February 1979

This document describes how to design and code procedures so that they can be installed in an object module library, or in a shareable image. It includes the standards and recommendations for modular programming in any language.

VAX-11
Guide to Creating Modular Library Procedures

Order No. AA-H500A-TE

SUPERSESSION/UPDATE INFORMATION: This is a new document for this release

OPERATING SYSTEM AND VERSION: VAX/VMS V1.5

SOFTWARE VERSION: VAX/VMS V1.5

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

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

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

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

Copyright © 1979 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

Page

FIGURES (Cont.)

TABLES

PREFACE

MANUAL OBJECTIVES

This manual is a tutorial guide to designing and coding modular procedures written in VAX-11 MACRO, BLISS-32, or FORTRAN IV-PLUS. Such procedures may be used for general programming or for inclusion in a procedure library. Such libraries include the system default object library, user-created object libraries, or user-created shareable images.

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

INTENDED AUDIENCES

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

STRUCTURE

All chapters in this manual are tutorial.

- Chapter 1 is an introduction that provides an overview of modular programming and of libraries, the options that you have in creating your own procedures and libraries, and information required to determine which 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, and VAX-11 FORTRAN IV-PLUS. This includes required and optional standards 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 (1) how to insert or replace a procedure in the system default object library, and (2) how to create and link with either a user object library or a user shareable image.

The appendixes provide useful background information:

- Appendix A summarizes the modular standards (both required and optional) and recommendations. Required standards must be followed. Optional standards must be followed or documented as not being followed. Recommendations should be followed, 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 Common Run-Time Procedure Library Reference Manual AA-D036A-TE
- VAX/VMS System Services Reference Manual AA-D018A-TE
- VAX-11 Linker Reference Manual AA-D019A-TE
- VAX-11 FORTRAN IV-PLUS User's Guide AA-D035A-TE
- VAX-11 FORTRAN IV-PLUS Language Reference Manual AA-D034A-TE
- VAX-11 MACRO User's Guide AA-D033A-TE
- VAX-11 MACRO Language Reference Manual AA-D032A-TE
- VAX-11 BLISS-32 User's Guide AA-D942A-RE
- VAX-11 BLISS-32 Language Guide AA-H019A-RE

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

CONVENTIONS USED IN THIS DOCUMENT

Unless otherwise noted, all numeric values are represented in decimal notation.

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

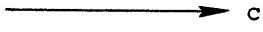
Variable information is indicated by lowercase characters; literal information, which you must enter exactly as shown, is indicated by uppercase characters.

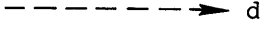
Brackets ([]) in procedure descriptions indicate optional arguments. An equal sign after an optional parameter indicates the default value if you omit the parameter.

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

Unless otherwise specified, the term MACRO will be used to mean VAX-11 MACRO, the term BLISS will be used to mean BLISS-32, and the term FORTRAN will be used to mean VAX-11 FORTRAN IV-PLUS.

In diagrams, the following conventions are used:

 control path

 data path

 interface

CHAPTER 1

INTRODUCTION

A procedure is a set of related instructions that performs a particular task. Typically, it is invoked by executing a VAX-11 CALLS or CALLG instruction. In MACRO, a procedure begins with a .ENTRY and terminates with a RET; in BLISS, a procedure is declared as a ROUTINE with the default linkage; in FORTRAN, a procedure is a main program, subroutine, or function.

A procedure is modular if it can be successfully linked and run in combination with any other modular procedure written by you or other programmers. You can design and code modular procedures by following the standards and recommendations of this book. Modular procedures may be used for general programming or for inclusion in procedure libraries.

The linker resolves references to procedures within a library by searching either the user libraries specified in the LINK command or 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 upon 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 shareable image save phsyical memory and disk space because all user processes access a single copy.

1.1 USING LIBRARIES WITH VAX/VMS

Procedures can be grouped together in two ways:

- As an object module library. A call to a procedure in a module causes the module to be individually copied and linked to the calling program's object file. The module and the program then become a single executable image.

- As a shareable image. A call to a procedure in a shareable image causes the linker to map the entire contents of the shareable image into the program's executable image.

The following subsections describe the default system libraries and recommendations for creating both object module libraries and shareable images.

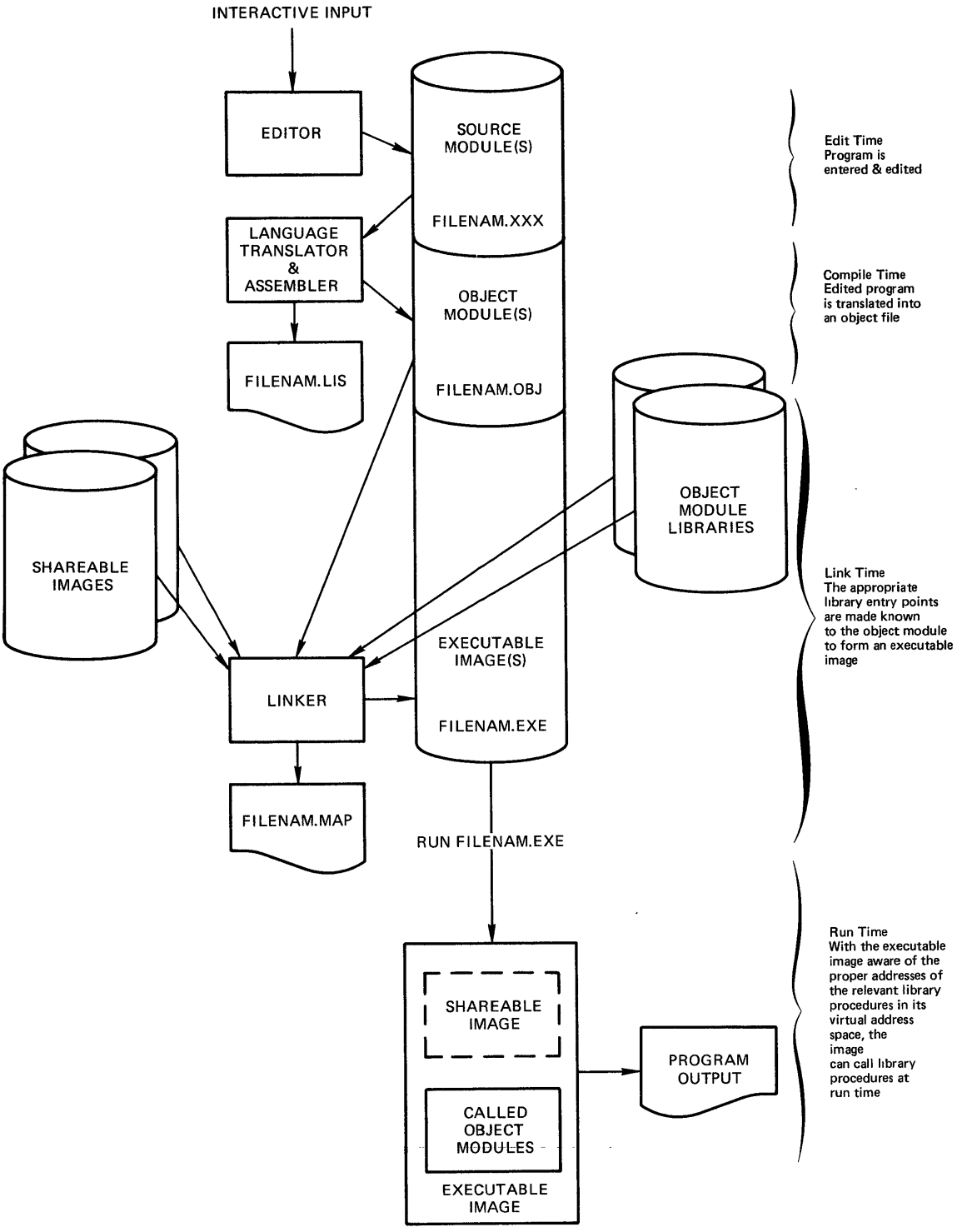


Figure 1-1 Developing a Program That Calls Library Procedures

1.1.1 DIGITAL-Supplied Libraries

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

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

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

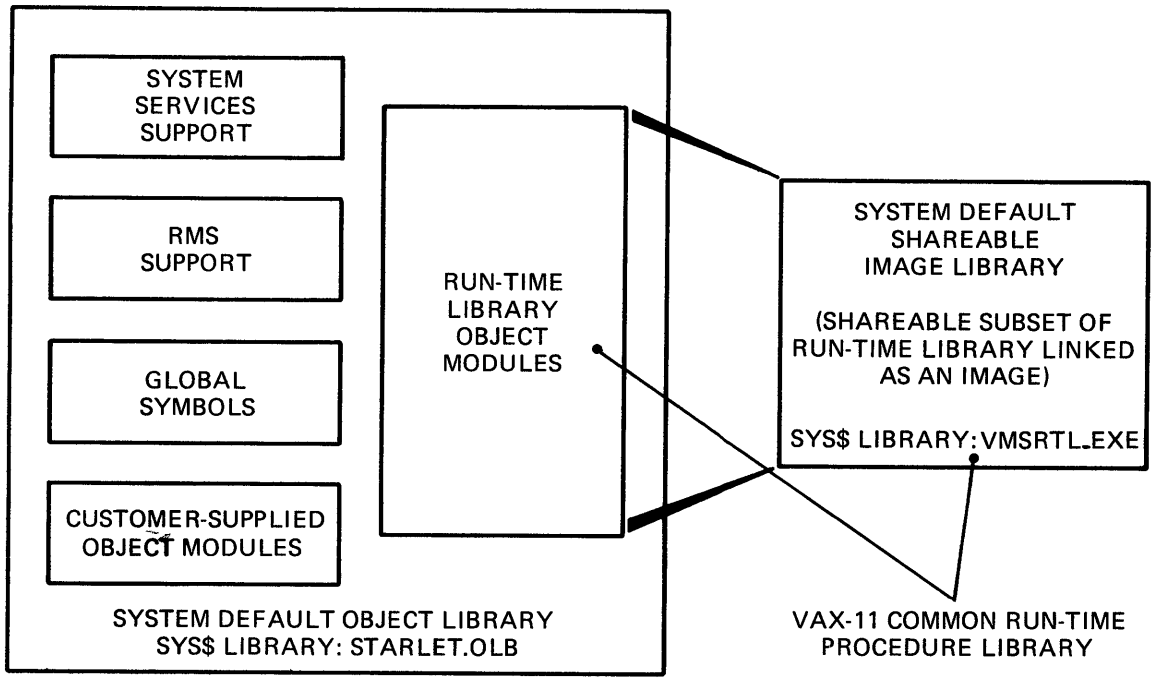


Figure 1-2 DIGITAL-Supplied Libraries

The linker automatically searches both of these libraries for unresolved references to global symbols during a LINK command. First, the linker searches VMSRTL.EXE, which is a shareable subset of STARLET.OLB. If the linker resolves a reference with this shareable image, it will map (as opposed to copying) the entire shareable image into 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.

1.1.2 User-Created Object Libraries

Figure 1-3 shows the development of a user-created object library.

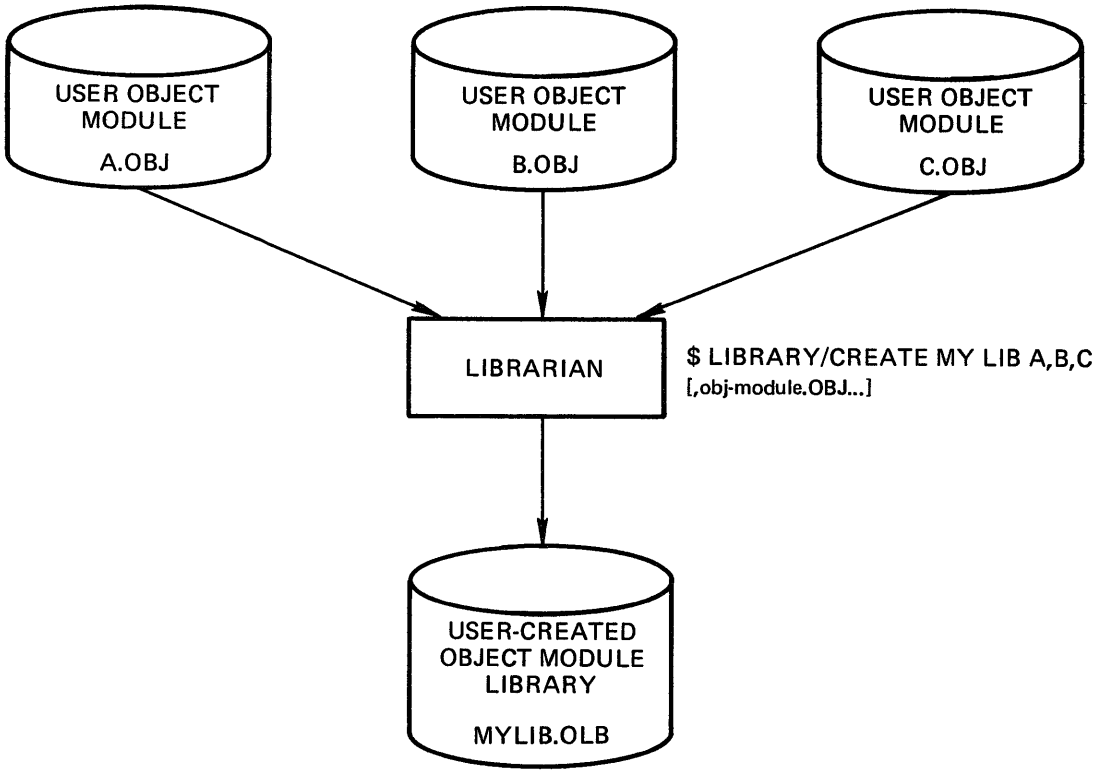


Figure 1-3 Creating an Object Module Library

A user-created object module library consists of procedures written by the user 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 library files is OLB. The default file type for input object files is OBJ.

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

- Implicit inclusion occurs when a module specified in the LINK command 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 qualifier 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.

More information on the linker's use of libraries may be found in Section 4.2 of the VAX-11 Linker Reference Manual.

1.1.3 User-Created Shareable Images

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

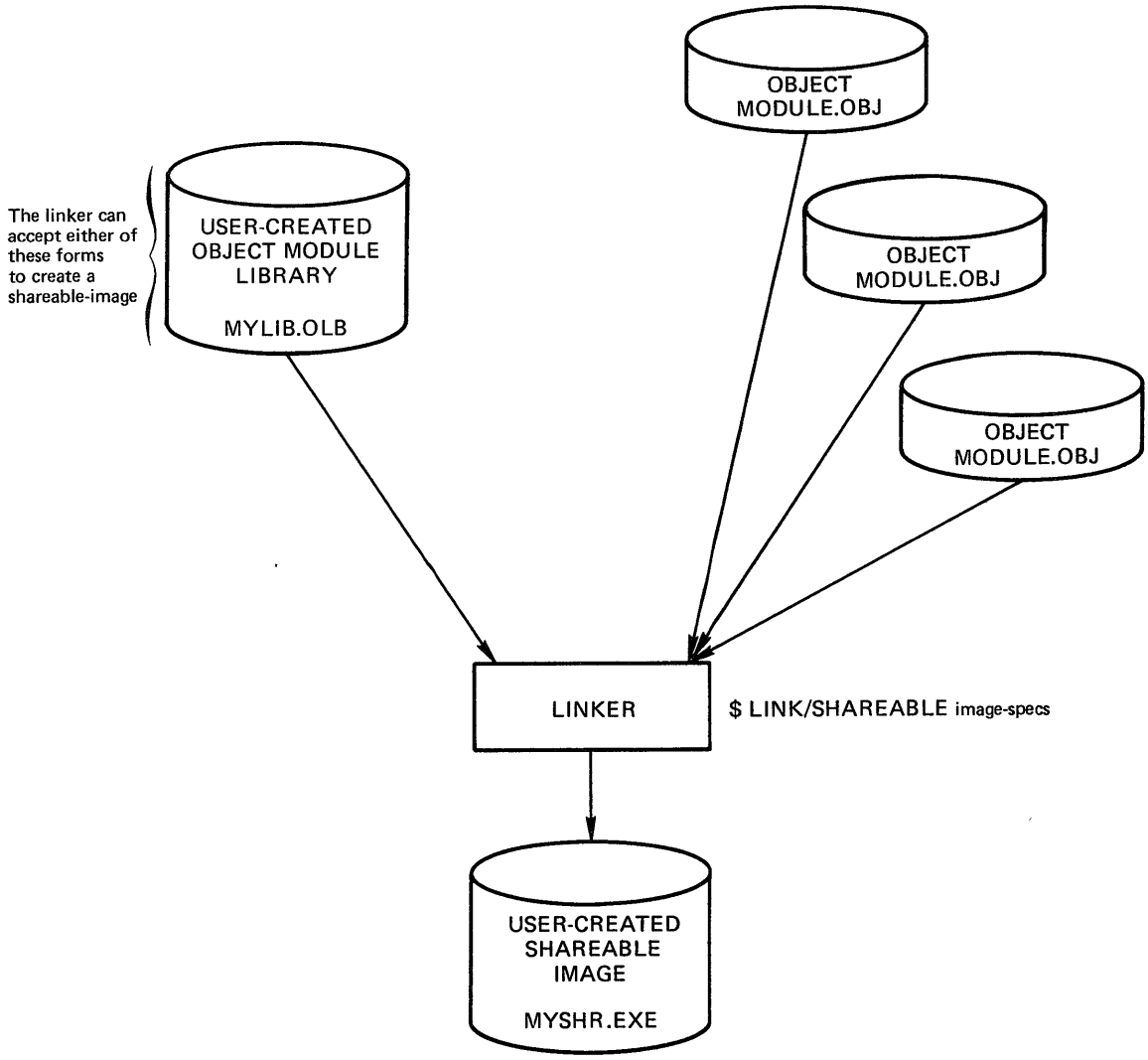


Figure 1-4 Creating a Shareable Image

A user-created shareable image may consist of a subset of a user-created object library. It contains modular procedures usually written in position-independent code that are used frequently enough to warrant being shared among processes. You can specify the user-created shareable image as input to the linker by using the /OPTIONS qualifier after the name of the options file in the LINK command. Section 8.1 of the VAX-11 Linker Reference Manual details the benefits and uses of shareable images.

