position-independent. Therefore FORTRAN sharable images have the following restrictions:

  - To use multiple, sharable, nonposition-independent images, the address space must be manually assigned to each image.
  - If new versions of the image are larger than old versions, rebuilding sharable images requires relinking all programs bound with the old one.
  - If a user-created sharable image has VMSRTL.EXE linked to it at creation, and you specify that user image last, you can install a new version of VMSRTL.EXE without relinking.
  - Transfer vectors can be used with FORTRAN-sharable libraries if all images are approximately the same size and padded with extra space between them. Transfer vectors let a sharable image be relinked without relinking all programs that called the old version.

- **PASCAL**

  You can create a sharable image containing PASCAL procedures because PASCAL generates position-independent code. (See the *VAX-11 PASCAL Language Reference Manual.*)

7.3.2 Building and Installing a User-Created Sharable Image

Figure 7-3 shows the transformation of the user-created library GRAPHICS from an object module library to a sharable image. To do this, you must create a command procedure (MAKESHAR.COM in Figure 7-3) to build the sharable image. For example:

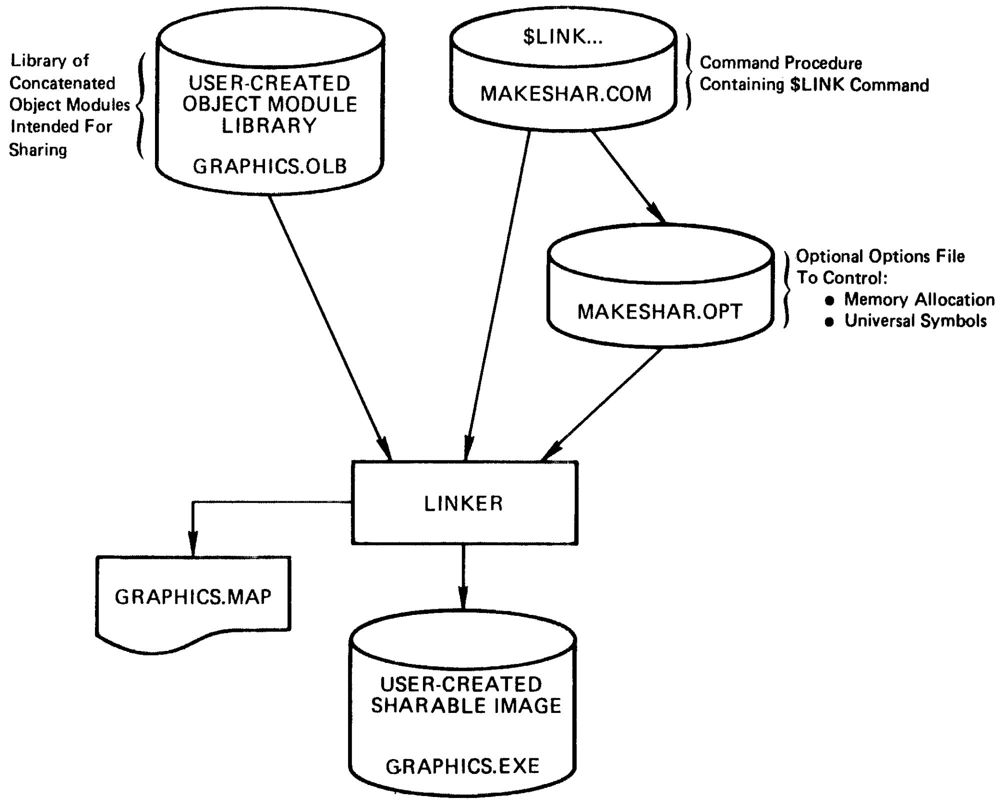
```
$ LINK/SHAREABLE=GRAPHICS/MAP/FULL-
GRAPHICS/INCLUDE=(GRA_SPHERE,...), MAKESHAR/OPTIONS
```

where:

- /SHAREABLE instructs the linker to build a sharable image called GRAPHICS.EXE.
- /MAP/FULL produces a detailed map of the image in (default) GRAPHICS.MAP.
- /INCLUDE specifies the list of objects to be taken from GRAPHICS.OLB for inclusion in this sharable image.
- MAKESHAR.OPT is an optional input file that provides additional information to the linker. Such information controls memory allocation and symbol tables. (See the *VAX-11 Linker Reference Manual.*)

You can optionally install your sharable image as a permanent global section. To do this, refer to the *VAX/VMS System Manager's Guide.*

Figure 7-3: Creating a Sharable Image



Library of Concatenated Object Modules Intended For Sharing
USER-CREATED OBJECT MODULE LIBRARY
GRAPHICS.OLB
$LINK...
MAKESHAR.COM
Command Procedure Containing $LINK Command
MAKESHAR.OPT
Optional Options File To Control:
● Memory Allocation
● Universal Symbols
LINKER
GRAPHICS.MAP
USER-CREATED SHARABLE IMAGE
GRAPHICS.EXE

7.3.3 Accessing a User-Created Sharable Image

You cannot run sharable images. They are incorporated in applications programs in a subsequent LINK operation. Figure 7-4 shows how an application program CARTOON.FOR might access the user-created sharable image GRAPHICS.EXE. To access a user-created sharable image, you must perform the following steps:

1. Create an OPTIONS file to specify the sharable image as an input to the linker. With this specification, it is also possible to request the linker to take a private copy of the content of the sharable image. Normally, during the linking of a sharable image into a user application program, the linker merely creates mapping information.

   In this example, the file NOWSHARE.OPT would contain:

   ```
   GRAPHICS.EXE/SHAREABLE[=COPY]
   ```

2. Link an application program with the user-created sharable image using the following command:

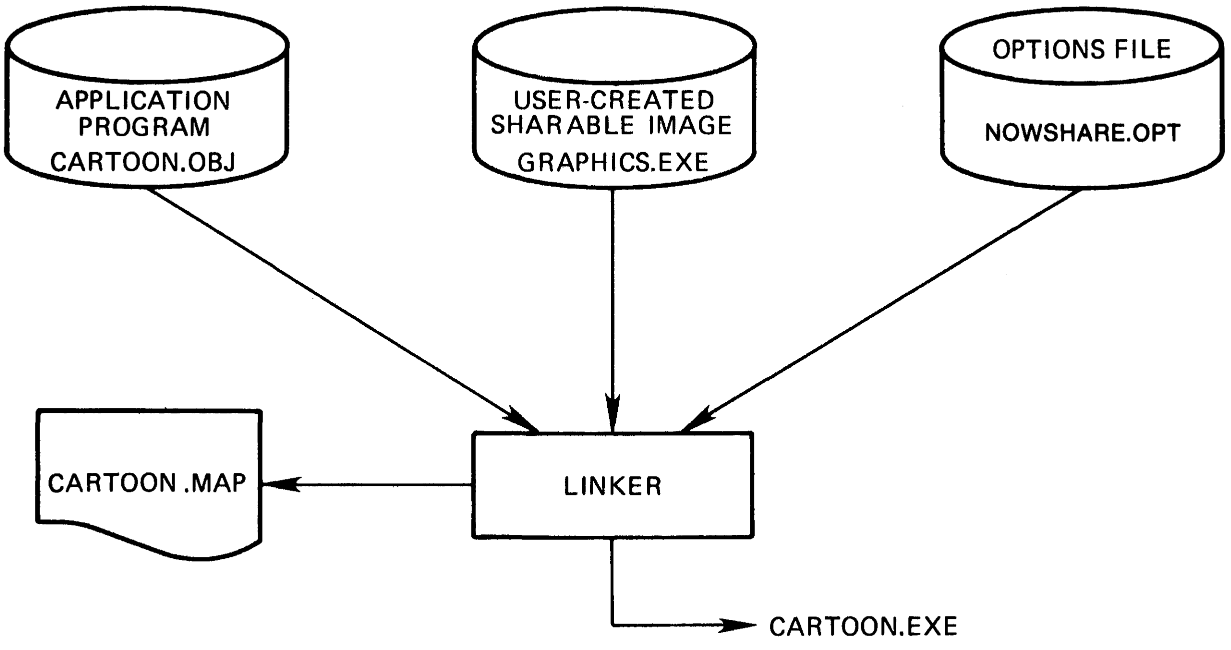
   $ LINK application-program, options-file/OPTIONS

