digital

Guide to Creating OpenVMS Modular Procedures

OpenVMS

Part Number: AA–PV6AA–TK

Guide to Creating OpenVMS Modular Procedures

Order Number: AA–PV6AA–TK

May 1993

This manual describes how to create a complex application program by dividing it into modules and coding each module as a separate procedure.

Revision/Update Information: This manual supersedes the *Guide to Creating OpenVMS Modular Procedures*, Version 1.0.

Software Version: OpenVMS AXP Version 1.5
OpenVMS VAX Version 6.0

Digital Equipment Corporation
Maynard, Massachusetts

May 1993

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK4518

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

5 Integrating Modular Procedures

6 Maintaining Modular Procedures

A Summary of Modular Programming Guidelines

Index

Examples

Figures

Tables

Preface

Intended Audience

This manual contains guidelines for developing, integrating, and maintaining modular procedures. It is intended for advanced system and applications programmers who are already familiar with OpenVMS operating system concepts. Readers should also be proficient in at least one supported language.

Document Structure

This book contains the following chapters and appendix:

- Chapter 1 defines modular procedures and discusses the benefits of modular programming.
- Chapter 2 covers design topics, such as organizing new applications, designing a modular procedure interface, using system resources, using input/output, writing internal documentation, and planning for signaling and condition handling.
- Chapter 3 presents general coding guidelines and information about initializing modular procedures. It also discusses guidelines for invoking optional user-supplied action routines, and writing AST-reentrant code.
- Chapter 4 describes methods for testing procedures for modularity, language-independence, and reentrancy. This chapter also provides general information about performance testing and monitoring procedures.
- Chapter 5 shows you how to create object module libraries, shareable images, and shareable image libraries from your completed procedures.
- Chapter 6 covers maintenance topics, such as upward compatibility, regression testing, updating procedures and procedure libraries, and changing the transfer vector or linker options file.
- Appendix A summarizes the modular programming guidelines presented in this manual.

Associated Documents

The following manuals contain more information about the programming tasks described in this book:

- *OpenVMS Programming Environment Manual*
- *OpenVMS Programming Concepts Manual*
- *OpenVMS Programming Interfaces: Calling a System Routine*
- *OpenVMS Calling Standard*
- *OpenVMS System Services Reference Manual*

- *OpenVMS Linker Utility Manual*
- The documentation set for your language processor

Conventions

In this manual, every use of OpenVMS AXP means the OpenVMS AXP operating system, every use of OpenVMS VAX means the OpenVMS VAX operating system, and every use of OpenVMS means both the OpenVMS AXP operating system and the OpenVMS VAX operating system.

The following conventions are used to identify information specific to OpenVMS AXP or to OpenVMS VAX:

AXP	The AXP icon denotes the beginning of information specific to OpenVMS AXP.
VAX	The VAX icon denotes the beginning of information specific to OpenVMS VAX.
♦	The diamond symbol denotes the end of a section of information specific to OpenVMS AXP or to OpenVMS VAX.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*), command lines (for example, /PRODUCER=*name*), and command parameters in text.
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.

1

Introduction to Modular Procedures

A procedure is a set of related instructions that performs a task. A module is a single body of code and text that can be assembled and compiled as a unit.

A procedure is modular if it contains all the definitions and calls it needs to perform a task. A modular procedure must also follow rules and principles that permit it to be successfully linked together with other procedures that follow the same rules and principles.

This chapter briefly discusses:

- Programming benefits of modular procedures
- Invoking modular procedures
- Using procedure libraries
- Existing OpenVMS system procedures
- Using translated images

1.1 Why Bother with Modular Procedures?

Procedures can be combined to form programs in the following ways:

- Your procedure calls other procedures
- Other procedures call your procedure
- A calling program calls either your procedure or other procedures

For procedures to execute successfully when they are combined to form a program, they must follow general guidelines. Modular procedures that do not follow these guidelines can cause other procedures in the program image to execute incorrectly.

The modular programming guidelines in this manual are designed to give programmers a common environment in which to write code. If all programmers follow these guidelines, then any modular procedure can be added to a procedure library without conflicting with procedures already in the library or with any that are added later.

Modular programming offers the following advantages:

- You can use any modular procedure in any program
- You can add a modular procedure to a library at any time
- You do not need to rewrite common algorithms for a new program
- You can reduce development time and complexity, and increase reliability

- You can modify or replace a procedure without modifying the calling program provided that you adhere to the guidelines for maintaining upward compatibility.
- You can control processwide resource allocation
- You can use different programming languages to write different procedures for a program

Many of the guidelines in this manual are recommendations, not requirements. By following all the guidelines, however, you can realize the following additional advantages:

- Shareable library procedures can save memory space, disk space, 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; therefore, they are easier to use and maintain

1.2 Invoking a Modular Procedure

Typically, you invoke a procedure by executing a VAX CALLS or CALLG instruction (on VAX systems) or JSR instruction (on AXP systems). If you are using a high-level language, the compiler generates the appropriate transfer instruction when you use the conventions required by your language to implement a procedure.

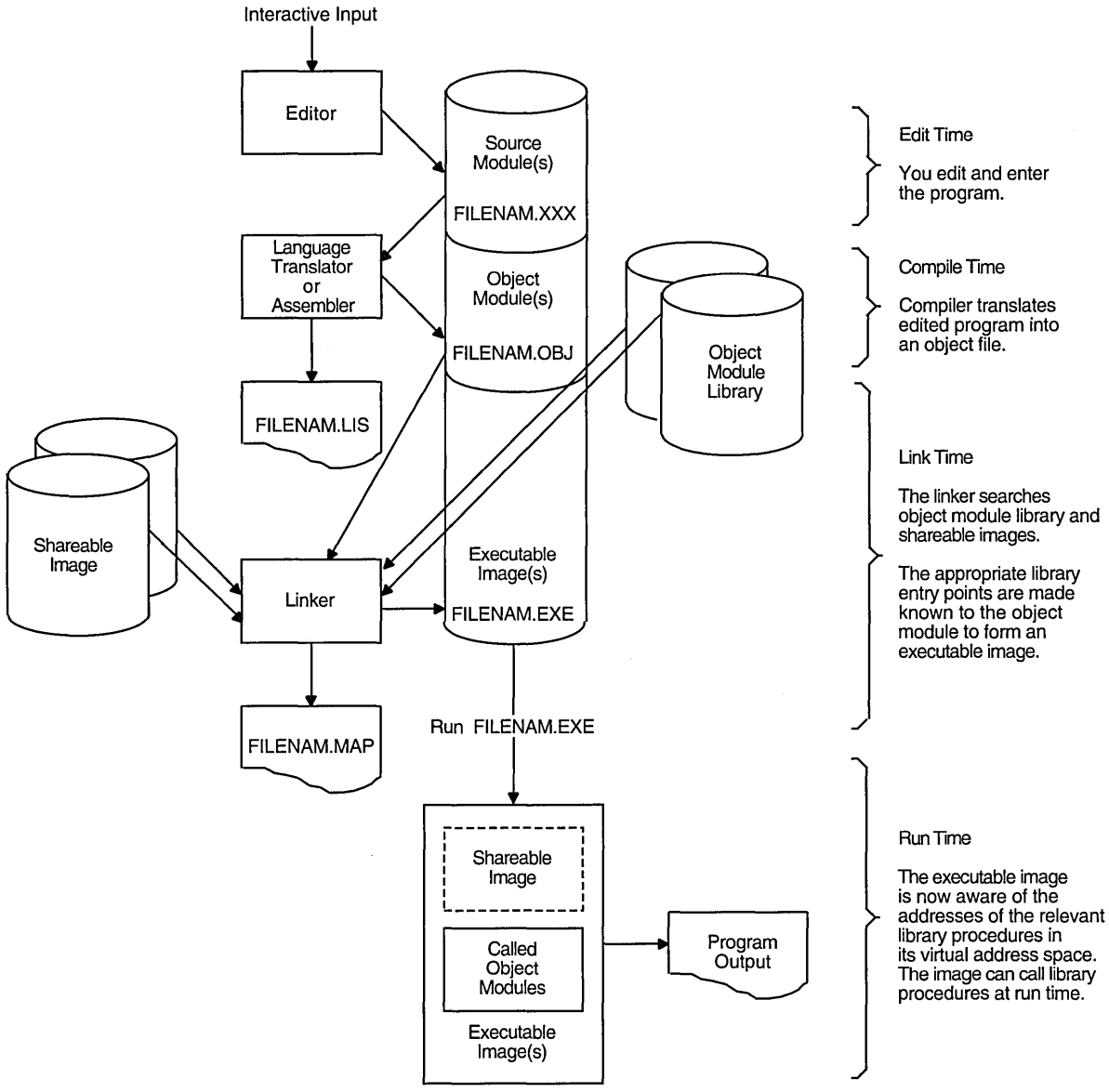
For more information about calling sequences, refer to *OpenVMS Programming Interfaces: Calling a System Routine*. To find out how specific languages implement procedures, refer to the documentation set for your language processor.