1.1.4 Linking Programs to Run-Time Libraries

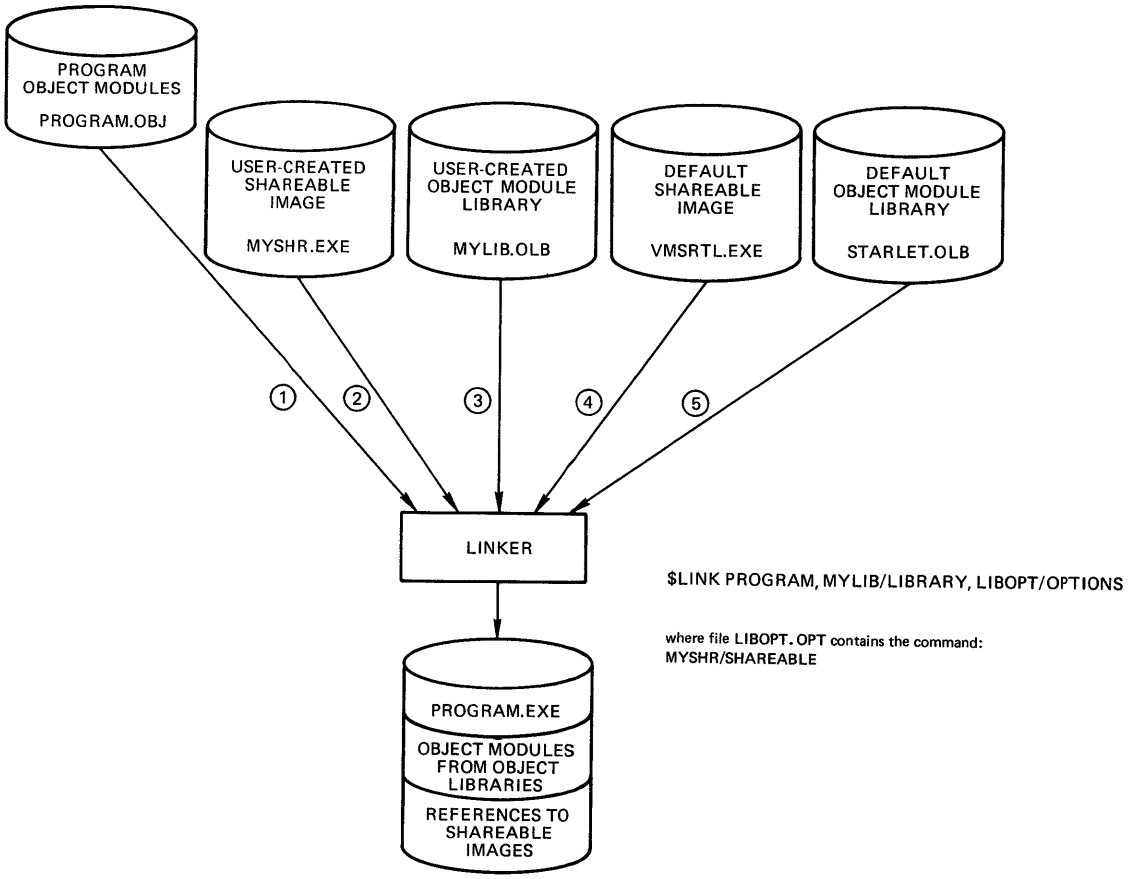


Figure 1-5 Linking Programs to Run-Time Libraries

Figure 1-5 shows how each type of run-time library is linked to a program object module to form an executable image. When the link command shown is given, the following events occur:

1. PROGRAM.OBJ is linked into the image.

2. MYSHR.EXE, the user-created shareable 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 linker will include a copy of the modules resolving those references in the image.

4. VMSRTL.EXE, the default shareable image, is automatically included if and only if it resolves any remaining unresolved references.[1]

[1] You can use the /NOSYSSHR qualifier to request the linker to omit the search of the default shareable image.

5. STARLET.OLB, the default object library, is automatically searched if any unresolved references remain.[1] If references are resolved, the linker will include a copy of the modules in the image that resolve those references.

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

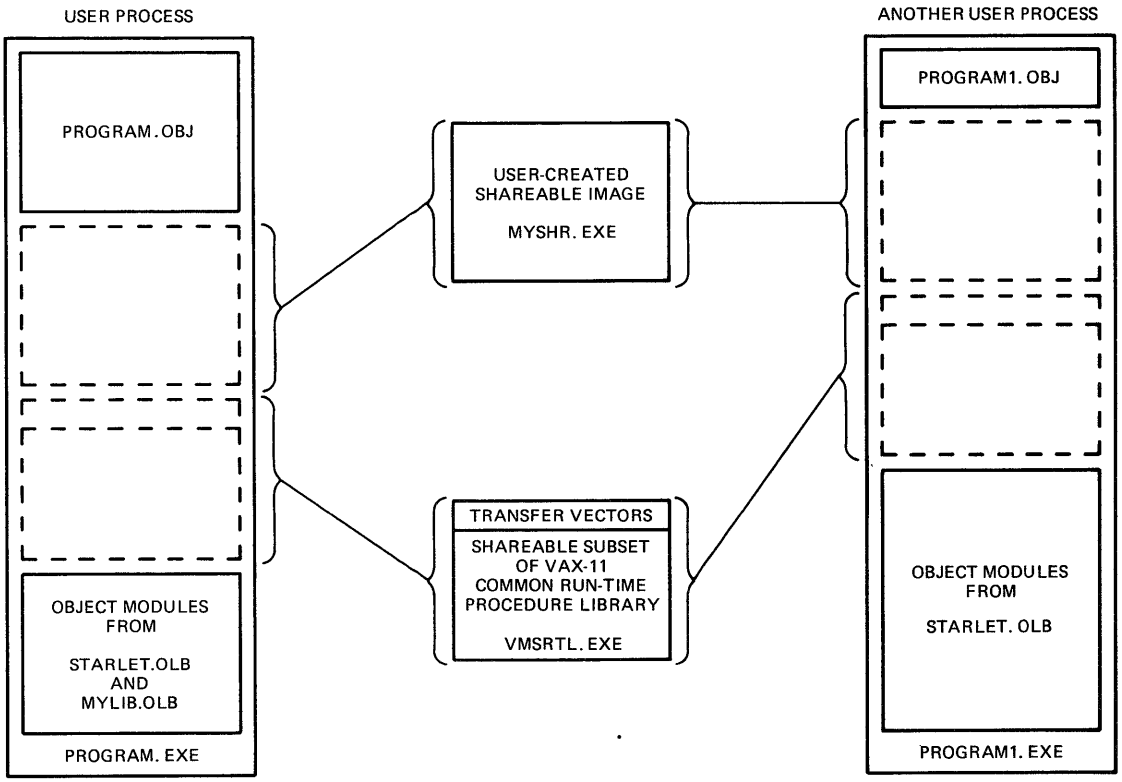


Figure 1-6 Executing an Image That Calls Library Procedures

Note that copies of the modules taken from STARLET.OLB and MYLIB.OLB are bound with each image that links with the object libraries, while the shareable modules in VMSRTL.EXE and MYSHR.EXE reside in a single image file that is shared.

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 standards and recommendations described in this manual.

---

[1] You can use the /NOSYSLIB qualifier to request the linker to omit the search of the default shareable image and the default object module library.

Modular programming standards are:

- A subset of the VAX-11 Procedure Calling Standard
- Additional standards and recommendations for modular programming

They are used internally by DIGITAL in the development of VAX/VMS library software.

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

It is recommended that any procedure placed in a shareable image also be placed in an object module library. Then in the rare case that a very large program is close to the virtual memory limit on your system, you can choose to include only called modules from the object library rather than allocating virtual memory for the entire shareable image.

1.2.1 Advantages of Modular Programming

The modular programming standards described in this manual offer several advantages over the practice of writing a complex program as a single source module. If you follow all the required standards, you will gain the following advantages:

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

If you follow the optional standards specified in this manual, you can also gain these additional advantages:

- Shareable 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.
- Structured programming recommendations enable your procedures to work together in a logical pattern.

1.2.2 Modular Programming Standards

Appendix A lists the modular programming standards explained in this manual. There are three types of standards:

- Required standards.
- Optional standards that are either followed or must be noted as not followed in the procedure's documentation.
- 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 the major aspects of the modular programming standards.

1.2.3 Storage

Most procedures use some type of storage to retain information either during a single procedure activation or between successive activations. While any modular procedure can use any of three types of storage, there are certain rules that are followed. These rules are explained in Chapter 3.

1.2.4 Naming Standards and Recommendations

This manual describes the required naming standards for procedures, modules, and program sections (PSECTS). It also describes the naming recommendations for file names. These are described in Sections 2.2 and 4.2.

1.2.5 Process-wide Resource Allocation

Process-wide resources are those resources that may be allocated as needed to any procedure in a process. They include blocks of virtual memory, dynamic string space, VMS event flags, and FORTRAN logical unit numbers. Moreover, you can create additional resources. Modular procedures follow the standard of allocating resources by calling a resource-allocating procedure rather than allocating the resource directly themselves. This prevents conflicts that could occur if two procedures were to allocate the same resource. The available resources and allocating methods are described in Section 4.4.

1.2.6 Use of System Services

Modular procedures may use system services that conform to the modular programming standards. Section 4.6 lists all the system services and indicates those that may be used by modular procedures.

1.2.7 Signaling and Condition Handling

Modular procedures follow certain standards to indicate errors. For example, all modular procedures either return a condition value or call system-signaling procedures to output all error messages. The programming standards for signaling and condition handling are discussed in Chapter 5. In addition, techniques of signaling between related procedures are described.

1.2.8 AST-Reentrant Procedures

VAX/VMS provides a mechanism that you use to interrupt the execution of an image 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 is capable of being interrupted and executed again before resuming successfully at the point of the interrupt. Thus, they may be called from AST-level and/or non-AST-level routines. Most modular procedures are designed to be AST-reentrant. Chapter 6 describes how to write AST-reentrant modular procedures.

1.2.9 Position-Independent Code

A position-independent piece of code will execute correctly no matter where it is placed in the virtual address space after it is linked. All shareable images are comprised of position-independent code. However, shareable images can have data that may or may not be position-independent.

Position-independent code is discussed in detail in Section 8.2.6 of the VAX-11 Linker Reference Manual, and in some language reference manuals.

1.2.10 Transfer Vectors

Transfer vectors are used to prevent the need to relink images that call procedures in a shareable image every time a new version of the image is installed. You can add transfer vectors to procedures in a shareable image at any time, as explained in Chapter 7.

1.3 CREATING AND MODIFYING LIBRARIES

You can add or modify procedures in the default system object library (STARLET.OLB), your own object library, or your own shareable image.

Adding procedures to existing libraries and installing your own libraries is discussed in Chapter 7.

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 as described in Chapter 7.

You can replace any module in any object library, including STARLET.OLB, also with the use of the LIBRARY command. Modules in STARLET.OLB may be replaced by examining the source files (available from DIGITAL) and substituting your module for the DIGITAL-supplied one.

1.3.2 Creating Shareable Images

You create a shareable image primarily to optimize storage space and access time. This involves the use of 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 shareable image is written in position-independent code and you have provided transfer vectors, the following advantage may also be gained:

- Elimination of the need to relink all images that called the old version when you install a new version.

You should observe the following rules-of-thumb when deciding whether to create a shareable image:

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

1.3.3 Updating Shareable Images

If you wish to add or modify anything in a shareable image, it is necessary to reinstall the entire image. You may do this to any user-created shareable image.

You cannot add or modify anything in the system default shareable image VMSRTL.EXE. However, by making a user-created shareable image that contains VMSRTL.EXE, you can modify it and substitute it for VMSRTL.EXE.

To substitute a user-created shareable image, OURSHRRTL.EXE, for VMSRTL.EXE, the following command is used:

```
$COPY OURSHRRTL.EXE SYS$LIBRARY:VMSRTL.EXE/NEW_VER
```

CHAPTER 2

DESIGN OF MODULAR PROCEDURE INTERFACES

The interface between a procedure and its caller must be modular so that 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
- Procedure names
- 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 standards that must be followed to ensure modularity, optional standards 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 is provided to help you:

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

The section numbers indicate where detailed information may be found.

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

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

- Access Type
- Data Type
- Passing Mechanism
- Form

Place the parameters in the calling sequence in the proper order (see Section 2.3.4).

3. Decide whether 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 2.3.6 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 (see Section 2.8.1)
   - Write procedure descriptions (see Section 2.8.2)

8. Decide how each procedure will utilize storage. Determine the type of storage to be used and steps required to maintain modular standards (see Chapter 3).

9. Make structured programming considerations (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 shareable

10. Check Appendix A for the complete list of modular programming standards 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 procedure to handle error conditions (see Chapter 5).

13. Decide whether to make procedures AST-reentrant (see Chapter 6).

14. Follow coding standards and recommendations while writing code (see Section 4.1). Be sure to follow standards in the following areas:

    - Initialization (if needed) (see Section 4.3)
    - Use of system services (if needed) (see Section 4.6)

If you are passing string parameters, see Section 4.5.

15. Debug procedures while maintaining modular standards.

16. (optional) Add debugged procedures to an object module library and/or install as a shareable image (see Chapter 8).

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.

A symbol generally consists of a verb followed by the object that together 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 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 these user-created internal global entry point names from user-created condition value symbols which have two underscores.

2.2.1 Facility Names

The DIGITAL-defined facility names are registered in a DIGITAL-maintained system-wide registry. The following facility names are used in the Common Run-Time Procedure Library:

LIB	General purpose
MTH	Mathematics
OTS	Language-independent support
FOR	FORTRAN support
BLI	BLISS transportable support
B32	BLISS-32 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 may also create your own facility names if none of the above are appropriate.

2.2.2 Condition Value Symbols

Condition value symbols are used to symbolically define unique system-wide 32-bit condition values that are used in return status codes and 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 means of a facility name and facility number. Bit 27 (STS$V_CUST_DEF) of a condition value indicates whether the condition value is user- or DIGITAL-supplied. This bit must be 1 if the facility number is user-created.

2.3 EXPLICIT PARAMETERS

Since explicit parameters are a procedure's primary interface with everything outside of itself, standards for parameter types and passing mechanisms must be carefully followed to maintain a modular interface.

2.3.1 Parameter Characteristics

Every parameter has the following characteristics:

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

Table 2-1 lists the possible alternatives that each of these characteristics can have. Each alternative is described in detail in Appendix C of this manual. This list is complete for all characteristics allowed by the VAX-11 Procedure Calling Standard.

The letter abbreviations next to each characteristic indicate a shorthand notation that is used in documentation to record the characteristics of each parameter. The format is:

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

For example the documentation for the calling sequence of LIB$GET_INPUT is:

```
ret-status.wlc.v = LIB$GET_INPUT (get-string.wt.dx [,prompt-string.rt.dx])
```

Table 2-1
Procedure Parameter Characteristics

<access type>	<data 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	a Absolute virtual address arb Byte containing relative virtual address arl Longword containing relative virtual address arw Word containing relative virtual address b Byte integer (signed) bu Byte logical (unsigned) c Single character cp Character pointer d Double precision floating-point f Single precision floating-point fc Complex floating-point h Integer value for counters l Longword integer lc Longword return status lu Longword logical (unsigned) nu Numeric string, unsigned nl Numeric string, left separate sign nlo Numeric string, left overpunched sign nr Numeric string, right separate sign nro Numeric string, right overpunched sign nz Numeric string, zeroed sign p Packed decimal string q Quadword integer (signed) qu Quadword integer (unsigned) t Text (character) string u Smallest unit of addressable storage 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
<passing mechanism>	<parameter form>
d By-descriptor r By-descriptor v By-reference	- Scalar a Array reference or descriptor d Dynamic string descriptor p Procedure reference or descriptor s Fixed length string descriptor x Class type in descriptor

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 such styles, you should follow the same quidelines. 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	By-Value	By-Reference	By-Descriptor
Scalars			
Input	OTS,FOR	LIB,MTH	–
Output	–	OTS,FOR,LIB	–
Arrays			
Input	–	OTS,FOR,LIB	FOR
Output	–	OTS,FOR,LIB	FOR
Strings			
Input	–	–	LIB, FOR, OTS
Output			
Fixed length	–	–	LIB, FOR, OTS
Dynamic	–	–	LIB, OTS

2.3.3 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 specified as described in the VAX-11 Procedure Calling Standard (see Section C.8 in VAX-11 Common Run-Time Procedure Library Reference Manual for 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			
	Class	Length	Pointer	Library Facility
Input Parameter to Procedures				
Input String Passed By-Descriptor	Ignored	Read	Read	OTS,FOR LIB
Output From Procedures: (class assumed by called procedure)				
Output String Passed-By-Descriptor (fixed-length)	Ignored	Read	Read	FOR
Output String Passed by Descriptor (dynamic)	Ignored	Always Written	May Be Written	LIB,OTS
Output Parameter from Procedures: (class specified by calling program)				
Output String (unspecified (DSC$K_CLASS_Z)	Read	Read	Read	LIB,OTS
Output String (fixed length) (DSC$K_CLASS_S)	Read	Read	Read	LIB,OTS
Output String (dynamic) (DSC$K_CLASS_D)	Read	Always Written	May Be Written	LIB,OTS

2.3.4 Optional Parameters

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

Note that for parameters passed by-value, there is no distinction between passing a zero value or passing a zero argument list entry.