   which in this example would be:

   ```
   $ LINK CARTOON.OBJ,NOWSHARE.OPT/OPTIONS
   ```

   This command produces CARTOON.EXE, the application program's executable image that can call GRAPHICS.EXE at runtime. (Note that .EXE, .OBJ, and .OPT are the default file types when /OPTIONS is used, and are included here for clarity only.)

Figure 7-4: Accessing a User-Created Sharable Image



7.4 Creating and Using Transfer Vectors

A transfer vector is a labeled virtual memory location that contains an address of, or a displacement to, a second location in virtual memory. This second location is the start of the instruction stream that is of actual interest. In the use of sharable images, transfer vectors are normally displacements rather than actual virtual addresses (for reasons of position independence). There are two reasons for doing this:

- Transfer vectors make it easy to modify and enhance the contents of the sharable image.
- Transfer vectors let you avoid relinking other programs bound to the sharable image.

7.4.1 Building Transfer Vectors

Transfer vectors must be written in MACRO; however, they can be used with procedures written in any language. The CALLS or CALLG instruction transfer vector has the form:

```
.TRANSFER   fac_symbol       ; Begin transfer vector to library
                             ; entry point, fac_symbol
.MASK       fac_symbol       ; Store register save mask
JMP         fac_symbol+2     ; Branch to routine at instruction
                             ; beyond the register save mask
```

The JSB instruction transfer vector has the form:

```
.TRANSFER fac_symbol::
JMP fac_symbol          ; Branch to JSB routine
```

In these examples, fac_symbol is the procedure's entry point name. For more information on how transfer vectors work, see the *VAX-11 Linker Reference Manual.*

7.4.2 Using Transfer Vectors

The linker automatically uses transfer vectors if they are present. Regardless of the procedures' languages, code the transfer vectors as shown in Section 7.4.1. Then assemble the program containing the transfer vectors. The resulting object module is used as input to the Linker. For example, the sharable image GRAPHICS, shown in Section 7.3.2, might be produced with the following command:

```
$ LINK/SHAREABLE=GRAPHICS/MAP/FULL TRANSVEC,-
GRAPHICS/INCLUDE=(...), MAKESHARE/OPTIONS
```

where TRANSVEC is the object module containing transfer vectors to all routines in the sharable image.

More detailed information about transfer vectors is in the *VAX-11 Linker Reference Manual.*

Appendix A
VAX–11 Modular Programming Standard

4 Jan 80 - Version 2.0

This appendix is the VAX–11 standard for writing modular procedures in any language, including MACRO and BLISS. This standard is the minimum necessary to interface your software at the callable procedure level with software written by others, and vice versa.

The standard contains required, optional, and recommended elements. Optional elements are indicated by asterisks (*). Non-conformance to optional elements must be indicated in the procedure's documentation. Recommendations are described in Section A–7. Non-conformance to recommendations need not be documented since modularity is not affected. Each element of the standard is described in greater detail in other sections of this manual, indicated by parenthesized references.

Most of this standard was derived by asking: "What general agreements are necessary between programmers to permit procedures to execute as expected when combined in arbitrary ways to form a program?"

This means that a procedure not following this standard could cause another modular procedure in the program image to execute incorrectly, or vice versa.

The arbitrary ways of combining procedures are:

- Your procedure calls other procedures.
- Other procedures call your procedure.
- A calling program calls either of the above.

Therefore, any modular procedure can be added to a collection of modular procedures without conflicting with them or those that might be added in the future.

A.1 Scope of Applicability

The required, optional, and recommended elements of this standard apply to library procedures and are recommended for other types of software, including utilities and application programs. Each programming language implemented on VAX lets you write your procedures to explicitly or implicitly follow the required elements of the standard for important language features. Therefore, compiler generated code for main programs and externally available subroutines and functions let you follow required elements of the standard. Furthermore, the language support procedures conform to the required, optional, and recommended elements, except as noted in Section A.3.

This standard applies to procedures that interface to a calling program; it does not apply to intra-module or inter-module calls that do not interface to the calling program as long as the entire set of procedures follows the standard.

A.2 Facility-Independent Part of the Standard

The following required and optional elements of the standard pertain to all facilities, whether in a library or not:

1. Calls to procedures follow the VAX-11 Procedure Calling Standard. (See Appendix C of the *VAX-11 Run-Time Library Reference Manual.*) Some elements of this standard restrict procedures to a subset of the VAX-11 Procedure Calling Standard to increase the ability for procedures to call one another.

2. A procedure does not accept data from or return data to the calling program using implicit overlaid PSECTs (COMMON in BASIC and FORTRAN, program or module level data in PASCAL) or implicit global data areas. Instead, all parameters accepted from or returned to the calling program use the argument list and function value registers (R0 and R0/R1). (See Section 2.4.1 and the VAX-11 Procedure Calling Standard.)

3. Modules must be relocatable. (See Section 4.2.1.)

4. Procedure entry point names contain at most 15 characters having the following forms: fac$name for DIGITAL-supplied procedures, and fac_name for user-supplied procedures, where fac can be LIB, MTH, OTS, STR, BLI, BAS, COB, FOR, PAS, or any other language abbreviation (and file type) or facility name.

   Global entry point names not intended for use by the calling program have two dollar signs ($$) or three underlines (___), respectively. The three underlines are needed to avoid conflict with user-defined condition value symbols that have two underlines. If alternate JSB entry points are provided, the name ends in _Rn (or just n if name would exceed 15 characters), where n indicates the highest register modified or used as an input parameter.

NOTE

> The limit of 15 characters will be increased to 31 after all VAX–supported language processors provide 31–character symbols and module names.

5. Generally each module contains a single procedure available to a calling program. This permits the greatest flexibility in linking procedures to form programs. Procedures can be grouped into a single module if they: (1) share the same static storage or (2) have a similar calling sequence, perform similar functions, and share a significant amount of code. (See Section 4.1.2.)