1.3 Using Procedure Libraries

You can use modular procedures for general programming or you can group them in procedure libraries. Grouping procedures into libraries is a way of collecting procedures so that calling programs can access them easily. When you link your program to a library, the OpenVMS Linker utility (linker) automatically searches that library to resolve any references that your program makes to procedures in the library. Because the linker searches the specified library automatically, your program can call many modular procedures without including the name of each procedure explicitly in the LINK command. The program's executable image and the procedures that it calls are executed in the proper sequence at run time.

Figure 1–1 shows the development of a program that calls one or more procedures in a library. Depending on the options you select when writing modular procedures, you can control the way the linker accesses your procedures, and therefore, the way procedures are invoked at run time. For example, if you place commonly used procedures within a shareable procedure library or shareable image library, you can save memory and disk space because all user processes can access a single copy of the shared procedures.

Figure 1–1 Developing a Program that Calls Library Procedures



ZK–4068–GE

1.4 Existing System Procedures

Many system routines that perform advanced applications are included in the OpenVMS operating system. These procedures are designed to perform various general functions and can be useful building blocks for your own procedures. Before you write a new procedure, make sure the application does not already exist. You should call an existing procedure from a system library whenever possible, instead of duplicating code.

The types of callable system procedures available as part of the OpenVMS operating system are:

Run-time library (RTL) Procedures
System Services
Utility Routines
Record Management Services (RMS)

For more information about the features of these procedures, refer to the *OpenVMS Programming Environment Manual*. For more information about how to use them, refer to *OpenVMS Programming Concepts Manual*.

1.5 Using Translated Images (AXP Only)

AXP

Programs that run on VAX systems can be converted to run on AXP systems by recompiling and relinking or by translating. A single application can include both native images (those that were recompiled and relinked) and translated images.

The most effective way to convert a program that runs on a VAX system to one that runs on an AXP system is to recompile the source code using a native AXP compiler and then to relink the object files and shareable images using the linker.

The alternative method, translation, involves using DECmigrate for OpenVMS AXP, which supports the migration of VAX applications to AXP applications by translating images. DECmigrate converts VAX images into functionally equivalent images that can run on AXP systems. DECmigrate includes the VAX Environment Software Translator (VEST) utility, which analyzes a VAX executable or shareable image and creates a functionally equivalent translated image.

The Translated Image Environment (TIE), which is part of the OpenVMS AXP operating system, provides the run-time support for translated images on OpenVMS AXP. The TIE includes an AXP shareable image that provides each translated image with an environment similar to OpenVMS VAX, interprets untranslated VAX instructions, and processes all interactions with the native AXP system. The TIE also includes a translated image that executes complex VAX instructions.

For more information about VEST and TIE, refer to *DECmigrate for OpenVMS AXP Version 1.0 Translating Images*. For more information about mixing native AXP and translated VAX modules in a single application, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*. ♦

2
Designing Modular Procedures

Well-designed procedures are more likely to be modular, well-written, and easy to maintain. Any time that you save by skimping at the design stage will be lost as you fix problems stemming from a poor design.

This chapter discusses the following aspects of designing a new application:

- Organizing new applications
- Defining a modular procedure interface
- Using JSB entry points
- Using system resources
- Using Input/output
- Documenting modules
- Planning for signaling and condition handling

2.1 Organizing New Applications

Before designing a new application, look at the overall organization. An application should be made up of one or more files, each containing one or more procedures. When linked, the procedures are organized into program sections (PSECTs). Each procedure, as well as the interface between the procedures, should conform to the modular guidelines described in this manual.

2.1.1 Organizing Files and Modules

Each application contains one or more files. Each file contains exactly one module. For information about naming files, refer to Section 3.1.1.3. For information about naming modules, refer to Section 3.1.1.4.

2.1.2 Organizing Procedures into Modules

Each module should contain a single procedure or a group of related procedures. The linker always brings the entire module containing a called procedure into the image if any of its entry points are referenced. Therefore, placing each procedure in a separate module reduces image size and allows more flexibility when using a procedure library. You can supply your own version of one procedure while using other procedures from the library. If many procedures have been grouped in a single module, the linker must link all or none of them.

Group procedures into a module if they share the same static storage or if they have a similar calling sequence, perform similar functions, and share a significant amount of code.

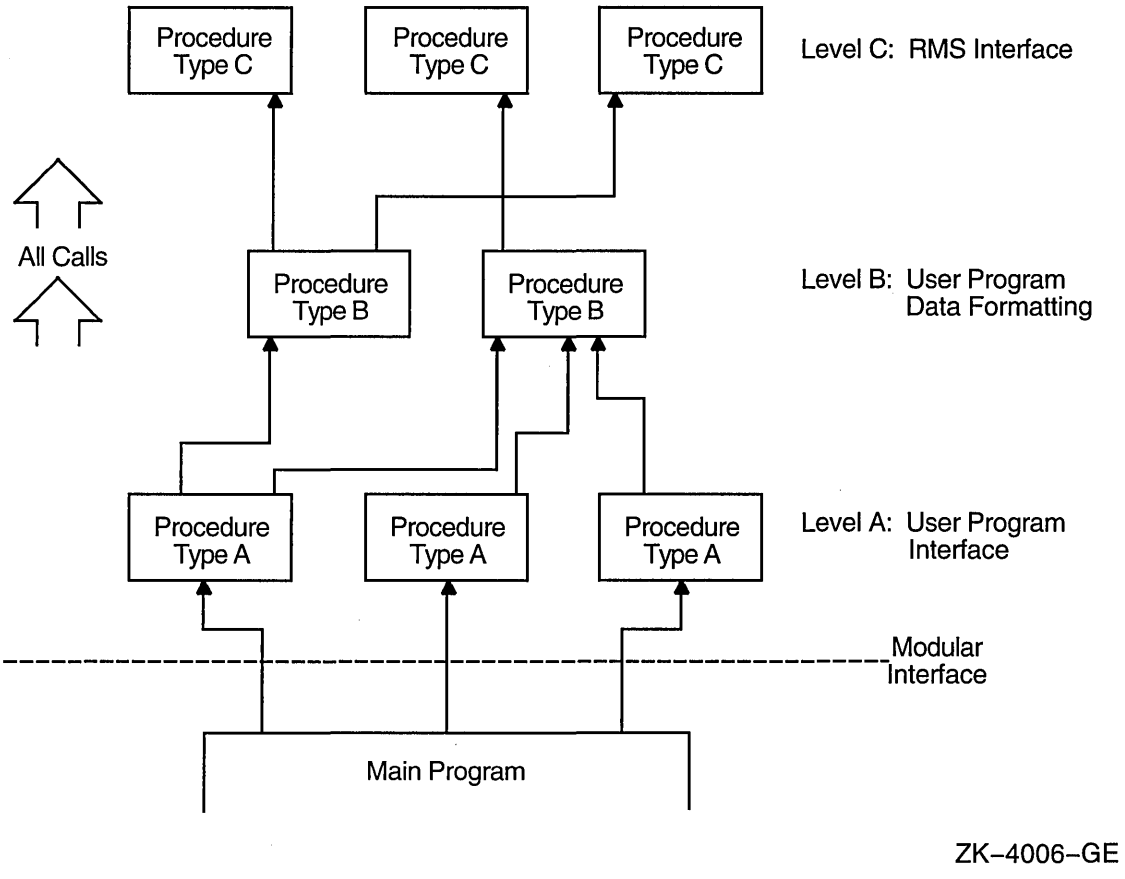
If you are writing a large number of related procedures that call one another or access common data blocks, make the relationship among those procedures as clear as possible. To do this, use the following guidelines to minimize the interaction between procedures, and between procedures and data structures:

- Organize procedures into levels of abstraction
- Make sure each level calls only the next lower level
- Restrict read/write access to data structures and system components to as few procedures as possible

Figure 2–1 shows the BASIC and FORTRAN record I/O processing procedures, which are implemented in the following three levels of abstraction:

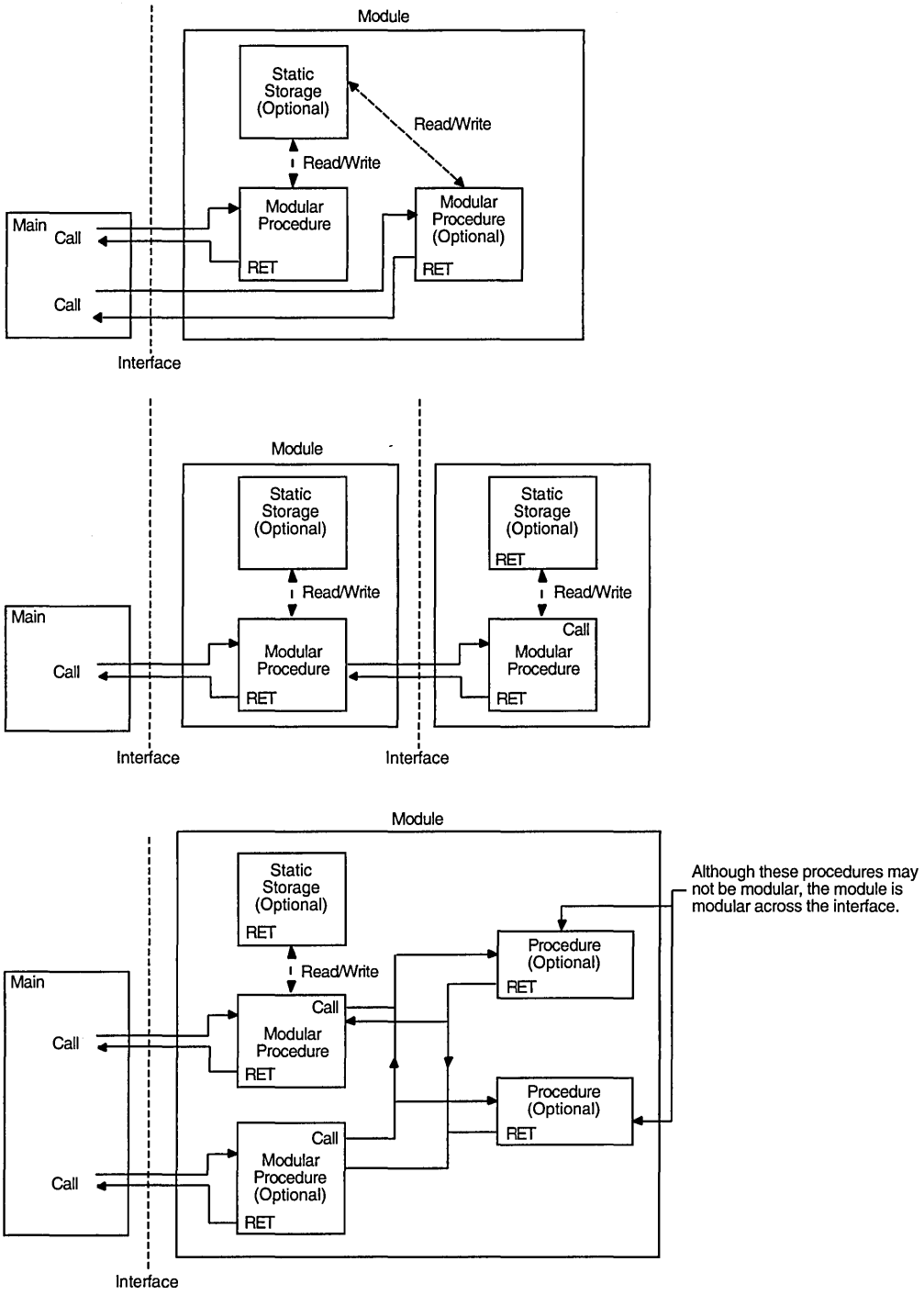
1. User program interface (UPI)
2. User program data formatting (UDF)
3. Record processing and OpenVMS RMS interface (REC)

Figure 2–1 Levels of Abstraction



All calls are made in one direction, to the next innermost level. Procedures at different levels should be in different modules. Figure 2–2 shows possible groupings of procedures.

Figure 2–2 Possible Procedure Groupings



ZK–4007–GE

2.2 Defining a Modular Procedure Interface

Procedures communicate with one another by passing arguments. To clarify the interactions between procedures and programs, you must define each argument when you are designing a procedure. There are two types of arguments: explicit arguments and implicit arguments. The following sections define explicit and implicit arguments and describe how to use them.

2.2.1 Explicit Arguments

Explicit arguments are a procedure's primary interface with other programs. Therefore, to maintain a modular interface, you must follow the rules for argument order, data types, and passing mechanisms. The following format is used to describe each argument:

```
argument-name

OpenVMS usage:   argument-data-structure
type:        argument-data-type
access:      argument-access
mechanism:   argument-passing-mechanism
```

For descriptions of each of these four argument attributes, see the *OpenVMS Programming Interfaces: Calling a System Routine*.

To make your procedures easier to call, be sure that the passing mechanism used for particular data types is consistent throughout all procedures in a facility. Passing all atomic data by reference and all string data by descriptor is recommended.

2.2.2 Implicit Arguments

An implicit argument is one that is not specified in the argument list. Implicit arguments provide additional information to your procedure from static storage locations. Two types of implicit arguments are:

- Arguments allocated by the calling program
- Arguments allocated by your procedure

Using implicit arguments is discouraged because they make the relationship across procedures less clear and tend to increase the interaction between procedures in a way that might go undetected. If your procedure must retain information from previous activations, see Section 2.2.3 for ways to avoid using implicit arguments.

2.2.2.1 Implicit Arguments Allocated by the Calling Program

The calling program can allocate implicit arguments as statically allocated variables in a named PSECT (for example, COMMON and MAP in BASIC, COMMON in FORTRAN, or variables declared in the outer block of a procedure or program in Pascal). The calling program can also allocate implicit arguments as statically allocated global variables (for example, symbols defined with a double colon [::] in MACRO and GLOBAL variables in BLISS).

Allocation of implicit arguments by the calling program is not recommended for the following reasons:

- Two programs could use the same PSECT name or global variable for different values. This error would be undetected.
- The calling program is no longer independent of the called procedure. Consequently, a change in one could inadvertently affect the other.

- In FORTRAN, the calling program declares all variables as COMMON regardless of the number of implicit inputs actually needed. All COMMON variables should also be declared by all modules that use the COMMON storage, further decreasing independence.

2.2.2.2 Implicit Arguments Allocated by the Called Procedure

Implicit arguments allocated by the called procedure are kept in local static storage.

These implicit arguments are usually used to keep track of resources (using resource allocating procedures) and shorten the explicit argument list. However, the use of implicit inputs by non-resource-allocating procedures can lead to unexpected results. For example, assume that procedure A is to leave information for a companion procedure B. This would result in B having both explicit inputs (from its caller) and implicit inputs (from A's storage). Next, consider that a calling program calls A, then calls procedure X, and finally calls B. For the calling program to get correct results from B, it must know that X (and any procedure that X calls) did not make a call to A, because such a call would change the implicit inputs A leaves for B.

Because one of the objectives of modular programming is to permit procedures to be combined arbitrarily without needing to understand each other's internal workings, using implicit arguments is not recommended. The same problems can occur with any non-resource-allocating procedure that leaves results for itself as future implicit arguments.

2.2.3 How to Avoid Using Implicit Arguments

Procedures that do not allocate resources can be written in the following three ways to avoid the implicit argument problems described in Section 2.2.2:

- When one procedure obtains results from another, combine the two procedures into a single call. (See Section 2.2.3.1.)
- Provide a single call to an action routine that is supplied by the calling program part way through the procedure's execution. (See Section 2.2.3.2.)
- Give the calling program responsibility for retaining information from a procedure activation. This is done with an explicit argument. (See Section 2.2.3.3.)

2.2.3.1 Combining Procedures

Often, non-resource-allocating procedures can be combined into a single procedure that returns all information explicitly in a single call.

Compare Example 2–1 with Example 2–2 to see the effects of combining procedures to avoid the use of implicit arguments.

Example 2–1 FORTRAN Program Showing the Improper Use of Implicit Arguments

```
!+
! This program demonstrates a situation where
! the input of a procedure depends on the output
! of a previously called procedure.
!-
        REAL*4 X, Y, RESULT
        X = 1
        Y = 1
!+
! Call the procedure that writes into a common data area.
!-
        CALL SUM_SQUARES (X, Y)
!+
! Call the procedure that reads from the common data area.
!-
        CALL GET_SQRT (RESULT)
!+
! Print the result obtained.
!-
        WRITE (6,10) X, Y, RESULT
10      FORMAT(1X, 'SQRT(', F6.2, '**2 + ', F6.2, '**2) =',F6.2)
        STOP
        END

!+
! This procedure sums the squares of its two inputs and
! places the result in a common area, for use by some
! other procedure.
!-
        SUBROUTINE SUM_SQUARES (A, B)
        COMMON /INTERNAL_STORAGE/ TEMP_RESULT
        TEMP_RESULT = (A ** 2) + (B ** 2)
        RETURN
        END
!+
! This procedure calculates the square root of whatever
! number is in the common area.
!-
        SUBROUTINE GET_SQRT (C)
        COMMON /INTERNAL_STORAGE/ TEMP_RESULT
        C = SQRT (TEMP_RESULT)
        RETURN
        END
```

2.2.3.2 User-Action Routine

Another way to combine several procedures into one call is to let the calling program gain control at a critical point in your procedure's execution. For this to happen, your procedure must specify an action routine argument that is called during execution. Therefore, your procedure can execute twice, before and after the action routine, with no implicit inputs. The OPEN statements in BASIC, FORTRAN, and Pascal use this technique by permitting the user to supply a user-action routine.