2.3.5 Order of Parameters

Procedures in the VAX-11 Common Run-Time Procedure Library follow a consistent pattern for the relative position of parameters. It is recommended that procedures group their parameters in the same left-to-right order as the VAX-11 hardware instructions, namely:

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)

Data is accessed in a left-to-right order. The only exceptions to the left-to-right rule are for functions in which the function value exceeds 64 bits 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.

2.3.6 Error and Condition Values

A procedure can indicate errors to its caller by either returning a condition value as a completion code or by 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 make a choice of recovery paths.

For a description of signaling, see Chapter 6 of VAX-11 Common Run-Time Procedure Library Reference Manual. 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 B32 BLI FOR	Returns completion code when a check of the code is not an excessive speed or space penalty; otherwise, it signals the error.

2.4 IMPLICIT PARAMETERS

In addition to explicit parameters, there are 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 if your procedure will have implicit parameters, you should consider the advantages and disadvantages discussed below. It is easier to maintain modularity by not using them. If your procedure needs to retain information from previous activations and you want to avoid using implicit inputs, read Section 2.5. If you must use implicit parameters, read the rest of this section as well as 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 that could be allocated by the calling program:

- Statically allocated variables in a named PSECT (for example, COMMON in FORTRAN)
- Statically allocated global variables (for example, symbols defined with a double colon :: in MACRO, and GLOBAL variables in BLISS)

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

- Two programmers may 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 inadvertantly affect the other.
- In 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.

Using implicit parameters that are allocated by the calling program violates modular programming standards.

2.4.2 Implicit Inputs Allocated by the Called Procedure

There is one type of implicit parameter allocated by the called procedure. The procedure declares static storage using .BYTE through .QUAD in MACRO, OWN in BLISS, and all variables in FORTRAN.

Implicit inputs of this type are normally used to keep track of resources (by resource allocating procedures), and to shorten the explicit parameter list. However, the use of implicit inputs by nonresource-allocating procedures can lead to unexpected results. Consider that procedure A and a companion procedure B are in a situation where A is specified to leave information for 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. In order for the calling program to get correct results from B, the calling program must know that X (and any procedures that X calls) did not make a call to A (as such a call would change the implicit inputs A leaves for B).

Figure 2-1 illustrates this configuration.

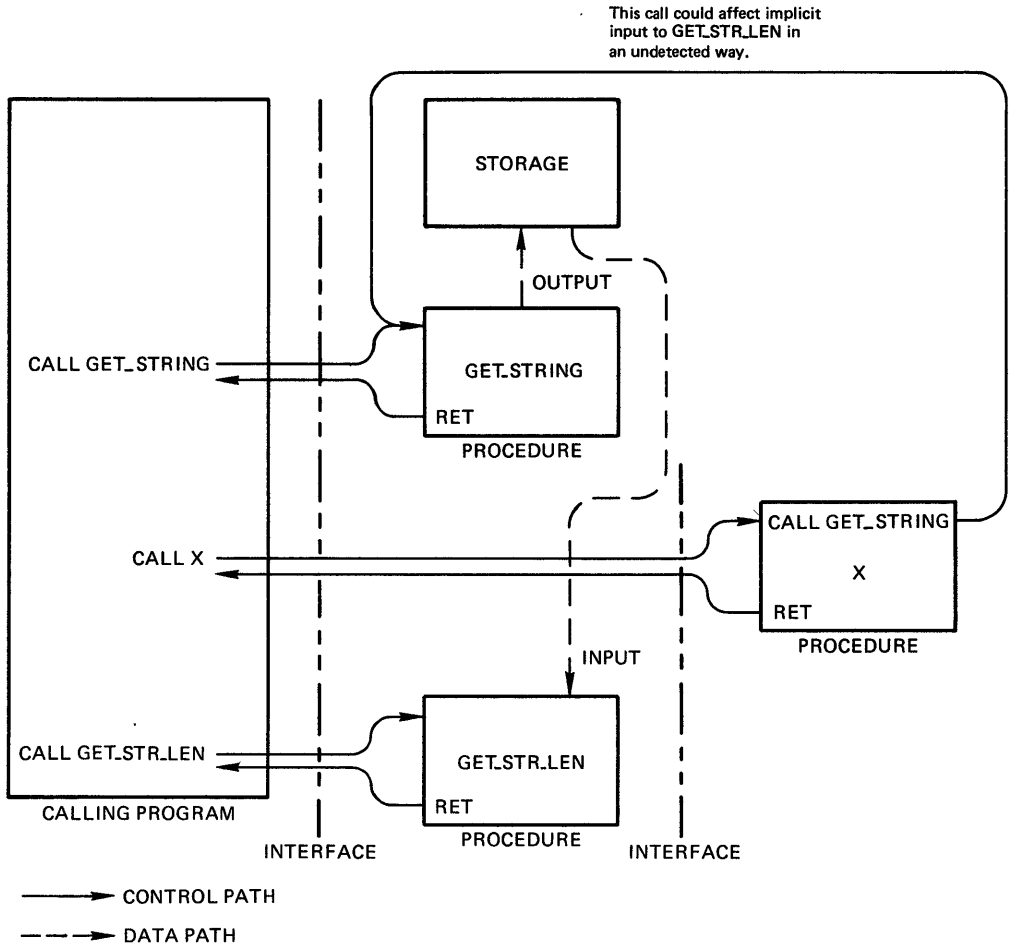


Figure 2-1 How Implicit Inputs Can Violate Modular Standards

The use of such implicit parameters violates the modular programming standards. The aforementioned problem will also occur if X is rewritten in the future 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.

Consider the following example of (LIB_GET_STRING) which reads a string from the terminal, and a companion procedure (LIB_GET_STR_LEN) which 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, A80)                 ! 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
```

The following 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())
```

The following section describes how to avoid the problems of implicit input parameters.

2.5 HOW TO AVOID IMPLICIT INPUTS

There are three ways to write nonresource-allocating procedures that avoid the implicit parameter problems described above:

- When one procedure obtains results from another, combine the two procedures into a single call (see Section 2.5.1).
- 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 of which saves the contents of any still active implicit parameters on a push down stack in heap storage, and the last of which restores 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 Combine Procedures Into One

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 example of the companion procedures LIB_GET_STRING and LIB_GET_STR_LEN in section 2.4.2.

By changing LIB_GET_STRING and LIB_GET_STR_LEN into a single procedure, Procedure X will no longer be able to 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
        ACCEPT 100, LEN, LIB_GET_STRING
100     FORMAT (Q, A80)                 !Set LENGTH to length of line input
        RETURN
        END
```

The following 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)
```

Another way to combine several procedures into one call is to allow the calling program to optionally gain control at a critical point in the execution of your procedure instead of your providing two procedures. This consists of specifying an optional action routine parameter in your procedure that will be called if the calling program provides it. Thus your procedure is able to execute twice: before and after the action routine with no implicit inputs. The FORTRAN OPEN statement uses 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 optional parameter which, if specified by the calling program, is passed along to the action routine. The calling sequence to your procedure is thus:

```
CALL my-proc (...[,action-routine.flc.rp [,user-arg.xx.x]])
```

The calling sequence for the action routine is:

```
CALL action-routine (...[,user-arg.xx.x])
```

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

2.5.2 Designating Responsibility to the Calling Program

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

- Cause the calling program to allocate the necessary storage required by your procedure. Then have it pass the storage

address as an explicit parameter on all calls to your procedure (see Section 2.5.2.1).

- Cause 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 will store the address allocated in the caller's longword. On subsequent calls, your procedure will use information left in the storage area from previous calls (see Section 2.5.2.2).

- Cause 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 2.5.2.3).

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

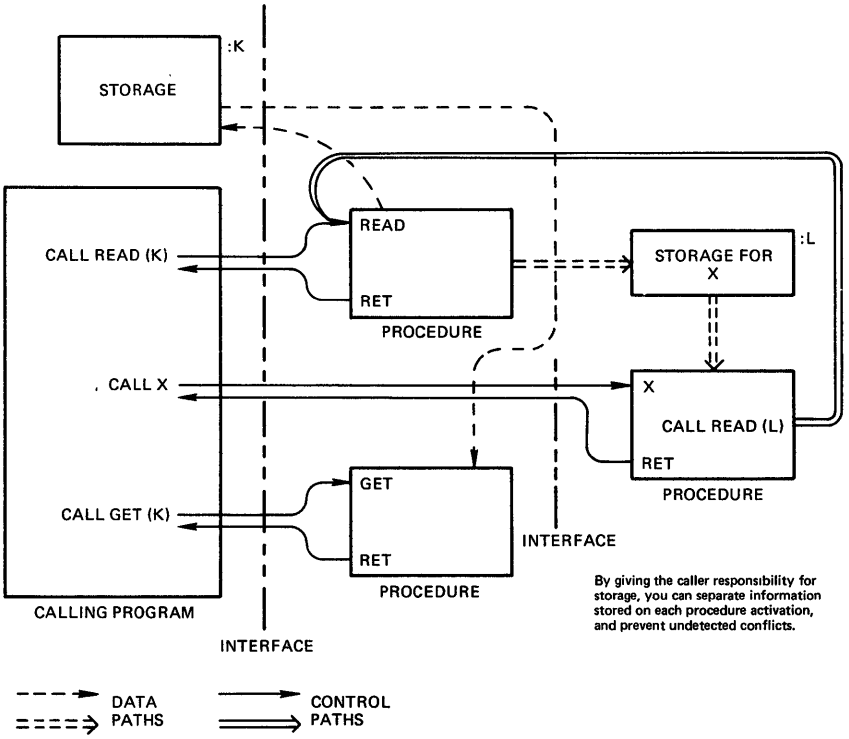


Figure 2-2 Designating Storage Responsibility to the Caller

2.5.2.1 **Calling Program Allocates Procedure Storage** - In this method, the calling program allocates all the storage needed and passes its address as an explicit parameter on each call. The library procedure MTH$RANDOM gets the storage it needs for the seed parameter from the calling program. 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 so that it will generate a different value on the next call (the code is shown in Section 3.3.1).

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 only use this method when you are confident that your procedure will not need to be revised in the future to use additional storage. The next two sections describe interface techniques which permit the size of storage to be changed without affecting interface with the calling program.

2.5.2.2 **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 and 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.
- Store the address of the allocated storage in the longword. On subsequent calls, the nonzero value instructs your procedure to use that value as the address of storage where information from previous calls can be found.

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

For example, the following procedure, LIB$INIT_TIMER which gets specified 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 will be stored.

If missing, they will be 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 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. If so, the block is reused, and fresh times and counts are stored in it.

Entry point LIB$FREE_TIMER deallocates the block of dynamic heap storage that had been allocated by a previous call to LIB$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.

2.5.2.3 **Calling Program Passes a Process-Wide Identifier** - In this method, the calling program passes a process-wide identifying value to identify the previous calls to which this call will be associated. This value indicates information from previous calls used on this call as implicit inputs. The process-wide identifier may be used by any calling program. Examples of process-wide identifiers include logical unit numbers in FORTRAN and I/O channel numbers in VMS system services.

Process-wide identifiers are a resource. Modular programming standards 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. Such a procedure should be added to the default system object library STARLET.OLB so that all programmers may use it.

An example of a resource-allocating procedure that allocates FORTRAN logical unit numbers is given in Section 4.4.3.

2.6 CONTROL OF HUMAN READABLE OUTPUT

A modular procedure allows its caller to control human readable output to the terminal, queued to a line printer, or written to a file. This is done 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 begins with a space (FORTRAN convention) and contains no ASCII carriage return (CR) or line feed (LF) characters. Thus, the line is suitable to be placed into any three of the four VAX-11 RMS record format files, namely, CR, FTN, or PRT.

The user-supplied action routine may output each record to any output device of its choosing, as well as returning 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.

In order to help your caller to write a single action routine that serves a number of purposes, 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 is particular 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:

```
      EXTERNAL PROC
      .
      .
      .
      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
      OUTPUT (10, *, ERR=100) RECORD
      PROC = 1                        ! Success
100   RETURN
      END
```

or as:

```
      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.

2.7 TIMER AND RESOURCE ALLOCATION PROCEDURES

It is recommended that all timer and resource allocation procedures make statistics available for performance evaluation and debugging. Such 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

The SHOW entry point provides formatted strings containing the desired information. 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 desired statistic. A separate code is defined for each statistic available and are the same for the SHOW and STAT entry points. Codes start at 1. 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 arglist 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 be of 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

This entry point returns 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 desired. A separate code is defined for each statistic available and are the same for the SHOW and STAT entry points. Codes start at 1.

value
is the value of the statistic returned.

2.8 DOCUMENTATION OF PROCEDURES AND MODULES

You must document your procedures so that you and others may be sure of your procedure's objective.

2.8.1 Write a Module Description

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

Title:

Gives the module name followed by a 1-line functional description.

Version:

Gives the level and modification number. Generally 0-01 is the original version.

Facility:

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

Functional Description: (or Abstract)

Gives a short 3 to 6-line functional description of the module. If an extensive functional description is needed, a short abstract should be put here; the longer description is added in a later section.

Environment:

This paragraph lists any special environmental assumptions that the module may make. These include assumptions made at both compilation time and execution time that may affect either the hardware or software environments.

For execution time, describe any situations that the module may assume or any optional standards that your module does not follow. Normally, 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 module description for BLISS or FORTRAN. (In MACRO the ! are changed to ;.)

```
!++
!
!  TITLE:  LIB$GET_INPUT              Get Line from SYS$INPUT.
!
!  FACILITY:  General Utility Library
!
!  ABSTRACT:
!
!        Inputs a string as a record from device SYS$INPUT.
!
!  ENVIRONMENT:   Runs at any access mode - AST re-entrant
!
!  AUTHOR:  Bert Byte, CREATION DATE:  8-Aug-1979
!
!  MODIFIED BY:
!
!        Frederick Float, 28-Sep-1979: VERSION 0
!  01    - original
!  04    - change to SYS$INPUT
!  05    - change to do OPEN at first time
!  06    - change to set up RAB for GET_STRING
!
!--
```

Figure 2-3 Example of a Module Description

2.8.2 Write A Procedure Description

You should add 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 the purpose of the module and documents its interfaces completely.

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

Calling Sequence:

A calling sequence to a procedure is described by (1) a return status, output 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 in which they are 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-string.wt.dx [,prompt-string.rt.dx])

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

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

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 VAX-11 MACRO, each parameter is symbolically defined as the offset relative to the argument pointer AP.

Input Parameters:

List any explicit input parameters in the calling sequence. Each input parameter should be listed in the order in which it appears, including a qualifying description.

Implicit Inputs:

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

Output Parameters:

List any explicit output parameters in the parameter list in the calling sequence. Each output parameter should be listed in the order in which it appears, including a qualifying descriptor.

Implicit Outputs:

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

Completion Codes:

List the condition value symbols that may be returned as completion codes. This includes exception conditions signaled, and a list of success and failure completion codes returned in R0.

Side Effects:

This section describes any functional side effects that are not evident from a procedure's calling sequence. This includes changes in storage allocation, process status, file operations, and signals. 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 2-4 shows a sample procedure description for BLISS or FORTRAN. IN MACRO the ! are changed to ;.