6. The form for module names is the same as that for procedure entry point names to avoid conflict when inserted into a library by the LIBRARY command. Modules containing one procedure have the same name as that procedure. Modules containing more than one procedure have a name formed from a combination or common subset of the entry point names. (See Section 4.2.2.)

7. Position-independent references (in a module) to writable data PSECTs use longword relative addressing. This is done so the VAX–11 Linker can correctly allocate the data PSECT anywhere with respect to the code PSECT no matter how many code modules are included. (See Section 4.2.3.)

8. External references use general-mode addressing so any of the referenced procedures can be put in a sharable image without requiring changes to the calling program.

9. A procedure does not print error or informational messages either directly or by calling the $PUTMSG system service. Instead, it either returns a condition value in R0 as a function value, or calls LIB$SIGNAL or LIB$STOP, directly or indirectly, to output all messages. (See Sections 5.2, 5.3, and A.3.)

10. If an error condition associated with a file is signaled, the expanded or resultant file name is included as one of the FAO arguments in the signal argument list. (See Section 5.3.)

11. If a procedure requires initialization once for each image activation, it is done without the caller's knowledge by: (1) initializing at compile time or (2) initializing at link time or (3) testing and setting a statically allocated first-time flag on each call or (4) adding a dispatch address to PSECT LIB$INITIALIZE. (See Section 4.3.)

    Using LIB$INITIALIZE is not recommended since your procedure cannot be placed in a sharable image. Furthermore, a procedure must not use LIB$INITIALIZE to establish a condition handler before the main program is called if its action might conflict with that of other condition handlers established before the main program.

12. If a procedure uses a process-wide resource, it calls the appropriate resource allocating library procedure or system service to allocate the resource to avoid conflict with allocations made to other procedures. To conserve resources, a procedure that requests resource allocation:

    - Calls the deallocation procedure before returning to the calling program, or
    - Remembers the allocation in static storage and calls the deallocation procedure later, or
    - Passes the responsibility for deallocation back to the calling program, or
    - Allocates a fixed number of the resources independent of the number of times it is called

    There are currently resource allocating and deallocating library procedures for:

    - Virtual memory in the program region
    - BASIC/FORTRAN logical unit numbers
    - Process-local event flags
    - Dynamic string memory

    (See Section 4.4 in this manual and Chapter 5 of the *VAX-11 Run-Time Library Reference Manual.)*

13. For each input and output string parameter (or string function value) the calling program either: (1) allocates a descriptor, or (2) passes along the address of a descriptor that had been passed to it. A procedure accesses a formal string parameter passed to it by:

    - Accessing the string's descriptor indirectly using the argument pointer (AP), or
    - Copying the address of the string descriptor, or
    - Copying the entire descriptor and changing the descriptor class code (in the copy only) to be fixed length (DSC$B_CLASS = 1), since there can only be one dynamic string descriptor per string (least preferred)

    NOTE

    The term "formal parameter" refers to the parameter's name as it is known to the called procedure, as opposed to either its actual value or its name as it is known to the calling program.

    The two semantics for writing formal string parameters are:

    - Fixed-length string semantics: The formal string is written with space filling or truncation on the right using the starting address and length specified in the descriptor passed by the calling program. Thus, the entire area described by the string is written. The descriptor is not modified.

- Dynamic string semantics: The formal string is either written by passing the address of the formal string descriptor and the string to be copied to

  - STR$COPY_DX,
  - STR$COPY_R,
  - LIB$SCOPY_DXDX,
  - LIB$SCOPY_R_DX,
  - OTS$SCOPY_DXDX,
  - OTS$SCOPY_R_DX,

  or allocated by calling

  - STR$GET1_DX,
  - LIB$SGET1_DD,
  - or OTS$SGET1_DD

  and written in pieces. Only the length and address of the descriptor is modified by any of the preceding dynamic string resource allocation procedures.

The two methods that you can choose for a procedure's interface specification to return a string as an output string parameter (or function value) are:

- Use fixed-length semantics (regardless of the class code in the descriptor passed by the calling program).
- Use the semantics indicated in the descriptor passed by the calling program. If DSC$B_CLASS contains DSC$K_CLASS_S=1 or DSC$K_CLASS_Z=0, use fixed-length string semantics. In contrast, if DSC$B_CLASS contains DSC$K_CLASS_D=2, use dynamic string semantics. (preferred)

With either semantics, a procedure also provides an optional, unsigned word output that indicates the length of the string in bytes, not including any space filling. If the string is truncated, the returned length reflects the truncation. Thus, the output parameter can always be used by the calling program to extract the significant data. (See Section 4.5.)

14. A procedure cannot require its caller to pass a dynamic string descriptor. (See Section 4.5.)

15. Some procedure interface specifications retain state information from one call to the next, even though the procedures are not resource allocating. The interface specification uses one of the following techniques to permit sequences of calls from independent parts of a program. These techniques either eliminate the use of static storage or overcome its limitations (in order of decreasing preference):

    - The interface specification consists of a sequence of calls to a set of one or more procedures — the first procedure allocates and returns (as an output parameter to the calling program): (1) the address of heap storage or (2) some other process-wide identifying value. This parameter is passed to the other procedures explicitly by the calling program, and the last procedure deallocates any heap storage or process-wide identifying value. (See Sections 2.5.3.2, 2.5.3.3, and 3.3.1.)

- The procedure's caller allocates all storage and passes the address on each call. (See Sections 2.5.3.1 and 3.3.2.)
- The interface specification consists of a single call, where the calling program passes the address of one or more action routines and arguments to be passed to them. The procedure calls the action routine(s) during its execution. Results are retained by the procedure across calls to the action routine(s). (No static storage used. See Section 2.5.2.)
- The interface specification consists of a sequence of calls to a set of one or more procedures. The first procedure, among other things,saves the contents of any still active static storage on a push down stack in heap storage, and the last procedure, among other things, restores the old contents of static storage. Thus, static storage is made available for implicit parameters to be passed from one procedure to the next in the sequence of calls (unknown to the calling program). However, if an exception can occur anywhere in the sequence, the calling program must establish a condition handler that calls the last procedure in the event of a stack unwind (to restore the old contents of static storage). (See Section 3.3.3.)

16. A procedure does not assume that the implicit outputs of procedures that it calls will remain unchanged if subsequently used as implicit inputs to those procedures or companion procedures. For example, your procedures cannot call SYS$CNTREG to contract the program region by the amount expanded previously by a call to SYS$EXPREG, since an intervening call to SYS$EXPREG might have been made by another procedure. Similarly, your procedure cannot make two calls to SYS$EXPREG and expect to have the second program region expansion allocated contiguously to the first. (See Sections 2.4.2 and 4.6.8)
17. * A procedure executes in any VAX-11 access mode and at any address. (You should not assume that address bit 31 is always 0.)
18. * A procedure does not depend on AST interrupts being enabled to execute correctly if there are other coding methods available. Therefore when doing synchronous VAX-11 RMS I/O, RMS completion routines are not used. (See Section 4.6.12.)
19. A procedure provides an interface to its callers that allows the callers to follow all required elements of this standard.
20. A procedure does not call other procedures or system services if the resulting combination violates this standard from the calling program's viewpoint. A procedure can call other procedures or system services that do not follow optional elements of this standard. However, if the resulting

combination (as seen from the calling program) does not follow the optional elements, the calling procedure must indicate such non-conformance in its documentation. (See Section 4.6.)