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

CALL my-proc (... ,action-routine ,user-arg)

Example 2–2 FORTRAN Program Combining Procedures to Avoid Implicit Arguments

```
!+
! This procedure shows the subroutines called in
! the previous example combined into a single subroutine
! that eliminates the use of COMMON.
!-
        REAL*4 X, Y, RESULT
        X = 1
        Y = 1
!+
! Call the new procedure.
!-
        CALL DO_IT_ALL (X, Y, RESULT)
        WRITE (6,10) X, Y, RESULT
10      FORMAT (1X, 'SQRT (', F6.2, '**2 + ', F6.2, '**2) = ',F6.2)

        STOP
        END

!+
! This procedure calculates the square root of the sum of
! the squares of its first two arguments, and returns the
! result in the third argument.  It combines the functions
! provided by the SUM_SQUARES and GET_SQRT
! procedures and eliminates the use of COMMON.
!-
        SUBROUTINE DO_IT_ALL (A, B, C)
        C = SQRT ((A ** 2) + (B ** 2))
        RETURN
        END
```

Then your procedure invokes the action routine as follows:

CALL action-routine (... ,user-arg)

For information on writing user-action routines, see Section 3.1.4.

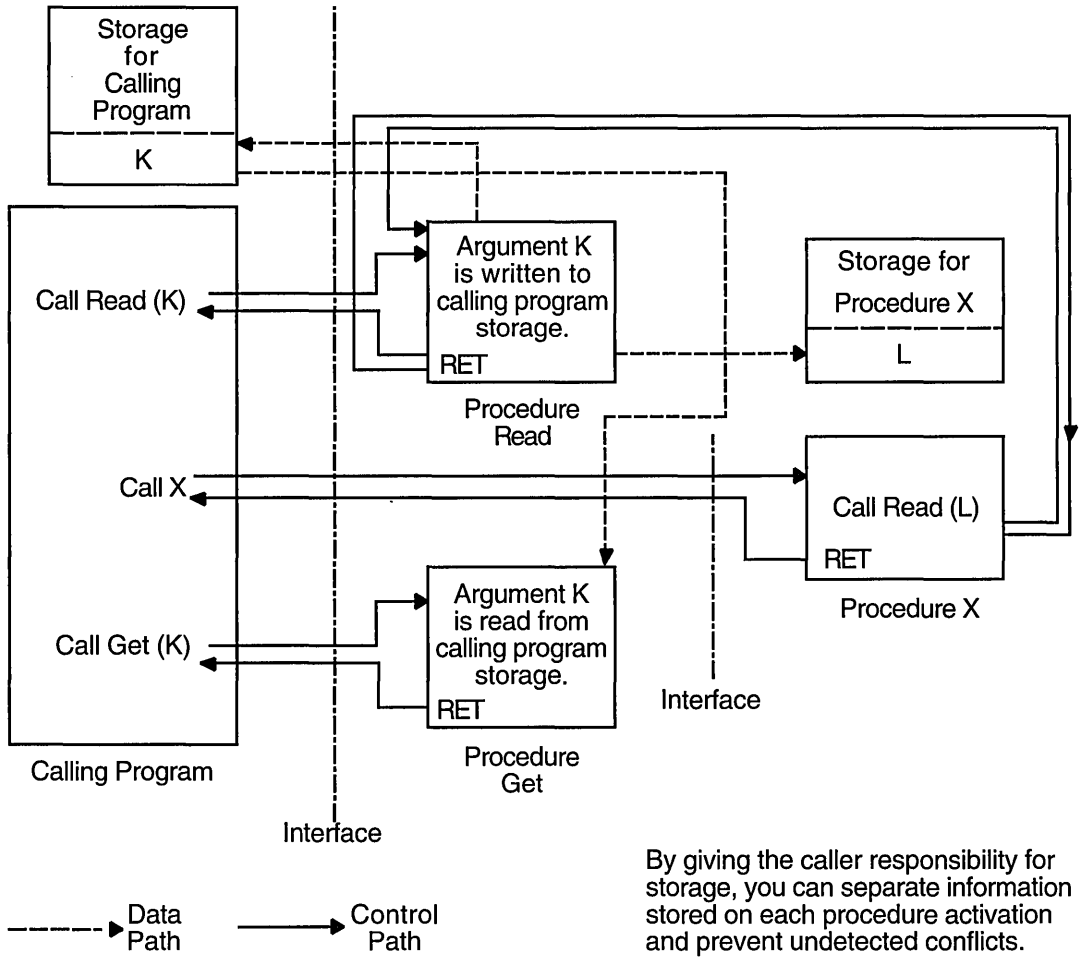
2.2.3.3 Designating Responsibility to the Calling Program

You can make the calling program responsible for retaining information from one procedure activation to another. There are three ways to do this:

- Require the calling program to allocate the storage your procedure needs. Then have the calling program pass the address of the storage location as an explicit argument on all calls to your procedure. The disadvantage of this method is that you cannot increase the amount of storage needed by your procedure without requiring all calling programs to be rewritten. Thus, you should use this method only when you are confident that your procedure will not be revised to use additional storage in the future.
- Require the calling program to allocate a longword pointer to the stored data and pass its address to your procedure as an explicit argument. On the first call, your called procedure will dynamically allocate storage (by calling LIB$GET_VM) and store its address in the caller's longword. On subsequent calls, your procedure will use information left in the storage area from previous calls.
- Require the calling program to pass a processwide identifying value to your procedure on all calls. The processwide identifier indicates which information from previous procedure activations is to be used as implicit inputs.

Figure 2–3 shows a calling program that has responsibility for explicitly indicating the storage to be used by the called procedure.

Figure 2–3 Designating Storage Responsibility to the Caller



Calling Program Allocates Procedure Storage

This method causes the calling program to allocate all storage needed and pass the address of the storage as an explicit argument on each call.

For example, the library procedure MTH$RANDOM requires that the calling program allocate storage for the longword seed and pass its address on each call. MTH$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. This ensures that the procedure will generate a different value on the next call.

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

Calling Program Passes Pointer

In this method, the calling program allocates only a longword pointer to the dynamic heap storage to be allocated by your procedure. It then passes the address of the longword as an explicit argument. The following two interface techniques can be used to indicate that storage is to be initialized:

- Provide a single entry point. If your called procedure finds the value zero in the longword that the calling program has allocated, the procedure allocates and initializes dynamic heap storage.
- Provide a second entry point. This entry point stores the address of the allocated storage in the longword. On subsequent calls, your procedure uses that value as the storage address of information from previous calls.

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

For example, the procedure LIB$INIT_TIMER, which gets times and counts from the operating system, uses a single optional argument **handle-adr** to determine where these values are to be stored. The **handle-adr** argument is the address of a longword pointing to a block of storage that contains the values of times and counts:

- If **handle-adr** is missing, the values are stored in static storage, making this call non-AST-reentrant.
- If **handle-adr** is zero, LIB$INIT_TIMER allocates a block of dynamic heap storage by calling LIB$GET_VM. The values are placed in that block, and the address of the block is returned in **handle-adr**.
- If **handle-adr** is nonzero, it is considered to be the address of a storage block previously allocated by a call to LIB$INIT_TIMER. The block is then used again and new times and counts are stored in it.

LIB$FREE_TIMER deallocates the block of dynamic heap storage allocated by a previous call to LIB$INIT_TIMER. The **handle-adr** argument to LIB$FREE_TIMER is the address of a longword that points to a block of dynamic heap storage where times and counts have been stored. That storage is returned to free storage by calling LIB$FREE_VM.

Calling Program Passes a Processwide Identifier

In this method, the calling program passes a processwide identifying value to identify implicit results produced on previous calls, which will be implicit inputs on this call. Any calling program can use the processwide identifier. Examples include BASIC or FORTRAN logical unit numbers and OpenVMS system services I/O channel numbers.