```
!++     FUNCTIONAL DESCRIPTION:
!
!       A line from the current controlling input device, SYS$INPUT, is
!       obtained. If an optional PROMPT_STRING is given, output will appear on
!       the device, SYS$INPUT, if the device is a terminal; otherwise
!       the PROMPT_STRING is ignored. On first call, device SYS$INPUT
!       is opened. Thus the user can assign the logical name to any
!       file name in order to redirect I/O.
!
!   CALLING SEQUENCE:
!
!       Rms-status.wlc.v = LIB$GET_INPUT (get_string.wt.dx
!       [,prompt_string.rt.dx])
!
!   INPUT PARAMETERS:
!
!       prompt_string   is the address of a string descriptor specifying
!                       an optional prompt which is output to the
!                       controlling input device. Where other conventions
!                       are not established, it is recommended for
!                       consistency to make prompts be an English word
!                       followed by a colon(:), one (1) space, and no
!                       CRLF.
!
!   OUTPUT PARAMETERS:
!
!       get_string      is the address of string descriptor of any type
!                       (unspecified, static, or dynamic, as
!                       specified by the DSC$B_CLASS field) which is to
!                       receive the string.
!
!   IMPLICIT INPUTS:
!
!       SYS_INPUT_ISI   Set on first call to RMS internal stream identifier.
!
!   IMPLICIT OUTPUTS:
!
!       SYS_INPUT_ISI   Set to RMS internal stream identifier
!                       on first call when SYS$INPUT is OPENed.
!
!   COMPLETION STATUS:
!
!        SS$_NORMAL if success.
!
!        For fixed-length strings, if RMS error RMS$_RTB
!        (RECORD TOO BIG) occurs, the truncated string is returned
!        with an error status of LIB$_INPSTRTRU (INPUT STRING TRUNCATED).
!        If any other RMS error occurs, the RMS error codes is returned.
!        If the descriptor class field is not a recognized code,
!        LIB$_INVARG (INVALID ARGUMENT) is returned.
!
!   SIDE EFFECTS:
!
!        Opens file SYS$INPUT on first call and remembers ISI for
!        subsequent calls.
!
!--
```

Figure 2-4 Example of a Procedure Description

CHAPTER 3

USE OF STORAGE

3.1 TYPES OF STORAGE

There are three types of storage: static, dynamic stack, and dynamic heap. Storage is allocated by assigning it to a virtual address. The three forms of storage differ in how each is allocated and for how long each remains allocated.

3.1.1 Static Storage

Static storage (statically allocated storage) is storage that is allocated by the linker and whose contents are initialized at program translation or link time. On a subsequent call to a procedure with static storage, the storage will have the same allocation and the previous contents.

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

VAX-11 MACRO

The following 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

BLISS

OWN Storage
GLOBAL Storage

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

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

FORTRAN IV-PLUS

All FORTRAN data storage is statically allocated. It is declared as local variables or arrays or is declared in a COMMON statement. 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. Futhermore, variable A is initialized to 10 at compile time while the 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 will not be reinitialized to 10 on subsequent calls to FUNC. Instead the value of A and all other statically allocated variables will retain the values left from the previous call.

3.1.2 Stack Storage

Dynamic stack storage (dynamically allocated stack storage) is allocated on the process stack at run time as it is needed. It is automatically deallocated when the procedure returns control to its caller.

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

```
SUBL    n*4,SP
```

In BLISS, stack storage can be allocated as follows:

```
LOCAL A: LONG;
```

Stack storage cannot be allocated by FORTRAN users.

3.1.3 Heap Storage

Dynamic heap storage (dynamically allocated heap storage) is allocated at run time to a procedure activation as it is needed from a process-wide pool (by calling LIB$GET_VM or the system service $EXPREG). Dynamic heap storage is deallocated -- that is, returned to the process-wide pool -- by calling LIB$FREE_VM.

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

The following example shows how heap storage can be allocated in BLISS:

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

Heap storage may be allocated in FORTRAN but it must then be passed to another procedure as an array parameter in order to be used. (See Section 5.1 in the VAX-11 Common Run-Time Procedure Library Reference Manual.)

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

3.1.4 Summary of Storage Use

Table 3-1 lists a summary of storage available to the programmer in various languages.

Table 3-1
Summary of Storage Use

Storage Type \ Language	MACRO	BLISS	FORTRAN
Static	Avoid if Possible	Avoid if Possible	If Any Variables are Present
Stack	Recommended	Recommended	Not Applicable
Heap	Use When Stack Storage May Be Exceeded; Or When Procedure Retains Information For Subsequent Activation	Same as MACRO	Difficult

3.2 CHOOSING A STORAGE TYPE

A procedure activation is the combination of instructions that implement the procedure and the associated stack frame storage allocated when the procedure is called and deallocated when the procedure returns. If a procedure is called again before it returns, two activations of the same procedure exist at the same time. This can occur if:

- The procedure is called by an AST-level routine.
- The procedure is called by a condition handler while it is executing.
- The procedure is called by another procedure that it has called.

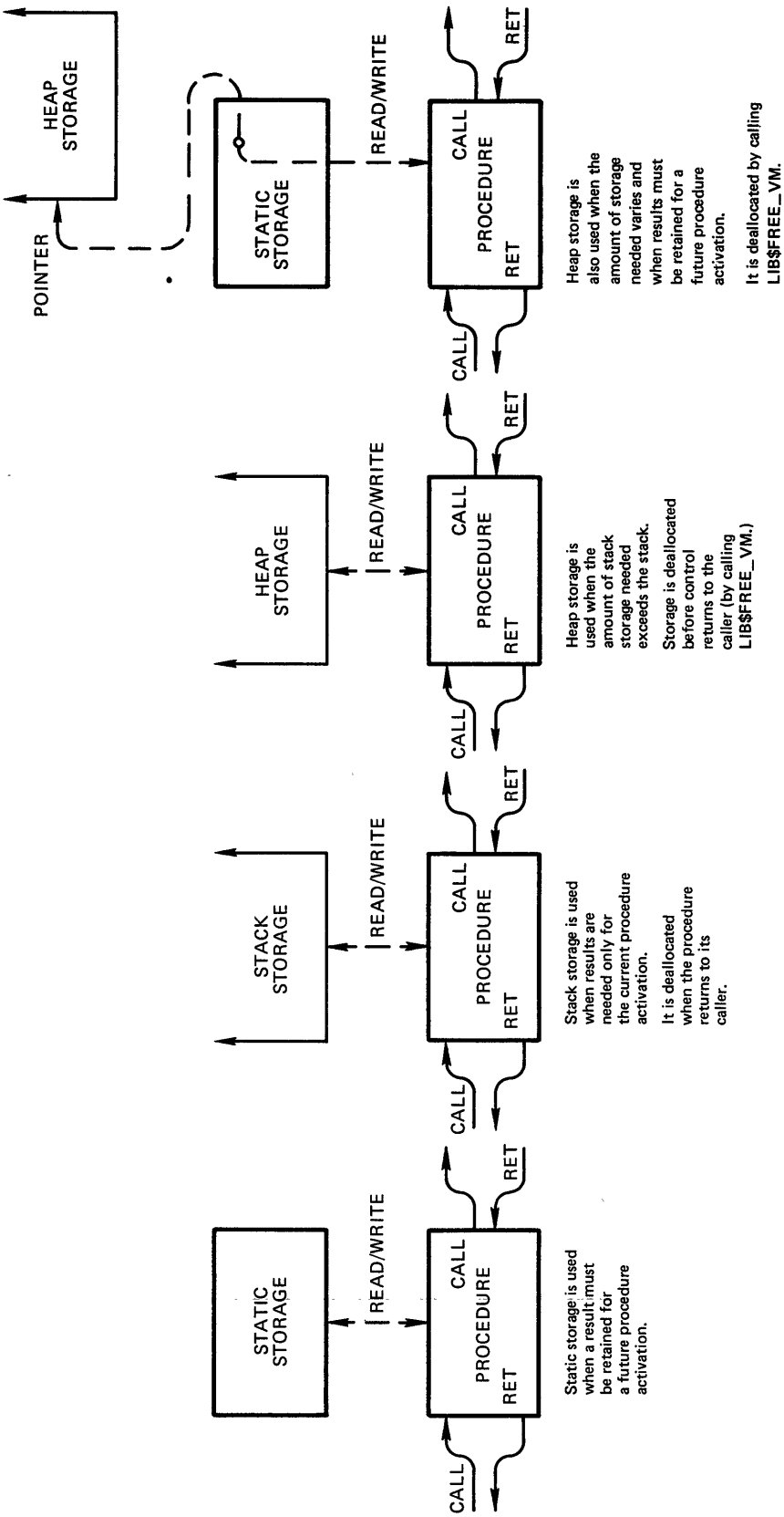


Figure 3-1 Use of Storage Types

If any of the results of a procedure must be retained 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 none of the results of a procedure activation need be retained for subsequent activations, then the procedure may use static, stack or heap storage.

Stack storage is always recommended. It is fast to allocate, and performs well in a paging system such as VMS.

Heap storage requires longer to allocate and also requires explicit deallocation. It is recommended for use instead of stack storage when the amount of space needed might exceed the stack, or when a variable amount of information must be retained after your procedure returns to its caller.

Static storage should be avoided wherever possible. It can cause unwanted side effects if it is used for implicit parameters (see Section 2.4.2), and when used, it is difficult to make your procedure 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 the use of static storage cannot be avoided, you can maintain modularity by using one of the following four techniques.

3.3.1 Pushing Down the Contents of Static Storage

Specify the interface with the calling program to consist of a sequence of calls, the first of which saves the contents of any still active implicit parameters on a push down stack in heap storage, and the last of which restores 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 within the module that automatically pushes the information stored in static storage onto a simulated software stack maintained in heap storage where it will remain during current and future procedure activations. Then write a termination procedure to automatically pop information back into static storage. When using this method, you must have the calling program also call the initialization and termination procedures. Additionally, the calling program must establish a condition handler that will call the termination procedure (that pops the data back into static storage) in case a stack unwind occurs.

For example, 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 is performed, with buffers and control blocks dynamically allocated. If an I/O statement is already being processed on another logical unit, 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 another logical unit had been "pushed," it is now "popped" back into static storage, thereby being restored as the current I/O statement.

For example, the FORTRAN statement:

```
WRITE (2) I,IFUNC(J),B
```

is compiled as:

```
PUSHL       #2                ; Unit Number
CALLS       #1,FOR$WRITE_SU   ; Initialize WRITE
                              ; sequential unformatted
PUSHAL      I                 ; Address of I
CALLS       #1,FOR$IO_L_R     ; Transmit integer
PUSHAL      J                 ; Address of ADB for A
CALLS       #1,IFUNC          ; Call function IFUNC
PUSHL       R0                ; Push function value
CALLS       #1,FOR$IO_L_V     ; Transmit by-value integer
PUSHAL      B                 ; Address of B
CALLS       #1,FOR$IO_F_R     ; Transmit Floating
CALLS       #0,FOR$IO_END     ; End of the I/O List
```

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

3.3.2 Caller Passes the Address of Storage

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

Example (in MACRO):

```
     .ENTRY  MTH$RANDOM, 0          ; no registers saved, clear IV

;+
; If this were to be placed as an inline expansion, then
; EMUL SEED, #69069,#1,R0 should replace the next two
; instructions because this would prevent the possibility
; of 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 ..  (2**24)-1
     CVTLF   R0,R0                  ; Convert to floating without
                                    ; rounding.  The result is
                                    ; positive and in the range
                                    ; 0.0 ..  (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 ..  1.0 exclusive

10$: RET

     .END
```

3.3.3 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 address of heap storage.

Example (in BLISS):

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

3.3.4 Using Static Storage in Procedures Not Needing to Retain Results

You can maintain modular standards in procedures that use static storage and do not need to retain values after control is returned to its caller by writing 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 using static storage. If your procedures are written in MACRO or BLISS, you should use stack storage exclusively when your procedure does not need to retain values from its previous activations. 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.

3.4.1 Using Stack Storage in MACRO

When using stack storage in MACRO, allocate it by subtracting the number of bytes required from the stack pointer (SP) provided on entry.

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:

The following steps take place:

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, srcl.rt.dx, src2.rt.dx)
```

```
        RESULT  = 4                     ; arg list offset for result
        SRC1    = 8                     ; arg list offset for sourcel
        SRC2    = 12                    ; arg list offset for source2

.ENTRY  LIB_CONC, ^M<R2,R3,R4,R5,R6>
        MOVZWL  @SRC1(AP), R6           ; R6 = length of sourcel 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 will keep 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
LOCAL AREA;
AREA = .HEIGHT * .WIDTH;
RETURN .AREA;
END;
```

3.5 USING HEAP STORAGE

You can use heap storage to dynamically allocate arbitrary amounts of storage. It should be used in preference to stack storage when the amount of required storage might exceed the stack size.

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) to the caller.

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

To allocate a buffer from heap storage in BLISS:

```
!+
! 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.

```
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.
!       ARG3.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:
!
!       May allocate or deallocate virtual memory.
!
!--

   BEGIN

   LOCAL
       RAN1,            !Interim random number
       RETURN_VALUE;    !Random number returned
```

```
   BUILTIN
       CBTLF;           !Convert integer (long) to floating

   IF (...DATA EQL 0)
   THEN

!+
! We must set up the data block that will remember old values for
!  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 we cannot get enough virtual memory, use the old algorithm.
!-

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

!+
!       We got the memory, now 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 with
!        value 1.0, which is invalid under all other circumstances.
!-
       BEGIN
!+
! Give the user back the latest seed so that he can run 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 will initialize
!  the data block again.
```

```
!-
        .DATA = 0;
!+
! Return the value 1.0.
!-
        CBTLF (%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 (usually) different seed.
!-
    ..DATA + 8 = .(..DATA + 8)*6909;
    ..DATA + 8 = .(..DATA +8) + 1;
    RAN1 = (.(..DATA + 8)<8, 24>);
!+
!       Reduce the 24-bit random number module 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));     !Get value from table
    (..DATA) + (.RAN1*4) = MTH$RANDOM ((..DATA) + 4); !Put another value in table
!+
!       Return to the caller of the random number generator
!        the value from the table.
!-
    RETURN (.RETURN_VALUE);
    END;                                        !of RANSUB
END

ELUDOM
```

CHAPTER 4

CODING MODULAR PROCEDURES

This chapter describes how to code modular procedures and modify existing procedures to be modular in MACRO, BLISS, and FORTRAN. The following areas are discussed:

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

Signaling and condition handling is discussed in Chapter 5.

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

4.1 STRUCTURED PROGRAMMING

Before you code individual procedures, consider how they might be grouped into modules. If you have a number of procedures that access common data or control blocks, you should also consider organizing them into separate levels, where each level has responsibility for different parts of the data base. These are called levels of abstraction.

4.1.1 Grouping Procedures

It is recommended that each module 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. It is also recommended if two or more procedures:

- Share the same static storage
- Have similar calling sequences
- Perform similar functions
- Share a significant amount of common code.

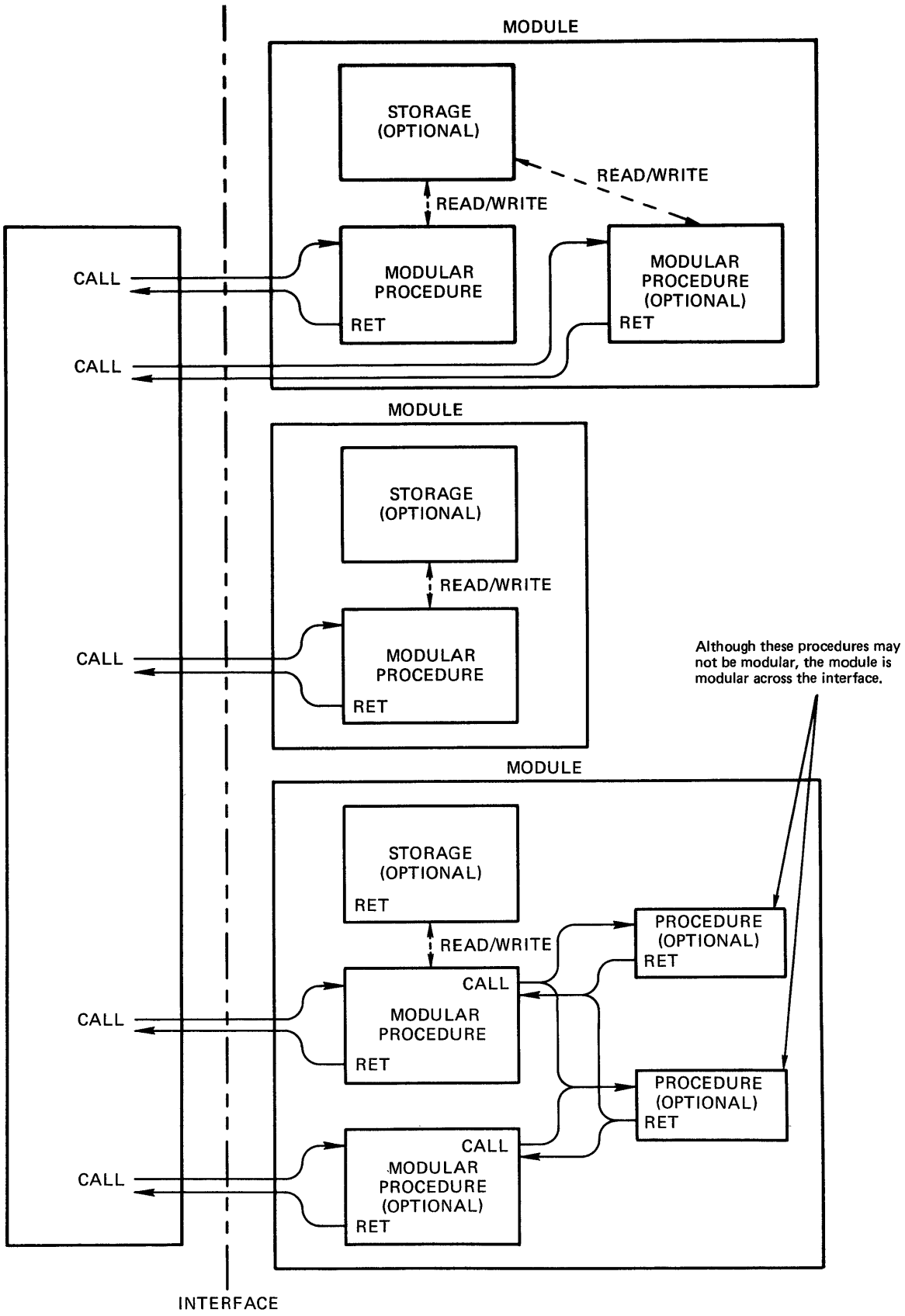


Figure 4-1 Examples of Modules

The 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 the size of your image.

Figure 4-1 shows various types of modules.

4.1.2 Levels of Abstraction

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

- Organize procedures in 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 4-2 shows the FORTRAN record I/O statement processing procedures. These are implemented in the following three levels:

- User program interface
- User program data formatting
- VAX-11 RMS interface

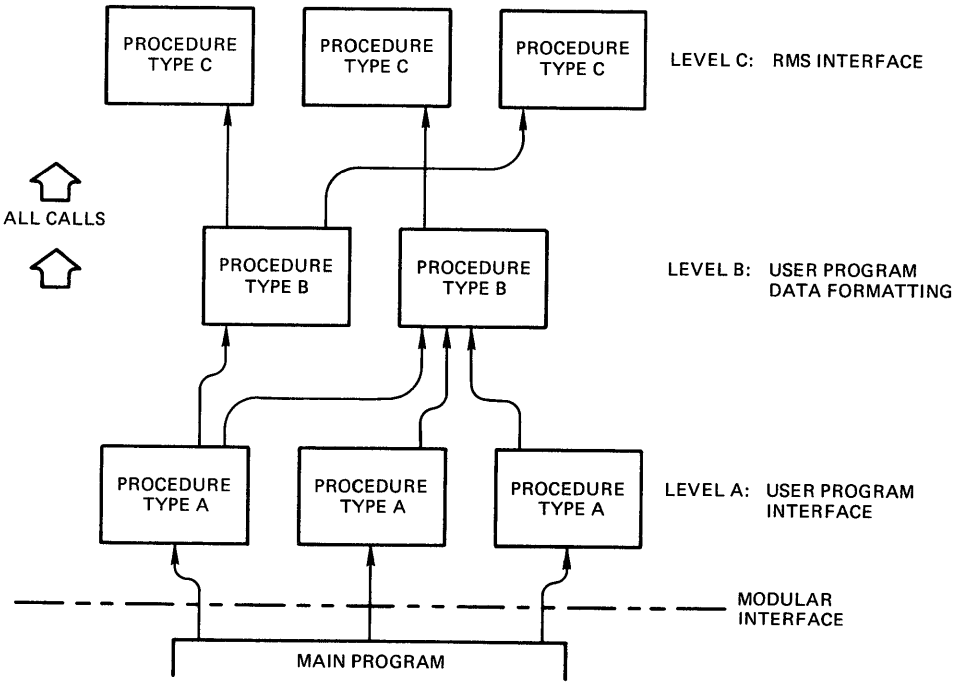


Figure 4-2 Levels of Abstraction

All calls are made in one direction: to the next highest level. It is recommended that procedures at different levels should also be in different modules.

4.2 CODING STANDARDS AND RECOMMENDATIONS

Coding standards and recommendations help maintain modularity and produce consistent software that is easy to read. You should choose simple ones. The following coding standards and recommendations are used by DIGITAL for all modular procedures. You must follow the sections marked "standard" for procedures to be modular. You may choose to follow sections marked "recommended" for procedures to be uniform.

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 modular standards since each absolute assignment might conflict with a similar assignment in another module.

4.2.2 Names for Files (recommended) and Modules (standard)

Module and file names are derived from the procedure names. If a module contains a single procedure, the file name consists of the first nine characters of the procedure name with the dollar signs and underscores eliminated. If the module contains more than one procedure, a more general name is used, composed of the facility prefix and the first noun common to all procedure names in the module.

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 library procedure are divided into two separate PSECTs with the names _fac_CODE and _fac_DATA respectively, where fac is the facility name. 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 any byte or word displacement addressing that the user program may contain. The appropriate declarations are:

in 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
```