21. A procedure makes no assumptions about its environment other than those of this standard. In particular, to operate as specified, a procedure neither makes assumptions about nor places requirements on the so-called main program.

A.3 Facility-Specific Part of the Standard

The following elements apply to procedures that are part of a specific library facility. The following facility names represent library facilities:

LIB	General Utility and Resource Allocation Procedures
MTH	Mathematics Procedures
STR	String Procedures
OTS	Language-independent Support Procedures
BAS	BASIC–specific Support Procedures
FOR	FORTRAN–specific Support Procedures
PAS	PASCAL–specific Support Procedures

22. For MACRO procedures, the PSECT declarations for library code and data, respectively, are:

```
.PSECT _fac$CODE PIC,USR,CON,REL,LCL,SHR,EXE,RD,NOWRT

.PSECT _fac$DATA PIC,USR,CON,REL,LCL,NOSHR,NOEXE,RD,WRT
```

For BLISS procedures, the code and PUT declaration is:

```
_fac$CODE READ, NOWRITE, EXECUTE, SHARE, PIC, CONCATENATE,
ADDRESSING_MODE (GENERAL)
```

and the OWN and GLOBAL (not available to caller) is:

```
_fac$DATA READ, WRITE, NOEXECUTE, NOSHARE, PIC, CONCATENATE,
ADDRESSING_MODE (LONG_RELATIVE)
```

The linker sorts the leading underline last so library modules cannot cause truncation errors due to byte or word displacement addressing performed by the user program.

NOTE

For user PSECTS, replace "$" with "_" in the preceding specifications; (see Section 4.2.3).

23. A procedure's caller, at its option, can indicate omitted trailing optional parameters either by passing argument list entries that contain zero or by passing a shortened argument list.

24. When a new version of a procedure replaces an existing library procedure, all added parameters are made optional to maintain upward compatibility. (See Section 2.3.4.)

General Utility Procedures (LIB)

25. LIB procedures follow the facility-independent part of this standard described in Section A.2.
26. LIB procedures pass arrays and strings by descriptor and input scalars by reference. LIB procedures can pass parameters by immediate value if the procedure provides a service for BLISS and MACRO programmers that is generally supplied as part of higher-level languages. For output string parameters (and string function values), LIB procedures use the semantics indicated in the descriptor passed by the calling program and return the string length as an optional output parameter. (See element 13 and Sections 2.3.2, 2.3.3, and 4.5.)
27. * The storage for input and output parameters can overlap at the option of the calling program. Therefore, a procedure is programmed to behave the same regardless of whether there is overlap.
28. LIB procedures return error conditions to the caller using completion codes returned in R0 as a function value rather than signaling. The condition value SS$_NORMAL (value of 1) is returned to indicate unqualified success. If LIB procedures call MTH, STR or other procedures that signal, the LIB procedures set up a handler, such as LIB$SIG_TO_RET, to convert any software signals to return status. (See Section 2.3.6 and Chapter 5.)

Mathematical Procedures (MTH)

29. MTH procedures follow the facility-independent part of this standard described in Section A.2.
30. MTH procedures pass input scalars by reference. (See Section 2.3.2.)
31. * The storage for input and output parameters can overlap at the option of the calling program. Therefore, a procedure is programmed to behave the same regardless of whether there is overlap.
32. MTH procedures signal errors since the function value (R0) is used to return a mathematical value. (See Section 2.3.6 and Chapter 5.)

String Procedures (STR)

33. STR procedures signal those errors that are programming errors, such as invalid descriptor, or environment errors that are very difficult to recover, such as exhausting virtual memory. Remaining errors are returned as function values.

Language-Independent Support Procedures (OTS)

34. Language-independent support procedures follow the higher-level, language-specific support procedure elements of this standard. (See elements 36–41.)
35. Language-independent support procedures return output string parameters (and string function values) using the semantics indicated in the descriptor passed by the calling program. (See element 13 and Section 4.5.2.)

Higher-Level Language-Specific Support Procedures (BAS, FOR, PAS)

36. Higher-level language-specific support procedures are programmed so, combined with the language-specific program unit, they follow the facility-independent part of this standard as seen from a program calling the program unit, except where prevented by the semantics of a language standard.

37. The standard CALL linkage is recommended, especially for large procedures that perform significant computation. However, JSB and other non-standard linkages can be used with no corresponding standard CALL entry point, since most language–specific support routines are not intended to be called explicitly by users. To ensure compatibility between compiler generated code and JSB entry points, each JSB entry point name ends in _Rn where n indicates the highest register modified or used as an input parameter. To ensure that all registers will be correctly restored when an unwind occurs, the compiler-generated code must save at least R2 thru Rn in its entry mask and the JSB code support routines cannot save, reference, or modify Rn+1 thru R11. (See Section 2.2.)

38. If a particular functional capability already exists in a LIB, MTH, OTS, or STR facility, then those procedures should be called by other language-specific procedures rather than implementing their own algorithms.

39. Higher-level language support procedures pass input scalars by immediate value if they are 32 bits or less, input scalars by reference if they exceed 32 bits, output scalars by reference, input and output arrays by reference or by descriptor, and input and output strings by descriptor. (See Sections 2.3.2 and 4.5.)

40. If a higher-level language statement does not indicate an error action, the error is signaled. Otherwise, higher-level language support procedures return a completion code to the caller on an error, where a compiled code check of R0 would not be an excessive speed or space penalty. However, when the penalty is excessive, the procedure can retain the error transfer address in the first of a series of calls, and transfer directly to it on an error after removing the stack frame. (See Section 2.3.6 and Chapter 5.)

41. For I/O errors that are signaled, the user PC is included as one of the FAO parameters so the user PC can be included in the message. Thus, the PC can be included even if traceback is disabled. (See Section 5.4.)

A.4 * AST–Reentrant Procedures (Optional)

The following elements are required for all AST–reentrant procedures. To be AST–reentrant, a procedure must execute correctly while allowing any procedure (including itself) to be called between any two instructions. The other procedure can be an AST–level procedure, a condition handler, or another AST–reentrant procedure (see Chapter 6). A procedure that uses no static storage and calls only AST–reentrant procedures is automatically AST–reentrant. (See element 15 for ways to eliminate the use of static storage.)