Processwide identifiers are a resource. Modular programming techniques require that all resources allocated by a procedure be allocated by calling a resource-allocating procedure. This prevents conflicts because a single procedure can keep track of multiple allocations to more than one procedure or procedure activation. Therefore, if you use the method described in this section, you will also have to write a resource-allocating procedure to control the resource. If you write a resource-allocating procedure, it is recommended that you place it in an object module library so that other programmers can use it.

The library procedures LIB$GET_LUN and LIB$FREE_LUN allocate and deallocate FORTRAN and BASIC logical unit numbers outside the range normally specified in user programs, that is, outside the range 0 to 99.

2.2.4 Order of Arguments

Procedures in the RTL follow a consistent pattern for positioning arguments. You should follow the same guidelines. Group procedure arguments from left to right in the following order:

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

Note that optional arguments follow required arguments. Therefore, when the calling program omits the optional arguments, the actual argument list passed to the procedure is shortened.

The called procedure accesses the required arguments from left to right, beginning with the first argument. The only exceptions are procedures that return a large function value of known size. In this case, the calling program uses the first argument to specify where the function value is to be stored, and the other arguments are shifted right one position. (For more information, refer to the *OpenVMS Calling Standard*.)

2.2.5 Using Optional Arguments

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

A zero argument list entry is simply a zero passed to the procedure by value. For example, if we call a procedure called GRA_CUBE and omit an optional argument **C**, the calling sequence from BASIC would be as follows:

```
15 CALL GRA_CUBE(A, B, 0 BY VALUE)
```

In this call, "0 BY VALUE" is the zero argument list entry.

Note

Most OpenVMS system services, unlike the run-time library procedures, cannot accept a shortened argument list. Omitted arguments must always be indicated with a zero argument list entry. For arguments passed by value, there is no distinction between passing a zero value and passing a zero argument list entry.

2.3 JSB Entry Points (VAX Only)

VAX

On VAX systems, Digital recommends that you do not use JSB[1] entry points in procedures that will be contained in a procedure library. Procedures that can be invoked only by JSB instructions are not callable by high-level languages. If a procedure does use a JSB entry point, it must also provide an equivalent call

[1] JSB is a MACRO instruction that means jump to subroutine.

entry point to maintain language independence. The call entry point must be provided because JSB instructions are only available in VAX MACRO and VAX BLISS-32.

If you provide a JSB entry point for your procedure, the name of the JSB entry point is the same as the name of the procedure, except that it ends in _Rn. The *n* indicates the highest register modified or used as an input argument.

For example, the JSB entry point of the run-time library procedure LIB$ANALYZE_SDESC is LIB$ANALYZE_SDESC_R2. ♦

2.4 Using System Resources

The system resources available to you are limited by your account quotas and by the amount of available resources on the system. Efficient use of system resources makes more resources available for all processes.

2.4.1 Choosing a Storage Type

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

2.4.1.1 Stack Storage

A procedure dynamically allocates stack storage on the process stack at run time, as needed. To allocate stack storage, the procedure moves the stack pointer up by decreasing its value. Note that stack storage is not initialized to zero because the stack is created once and reused many times for subsequent stack frames.

The procedure deallocates stack storage by moving the stack pointer down (increasing its value) when that procedure returns control to the calling program. Stack storage exists only for the duration of the procedure activation that creates it.

2.4.1.2 Heap Storage

Dynamic heap storage is allocated at run time from a processwide pool, as the procedure activation needs it and as the account quotas and virtual address space of your process permits.

To allocate heap storage, your procedure calls a system routine such as the Run-Time Library procedure LIB$GET_VM or the system service $EXPREG. The call to the system routine may be within the procedure itself, or you may use a general resource-allocating procedure to centralize your resource allocations.

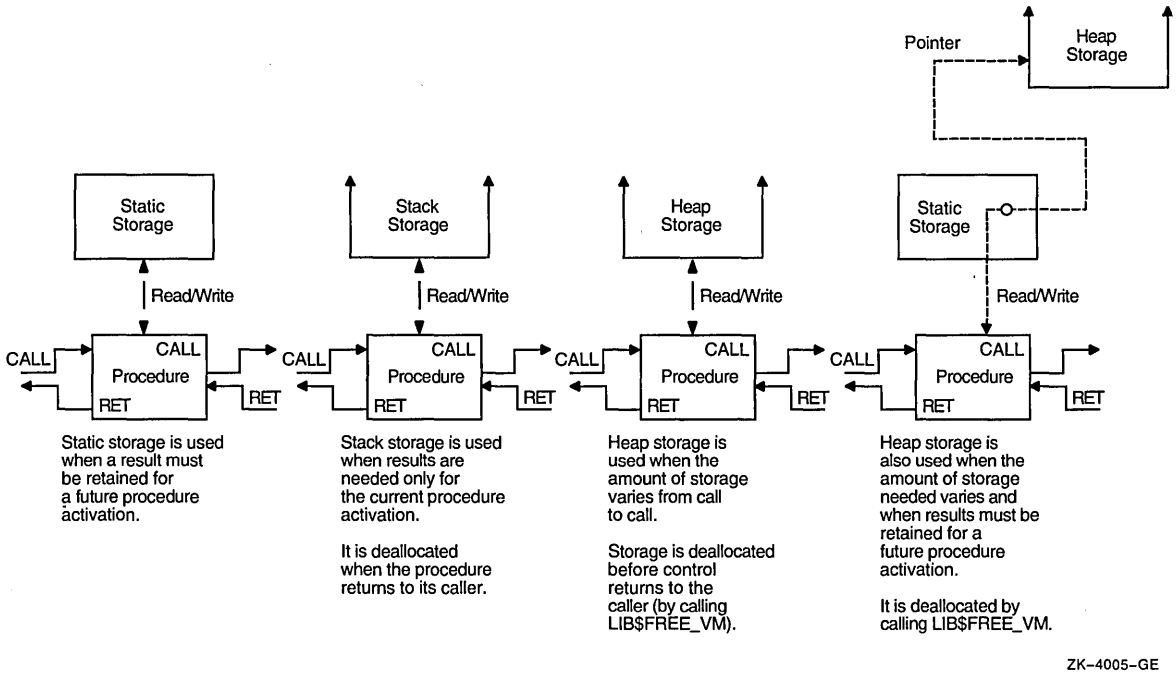
Heap storage is deallocated—that is, returned to the processwide pool—by calling LIB$FREE_VM. The system service $CNTREG cannot be used to deallocate heap storage.

Figure 2–4 shows how the different types of storage are used.

Note

The type of storage to be used can be determined by the duration or quantity of the storage. Any storage that is of long duration and unknown quantity (at compile time) should be heap storage. Storage of short duration (during the current invocation of the procedure) should be stack storage. Storage of long duration that is needed in only one instance should be static storage.

Figure 2–4 Use of Storage Types



2.4.1.3 Static Storage

At link time, the linker collects storage in similar PSECTs into a single image section. The initial contents of this storage are specified in the source program. The OpenVMS operating system initializes any noninitialized static storage to zero. On calls to a procedure after initialization, the static storage has the same allocation and the contents left from the previous call.

2.4.1.4 Avoiding Use of Static Storage

Several disadvantages to using static storage are:

- It is an inefficient use of memory. When using static storage, you must provide for the largest possible memory use.
- An image size is larger because of the inefficient use of memory.
- It can easily lead to problems with AST reentrancy, as seen in Example 2–3. This example circumvents the problem of an AST corrupting data by setting a first-time flag. Another method of preventing this problem is to use "test and set" instructions. For more information, see Section 3.3.4.2.

Example 2–3 Static Storage and AST-Reentrancy

```
10      !+
        ! Program to demonstrate corruption
        ! of static storage due to ASTs.
        !-
        DECLARE LONG CURRENT_NUMBER

        !+
        ! Enable CTRL/C AST handling.
        !-
        ON ERROR GOTO 19000
        X% = CTRLC

        !+
        ! Increment the number and print the
        ! current value.  When the number
        ! reaches 1000, exit.
        !-
        FOR CURRENT_NUMBER = 1% TO 1000%
100     PRINT CURRENT_NUMBER;
        NEXT CURRENT_NUMBER
        GOTO 32767

19000   !+
        ! Error-handling routine.  If this routine is
        ! entered due to a CTRL/C
        ! AST, corrupt CURRENT_NUMBER by setting it to -1.
        !-
        IF ERR = 28 THEN CURRENT_NUMBER = -1%
        RESUME 100

32767   END
```

2.4.1.5 Summary of Storage Use by Language

Table 2–1 summarizes storage available to the programmer in various language procedures.