in 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));
```

in 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 modular programming standards.

Position-independent constant data is included in the _fac_CODE PSECT to shorten the references. For example, _LIB$CODE and _LIB$DATA are the only two PSECT names used by LIB$ procedures.

4.2.4 Using Parameter Definition Files (recommended)

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

In MACRO, an auxiliary source file or macro library can be specified in the command line.

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

In FORTRAN, your source program may contain an INCLUDE statement 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 that is accessed from several modules.

4.2.5 Using Symbols vs Numbers (recommended)

Symbols rather than numbers are used as much as possible. This improves understanding and provides more information for cross-references. In 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 pointer to a general value is %UPVAL (addressable units per value) instead of 4.

4.2.6 Line Length (recommended)

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

MACRO	80
BLISS	124
FORTRAN	72

4.2.7 Using Uppercase and Lowercase (recommended)

Uppercase is used for all source code except comments, while upper- and lowercase is used for all comments. Comments that are complete sentences start with a capital letter and end with a period.

4.2.8 Using Optional Spaces (recommended)

A single space always follows a comma (no exceptions) and precedes and follows an equal sign (=). A single space precedes a left parenthesis or a left bracket (no exceptions), 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 Using Block Comments (recommended)

Blocks of statements are commented by one or more lines preceding the block. Comments start in column 1 independent of the indentation of the code. Block comments begin with a blank line to separate them from the preceding code. The first comment line contains a single plus sign (+); the last comment line contains a single minus sign (-), followed by a blank line. Exclamation marks and semicolons are followed by one space, except when followed by the first + and the last -, as shown below:

```
        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 Using Branch and Jump Instructions in MACRO (recommended)

In MACRO, code should be arranged so that branch and jump instructions refer to labels located forward in the program listing (except for loops and first-time initialization).

4.3 INITIALIZING MODULAR PROCEDURES

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

- Store a value in static storage that can only be determined at run time.
- Declare an exit handler using the $DCLEXH system service.
- Allocate a process-wide resource once.
- Open a process permanent file the first time, in case it was not already opened.

Note that initialization of dynamically allocated stack and heap data only involves writing it after each allocation before reading it.

You must perform initialization carefully to avoid violating modularity principles:

- You must perform any initialization without the calling program being aware of it. Therefore you cannot perform initialization by providing an entry point that must be called before any other entry point is called, as this would force the calling program in turn to provide an initialization entry point to its caller, etc. Also you would not be able to replace a module that does not have an initialization call with one that does, without requiring your calling programs to be reprogrammed.
- If your procedure uses LIB$INITIALIZE, you must preserve a modular environment that will not conflict with the environment established by any other procedure using LIB$INITIALIZE.

Table 4-1 shows the methods that your procedures can use to perform initialization. Each method is explained in the following sections.

Table 4-1
Methods of Initialization

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 Initialization of Storage Areas

In order 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 only happen once per 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, or
- at run time on the first call to the procedure.

4.3.2 Initialization of Static Storage

If your procedure has static storage, you will usually initialize it to zero. You do this explicitly with a data initialization statement or implicitly with the linker.

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

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

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

4.3.3 Testing and Setting First-Time Flag

Occasionally your procedure will require initialization that can only be performed at runtime. Examples are:

- Initialize static storage to a value that can only be determined at run time.
- Establish an EXIT handler

- Allocate a resource for the first time
- Open a process permanent file for the first time

These types of initialization are restricted to a first call to a procedure. To do this, your procedure tests and then sets (to 1) a statically allocated first time flag each time it is called. This flag is initialized to 0 at compile or link time. Setting and testing the flag with the VAX instruction BBSS (Branch on Bit Set and Set) will insure that initialization will be executed exactly once.

For example, your procedure may 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 LIB$SGET procedure uses this technique to initialize dynamic string storage to a set of empty queues. Each allocation of dynamic string storage is performed by first attempting to remove a pre-allocated block from the appropriate queue.

The following MACRO example is a resource-allocating procedure that keeps a single queue of quadword blocks. When it runs out of blocks, it creates more and inserts them in the queue:

```
        .PSECT  _LIB_DATA PIC,USR,CON,REL,LCL,NOSHR,NOEXE,RD,WRT
FLAG:   .LONG   0                       ; first-time flag
Q_HED:  .LONG   0,0                     ; queue header
        .PSECT  _LIB_CODE PIC,USR,CON,REL,LCL,SHR,EXE,RD,NOWRT

.ENTRY  LIB_GET_X,^M<>
        BBCS    FLAG, 10$               ; branch on call only
TRY:    REMQUE @Q_HED, R0               ; R0 = address of queue
        BVS     FILL                    ; Branch if empty and fill
        RET

;+
; Here on first call only
;-

10$:    MOVAL   Q_HED,  Q_HEAD          ; Make queue empty
        MOVAL   Q_HED,  Q_HEAD+4        ; Back pointer too
FILL:   get space for 10 quadwords by calling LIB$GET_VM
        and insert in queue using INSQUE
        BRB     TRY                     ; Try to remove one again
```

In BLISS, you can use the BUILTIN functions TESTBITSCS and TESTBITSS to test and then set a bit in one uninterruptible operation.

Another example of performing first-time initialization transparent to the caller is to establish an EXIT handler to perform some cleanup operation once 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 Making a PSECT Contribution to LIB$INITIALIZE

This method makes a PSECT contribution to PSECT LIB$INITIALIZE that contains one or more addresses of procedures to be called by the library initialization procedure LIB$INITIALIZE before the main program is called. (Examples of this method are shown in Appendix E of the VAX-11 Common Run-Time Procedure Library Reference Manual.)

Note that a module in a shareable image cannot use this method because the PSECT contribution would be to the shared image and not to the user program image. Furthermore, if this method is used, a modular procedure cannot 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 established by another procedure.

4.4 RESOURCE ALLOCATION

A resource is a part of the hardware or software system that can be allocated and deallocated. A resource is 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 upon so that many allocators may make nonconflicting allocations.

Examples of the single allocator approach are:

- The linker allocates relocatable virtual addresses among competing procedures within an image.
- The $ASSIGN system service assigns I/O channel numbers to competing procedures within a single process for each procedure that needs a separate channel.
- The library procedures LIB$GET_VM and LIB$FREE_VM allocate and deallocate virtual memory to requesting procedures in an image.

Examples of the multiple-allocator approach are:

- 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 Use of Storage with Resource-Allocating Procedures

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

4.4.2 Allocating Identification Numbers In MACRO

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

```
.TITLE  LIB_GET_INUM  -- Allocate and deallocate identifying numbers
TAB:    .WORD   0                       ; bitmap for event 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
```

Note that in order to make this procedure AST-reentrant, move the label 10$ from the MOVL instruction to the FFC instruction. (See Chapter 6).

4.4.3 Allocating Logical Unit Numbers in FORTRAN

The following FORTRAN procedure, LIB_GET_UNIT, allocates logical unit numbers:

```
        FUNCTION LIB_GET_UNIT (UNIT)
        INTEGER*4 UNIT, UNIT_TABLE(100)
        LIB_GET_UNIT = 1                ! Assume Success
        DO 10 I=1,100
          IF(UNIT_TABLE(I).EQ. 0) THEN
            UNIT_TABLE (I) = 1          ! Flag unit as in use
            UNIT = I-1
            RETURN                      ! return
        ASSIGNED
          ENDIF
10      CONTINUE
        LIB_GET_UNIT = 0                !  Indicate Failure
        RETURN

C       Deallocate Logical Unit

        ENTRY LIB_FREE_UNIT

        IF (UNIT_TABLE(UNIT+1) .EQ. 1 THEN
          UNIT_TABLE(UNIT+1) = 0                ! Flag unit as free
          LIB_FREE_UNIT = 1                     ! Indicate success
        ELSEIF
          LIB_FREE_UNIT = 0                     ! Indicate already free
        ENDIF
        RETURN
        END
```

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

```
        .
        .
        .
```

```
IF .NOT.  (LIB_GET_UNIT(I)) THEN GO TO error
OPEN (UNIT = I, ...
.
.
.
```

4.4.4 Process-wide Resources

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

Table 4-2
Allocation Methods for Resources

Resource	Allocation Method
R0, R1	not a shared resource.
R2:R15	Preserved using stack frame discipline. (See Appendix C.)
PSW	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$FREE_VM.
Static storage for nonresource allocation procedures	procedure to push old contents onto a stack in heap storage and another to pop old contents back.
	caller allocates storage.
Process-wide identifiers for static storage	procedure to assign process-wide identifiers.
Dynamic string memory	written only by calling LIB$SCOPY_DXDX, LIB$SCOPY_R_DX, OTS$SCOPY_DXDX, OTS$SCOPY_R_DX, LIB$SGET1_DD, OTS$SGET1_DD, and deallocated by LIB$SFREE1_DD, LIB$SFREEN_DD, OTS$SFREE1_DD, OTS$SFREEN_DD. (See Chapter 5 of the VAX-11 Common Run-Time Procedure Library Reference Manual.)
VMS Event Flags	Process local event flags 32-63 allocated by calling LIB$GET_EF. Process local event flags 1-23 and 32-63 may be reserved by calling LIB$RESERVE_EF and may be freed by calling LIB$FREE_EF.

Table 4-2(cont.)
Allocation Methods for Resources

Resource	Allocation Method
Condition codes (message IDs)	bits 32:16 contain the facility number. Bit 27 is 0 for those signed out by DIGITAL, and is 1 for those signed out by customers. Each allocator must insure 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 for use by users have a single "$" in them. 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.

VAX/VMS does not provide resource allocation procedures or allocation discipline for the following resources. However, if a library resource allocation procedure does not exist, you can write your own as indicated by the examples in Sections 4.4.2 and 4.4.3:

- FORTRAN logical unit numbers
- Logical Names
- Process Names
- Event flag cluster numbers 2 and 3

4.5 PASSING STRINGS AS PARAMETERS

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

For both string types, the calling program allocates the string's descriptor or passes the address of its descriptor (which is allocated 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 since the string length, string address, and data type fields appear in the same place in the two classes of descriptor. Thus, a procedure can accept either class of string. Modular procedures may read strings by any of the following methods:

- Access the length and address field indirectly through the parameter list.
- Copy the address of the string descriptor.
- Copy the contents of the string descriptor.

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 large enough, the called procedure fills the string with trailing ASCII spaces or truncates on the right.
- If truncation occurs, the called procedure may return either the success condition code LIB$_STRTRU or an appropriate error condition value as a completion status (in R0) depending upon the application.

The semantics for returning a dynamic string are:

- The called procedure may modify the string descriptor passed by the calling program only if the descriptor class code is dynamic (DSC$K_CLASS_D = 1) and only by calling the dynamic string allocation procedures (LIB$GET1_DD, LIB$SCOPY_DXDX, LIB$SCOPY_R_DX, OTS$GET1_DD, OTS$SCOPY_DXDX, or OTS$SCOPY_R_DX).
- Using the dynamic descriptor passed by the calling program, the called procedure can use either of two methods:
    1. Creates the entire string to be returned and passes it to LIB$SCOPY_DXDX, LIB$SCOPY_R_DX, OTS$SCOPY_DXDX, or OTS$SCOPY_R_DX to be copied, or
    2. Allocates the next amount of string space needed (by calling LIB$SGET1_DS or OTS$SGET1_DD using the descriptor passed by the calling program) and fills it piece-by-piece starting at the address specified in DSC$A_POINTER.
- If the resource-allocating string procedure exhausts the virtual memory for your process, it is recommended that your procedure also indicate the error to the calling program by

either returning the error condition value (in R0) (LIB$ convention) or signaling the error condition (OTS$ convention).

- The called procedure cannot make a copy of the dynamic string descriptor since its contents may 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 a fixed-length string or 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 fixed-length string semantics or the semantics indicated by the calling program in the descriptor (preferred).

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

- Method 1 (above), your procedure can always call the library procedure since it performs the semantics indicated in the descriptor.

- Method 2 (above), before calling LIB$SGET1_DD, your procedure must check the class code in the string descriptor (DSC$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 indicates the action to be taken by your procedure for all combinations of interface specification and descriptor class passed by the calling program:

Table 4-3
Procedure Action Taken on Strings Passed by Calling Program

	Interface Specification for Output String	
Calling program passes	Fixed-length semantics (–.wt.ds) (ignore DSB$B_CLASS)	Semantics specified by calling program (–.wt.dx) (observe PSB$B_CLASS)
Fixed-length string (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$A_POINTER
Dynamic string (DSC$B_CLASS=2)	Space fill or DSC$W_LENGTH and DSC$A_POINTER*	Use library dynamic string procedures

*Note that in this case the calling program must first allocate sufficient space for the dynamic string (using the library dynamic string procedures) to contain the string to be returned.

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 on to another procedure:

- If you have specified that your procedure (and any it calls) will only access the string as an input parameter, your procedure may 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) will access the parameter as an output parameter using fixed-length semantics (wt.ds), your procedure may 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) will access the parameter as an output parameter using the semantics specified by the calling program, your procedure must pass the address of the original descriptor because a dynamic string must have one and only one descriptor pointing to it.

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

4.6 USE OF VAX/VMS SYSTEM SERVICES BY MODULAR PROCEDURES

The operating system services are listed by categories in the following sections. The first column in each section indicates whether the service is modular; the numbers refer to explanatory notes in Section 4.6.13. 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.

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(1)	$DCLAST	Declare AST
yes(1)	$SETPRA	Set Power Recovery AST

4.6.3 Logical Name System Services

Logical names are stored in process-wide storage by VMS and therefore 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 Flag
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

4.6.5 Process Control Services

When using the process control services, you must specify the process name parameter as zero; otherwise, there would need to be a resource allocation procedure 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

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

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

4.6.8 Memory Management System Services

yes(11)	$EXPREG	EXPAND Program/Control Region
no(6)	$CNTREG	Contract Program Control Region
yes(18)	$CRETVA	Create Virtual Address Space
yes(18)	$DELTVA	Delete Virtual Address Space
no(7)	$CRMPSC	Create and Map Global Section
no(7)	$UPDSEC	Updata Global Section File on Disk
no(7)	$MGBLSC	Map Global Section
no(7)	$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 a person who ensures uniqueness of bits 15:3. However, for proper modularity all modular procedures must use LIB$SIGNAL (or LIB$STOP) error messages rather than actually outputting an error message. Only the catch-all handler can actually use this service.

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

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 may conflict with one that already exists. Do not use the RMS optional success and error action routines since they depend on AST interrupts being enabled even for synchronous I/O. This dependency is not made by modular procedures.

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 an event flag as an input parameter which is a process-wide resource. Process local event flags must be allocated by calling the library event flag-allocating procedures LIB$GET_EF and LIB$FREE_EF to allocate 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 would cause a conflict. Further problems result if an AST interrupt occurred while static storage was in a nondefault state.

3. A module can only deallocate items that are known to have been allocated by it.

4. No process name can be specified since there would have to be an allocation procedure to allocate it.

5. This service may 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 may 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 be derived from explicit arguments passed to a modular procedure from a nonmodular procedure or from a user via the controlling device.

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

11. If LIB$FREE_VM is used to deallocate space in the program region, LIB$GET_VM must be called in order to reuse the deallocated space. (See Chapter 5 of the VAX-11 Common Run-Time Procedure 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. In order to be AST reentrant when using $GET and $PUT, check for record stream active error (RMS$_RSA). If the error it encountered, call $WAIT and try again. (see Section 6.4).

15. In order 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. This service requires a resource-allocating procedure to allocate event flag cluster numbers 2 and 3, which are not provided for.