42. * A procedure that uses static storage uses one of the following methods (or equivalent) to be called from AST and non-AST levels (in order of decreasing preference):

    - Perform access and modification of the data base in a single uninterruptable instruction. (See Section 6.3.1.)
    - Detect concurrency of data base access with "test and set" instructions at each access of the data base. (See Section 6.3.2.)
    - Keep a call-in-progress count incremented upon entry to the procedure and decremented upon return. The count is used as an index into separately allocated areas. (See Section 6.3.3.)
    - Disable AST interrupts on entry to the procedure and restore the state of the AST enables on return. The procedure must also establish a condition handler that restores the state of the AST enables in case an exception condition or stack unwind occurs. Since this technique could affect the real time response of the calling program, it must be documented if used. Furthermore, the length of time that ASTs are disabled should be minimized. (See Section 6.3.4.)

43. * If a procedure performs I/O from the AST level by calling VAX-11 RMS $GET and $PUT system services, it must check for the record stream active error status (RMS$_RSA). If the error is encountered, the procedure issues the $WAIT system service and then retries the $GET or $PUT system service. (See Section 6.4.)

A.5 * Sharable Images (Optional)

A procedure that adheres to the following elements can be included in a sharable image at any time.

44. * A procedure's code is position-independent. All references to relocatable data use PC relative addressing mode (n(PC)). All references to absolute locations such as VMS System Service entry points, use absolute addressing mode (@(PC)+). (See Section 1.2.9)

45. The data need not be position-independent. However, for improved performance, data should be initialized to zero at compile or link time to avoid either position-independent constants or position-dependent addresses. (See Section 4.3.1.)

46. A procedure cannot use LIB$INITIALIZE to initialize data, since a sharable image cannot make a PSECT contribution to a user program at link time. (See Section 4.3.4.)

A.6 * Upwards Compatible Sharable Images (Optional)

To be compatible with all future versions of the sharable image, sharable image procedures follow these additional rules:

47. A procedure's entry points are vectored using a separate MACRO module containing TRANSFER declarations. (See Section 7.4.)

48. A procedure's code and data is position-independent. Because VMS can provide a demand-zero page when the page is first accessed, initializing the data to zero is recommended.

A.7 Modular Programming Recommendations (Optional)

The following elements are recommended in the hope that modular procedures will be similar in form and format and thereby more usable by others. These recommendations deal with matters of style. Therefore, nonconformance to them will not affect modularity and need not be documented.

49. The order of required parameters should be the same as that of the VAX–11 hardware instructions, namely, read, modify, and write. Optional parameters follow in the same order. However, (according to the VAX–11 Procedure Calling Standard) if a function value cannot be represented in 64 bits, the first parameter specifies where to store the function value, and all other parameters are shifted one position to the right. (See Section 2.3.5.)

50. A procedure should not have static storage unless it: (1) is a process-wide, resource-allocating procedure, or (2) must retain results for implicit inputs on subsequent activations. Most of the techniques in element 15 avoid static storage. If a procedure cannot eliminate static storage and does not need to retain information from one procedure activation to the next, it writes each static storage location before the first read access to it. (See Section 2.5 and Chapter 3.)

51. If a procedure produces human-readable text and outputs it to a file or device by default, it provides the caller with the option of specifying a parameter that consists of an action routine to accept the text instead (see Section 2.6). The procedure calls the action routine with each line of text as a string containing a leading space (in case of FORTRAN carriage control) and no ASCII CR, LF, VT, or FF. Thus, the string can be put in three of the four types of record attribute files (CR, FTN, or PRN). The string is passed by descriptor and should not exceed 80 characters so it can be printed on most terminals. The action routine returns a condition value that is either: success (the procedure continues), or failure (the procedure stops further calls to the action routine). (See Sections 2.6 and 4.7.)

52. A procedure that allocates process-wide resources provides an entry point that shows the state of the resource for debugging and performance statistics). If such an entry point produces human readable output to a file or device, it must also conform to element 51. (See Sections 2.7, 4.4, and 4.7.)

53. Timing procedures and resource allocation procedures should make statistics available for performance evaluation and debugging. (See Section 2.7.) Such procedures should provide two entry points that accept an input parameter code (1,...,n) indicating the desired statistic and return a

completion status in R0:

fac$SHOW_name**	Provides formatted strings according to element 51. A zero input parameter code requests all available statistics and can produce one or more calls to the action routines, each of which passes a single line. If the calling program does not supply the optional action routine parameter, the string(s) are output to SYS$OUTPUT.
fac$STAT_name **	Returns the binary value of the statistic you want.

** (User versions use _ instead of $)

54. The recommended format for prompt strings is: an English word or words followed by a colon (:), one space, and no CRLF. (RSX utilities use > with no trailing spaces.)

55. Procedures should follow structured programming guidelines. This includes placing a minimum number of procedures — typically one — in a module, and arranging procedures in levels of abstraction. Related procedures, such as those that access the same static storage, should be in the same module. (See Section 4.1.)

56. Procedures should be placed in a module that is documented with a module description. Every procedure should be documented with a procedure description. (See Section 2.8 for the template.)

57. File names should be identical to the first nine characters of the module name, with $ and _ characters omitted. (See Section 4.2.2.)

58. When symbol definitions are to be coordinated between more than one module, (such as control blocks, procedure parameter values, and completion status codes), the definitions should be centralized in one place. The preferred method is for procedures to make external declarations to obtain the symbolic value. Then, a source module can be compiled or assembled independently from any other source files. When the use of external symbols is not practical or possible, procedures should use these techniques (see Section 4.2.4):

• **MACRO:**	Macro library file
• **BLISS:**	REQUIRE or LIBRARY file
• **BASIC:**	APPEND file
• **FORTRAN:**	INCLUDE file
• **PASCAL:**	INCLUDE file

59. Procedure name formats contain a verb followed by the object of the action: for example, LIB$GET_VM and LIB$FREE_VM. (See Section 2.2.)

60. JSB calling sequences should be avoided because they are not available to most languages. When a procedure uses a JSB entry point, it should also provide an equivalent CALL entry point.

61. Instructions and statements are upper-case, while comments are in upper- and lower-case. A space follows every comma, semicolon, and exclamation point. A space precedes a left parenthesis or square bracket (except in MACRO), but not a left angle bracket. Block comments start in column 1 and have the following form (use ; or ! depending on the language):

```
;+
; Put one or more lines of block comment here
;-
```

62. Use symbols rather than numbers in the body of the procedure. In MACRO, procedures use numeric labels (n$) in logical blocks of code that fit on the same listing page. (See Section 4.2.5.)

A.8 Change History

The following changes have been made to the VAX–11 Modular Programming Standard from the previous version of the *VAX–11 Guide to Creating Modular Library Procedures* (AA-H500-A-TE, February 1979).

1. Added Element 5; when procedures can be grouped in a single module.
2. Added Element 10; error condition associated with a file.
3. Added new paragraph to Element 13; the optional, string length parameter.
4. Expanded Element 21; a procedure and its environment.
5. Added Element 25; LIB$ procedures follow the facility-independent part of this standard.
6. Moved Element 27 from Section A–2.
7. Expanded Element 28; error condition handling in LIB$ procedures.
8. Added Element 29; MTH$ procedures follow the facility-independent part of this standard.
9. Moved Element 31 from Section A–2.
10. Added Element 33; error handling in STR$ procedures.
11. Added Element 36; high-level language-specific support procedures follow the facility-independent part of this standard except where prevented by individual language semantics.
12. Added Element 37; using JSB entry-points in high-level language-specific support procedures.
13. Added Element 41; availability of the user PC when an I/O error is signaled.
14. Expanded Element 51; 80–character limit on human-readable, output text.
15. Expanded Element 62; using numeric labels in MACRO procedures.