Table 2–1 Summary of Storage Use by Language

Language	Storage Type		
	Static	**Stack**	**Heap**
Ada	Constants and fixed-size objects contained in library packages	Local subprogram and task variables	Dynamically sized objects in library packages and objects created by allocators
BASIC	All COMMON and MAP data storage	Local variables	Dynamic strings
	Most arrays	Executable DIMENSION statement	
BLISS	OWN and GLOBAL	STACK LOCAL	By calling LIB$GET_VM
C	Objects declared with external or static internal linkage	Objects declared inside a function with "automatic" linkage	By calling malloc, calloc, or realloc
COBOL	All data storage	Not applicable	By calling LIB$GET_VM

(continued on next page)

Table 2–1 (Cont.) Summary of Storage Use by Language

Language	Storage Type		
	Static	Stack	Heap
DIBOL	All RECORD, COMMON, and LITERAL data storage	Not applicable	Not applicable
VAX FORTRAN	All data storage	Not applicable	By calling LIB$GET_VM[1]
Assembly language	Block storage	Decrementing stack pointer	By calling LIB$GET_VM
Pascal	All program or module level storage	PROCEDURE and FUNCTION local	By calling NEW[2]
PL/I	STATIC	AUTOMATIC	ALLOCATE statement (BASED)[3]
RPG II	All data storage	Not applicable	By calling LIB$GET_VM
SCAN	STATIC, GLOBAL, COMMON, EXTERNAL	When AUTOMATIC is used in a procedure or macro	DYNAMIC STRING values, TREE pointers, and the ALLOCATE function

[1]Storage for DEC Fortran for OpenVMS Alpha is the same as for VAX FORTRAN, except that stack storage is available as a compile time option for some variables.

[2]Although this is true most of the time, there are other rules that can also determine STATIC versus STACK allocation. For more information, see the Pascal user documentation.

[3]BASED is the storage class used to allocate heap storage in PL/I. The ALLOCATE statement does the actual allocation.

2.4.2 Using Event Flags

Event flags allow modular procedures to communicate with each other and to synchronize their operations. Because they can be allocated at run time, event flags allow one procedure to run independently of other procedures existing in the same process.

Event flags are allocated and deallocated by the run-time library procedures LIB$GET_EF and LIB$FREE_EF. (For more information, see the descriptions of the LIB$GET_EF and LIB$FREE_EF procedures in the *OpenVMS Programming Concepts Manual* and the *OpenVMS RTL Library (LIB$) Manual.*)

2.4.3 Using Logical Unit Numbers

A logical unit number is used to define the device or file a program uses to perform input and output. Modular procedures do not need to know the unit numbers of other procedures running at the same time.

Logical unit numbers are used only in BASIC and FORTRAN.

Logical unit numbers should be allocated and deallocated using the LIB$GET_LUN and LIB$FREE_LUN RTL procedures. (For more information about using logical unit numbers, see the descriptions of the LIB$GET_LUN and LIB$FREE_LUN procedures in the *OpenVMS Programming Concepts Manual* and the *OpenVMS RTL Library (LIB$) Manual.*)

2.5 Using Input/Output

In general, your procedure's input/output (I/O) is directed to either the terminal or a file. (In some cases, you may need to use mailbox I/O and network operations. For information about these areas, see the *DECnet for OpenVMS Networking Manual.*) Regardless of whether you are directing input/output to the terminal screen or to a file, you must follow two rules to maintain modularity:

1. A procedure must not print error or informational messages either directly or by calling the $PUTMSG system service. It must either return a condition value in R0 as a function value, or call LIB$SIGNAL or LIB$STOP to output all messages. (LIB$SIGNAL and LIB$STOP may be called either directly or indirectly.)
2. A procedure should use device independent services and procedures for input/output.

2.5.1 Terminal Input/Output

The methods available for performing input/output to the terminal include the following:

- Queue I/O Request system service ($QIO)

  Using a $QIO to perform terminal I/O is very efficient. However, $QIOs use device-dependent services and are the most difficult to use from high-level languages of all methods discussed here, because there are more steps involved and because the calling interface requires more knowledge from the caller than RMS services. Using a $QIO in your procedure may require additional steps, such as constructing item lists, writing AST routines, assigning an I/O channel, queueing an I/O request, testing to ensure that the request was successfully queued and completed, and deassigning the I/O channel. (For more information about $QIOs, see the *OpenVMS System Services Reference Manual.*)

- OpenVMS Record Management Services (RMS)

  The RMS facility provides device-independent and general-purpose services that are easier to call than $QIOs. However, it is often not convenient to construct the access control blocks (FAB, RAB, and so forth) required by RMS from a high-level language. (For more information about RMS, see the *OpenVMS Record Management Services Reference Manual.*)

- Language I/O statements

  Language I/O statements are provided for all high-level languages. These statements are easy to use and provide simple I/O and data formatting for the high-level language user. Native language I/O statements make it unnecessary for the high-level language user to call $QIO or RMS directly; these calls are made by the compiled code on your behalf. However, low-level and medium-level languages (VAX MACRO and BLISS-32) have no built-in language I/O statements and must use $QIO and RMS for terminal and file I/O. (For more information, see the appropriate language reference manual.)

- Screen Management Procedures in the run-time library (SMG$)

  SMG$ procedures provide an easy-to-call interface for high-level languages. They are device-independent and aid in the composition of complex screen images. The SMG$ facility in the run-time library provides screen composition operations; that is, SMG$ makes it easy for an application to

divide its screen into multiple regions and provides functions for manipulating those regions. Other features provided by SMG$ procedures are as follows:

- Output to virtual displays
- Input from a virtual keyboard or locator device
- The ability to perform asynchronous input
- Built-in minimal screen updating
- Optional buffering and batching to optimize performance
- The ability to trap broadcast messages
- The option of performing output to a file or a hardcopy device
- Support for foreign (not Digital) terminals
- Subprocess manipulation

For more information about SMG$ procedures, see the *OpenVMS RTL Screen Management (SMG$) Manual* and the *OpenVMS Programming Concepts Manual.*

During I/O to the terminal, it is important that the procedure and the main program cooperate in controlling the terminal screen. For example, an I/O procedure may write something to the terminal screen that the calling program wants to erase. The calling program must know both what and where that information is, in order to erase it. The calling program and the called procedure must communicate by passing arguments that define which part of the screen will be accessed by each. The run-time library contains Screen Management (SMG$) procedures for this purpose.

Do not combine different methods of I/O within your application. Problems can arise if the calling program and the called procedure use different methods of I/O. Each method of performing input/output maintains some knowledge of what is on the terminal screen. At the very least, the current cursor position is remembered. If another type of I/O is performed, that information is not updated and, therefore, becomes incorrect. The results of any subsequent I/O would be unpredictable. If you must combine other methods with uses of SMG$ procedures, use the SMG$ procedures that aid such an integration.

2.5.2 File Input/Output

File I/O can be performed by the following methods:

- Block I/O

  Uses system services to map a section of the file to the process virtual address space. No notion of records.

- OpenVMS Record Management Services (RMS)

  RMS provides a variety of file organizations and access modes from which you can select the processing techniques best suited to your application. RMS supports the sequential, relative, and indexed-sequential file organizations. These modes allow you to access records within these files sequentially, randomly by key value or relative record number, or randomly by the records file address (RFA). It is usually not necessary to call RMS directly from high-level languages. For specific information about performing record management operations in the language you are using, consult your language reference manual. (For more information about RMS, see the *OpenVMS Record Management Services Reference Manual.*)

- Language I/O

  The compiled code in most high-level languages calls a run-time library language support procedure for file operations. The run-time library procedures normally call RMS. Therefore, most RMS features are available to the high-level language user without calling RMS directly. Language I/O statements are suitable for either data files or output files. Low- and medium-level languages (VAX MACRO and BLISS-32) do not have any language I/O statements and must call RMS directly. (For more information, see the appropriate language reference manual.)

2.6 Documenting Modules

You should document every module you create so that you and others know what the procedure does. Each module should include:

- A preface that identifies the procedure
- A description of the procedure

In most cases, a module should contain only one procedure.

2.6.1 Writing a Module Preface

At the beginning of every module, include a preface that contains the following information:

Title:	Module name followed by a one-line functional description.
Version:	Version and a three-digit edit number. Generally 1-001 is the original version.
Facility:	Description of the library facility, such as general utility library (LIB).
Abstract:	Short (three to six lines) functional description of the module.
Environment:	Describe any special environmental assumptions that the module can make. These include assumptions made at both compilation and execution time that could affect either the hardware or software environments. Describes situations that the module assumes during execution time and optional modular programming elements that your module does not follow. Indicates the reentrancy characteristics of the procedures in this module. Each procedure is either fully-reentrant, AST-reentrant, or non-reentrant.
Author:	Your name and date the module was created.
Modified by:	Modification number, name of modifying programmer, modification date, and a list of the modifications.

End the preface with a page delimiter. After the preface, include the code for the procedure.

Example 2–4 shows a sample module description.