17. $SETPRT must only be used to change the protection of pages which were statically or dynamically allocated to your procedure.

18. $DELTVA and $CRETVA must only be used on pages which were statically or dynamically allocated to your procedure.

4.7 INVOKING OPTIONAL USER ACTION ROUTINES

An optional user action routine is a useful way to allow the calling program to gain control at a critical point within the algorithm of your procedure. To provide a user action routine, your procedure should have the following calling sequence:

```
CALL myproc (...[,action-routine.flc.rp[,user-arg.xx.x]])
```

The user action routine has the calling sequence:

```
status.wlc.v = action-routine (...[,user-arg.xx.x])
```

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 illustrates how to test for the presence of an optional user action routine and pass it the optional user arg:

```
        .
        .
        .
        MOVAQ   ..., R0                 ; R0 = adr. of string descr. for line
        CMPB    (AP), #1                ; test no. of caller parameters
        BLEQU   30$                     ; branch if no action routine specified
        TSTL    8(AP)                   ; test for 0 action routine adr.
        BEQU    30$                     ; branch if no action routine specified
                                        ; (LIB$ convention)
        CMPB    (AP), #2                ; test no. of caller parameters
        BLEQU   20$                     ; branch if no optional user-arg par.

;+
; Call user action routine with optional user-arg parameter
;-

        PUSHL   12(AP)                  ; 2nd par = user-arg list entry
        PUSHL   R0                      ; 1st par = adr. of string descr.
        CALLS   #2, @8(AP)              ; call user supplied action routine
        BRB     40$                     ; join common code

;+
; Call user action routine without optional user-arg parameter
;-

20$:    PUSHL   R0                      ; 1st par = adr. of string descr.
        CALLS   #1, @8(AP)              ; call user supplied action routine
        BRB     40$

;+
; Call LIB$PUT_OUTPUT
;-

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
```

CHAPTER 5

SIGNALING AND CONDITION HANDLING

A modular procedure should not print error or informational messages. Instead, it should use condition values to indicate success and failure, including failure type.

A modular procedure uses either of the following techniques:

- Return a condition value as a function value (preferred). (See Section 5.2.)
- Signal 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 Common Run-Time Procedure Library Reference Manual.)

When an exception condition occurs, your procedure should use one of these methods rather than output an error message directly. Otherwise, the calling program will be unable to control or change effects caused by your procedure, thereby precluding use of it in certain situations. For example, an applications program that was used by a nonprogramming clerk should output an applications-specific message (such as "Please start over") rather than a systems programming-oriented message (such as 'MRS, maximum record size invalid').

5.1 CONDITION VALUES

A condition value is a 32-bit quantity. In addition to indicating success or failure, it can provide the following information:

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

Success or failure is indicated in bit 0 as a 1 or 0, respectively. Thus, the simplest form of a condition value is a 1 or a 0, meaning success or failure, respectively.

5.2 RETURNING A CONDITION VALUE AS A FUNCTION VALUE

The following 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.
- Error contingencies are considered when the calling program is written, thereby increasing 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 In MACRO

The following example shows how to indicate success or failure in a procedure written in MACRO:

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

The following example shows how a MACRO calling program can check for success or failure in a called procedure:

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

5.2.2 Returning and Checking an Error Status in BLISS

The following example illustrates a procedure returning a success or failure status in BLISS:

```
GLOBAL ROUTINE PROC (X,Y,Z) =

    BEGIN
        .
        .
        .
    IF ...  THEN RETURN 1 ELSE RETURN 0
    END
```

The following example shows how a BLISS calling program can check for success or failure in a called procedure:

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

5.2.3 Returning and Checking Error Status in FORTRAN

The following example illustrates how a FORTRAN procedure can return a success or failure status:

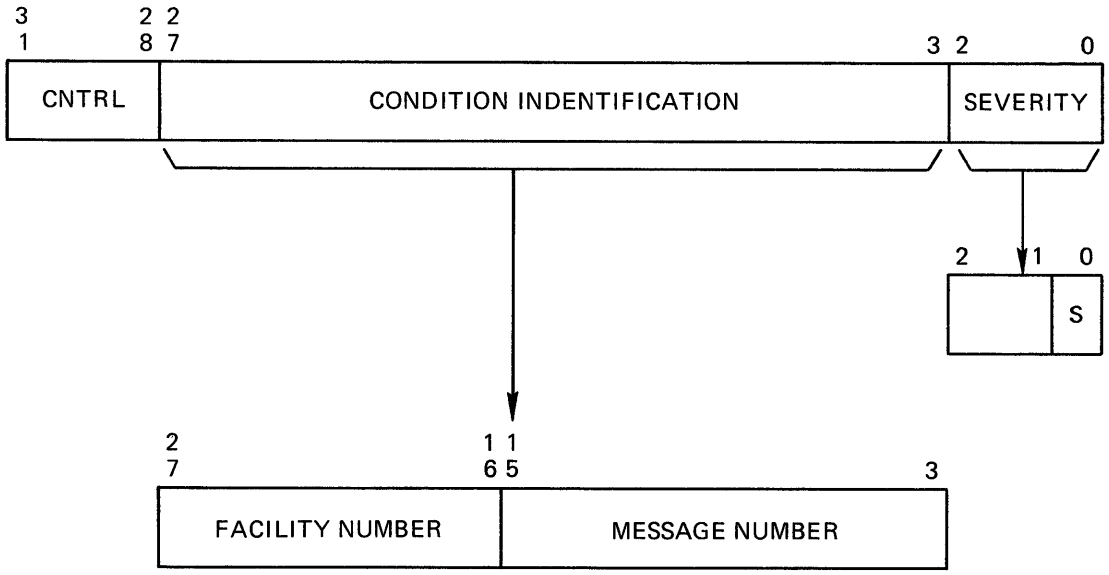
```
FUNCTION PROC (X,Y,Z)
INTEGER*4 PROC
.
.
.
IF (...) THEN
  PROC = 1
ELSE
  PROC = 0
ENDIF
RETURN
END
```

The following example shows how a FORTRAN calling program can check for a success or failure status:

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

5.2.4 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 status codes.

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 three 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_INFO	3 = information
STS$K_SEVERE	4 = severe-error

cntrl
Four control bits. The software symbols are defined for these fields in Section C.4 of the VAX-11 Common Run-Time Procedure Library Reference Manual.

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

5.2.5 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 detected by your procedure. The global symbols should have the form:

fac__error-name

You may also wish to 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, respectively.

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

In order to uniquely define condition value symbols so that neither the name nor the value can possibly be the same as that defined by another user or by DIGITAL, you must:

1. Choose a DIGITAL facility name (LIB, MTH,...) or create one. If you create one, you must register the name with a person at your installation who maintains responsibility for the uniqueness of such symbols.

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.

5. Set bits 26 through 16 to the proper facility number. (DIGITAL-supplied numbers are listed in Appendix B.)

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

```
LIB__NOSUCHFILE -  no such file
LIB__NOSUCHDEV  -  no such device
LIB__NOSUCHDIR  -  no such directory
```

in MACRO:

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
```

in 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;
```

IN FORTRAN:

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

5.2.6 Using Global Condition Values in a Calling Program

A calling program can identify a condition value returned by a procedure and take appropriate 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 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 Common Run-Time Procedure 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-val-i
: Address of longword containing the condition value to be compared with condition-value

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

The following sections show examples that use the condition values described above in a program to branch to different instructions on each different condition value. Examples are given in MACRO, BLISS, and FORTRAN both with and without the use of LIB$MATCH_COND.

in 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 that shown below:

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

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 that shown below:

```
      EXTERNAL ROUTINE
         LIB_PROC:  ADDRESSING_MODE (GENERAL),
         LIB$MATCH_COND:  ADDRESSING_MODE (GENERAL);
```

```
EXTERNAL LITERAL
   LIB__NOSUCHFIL, LIB__NOSUCHDEV, LIB__NOSUCHDIR;
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,
                LIB__NOSUCHFIL,
                LIB__NOSUCHDEV,
                LIB__NOSUCHDIR) FROM 1 TO 3
           SET
           [1]:  ...  ;
           [2]:  ...  ;
           [3]:  ...  ;
           [OUTRANGE]:  ...  ;
           TES;
```

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

```
      EXTERNAL LIB_PROC, LIB__NOSUCHFIL, LIB__NOSUCHDEV,
      1LIB__NOSUCHDIR
      INTEGER*4 LIB_PROC, COND_VAL
      .
      .
      .
      COND_VAL = LIB_PROC (...)
      IF (.NOT.  COND_VAL) THEN
        IF (COND_VAL .EQ.  %LOC(LIB__NOSUCHFIL)) THEN
          ...
        ELSEIF (COND_VAL .EQ.  %LOC(LIB__NOSUCHDEV)) THEN
          ...
        ELSEIF (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) THEN
        GOTO LIB$MATCH_COND (COND_VAL, %LOC(LIB__NOSUCHFIL),
      1 %LOC(LIB__NOSUCHDEV), %LOC(LIB__NOSUCHDIR) 20,30,40
```

5.3 SIGNALING ERROR CONDITIONS

Currently you cannot add messages to the system message file, or have a private message file. Thus, if your procedure signals a user-created condition value, no associated message is printed by the

catch-all handler as would normally be the case. Instead, only the error message number will be printed.

You can however, signal existing condition values by calling LIB$SIGNAL or LIB$STOP.

5.3.1 LIB$SIGNAL - Signal Exception Condition

LIB$SIGNAL is called whenever it is necessary to indicate an exception condition or output a message rather than return a status code to the calling program. LIB$SIGNAL 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-value).

parameters
: Optional additional FAO (formatted ASCII output) parameters for message. (passed by-value).

5.3.2 LIB$STOP - Stop Execution Via Signaling

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

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

The LIB$STOP parameters, are identical to those described above for LIB$SIGNAL.

LIB$SIGNAL and LIB$STOP are discussed in more detail including MACRO and FORTRAN examples in Section 6.6 of the VAX-11 Common Run-Time Procedure Library Reference Manual. The pattern for FAO arguments is also described so that a series of messages can be produced.

5.4 INTERNAL SIGNALING

Because you may choose to organize procedures in levels of abstraction (See Section 4.2) some procedures will not be available to the calling program across the modular interface. You may want to use internal signaling between these internal procedures.

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 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 converts 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 bit field.
- 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.

Your specific condition handler is then entered. It can decide how to proceed from this point. Usually, it will unwind to the caller of the establisher (which is the program calling across the modular interface). The signaled condition value is the value returned to the calling program that called the establisher (outermost layer). This can be done by making 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 PC of the calling program is not known by the internal signaling procedure FOR$$SIGNAL. 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 means of identifying which language the procedure was written in. This is useful for language support procedures which need to know the layout of the stack frame.

Such a handler takes appropriate language-specific action on software errors signaled by mathematics (MTH) or language support (FOR, BLI, .,) 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.

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 said to be AST-reentrant[1] if it:

- Can be interrupted between any two instructions, permitting it or any related procedure to be called (reentered)
- Will execute 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

It is recommended that all modular procedures be AST-reentrant so that they may be called from any program. If your procedure is not AST-reentrant or calls any procedure that is not AST-reentrant, your documentation should state that it is not AST-reentrant to warn others using your procedure.

6.1 AST INTERRUPTS WITHIN A PROCESS

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

---

1 The term AST-reentrant should not be confused with reentrant, which refers to a more restrictive set of conditions encountered when static storage is shared between processes (PSECT attribute SHR). In such a situation, there can be more than two threads of concurrent execution and each thread may 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 (PSECT attribute NOSHR).

While at AST level, a process cannot be interrupted again. The process runs to completion at the AST level before the non-AST level procedure resumes and is able to execute another instruction. Hence, a process is either executing at AST level or at non-AST level at any instant of time and thus consists of two "threads of execution", one thread at each level. Note that 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 (which is the operating system). This automatically continues the execution of the interrupted procedure at the point at which it had been interrupted.

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 Control ^C on the terminal.

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

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

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

6.1.1 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 the appropriate bits in the entry mask so that the contents of the registers are saved.
- If it calls any other procedures, they must all be AST-reentrant.
- It must return with a RET instruction.

6.2 WRITING AST REENTRANT MODULAR PROCEDURES

You must observe the following standards when writing AST reentrant procedures:

- Only AST reentrant procedures may 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 interruptible, an AST interrupt may also occur within that instruction. (For more information, see the VAX-11 Architecture Handbook.)

- An AST-reentrant procedure cannot call any procedures that are not AST-reentrant.
- If both an AST level and a non-AST level procedure access a data base in static storage concurrently, each procedure must make sure that a race condition[1] interference does not occur. (See Section 6.3)
- If I/O at the AST level is performed, you must be careful not to attempt simultaneous I/O of the same data base from both the AST level and non-AST level procedures. (See Section 6.4)

A procedure with no static storage is automatically AST-reentrant. Hence, it is recommended that AST-reentrant procedures use dynamic storage whenever possible.

A procedure with static storage may be AST-reentrant, although this is difficult to program since statically allocated data may be being changed 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 interferences when accessing and modifying data in its static storage:

- Perform all accessing or modification in a single uninterruptible 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, the entire modification of data in static storage can be performed in a single uninterruptible instruction. For example, you can use queue instructions at the beginning and end of your procedure to control resource allocation.

---

[1] 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 the AST and non-AST level.

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, LIB$SCOPY 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<>
        BBCS    FLAG,  10$              ; branch on call only
TRY:    REMOVE  @Q_HED,  R0             ; R0 = address of queue
        BVS     FILL                    ; branch if empty and fill
        RET

;+
; Here on first call only
;-

10$:    MOVAL   Q_HED,  Q_HEAD          ; Make queue empty
        MOVAL   Q_HED,  Q_HEAD+4        ; Back pointer too
FILL:   get space for 10 quadwords by calling LIB$GET_VM
        and insert in queue using INSQUE
        BRB     TRY                     ; Try to remove one again
```

In other applications, the static storage can be divided into two or more pieces that are each placed in a queue.

A single queue instruction can be used at the beginning of your procedure to remove one piece, and another can be used at the end to insert the piece back in the queue.

While a piece is removed from the queue, your procedure may modify data in that piece. If an AST-interrupt occurs while the piece is removed, a different piece of data will be 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 access 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 will not work if an exception condition could occur between execution of the BBSS instruction and the BBCC instruction, and 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.

- Repeat the execution of 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 VMS event flags:

```
.TITLE  LIB_GET_INUM  -- Allocate and deallocate identifying numbers 1 - 10
TAB:    .WORD   0                       ; bitmap for event 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

You can keep track of when your procedure is called if the data base is to be kept separate between each call by using a call-in-progress count. Before data base access, the count is incremented and used to index into a table of base addresses for the separate data bases. A check for depth being exceeded should be made. After the data base has been accessed, the count is decremented. This technique has an advantage over the BBxx technique in that 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, causing an eventual procedure malfunction. This can be avoided 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 may adversely affect performance of real-time programs using AST interrupts, and hence should be avoided whenever any technique described in the previous sections can be used.

The number of instructions during which the AST interrupts are disabled should be minimized. Before disabling AST interrupts, establish a condition handler to restore the AST level in case an exception or stack unwind takes place.

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

```
        .ENTRY  PROC, ^M<>
        $SETAST_S     ENBFLG=0      ; disable ASTs, R0, = SS$_WASSET or SS$_W
        .
        .
        .
        CMPL    R0, #SS$_WASSET     ; were ASTs enabled?
        BNEQ    10$                 ; branch, if not
        $SETAST_S     ENBFLG=1      ; enable AST
10$:    RET                         ; return with AST delivery restored
```

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 that you must observe in order 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. Such an error indicates that a record operation had already started for the process permanent file. This error 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. Such an 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 FORTRAN I/O support procedures use this technique so that FORTRAN I/O can be done at AST and non-AST level. The VAX/VMS Put Message system service ($PUTMSG) also uses this technique so that error message signaled at AST level will be output on SYS$OUTPUT even though the non-AST level is also calling $PUT.