Appendix B
Naming Conventions

The conventions described in this appendix were derived to aid implementors in producing meaningful public names. Public names are all names known to the linker ("global") in parameter or macro definition files.

Reasons for the public naming conventions include the following:

- Using reserved names ensures that customer-written software is not invalidated by subsequent releases of DIGITAL products which add new symbols.
- Using definite patterns for different uses lets you judge the type of object being referenced. For example, the form of a macro name differs from that of an offset, which differs from that of a status code.
- Using certain codes in a pattern associates the size of an object with its name. This increases the likelihood that the reference uses the correct instructions.
- Using a facility code in symbol definitions gives the reader an indication of where the symbol is defined. Separate groups of implementors can choose facility codes names that do not conflict with one another.

Never define local synonyms for public symbols. You should use the full public symbol in every reference to give maximum clarity to the reader.

B.1 Public Symbol Patterns

All DIGITAL public symbols contain a dollar sign. Thus, customers and applications developers are strongly advised to use underscores instead of dollar signs to avoid future conflicts.

Public symbols should be constructed to convey as much information as possible about the entity they name. These are used both in a module and globally between modules of a facility. All names that might ever be bound into a user's program must follow the rules for public names; for internal names, you can use a double dollar sign convention. (See the following numbers 3 and 5.) However, DIGITAL is free to change the interface of entities identified with more than one dollar sign.

Public names are of the following forms:

1. Service macro names are of the form:

   $macroname

   A trailing _S or _A distinguishes the stack and separate arglist forms. These names are in the system macro library and represent a call to one of many facilities. The facility name usually is not in the macro name.

2. Facility-specific public macro names are of the form:

   $facility_macroname

3. System macros using local symbols or macros always use those of the form:

   $facility$macroname

   This is the form to be used both for symbols generated by a macro and included in calls to it, and for internal macros that are not documented.

4. Status codes and condition values are of the form:

   facility$_status

5. Global entry point names are of the form:

   facility$entryname

   Global entry point names intended for use only in a set of related procedures (but not by any calling programs outside the set) are of the form:

   facility$$entryname

6. Global entry point names that have nonstandard calls (JSB entry point names) are of the form:

   facility$entryname_Rn

   where values in registers R0 to Rn are not preserved. The caller of such an entry point must include at least registers R2 through Rn in its own entry mask so a stack unwind correctly restores all registers.

7. Global variable names are of the form:

   facility$Gt_variablename

   The G stands for global variable, and the t represents the type of variable, as defined in Section B.2.

8. Addressable global arrays use the A (instead of the G) and are of the form:

   facility$At_arrayname

   The A stands for global array, and t represents the type of array element, as defined in Section B.2.

9. In the assembler, public structure offset names are of the form:

   structure$t__fieldname

   The t represents the data type of the field, as defined in Section B.2. The value of the public symbol is the byte offset to the start of the field in the structure.

10. In MACRO, public structure bit field offset and single bit names are of the form:

    structure$V__fieldname

    The value of the public symbol is the bit offset from the start of the containing field (not from the start of the control block).

11. In MACRO, public structure bit field size names are of the form:

    structure$S__fieldname

    The value of the public symbol is the number of bits in the field.

12. In BLISS, the functions of the symbols in the previous three items are combined into a single name used to reference an arbitrary datum. Names are of the form:

    structure$x__fieldname

    where x is T for standard-sized data and x is V for arbitrary and bit fields. The macro includes the offset, position, size, and sign extension suitable for use in a REF BLOCK structure. Typically, this name is definable as:

    MACRO

    ```
            structure$V__fieldname =
            structure$T__fieldname,
            structure$V__fieldname,     !assembler meaning
            structure$S__fieldname,
            <sign extension> %&;
    ```

    or

    FIELD

    ```
            structure$V__fieldname =
                [structure$T__fieldname,
                structure$V__fieldname,
                structure$S__fieldname,
                <sign extension> %& ];
    ```

13. Public structure mask names are of the form:

    structure$M__fieldname

    The value of the public symbol is a mask with bits set for each bit in the field. This mask is not right justified; it has structure$V__fieldname zero bits on the right.

14. Public structure constant value names are of the form:

    structure$K_constantname

15. PSECT names are of the form:

    facility$mnemonic

    and when put in a library:

    _facility$mnemonic

16. Module names are of the form:

    facility$mnemonic

    The module is stored in a file with file name "facilitymnemonic".

17. Public structure definition macro names are of the form:

    $facility_structureDEF

    Invoking this macro defines all the structure$xxx symbols.

Example of usage:

IOC$IODONE	Entry point of the routine IODONE in the I/O subsystem.
UCB$B_FORK_PRI	Offset in the UCB structure to a byte datum containing the fork priority.
UCB$L_STATUS	Offset in the UCB structure to a longword datum containing status bits.
CRB$M_BUSY	Mask pattern for the busy bit in the CRB structure.
CRB$V_BUSY	Bit offset in the CRB structure of the busy bit.

B.2 Object Data Types

The following letters are used for data types or are reserved:

Letter	Data Type or Usage
A	address
B	byte integer
C	single character
D	double precision floating
E	reserved to DIGITAL
F	single precision floating
G	general value
H	integer value for counters
I	reserved for integer extensions

Letter	Data Type or Usage
J	reserved to customers for escape to other codes
K	constant
L	longword integer
M	field mask
N	numeric string (all byte forms)
O	reserved to DEC as an escape to other codes
P	packed string
Q	quadword integer
R	reserved for records (structure)
S	field size
T	text (character) string
U	smallest unit of addressable storage
V	field position (assembler); field reference (BLISS)
W	word integer
X	context dependent (generic)
Y	context dependent (generic)
Z	unspecified or non-standard

N, P, and T strings are typically variable-length. In structures or I/O records, they frequently contain a byte-sized digit or character count preceding the string. If so, the location or offset is to the count. Counted strings cannot be passed in CALLs; instead, a string descriptor is generated.

The letters A, C, G, H, and U should be used in preference to L, B, L, W, and B when transportability is involved. This table defines their sizes:

Letter	16-bits	32-bits	36-bits
A	16	32	18
C	8	8	7
G	16	32	36
H	16	16	18
U	8	8	36

B.3 Facility Prefix Table

The following list shows some of the facility prefixes for DIGITAL-supplied software. This list will grow as new facility prefixes are chosen. You should not use a new code without registering it in a common place.

Prefix	Facility	Condition (27:16)
BAS	BASIC support	26
BLI	BLISS transportable support	20
B32	BLISS-32 support	27

(continued on next page)

B.3 Facility Prefix Table (Cont.)