Example 2–4 Sample Module Description

```
        PROGRAM GRA_CUBE                ! Create representation of a cube

!+
! VERSION:      1-002
!
! FACILITY:     User Graphics Computation Library
!
! ABSTRACT:     This module contains a procedure to create a mathematical
!               representation of a cube, GRA_CUBE.
!
! ENVIRONMENT:  User Mode, AST-reentrant
!
! AUTHOR:       John Smith                CREATION DATE:  14-Sep-1993
!
! MODIFIED BY:
! 1-001 - Original.  DWS 14-Sep-1993
! 1-002 - Fix a minor bug in cube volume computation.  MDL 15-Mar-1993
!-
```

2.6.2 Writing a Procedure Description

At the beginning of every procedure in a module, describe the procedure by including the information in this section. Include all the description elements, even if they are not in the procedure. For example, if a procedure has no implicit inputs, write the following:

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

Every procedure description should include the following information:

Functional description:

Describes a procedure's purpose and completely documents its interfaces.

Includes the basis for any critical algorithms used, including literature references where applicable, and explains why a particular algorithm was chosen.

Indicates the reentrancy characteristics of this procedure if they differ from those given in the module description.

Calling sequence:

Includes these elements in the following order:

1. A return status, value argument, or CALL statement
2. The procedure name
3. The argument list (typically a list of registers or arguments)

In VAX MACRO, each argument is symbolically defined as the offset relative to the argument pointer (AP).

Lists the arguments in the order they will appear in a high-level language. Each argument characteristic should also be included, using the procedure argument notation described in *OpenVMS Programming Interfaces: Calling a System Routine*.

Formal arguments:	Lists any explicit input, input-output, or output arguments. Includes a qualifying description with each argument. The arguments should be listed in the order they are listed in the calling sequence.
Implicit inputs:	Lists any inputs from storage, internal or external to the module, that are not specified in the argument list. Usually all that will appear here is "NONE". See Section 2.2.2.
Implicit outputs:	Lists any outputs to internal or external storage that are not specified in the argument list.
Completion status or routine value:	Lists the success or failure condition value symbols that could be returned. If your procedure returns a function value other than a condition value, change the heading to "Routine value".
Side effects:	Describes any functional side effects not evident from a procedure's calling sequence. This includes changes in storage allocation, process status, file operations, and possible signaled conditions. In general, you should 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.

Example 2–5 shows a sample procedure description.

Example 2–5 A Sample Procedure Description

```
!++
! FUNCTIONAL DESCRIPTION:
!
!       Return the system date and time, using the caller's
!       semantics for his/her string.
!
!       Non-reentrant; uses static storage.
!
! FORMAL ARGUMENT(S):
!
!       RESULT_ADDR
!       VMS USAGE : char_string
!       TYPE      : character string
!       ACCESS    : write only
!       MECHANISM : by descriptor
!
!      Address of the descriptor into which the
!      system date and time is written.
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
```

(continued on next page)

Example 2–5 (Cont.) A Sample Procedure Description

```
! COMPLETION CODES:
!
!       SS$_NORMAL      Procedure successfully completed
!       LIB$_STRTRU     Success, but source string truncated
!
! SIDE EFFECTS:
!
!       Requests the current date and time from VMS.
!
!--
```

2.7 Planning for Signaling and Condition Handling

Two methods are available to a procedure for indicating to its caller whether it completed successfully. One method is to return a condition value. The other method is to signal an error condition.

To provide a better user interface, all procedures in a facility should either return condition values or signal error conditions. Regardless of which method you choose, you should be consistent within the facility to make the procedures easier for the user to call.

2.7.1 Guidelines for Signaling Error Conditions

The signaling of an error condition is, in some instances, mandatory.

Procedures that return a function value cannot also return a condition value and therefore must signal any error conditions encountered.

However, to maintain efficiency, you might want other procedures to signal error conditions also. Checking the return status of a called procedure for repetitive calls can be time consuming and adversely affect the performance of the calling program. For example, if you are going to call a procedure 100 times within a loop and the chances of that procedure's failure are relatively small, you may not want to take the time to check the return status after each call to make sure that the condition value returned was SS$_NORMAL. Signaling error conditions is far more efficient in this type of application.

From the point of view of the calling program, handling a signaled condition is slightly more difficult than checking a returned condition value because it involves writing a condition handler to be invoked in the event that an error condition is signaled. However, handling a signaled condition allows the calling program to execute more efficiently.

To signal an error condition, your procedure uses either a condition-handling mechanism provided by the source language, or it calls the Run-Time Library procedure LIB$SIGNAL. To use LIB$SIGNAL, your procedure calls LIB$SIGNAL and specifies the condition code and zero or more arguments specifying the environment of the condition. For more information about using LIB$SIGNAL, see the *OpenVMS RTL Library (LIB$) Manual*.

2.7.2 Guidelines for Returning Condition Values

From the point of view of the calling program, it is considerably easier to check returned condition values than to handle signaled error conditions. When the condition value is being returned, the calling program does not need to include a condition handler. The calling program needs only to check the status of the returned value.

However, if you return condition values rather than signal error conditions, you return less information about the error condition to the calling program. It is recommended that you return condition values when the explanation of the error condition is simple and self-contained. For example, LIB$GET_VM returns a condition value, because the possible status conditions are self-contained and simple (for example, insufficient virtual memory).

According to the *OpenVMS Calling Standard*, the status returned must be a condition value. (For more information, see *OpenVMS Programming Interfaces: Calling a System Routine*.)

2.7.3 When to Signal or Return Condition Values

To some degree, whether you decide to signal an error condition or return a condition value depends on the language you are using for your procedure. In some high-level languages, it is very difficult to write a condition handler to be invoked in the event that an error condition is signaled. (For more information about condition handling in your language, consult the appropriate language reference manual.)

Regardless of which language you are using, there are general guidelines for when to return a condition value and when to signal an error condition.

You should signal an error condition in the following situations:

- Your procedure returns a value in R0 and cannot return a condition value.
- Your procedure must execute quickly and checking the return status of a condition value would be inefficient.
- Your procedure will be executed repetitively and, therefore, checking the condition value returned would adversely affect your procedure's performance.
- The amount of information you want to return about the error condition cannot be contained in a condition value.
- A useful error message requires information that cannot be determined until run time. For example, the FDL$PARSE procedure must tell you which line of the FDL file was the cause of an error. Because the line number of the line containing the error cannot be determined until run time, the signal mechanism is preferred.
- You want to execute a specific condition handler in the event that an error condition is signaled.

You should return a condition value in the following situations:

- You want to keep the error-handling mechanism simple.
- The speed of the error-checking mechanism is not of great concern.
- The total possible errors that may be returned is a small number and sufficient information about those errors can be contained in the condition value returned.
- The functions provided by the procedure are so general that the procedure will be used in various levels and environments.

3

Coding Modular Procedures

This chapter describes how to code modular procedures. Specifically, it covers the following topics:

- Coding guidelines
- Initializing modular procedures
- Writing AST-reentrant code

Appendix A summarizes many of these guidelines. Refer to the appendix to review the guidelines or use it as a checklist.

3.1 Coding Guidelines

The coding guidelines discussed in this section are of two types: required and recommended. You must follow the sections marked required to ensure that your application is modular. Digital highly recommends that you adhere to the guidelines presented in the sections marked recommended. Following these additional rules will help you produce consistent, uniform applications.

3.1.1 Adhering to the Naming Conventions

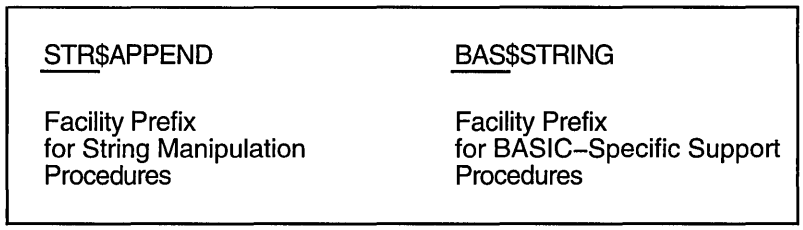
The following guidelines apply to the naming of facilities, procedures, files, modules, and program sections. You must follow these conventions when choosing names for modules, PSECTs, and status codes.

3.1.1.1 Facility Naming Conventions (Recommended)

To make it easy to locate a set of related procedures, Digital recommends that you group your procedures into facilities. Providing related procedures with a common facility prefix is a convenient method for organizing procedures. The facility prefix is the first part of any procedure name.

As shown in Figure 3–1, the first three (or sometimes four) characters of a procedure name are used to indicate the facility of a run-time library (RTL) procedure.