- Avoid storing data in a record access block (RAB) which VAX-11 RMS may still be accessing. Your procedure may do this by one of two techniques:

    1. Allocate the RAB on the stack so that AST level and non-AST level have separate RABs.

    2. Allocate 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 may use either of the techniques described above. However, the first is more reliable than the second 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 that is 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 shareable image. Before starting, make sure the modular procedures conform to the standards 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 may use the following command to add procedures to STARLET.OLB: The general form is:

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

If you wish 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 within a module LIBCONVTIM.OBJ. The following command will add the module to the system default library STARLET.OLB:

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

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

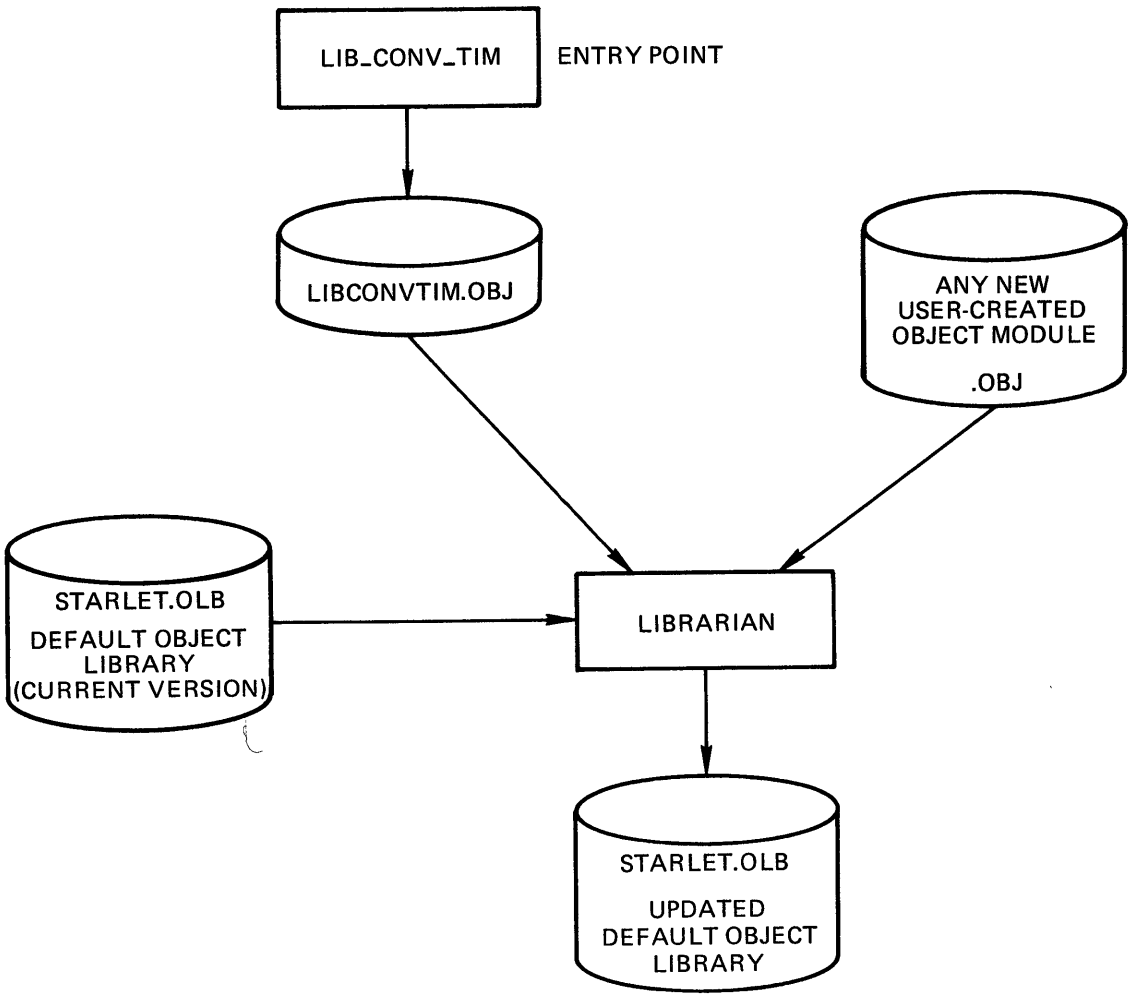


Figure 7-1 Adding a User-Created Procedure to the Default Object Library

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 shareable image is searched, and if any unresolved strong references remain. If references are found in STARLET.OLB, the linker will include 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 shareable subset of STARLET.OLB, VMSRTL.EXE.

More detailed information on both the LIBRARY and LINK commands may be found 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 written by you 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.

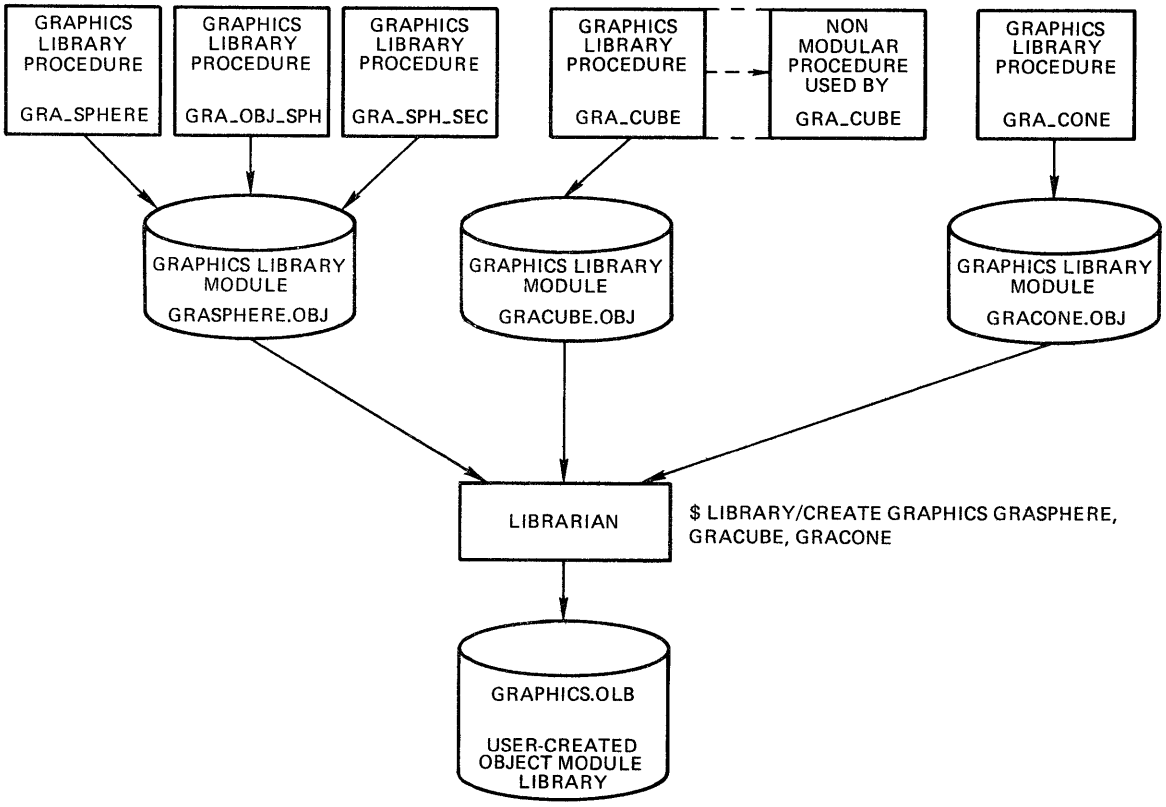


Figure 7-2 Development of a User-Created Object Module Library

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 together 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 of a library since it instructs the linker to explicitly include a specified module in the image.

Any object library specified in an application program's LINK command will be linked with 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 will be 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 SHAREABLE IMAGE

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

A shareable image may be built from either position-independent or nonposition-independent code.

The linker and image activator treat shareable images as follows: The size of each shareable image section must remain identical from one update to the next unless it is the last shareable image in the P0 address space, in which case it is free to grow. Since a position-independent piece of code will always be placed after all position-dependent code, position-independent shareable images are always placed last. Creating a position-independent shareable image will cause it to appear last in the P0 address space and thus able to

be appended. If none of the shareable images consists of position-independent code, the last image specified in the LINK command becomes the last image in the address space.

7.3.1 Creating Shareable Images in FORTRAN

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

- To use multiple shareable NONPIC images, the address space must be manually assigned to each image.
- If new versions of the image are larger than old versions, rebuilding shareable images will require relinking all programs that were bound with the old one.
- If a user-created shareable 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 may be used with FORTRAN-shareable libraries if all images are approximately the same size and padded with extra space between them. Transfer vectors enable a shareable image to be relinked without relinking all the programs that called the old version.

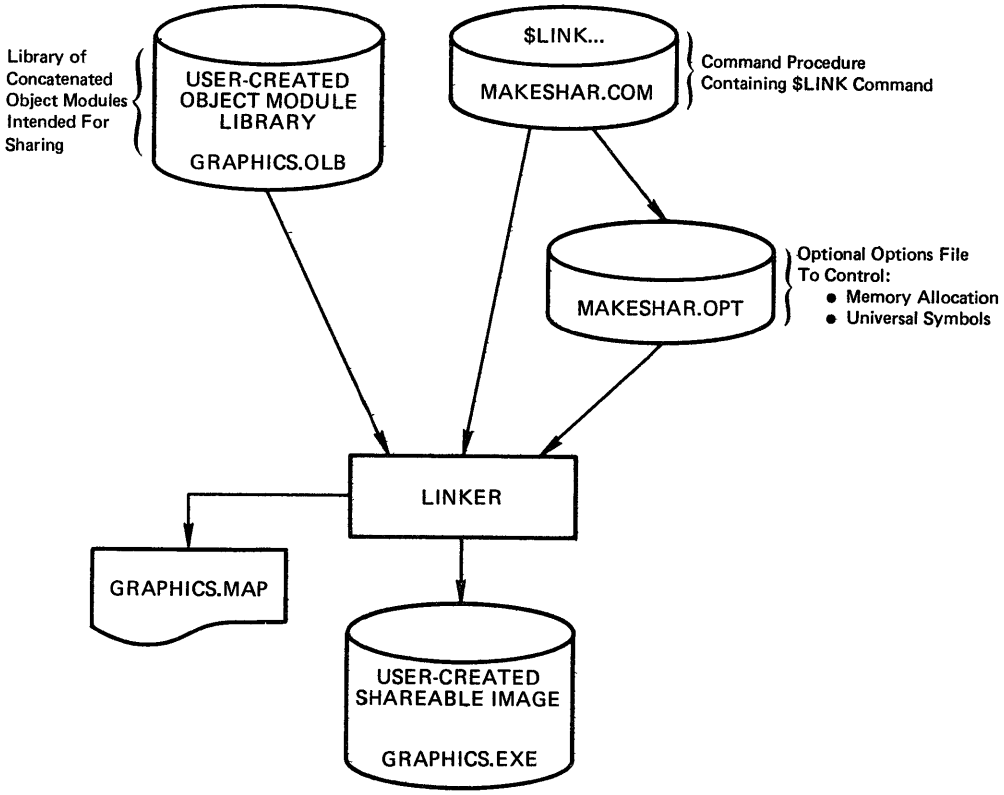


Figure 7-3 Creating a Shareable Image

7.3.2 Building and Installing A User-Created Shareable Image

Figure 7-3 shows the transformation of the user-created library GRAPHICS from an object module library to a shareable image. To do this, you must perform the following:

Create a command procedure (MAKESHAR.COM in Figure 7-3) to build the shareable image. For example:

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

where:

- /SHAREABLE instructs the linker to build a shareable image called GRAPHICS.EXE.
- /MAP/FULL produces a detailed map of the image in (by default) GRAPHICS.MAP.
- /INCLUDE is used to specify the list of objects to be taken from GRAPHICS.OLB for inclusion in this shareable image.
- MAKESHAR.OPT is an optional input file that provides additional information to the linker. Such information controls memory allocation and symbol tables. (See Chapter 6 and Section 8.3 of the VAX-11 Linker Reference Manual.)

You can optionally install your shareable image as a permanent global section. If you want to do this, refer to Section 10.2 of the VAX/VMS System Manager's Guide.

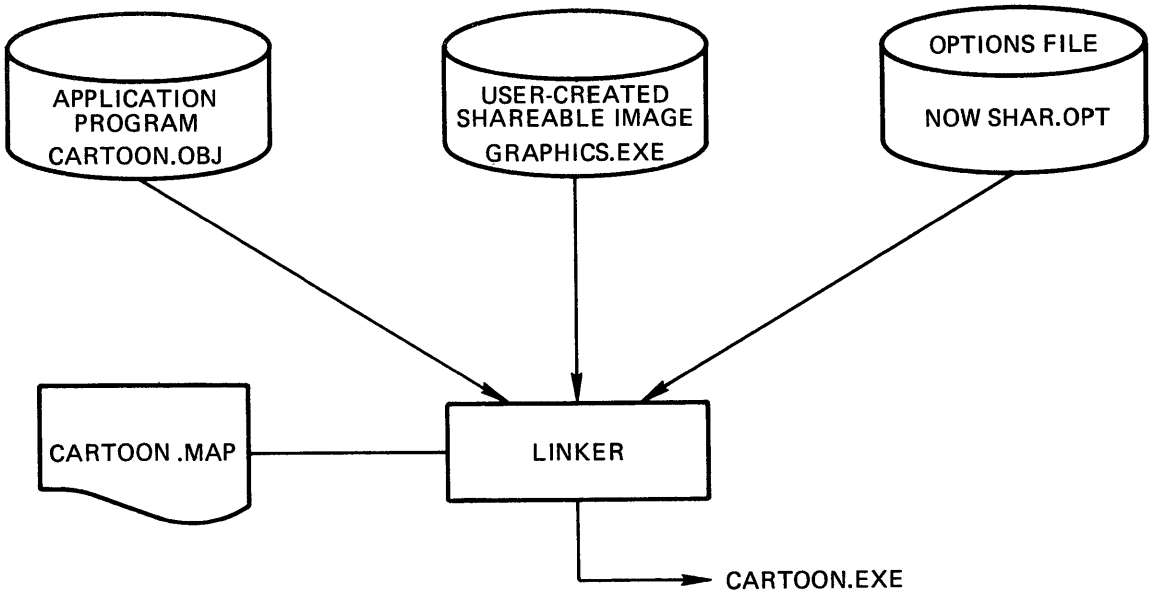


Figure 7-4 Accessing a User-Created Shareable image

7.3.3 Accessing a User-Created Shareable Image

Note that you cannot run shareable 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 shareable image GRAPHICS.EXE. To access a user-created shareable image, you must perform the following steps:

1. Create an OPTIONS file to specify the shareable 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 shareable image. Normally, during the linking of a shareable 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 shareable 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 will produce 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 is included here for clarity only.)

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 shareable 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 shareable image.
- Transfer vectors allow you to avoid relinking other programs that are bound to the shareable 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
```

```
BRW         fac_symbol+2    ;Branch to routine at instruction
                             beyond the register
                            ;save mask.
```

The JSB instruction transfer vector has the form:

```
.TRANSFER fac_symbol::
BRW 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 will automatically use transfer vectors if they are present. Regardless of the procedures' languages, code the transfer vectors as shown above. Then assemble the program containing the transfer vectors. The resulting object is used as an input to the link of the shareable image. For example, the shareable image GRAPHICS, shown above 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 shareable image.

More detailed information about transfer vectors can be found in Section 7.2.4 and is illustrated in Figures 7-3 through 7-5 in the VAX-11 Linker Reference Manual.

APPENDIX A

VAX-11 MODULAR PROGRAMMING STANDARD

This appendix is the VAX-11 standard for writing modular procedures in any language, including MACRO and BLISS. This standard is the minimum necessary so that a programmer can interface his software at the callable procedure level with that written by others and vice versa.

This standard is divided into required, optional, and recommended parts. The optional parts are indicated by asterisks (*). Any non conformance to optional parts must be indicated in the procedure's documentation. The recommended parts are documented in Section A-7. Non conformance to the recommended parts need not be documented since modularity is not affected. Each part of the standard is described in greater detail in the section or sections of this manual indicated in parentheses.

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 that does not follow this standard may cause another modular procedure (known or unknown) 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 any of the above.

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

A.1 SCOPE OF APPLICABILITY

The required, optional, and recommended parts 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 permits you to explicitly or implicitly follow the required parts of the standard for all important language features. Therefore, the compiler generated code for main programs and externally available subroutines and functions permit you to follow the required parts of the standard. Furthermore, the language support procedures conform to the required, optional, and recommended parts.

This standard applies to procedures that interface to a calling program; they do 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 REQUIRED AND OPTIONAL (*) PARTS OF THE STANDARD

The following required and optional parts 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 Common Run-Time Procedure Library Reference Manual.) Some of the following parts of the 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 FORTRAN) or implicit global data areas. Instead all parameters that are 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, FOR, BLI, B32, MTH, OTS, or any other language abbreviation (and file type) or meaningful facility name. Global entry point names that are not intended for use by the calling program have two dollar signs ($$) or three underlines (___), respectively. 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. (See Section 2.2.)

5. The form for module names is the same as that for procedure entry point names. 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.)

6. Position-independent references (within a module) to writeable data PSECTs use longword relative addressing. This is done so that the data PSECT can be allocated anywhere with respect to the code PSECT by the linker, and will link correctly no matter how many code modules are included. (See Section 4.2.3.)

7. External references use general-mode addressing so that any of the referenced procedures can be put in a shareable image without requiring change to the calling program.

8. 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 to output all messages. (See Sections 5.2, 5.3, and A.3.)

9. If a procedure requires initialization once per image activation, it is done without the caller's knowledge by: (1) the compiler at compile time or (2) the linker at link time or (3) testing and setting a statically allocated first-time flag on each call or (4) making a PSECT contribution to LIB$INITIALIZE. (See Section 4.3.)

   Using LIB$INITIALIZE is not recommended since your procedure cannot be placed in a shareable image. Furthermore, a procedure must not use LIB$INITIALIZE to establish a condition handler before the main program is called if it might interact with other condition handlers established before the main program.

10. 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 prevent needless exhaustion of resources, a procedure that requests allocation of a resource:

    - 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 (1) virtual memory in the program region, (2) dynamic string memory, and (3) process-local event flags. (See Section 4.4 and Chapter 5 of the VAX-11 Common Run-Time Procedure Library Reference Manual.)

11. For each input and output string parameter (or string function value) the calling program either: (1) allocates a descriptor, or (2) passes the address of a descriptor passed to it. A procedure accesses a formal[1] 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
    - (Least preferred) 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.

[1] 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 using the starting address and length specified in the descriptor passed by the calling program with space filling or truncation on the right. The descriptor is not modified.

- Dynamic string semantics: The formal string is either (1) written by passing the address of the formal string descriptor and the string to be copied to LIB$SCOPY_DXDX, LIB$SCOPY_R_DX, OTS$SCOPY_DXDX, or OTS$SCOPY_R_DX, or (2) allocated by calling LIB$SGET1_DD or OTS$SGET1_DD and written in pieces. Only the length and address of the descriptor is modified and only by any of the above dynamic string resource allocation procedures.

The two methods that you may 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).

- (Preferred) Use the semantics indicated in the descriptor passed by the calling program. If DSC$B_CLASS contains DSC$K_CLASS_S=1 or DSC$K_CLASS_Z=0, use fixed-length string semantics. If DSC$B_CLASS contains DSC$K_CLASS_D=2, use dynamic string semantics.