Prefix	Facility	Condition (27:16)
COB	COBOL support	25
FOR	Fortran support	24
LIB	General Utility	21
MTH	Math	22
OTS	Language independent (Object Time System)	23
PAS	PASCAL support	33
RMS	RMS internals and status codes	1
SORT	VAX-11 SORT	28
SS	System Service Status Codes	0
STR	String	36
XPO	BLISS transportable	32

Individual products such as compilers also get unique facility codes formed from the product name. They must be signed out in the registry. You should choose facility prefixes to avoid conflict with file types.

Structure name prefixes are typically local to a facility. Refer to the individual facility documentation for its structure name prefixes. This method does not cause problems, since these names are not global and are therefore not known to the Linker. They become known at assembly or compile time only by explicitly invoking the macro defining the facility structure.

NOTE

DIGITAL does not provide a registration service for the customer facility codes.

Appendix C
Notation for Describing Procedure Parameters

This appendix describes a language-independent notation for procedure parameters, including the type of access, the data type, the parameter passing mechanism, and the form of the parameter.

C.1 Routine Interface Types

To achieve the VAX–11 goal of being able to mix languages in a program, all routines are designed with certain common attributes. The data types and mechanism passing rules are designed to maximize the ability to interface to routines. A common notation is used to express the specification of the interface.

The access types, data types, mechanisms, and parameter forms are defined in the *VAX–11 Run-Time Library Reference Manual.* In the design of a procedure interface, the data types must be specified. Four other considerations are also important:

1. Whether the routine follows the VAX–11 procedure calling standard.
2. Whether its scalar input parameters are by immediate value or by reference.
3. How output strings are returned; this is discussed in the next paragraph.
4. Whether the routine has a function value and whether the value is a status code or a scalar result.

In any given facility, it is generally preferable to have only one style of these interface choices. Other combinations can be chosen, but the prospect of user confusion must be weighed against the possible inefficiency of forced consistency.

There are two string semantics for returning a string to a calling program as an output parameter or a function value:

- Fixed-length string semantics: The called procedure writes the string starting at the address specified in the descriptor and blank fills or truncates on the right. It does not modify the contents of the descriptor.

- Dynamic string semantics: The called procedure allocates the string buffer and places both the address and the length into the dynamic descriptor by calling library dynamic string allocating procedures.

The calling program can always pass a fixed-length or dynamic string at its option to any procedure.

There are two choices for the interface specification of a procedure:

- Return string using fixed-length semantics (notation __.wt.ds)
- Return string using either fixed-length or dynamic semantics as specified by the caller in the descriptor (notation __.wt.dx)

The choice depends on the environmental assumptions made in procedure design.

The most common combinations of interface specifications are given in the following table.

The column "Passing Scalars" shows how scalars are passed. The column "Output Strings" shows how output strings are returned. The column "Function Value" shows what kind of function value is returned.

Type of Call	Instruction	Passing Scalars	Output Strings	Function Value
J (non-CALL)	JSB	in register	-	-
V (immed val)	CALL	AP by immed val	length,descr	.lc
F (Function)	CALL	AP by reference	none	scalar
BASIC	CALL	AP by reference	dynamic	any
COBOL	CALL	AP by reference	fixed	any
FORTRAN	CALL	AP by reference	fixed	any
PASCAL	CALL	AP by reference	byte array	any

C.2 Notation for Describing Procedure Parameters

A concise, language-independent notation describes each procedure parameter. The notation is a compatible extension to the one used in the *VAX-11 Architecture Handbook.*

The notation specifies for each parameter:

1. A mnemonic name
2. The type of access the procedure will make (read, write, ...)
3. The data type of the parameter (longword, floating, ...)
4. The argument passing mechanism (immediate value, reference, descriptor)
5. The form of the parameter (scalar, array, ...)

If a parameter is an address saved for later access by another procedure, the notation should reflect the ultimate access made by the second procedure.

C.2.1 Procedure Parameter Characteristics

Subroutines are described as:

CALL subroutine_name(parameter1, parameter2, ..., parametern)

and functions are described as:

function_value = function_name(parameter1, parameter2, ..., parametern)

where parameter and function_value are:

<name>.<access type><data type>.<passing mechanism><parameter form>

where:

1. <name> is a mnemonic for the procedure formal specifier or function value specifier.

2. <access type> is a single letter denoting the type of access that the procedure will (or can) make to the argument:

   r - parameter can be read only.

   m - parameter can be modified, that is, read and written.

   w - parameter can be written only.

   j - parameter is an address to be (optionally) jumped to after stack unwind (return). No <data type> field is given, since the argument is a sequence of instructions, for example, FORTRAN ERR=.

   c - parameter is an address of a procedure to be (optionally) CALLed after stack unwound (return). No <data type> field is given, since the argument is a sequence of instructions.

   s - parameter is an address of a procedure subroutine to be (optionally) CALLed without unwinding the stack. The <data type> field indicates the data type used to represent the subroutine (ZEM or BPV).

   f - parameter is an address of a function to be (optionally) CALLed without unwinding the stack. The <data type> field indicates the data type used to represent the function (ZEM or BPV). Immediately following ZEM or BPV is the data type of the function value. For example, func.fzeml.r indicates that the arg list entry contains the address of a function that returns a signed longword value in R0.

   a - reserved for use in the *VAX-11 Architecture Handbook* (address). Not used here since the object pointed to is specified.

   b - reserved for use in the *VAX-11 Architecture Handbook* (branch destination). Not used here since a branch destination cannot be a procedure formal.

v –reserved for use in the *VAX-11 Architecture Handbook* (variable bit field).

3. <data type> is a letter denoting the primary data type with trailing qualifier letters to further identify the data type. Note that the routine must reference only the size specified to avoid improper access violations.

Data Type Code	Letters	Use
		Atomic Data Types
0	z	Unspecified
1	v	Bit (variable bit field)
2	bu	Byte logical (unsigned)
2	c	Single character
2	u	Smallest unit for addressable storage
3	wu	Word logical (unsigned)
4	lu	Longword logical (unsigned)
4	a	Virtual address
4	cp	Character pointer
4	lc	Longword containing a completion code
5	qu	Quadword logical (unsigned)
25	ou	Octaword logical (unsigned)
6	b	Byte integer (signed)
6	arb	Byte containing a relative virtual address (*)
7	w	Word integer (signed)
7	arw	Word containing a relative virtual address (*)
8	l	Longword integer (signed)
8	arl	Longword containing a relative virtual address (*)
9	q	Quadword Integer (signed)
26	o	Octaword integer (signed)
10	f	Single-precision F-Floating
11	d	Double-precision D-Floating
27	g	Double-precision G-Floating
28	h	Quadruple-precision H-Floating
12	fc	F-Floating complex
13	dc	D-Floating complex
29	gc	G-Floating complex
30	hc	H-Floating complex
31	cit	COBOL intermediate temporary

(continued on next page)

Data Type Code	Letters	Use
		String Data Types
14	t	text (character) string
15	nu	Numeric string, unsigned
16	nl	Numeric string, left separate sign
17	nlo	Numeric string, left overpunched sign
18	nr	Numeric string, right separate sign
19	nro	Numeric string, right overpunched sign
20	nz	Numeric string, zoned sign
21	p	Packed Decimal string
		Miscellaneous Data Types
–	x	Data type indicated in descriptor
22	zi	Sequence of instructions
23	zem	Procedure entry mask
24	dsc	Descriptor (for use in descriptors)
32	bpv	Bound procedure value

* – arl, arw, and arb are self-relative addresses using the same format as the hardware displacements. That is, the self-relative address is a signed offset in bytes with respect to the first byte following the parameter.

4. <passing mechanism> is a single letter indicating the parameter passing mechanism that the called routine expects:

v – immediate value, that is, call by immediate value where the contents of the parameter list entry is itself the parameter of the indicated data type. Note that call by immediate value parameter list entries are always allocated as a longword. The quadword data types can be used as values only for function values, never as a formal parameter. Note also that the VAX–11 calling standard requires that <access type> must be r whenever <passing mechanism> is v, except for function values where <access type> is always w and <passing mechanism> is usually v.

r – reference, that is, call by reference where the contents of the parameter list entry is the longword address of the argument of the indicated data type. If the parameter is a scalar of the indicated data type or is a label, <parameter form> must be absent. If the parameter is an array, <parameter form> must be present.

d – descriptor, that is, call by descriptor where the contents of the parameter list entry is the longword address of a descriptor. The descriptor is two or more longwords that specify further information about the parameter; see Appendix C of the *VAX–11 Run-Time Library Reference Manual.* Note that when <passing mechanism> is d, <parameter form> must be present to indicate the type of descriptor.

5. <parameter form> is a letter denoting the form of the argument:

Class Code	Letters	Meaning
–	–	null means scalar of indicated data type.
4	a	array reference or array descriptor, that is, call by reference or call by descriptor, as indicated by <parameter mechanism>. For array call by reference, the contents of the parameter list entry is the address of an array of items of the indicated data type. The length is fixed, implied by entries in the array (for example, a control block), determined by another parameter, or specified by prior agreement. For array call by descriptor, the contents of the parameter list entry is the longword address of an array descriptor block. (See Appendix C of the *VAX-11 Run-Time Library Reference Manual.)*
1	s	scalar or string descriptor (call by descriptor). The contents of the parameters list entry is the longword address of a 2–longword scalar descriptor. When the data type field (DSC$B_DTYPE) indicates ASCII text (DSC$K_DTYPE_T), the descriptor contains the length, data type, and address of a fixed-length string. When the string is written, neither the length nor the address fields in the descriptor are modified, and the string is filled with trailing spaces or a separate parameter is updated with the written length.
2	d	dynamic string descriptor, that is, passed by descriptor, where the contents of the parameter list entry is the longword address of a 2–longword string descriptor of the same format as that of s. However, when the string is written, both the length and address fields may be modified. Space is allocated dynamically by routines in the procedure library.
5	p	procedure descriptor, that is, passed by descriptor, where the contents of the parameter list entry is the longword address of a two longword procedure descriptor. The descriptor contains the address of the procedure and the data type that the procedure returns if it is a function. <access type> must be c, f, j, or s.
–	x	either fixed-length or dynamic descriptor, as indicated by the calling program in the DSC$B_CLASS field of the descriptor that it passes to the called procedure.
9	sd	scalar decimal descriptor. First two longwords are like the s descriptor. The third longword contains scale factor byte, and number of decimal digits byte.
10	nca	non-contiguous array descriptor. Used when ***elements are not allocated contiguously to one another.

C.2.2 Optional Parameters and Default Values

The caller can omit optional parameters at the end of a parameter list by passing a shortened list. The caller can also omit optional parameters anywhere by passing a 0 value as the contents of the parameter list entry. However, a caller cannot omit a parameter that is not indicated as optional. The called procedure is not obligated to detect such a programming error. Optional parameters are enclosed in square brackets, as follows:

CALL FOR$READ_SU (unit.rb.v [,err.j.r [,end.j.r]]).

An equal sign (=) after a parameter inside square brackets indicates the default value if the parameter is omitted, as in the following example:

success.wlc.v = LIB$DELLOG (lognam.rt.ds [,tblflg.rb.v=0]).

NOTE

> VAX/VMS system services have optional parameters, but the list cannot be shortened. This type of optional parameter is indicated with the comma outside of the square brackets. For example:
>
> success.wlc.v=SYS$DELLOG([tblflg.rl.v], [lognam.rt.dx], [acmode.rl.v])

C.2.3 Repeated Parameters

Parameters that can be repeated one or more times are indicated using ellipses, for example, CALL FOR$OPEN (keywd.rw.v,info.rl.v...). Repeated parameters that can be omitted entirely are indicated with ellipses inside square brackets, for example, CALL FOR$CLOSE ([logical_unit.rl.v...]).

C.2.4 Examples

sine_of_angle.wf.v = MTH$SIN (angle_in_radians.rf.r)

CALL FOR$READ_SF (unit.rb.v, format.mbu.ra [,err.j.r [,end.j.r]])

Note that: (1) end can be omitted and that (2) err and end can both be omitted. However, unit and format must always be present. The parameter count byte in the parameter list specifies how many parameters are present. Alternatively, err, end, or both could have a 0 parameter list entry.

Common combinations are:

completion code:	status.wlc.v =...
longword call by immediate value input arg:	no_of_pages.rlu.v
address of an array of signed words for input:	array.rw.ra
address of a control block:	fab.mz.ra
address of a precompiled format statement:	format.rbu.ra
label to jump to:	error_label.j.r
floating input call by reference arg:	angle_in_rad.rf.r
floating complex call by reference input arg:	angle.rfc.r
read only character string:	string.rt.ds
output fixed-length string:	string.wt.ds
output fixed-length or dynamic string:	string.wt.dx
user action routine that returns a cond. value:	func.fzemlc.r

C.2.5 Summary Chart of Notation

<name>.<access type><data type>.<pass mech><parameter form>

<access type>		<data type>	
c	Call after stack unwind	a	Virtual address
f	Function call (before return)	arb	8-bit relative virtual address
j	JMP (after unwind) access	arl	32-bit relative virtual address
m	Modify access	arw	16-bit relative virtual address
r	Read-only access	b	Byte integer (signed)
s	Call without stack unwinding	bpv	Bound procedure value
w	Write-only access	bu	Byte logical (unsigned)
		c	Single character
		cit	COBOL intermediate temporary
		cp	Character pointer
		d	Double precision D-floating
		dc	D-floating complex
		dsc	Descriptor (used by descriptors)
		f	Single precision F-floating
		fc	F-floating complex
		g	Double precision G-floating
		gc	G-floating complex
		h	Quadruple precision 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 integer (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

(continued on next page)

<passing mechanism>		<parameter form>	
d	By descriptor	–	Scalar
r	By reference	a	Array reference or descriptor
v	By immediate value	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

The notation xy.z means that the argument is passed only to a user-supplied procedure, and so can have any access type (x), data type (y) and passing mechanism (z).

Index

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 —