A procedure cannot require its caller to pass a dynamic string descriptor. (See Section 4.5.)

12. Some procedure interface specifications retain results from one call to the next, even though the procedures are not resource allocating procedures. 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 of which 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. The remaining procedures are passed to this parameter explicitly by the calling program, and the last of these deallocates any heap storage or process-wide identifying value. (See Sections 2.5.2.2, 2.5.2.3, and 3.3.3.)

    - The procedure's caller allocates all storage and passes the address on each call. (See Sections 2.5.2.1 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 2.5.1.)

- The interface specification consists of a sequence of calls to a set of one or more procedures, the first of which saves the contents of any still active static storage on a push down stack in heap storage, and the last of which 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 3.3.1.)

13. 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 be allocated contiguously to the first. (See Sections 2.4.2 and 4.6.8)

14. * A procedure executes in any VAX-11 access mode and at any address. (You should not assume that address bit 31 is always 0.)

15. * The storage for input and output parameters may overlap at the option of the calling program. Therefore, a procedure is programmed to behave the same regardless of whether there is overlap.

16. * 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 action routines are not used. (See Section 4.6.12.)

17. A procedure provides an interface to its callers that allows the callers to follow all required parts of this standard.

18. A procedure does not call other procedures or system services such that the resulting combination violates any required part of this standard from the point of view of the calling program. A procedure may call other procedures or system services that do not follow optional parts of this standard. However, if the resulting combination as seen from the calling program does not follow the optional parts, the calling procedure must indicate such non-conformance in its documentation. (See Section 4.6.)

19. A procedure makes no assumptions about its environment other than those of this standard.

A.3 FACILITY SPECIFIC REQUIRED AND OPTIONAL (*) PARTS OF THE STANDARD

The following parts apply to procedures that are part of a specific library facility.

The facility names below are used to represent the corresponding library facilities:

LIB General Utility and Resource Allocation Procedures
MTH Mathematics Procedures
OTS Language Independent Support Procedures
FOR FORTRAN-specific Support Procedures
BLI Transportable BLISS-specific Support Procedures
B32 VAX-11 Unique Native Mode BLISS-specific Support Procedures

20. The PSECT declarations for library code and data respectively are:

    in 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
    ```

    in BLISS:

    ```
    _fac$CODE READ, NOWRITE, EXECUTE, SHARE, PIC, CONCATENATE,
    ADDRESSING_MODE (GENERAL)

    _fac$DATA READ, WRITE, NOEXECUTE, NOSHARE, PIC,
    CONCATENATE, ADDRESSING_MODE (LONG_RELATIVE)
    ```

    Note that the leading underline is sorted last by the linker so that library modules cannot cause truncation errors due to byte or word displacement addressing performed by the user program.

    (In the examples above, user PSECTS replace $ with _) (See Section 4.2.3.)

21. A procedure's caller may indicate omitted trailing optional parameters either by passing argument list entries that contain zero, or by passing a shortened argument list.

22. 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.)

LIB Procedures:

23. LIB procedures pass arrays and strings by-descriptor and input scalars by-reference. LIB procedures can pass parameters by-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. (See Part 11 above and Sections 2.3.2, 2.3.3, and 4.5.)

24. LIB procedures return error conditions to the caller using completion codes returned in R0 as a function value rather than signaling. (See Section 2.3.6 and Chapter 5.)

MTH Procedures:

25. MTH procedures pass input scalars by-reference. (See Section 2.3.2.)

26. MTH procedures signal errors since the function value (R0) is used to return a mathematical value. (See Section 2.3.6 and Chapter 5)

Higher Level Language-specific Support Procedures (FOR, B32, BLI):

27. If a particular functional capability already exists in a LIB, OTS, or MTH facility, then those procedures should be called by other language-specific procedures rather than implementing their own algorithms.

28. Higher-level language support procedures pass input scalars by-value if 32 bits or less, input scalars by-reference if exceeding 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.)

29. 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 retains the error transfer address in the first of a series of calls, and transfers directly to it on an error after removing the stack frame. (See Section 2.3.6 and Chapter 5.)

Language-Independent Support Procedures (OTS):

30. Language-independent support procedures follow the language-specific support procedures parts of this standard described above.

31. 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 Part 11 above and Section 4.5.2.)

A.4 * AST-REENTRANT PROCEDURES (OPTIONAL)

The following parts are required for all AST-reentrant procedures. To be AST-reentrant, a procedure allows any procedure (including itself) to be called between any two instructions. This other procedure may be an AST-level procedure, a condition handler, or another procedure (see Chapter 6). A procedure that uses no static storage and calls only AST-reentrant procedures is automatically AST-reentrant. (See Part 12 above for ways to eliminate the use of static storage.)

32. * 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 uninterruptible 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 that is incremented upon entry to the procedure and decremented upon return. The count is used as an index into separate 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 may 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.)

33. * 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 * SHAREABLE IMAGES (OPTIONAL)

The following additional parts are required for procedures that are to be included in a shareable image. A procedure that adheres to the following parts can be included in a shareable image at any time.

34. * A procedure's code is position-independent.

35. The data need not be position-independent. However, for improved performance, it is recommended that it be initialized to 0 to avoid either position-independent contents or position-dependent addresses. (See Section 4.3.1.)

36. A procedure cannot use LIB$INITIALIZE to initialize data since a shareable image cannot make a PSECT contribution to a user program at link time. (See Section 4.3.4.)

A.6 * UPWARDS COMPATIBLE SHAREABLE IMAGES (OPTIONAL)

To be compatible with all future versions of the shareable image, shareable image procedures follow these additional parts of the standard:

37. A procedure's entry points are vectored. (See Section 7.4.)

38. A procedure's code and data is position-independent. Because the operating system 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 parts of the standard are recommended in the hope that modular procedures will be similar in form and format and thereby easier to be used by others. However, nonconformance will not affect modularity and need not be documented.

39. 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.)

40. A procedure should not have static storage unless it is a process-wide resource-allocating procedure, or must retain results for implicit inputs on subsequent activations. Most of the techniques used in Part 12 above avoid the use of static storage. If a procedure cannot eliminate the use of static storage and does not need to retain information from one procedure activation to the next, it writes each static storage location before using it. (See Sections 2.5 and Chapter 3.)

41. 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. 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.)

42. A procedure that allocates process-wide resources provides an entry point that shows the state of the resource (for degugging and performance statistics.) If such an entry point produces human readable output to a file or device, it must conform to part 41. (See Sections 2.7, 4.4, and 4.7.)

43. Timing procedures and resource allocation procedures should make statistics available for performance evaluation and degugging (See Section 2.7.) Such procedures should provide two entry points that accept an input parameter code (1,...,n) indicating the desired statistic and that return a completion status in R0:

    fac$SHOW_name ** Provides formatted strings according to Part 41 above. A 0 input parameter code requests all available statistics. If the calling program does not supply the optional action routine parameter, the string(s) are output to SYS$OUTPUT.

    fac$STAT_name ** Returns the binary value of the desired statistic.

    ** (User versions use _ instead of $)

44. 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.)

45. 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 placed in the same module. (See Section 4.1.)

46. 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.)

47. File names should be identical to the first nine characters of the module name, with $ and _ characters omitted. (See Section 4.2.2.)

48. 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 the following techniques:

    in MACRO: Macro library file
    in BLISS: REQUIRE or LIBRARY file
    in FORTRAN: INCLUDE file

49. Procedure names should have the form of a verb followed by the object of the action: for example, LIB$GET_VM and LIB$FREE_VM. (See Section 2.2.)

50. JSB calling sequences should be avoided as they are not available to most languages. When a procedure uses a JSB entry point, it should also provide an equivalent CALL entry point.

51. Instructions and statements are uppercase, comments are in upper- and lowercase. A space follows every comma, semicolon, and exclamation point. A space precedes a left parenthesis but not a left angle bracket or square bracket. Block comments start in column 1 and have the following form (use ; or ! depending on the language):

```
<blank line>
;+
;  Put one or more lines of block comment here
;-
<blank line>
```

52. Use symbols rather than numbers in the body of the procedure. (See Section 4.2.5.)

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 which are global (known to the linker) or which appear in parameter or macro definition files.

These public names are all constrained to follow these rules for the following reasons:

- Using reserved names ensures that customer-written software will not be invalidated by subsequent releases of DIGITAL products which add new symbols.
- Using definite patterns for different uses allows you to judge the type of object being referenced. For example, the form of a macro name is different from that of an offset, which is different from that of a status code.
- Using certain codes within a pattern associates the size of an object with its name. This increases the likelihood that the reference will use 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 are allowed to choose facility codes names which will not conflict with one another.

Never define local synonyms for public symbols. The full public symbol should be used in every reference to give maximum clarity to the reader.

B.1 PUBLIC SYMBOL PATTERNS

All DIGITAL public symbols contain a currency sign. Thus, customers and applications developers are strongly advised to use underscores instead of currency 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 within 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; in the case of internal names, a double currency sign convention can be used such as in (3) or (5) below.

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 appear in the system macro library and represent a call to one of many facilities. The facility name usually does not appear 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 which 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 that are intended for use only within 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 registers R0 to Rn are not preserved. Note that the caller of such an entry point must include at least registers R2 through Rn in its own entry mask so that a stack unwind will restore all registers properly.

7. Global variable names are of the form:

   facility$Gt_variablename

   The letter G stands for global variable and the t is a letter representing the type of the variable as defined in Section B.2.

8. Addressable global arrays use the letter A (instead of the letter G) and are of the form:

   facility$At_arrayname

   The letter A stands for global array and t is one of the letters representing the type of the array element according to the list in Section B.2.

9. In the assembler, public structure offset names are of the form:

   structure$t_fieldname

   The t is a letter representing 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 descriptor 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. For 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. Most typically, this name is definable as

```
MACRO
     structure$V_fieldname =
         structure$t_fieldname,
         structure$V_fieldname,   !assembler meaning
         structure$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; rather 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 are the letters used for the various data types or are reserved for the following purposes:

Letter	Data Type or Usage
A	address (*)
B	byte integer
C	single character (*)
D	double precision floating
E	reserved to DEC
F	single precision floating
G	general value (*)
H	integer value for counters (*)
I	reserved for integer extensions
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 respectively when transportability is involved. The following table defines their sizes:

Letter	16	32	36
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

Following is a list of all the facility prefixes for DIGITAL-supplied software. This list will grow over time as new facility prefixes are chosen. No one should use a new code without registering it in a common place.

Prefix	Facility	Interface Type	Condition <31:16>
B32	BLISS-32 support library	V	27
BLI	BLISS transportable support library	V	20
C74	COBOL - 74	V	29
FOR	Fortran support library	V	24
MTH	Math library	F	22
OTS	Language independent Object Time System	V	23
RMS	RMS internals and status codes	V	1
SORT	VAX-11 SORT	any	28
SS	System Service Status Codes	-	0
XPO	BLISS transportable	v	32

Individual products such as compilers also get unique facility codes formed from the product name. They must be signed out in the above list. Facility prefixes should be chosen 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 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.

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

In order to achieve the VAX-11 goal of being able to mix langauges within 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 Common Run-Time Procedure 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-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.

Within any given facility, it is generally preferable to have only one style of these interface choices. These are defined below. 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 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 between these methods is dependent on the environmental assumptions made in the design of the procedure.

The most common combinations of interface specifications are given in the following table. The column "Scalars" shows how scalars are passed. The column "Strings" shows how output strings are returned. The column "Function" shows what kind of function value is returned.

Type of call	Instruction	Passing Scalars	Output String	Function Value
J (non-CALL)	JSB	in register	-	-
V (by Value)	CALL	AP by value	length,descr	.lc
F (Function)	CALL	AP by reference	none	scalar
FORTRAN	CALL	AP by reference	fixed	any
COBOL	CALL	AP by reference	fixed	none

C.2 NOTATION FOR DESCRIBING PROCEDURE PARAMETERS

A concise language independent notation is used to describe 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 (value, reference, descriptor)
5. The form of the parameter (scalar, array,...)

Note that if a parameter is an address which is saved for later access by another procedure, the notation should reflect the ultimate access which will be 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 may) make to the argument:

   r - parameter may be read only.

   m - parameter may be modified, i.e., read and written.

   w - parameter may 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, e.g., 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. No <data type> field is given since the argument is a sequence of instructions.

   f - parameter is an address of a function to be (optionally) CALLed without unwinding the stack. The <data type> field indicates the data type of the function value.

   a - reserved for use in the System Reference Manual (address). Not used here since the object pointed to is specified.

   b - reserved for use in the System Reference Manual (branch destination). Not used here since a branch destination cannot be a procedure formal.

   v - reserved for use in the System Reference Manual (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. See Appendix C for the numeric codes assigned to these data types for use in descriptors, parameter validation, etc.

Letters	Use
z	Unspecified
v	Bit (variable bit field)
bu	Byte Logical (unsigned)
c	Single character
u	Smallest unit for addressable storage

Letter	Use
wu	Word Logical (unsigned)
lu	Longword Logical (unsigned)
a	Absolute virtual address
cp	Character pointer
lc	Longword containing a completion code
qu	Quadword Logical (unsigned)
b	Byte Integer (signed)
arb	Byte containing a relative virtual address (*)
w	Word Integer (signed)
h	Integer value for counters
arw	Word containing a relative virtual address (*)
l	Longword Integer (signed)
g	General value
arl	Longword containing a relative virtual address (*)
q	Quadword Integer (signed)
f	Single-Precision Floating
d	Double-Precision Floating
fc	Complex (Floating)
dc	Double-Precision Complex
t	text (character) string
nu	Numeric string, unsigned
nl	Numeric string, left separate sign
nlo	Numeric string, left overpunched sign
nr	Numeric string, right separate sign
nro	Numeric string, right overpunched sign
nz	Numeric string, zoned sign
p	Packed decimal string
x	Data type indicated in descriptor
zi	Sequence of Instructions (parameter validation)
zem	Procedure Entry Mask (parameter validation)

* - arl, arw, and arb is a self-relative address 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 mechanism that the called routine expects:

   v - value, i.e., call-by-value where the contents of the parameter list entry is itself the parameter of the indicated data type. Note that call-by-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, i.e., 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, <passing form> must be absent. If the parameter is an array, <passing form> must be present.

   d - descriptor, i.e., 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 Common Run-Time Procedure Library Reference Manual. Note that when <passing mechanism> is d, <arg form> must be present to indicate the type of descriptor.

5. <parameter form> is a letter denoting the form of the argument:

   Null means scalar of indicated data type.

   a - array reference or array descriptor, i.e., call-by-reference or call-by-descriptor as indicated by <arg 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; Appendix C of the VAX-11 Common Run-Time Procedure Library Reference Manual.

   s - scalar descriptor, i.e., call-by-descriptor where 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.

   d - dynamic string descriptor, i.e., 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.

   p - Procedure descriptor, i.e., 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.

   pi - Procedure incarnation descriptor, i.e. passed-by-descriptor which is identical to a procedure descriptor with the addition of its call frame address. This is used to refer to a specific incarnation of a procedure, such as ALGOL or PL/I.

   j - Label descriptor i.e., passed-by-descriptor for a label specifying the start of its code.

   ji - Label incarnation descriptor, i.e., passed-by-descriptor. This is identical to a label

descriptor with the addition of its call frame address. This is used to refer to a specific incarnation of a procedure, such as ALGOL or PL/I.

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.

C.2.2 Optional Parameters And Default Values

The caller may omit optional parameters at the end of a parameter list by passing a shortened list. The caller may also omit optional parameters anywhere by passing a 0 value as the contents of the parameter list entry. However, a caller may not 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].r [,end.].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 has 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 may be repeated one or more times are indicated using ellipses, e.g., CALL FOR$OPEN (keywd.rw.v,info.rl.v...). Repeated parameters that may be omitted entirely are indicated within ellipses inside square brackets, e.g., CALL FOR$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 may be omitted and that (2) err and end may 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 in the above.

Common combinations are:

completion code:	Status.wlc.v =...
longword call-by-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

C.2.5 Summary Chart of Notation

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

<access type>

r	Read
m	Modify
w	Write
j	RET and JMP
c	RET and CALL
s	sub CALL
f	function CALL

<data type>

z	Unspecified
v	Bit (variable bit field)
bu	Byte Logical (unsigned)
c	Single character
u	Smallest unit for addressable storage
wu	Word Logical (unsigned)
lu	Longword Logical (unsigned)
a	Absolute virtual address
cp	Character Pointer
lc	Longword containing a completion code
qu	Quadword Logical (unsigned)
b	Byte Integer (signed)
arb	Byte-sized relative virtual address
w	Word Integer (signed)
h	Integer value for counters
arw	Word-sized relative virtual address
l	Longword Integer (signed)
g	General value
arl	Longword-sized relative virtual address
q	Quadword Integer (signed)
f	Single-Precision Floating
d	Double-Precision Floating
fc	Complex (Floating)
dc	Double-Precision Complex
t	text (character) string
nu	Numeric string, unsigned
nl	Numeric string, left separate sign
nlo	Numeric string, left overpunched sign
nr	Numeric string, right separate sign
nro	Numeric string, right overpunched sign
nz	Numeric string, zoned sign
p	Packed decimal string
x	Data type indicated in descriptor
zi	Sequence of Instructions
zem	Procedure Entry Mask (arg validation)

NOTATION FOR DESCRIBING PROCEDURE PARAMETERS

<passing mechanism>	<parameter form>
v Value	<null> scalar
r Reference	a array
d Descriptor	s fixed-length string
	d dynamic string
	p procedure
	pi procedure incarnation
	j label
	ji label incarnation
	x fixed-length or dynamic string as specified in descriptor

INDEX

VAX-11 Guide to Creating
Modular Library Procedures
AA-H500A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of 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

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

— — Do Not Tear - Fold Here —