Figure 3–1 Examples of Facility Prefixes as Used in Procedure Names



ZK–3084–GE

Facility names represent library facilities. A procedure is characterized as belonging to a particular facility according to the types of operations it performs. Facilities may differ in the conventions they use for handling errors and receiving arguments, as well as in primary function. Table 3–1 lists some common Digital facility prefixes.

Table 3–1 Common Library Facilities — Prefixes and Content

Prefix	Content
ADA	Ada Run-Time Library procedures
APL	APL Run-Time Library procedures
BAS	BASIC Run-Time Library procedures
B32	BLISS-32 Run-Time Library procedures
CDU	Command Definition utility
CLI	Command language interpreter
COB	COBOL Run-Time Library procedures
COR	CORAL Run-Time Library procedures
C74	COBOL-74 Run-Time Library procedures
DBG	Debugger
DBL	DIBOL Run-Time Library procedures
DECC	C RTL
ERF	Error Log Formatter
FDV	FMS Forms Driver Library procedures
FOR	FORTRAN Run-Time Library procedures
LBR	Librarian utility procedures
LIB	RTL General-Purpose procedures
MATH	Portable Math Library
MTH	RTL Mathematics procedures
OTS	RTL language-independent procedures
PAS	PASCAL Run-Time Library procedures
PLI	PL/I Run-Time Library procedures
RMS	Record Management Services
RPG	RPG II Run-Time Library procedures

(continued on next page)

Table 3–1 (Cont.) Common Library Facilities — Prefixes and Content

Prefix	Content
SMG	RTL screen management procedures
SOR	Sort utility procedures
STR	RTL string manipulation procedures
VAX	VAX Architecture Emulation

You can create your own facilities by defining a unique facility name and facility number. The name for your facility should be a unique name between 1 and 27 characters in length. Facility names supplied by Digital all contain a dollar sign ($) after the prefix. User-supplied facility names should use an underscore (_) rather than a dollar sign ($) to avoid any name conflicts.

The facility number is used in defining condition values for the facility. Bit 27 (STS$V_CUST_DEF) of a condition value indicates whether the value is supplied by Digital or by the user. This bit must be 1 if the facility number is created by the user. For more information, use the Help Message utility (MSGHLP) to access online descriptions of system messages from the DCL ($) prompt. For more information about using MSGHLP, refer to the *OpenVMS System Messages: Companion Guide for Help Message Users*.

3.1.1.2 Procedure Naming Conventions (Recommended)

When you create a procedure and make its name global, you allow other procedures in the same image to call that procedure. The common RTL procedures are examples of procedures with global names. In such an environment, a naming convention is required to prevent any name conflict between global procedures in the same image.

The rules for naming entry points to procedures have the following general form:

```
fac$symbol (Digital supplied)
fac_symbol (user-supplied)

  fac = a two- to four-character facility name.

  symbol = a symbol from one to 27 characters long.
           (The entire procedure name may not exceed
           31 characters in length.)
```

The facility name and symbol name are separated by a dollar sign ($) if the procedure is supplied by Digital and by an underscore (_) if the procedure is supplied by the user. This convention should be used to avoid conflict between Digital and user procedure names.

The procedure name usually consists of a verb and an object that together describe the action of the procedure. For example, the Run-Time Library procedure intended to get virtual memory is called LIB$GET_VM.

Some procedures, even though they have global names, are not intended to be called from outside the facility in which they are located. These procedures are only available internally, within a set of procedures, and do not by themselves provide any functionality for the facility. The names for these procedures contain a double dollar sign ($$) if they are supplied by Digital or a triple underscore (_ __) if they are supplied by the user. (Three underscores are necessary to avoid conflict with user-defined condition value symbols, which use two underscores.)

Table 3–2 shows examples of procedure entry point names.

Table 3–2 Naming Procedure Entry Points

Procedure Name	Description
LIB$GET_VM	Digital supplied global procedure
LIB_PRINT_REPORT	User-supplied global procedure
OTS$$INTERNAL	Digital supplied internal procedure
LIB_ _ _ADD_TAX	User-supplied internal procedure

3.1.1.3 File Naming Conventions (Recommended)

You should derive your file name from the names of the procedures contained in the module that comprises the file.

If a module contains a single procedure, the file name consists of the procedure name. You can remove dollar signs and underscores, but this is not required. File types are the standard default file types for the source language. For example, the file containing the RTL procedure MTH$EXP is named MTHEXP.MAR. This name makes it obvious that the file MTHEXP.MAR contains the procedure MTH$EXP and is written in VAX MACRO.

Sometimes, the module comprising the file will contain more than one procedure. For example, the RTL procedures LIB$GET_VM and LIB$FREE_VM are contained in the same module and thus in the same file. In this case, a more general file name is used, composed of the facility prefix (LIB) and the first nouns common to all procedure names in the module (VM). Thus, the name for the file containing procedures LIB$GET_VM and LIB$FREE_VM is LIBVM.B32. (The file type B32 indicates that the module is written in VAX BLISS-32.)

3.1.1.4 Module Naming Conventions (Required)

Module names are identical to file names except that module names do not have extensions, and the dollar sign ($) or underscore (_), which separates the facility prefix and symbol name, is not removed.

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.

3.1.1.5 PSECT Naming Conventions (Required)

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

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

The collating sequence for leading underscores causes the linker to place all library procedures after the user program in the executable image. This prevents a library procedure from being placed between two user modules and adversely affecting any byte or word displacement addressing contained in the user programs.

Not all languages give you control over PSECT names. In VAX BASIC and VAX Pascal, it is not possible to control PSECT names except through use of COMMON. However, using COMMON is not recommended.

For additional information about declaring PSECTs, see the appropriate language reference manual.

3.1.1.6 Lock Resource Naming Conventions (Recommended)

When using the lock manager, the resource names of root-level locks (locks without a parent) should be derived from the facility name. The naming convention used is:

```
fac$name = Digital-supplied resource name
fac_name = user-supplied resource name
```

Following this convention will prevent unintended resource conflicts.

3.1.1.7 Global Variable Naming Conventions (Recommended)

Global variables should be named using the following format:

```
fac$Gt_variablename = Digital-supplied global variable name
fac_Gt_variablename = user-supplied global variable name
```

The letter *t* indicates the contents and usage of the global variable. The possible values of *t* are listed in Table 3–3.

Likewise, the format for addressable global arrays is as follows:

```
fac$At_variablename = Digital-supplied global variable name
fac_At_variablename = user-supplied global variable name
```

Again, the letter *t* indicates the contents and usage of the addressable global array. The possible values of t are listed in Table 3–3.

Table 3–3 Code for the Content and Usage of Global Variables

t	Content and Usage of Global Variable
A	Address
B	Byte integer
C	Single character
D	D_floating
E	Reserved for Digital
F	F_floating
FS	S_floating
FT	T_floating
G	G_floating
†H	H_floating
I	Reserved for integer extensions
J	Reserved for customers for escape to other codes
K	Constant
L	Longword integer

†VAX specific

(continued on next page)

Table 3–3 (Cont.) Code for the Content and Usage of Global Variables

t	Content and Usage of Global Variable
M	Field mask
N	Numeric string (all byte forms)
O	Octaword
P	Packed string
Q	Quadword integer
R	Records (structure)
S	Field size
T	Text (character) string
U	Smallest unit of addressable storage
V	Bit field
W	Word integer
X	Context dependent (generic)
Y	Context dependent (generic)
Z	Unspecified or nonstandard

3.1.1.8 Status Code and Condition Value Naming Conventions (Required)

The format of status codes and condition values is as follows:

```
fac$_status = Digital-supplied status code or condition value
fac__status = user-supplied status code or condition value
```

3.1.2 Using Common Source Files (Recommended)

For some applications, it may be necessary to make identical argument declarations in several modules. Languages supported by the OpenVMS operating system let you centralize these declarations in one place by using common source files. Table 3–4 summarizes the common source file declarations for languages supported by the OpenVMS operating system.

Table 3–4 How to Declare Common Source Files

Language	Common Source File Declaration
DEC Ada	To share common declarations among DEC Ada programs, you include the declarations in a package (as a separate compilation unit) and provide visibility to the package by using a WITH clause in programs you want to share the common declarations.
BASIC	You can use the BASIC %INCLUDE directive in your program to include the common source file, or a CDD record.
BLISS-32	Your source program can contain a REQUIRE or LIBRARY list option that specifies a file to be included at the point of the declaration.
C	Include a preprocessor directive to include a file or a dictionary.

(continued on next page)

Table 3–4 (Cont.) How to Declare Common Source Files

Language	Common Source File Declaration
COBOL	The COPY statement specifies source text from a COBOL library file, a Librarian file, or a Common Data Dictionary (CDD) record description that is to be included in the source program.
DIBOL	The INCLUDE directive will include a common source from a separate file, text library, or CDD record.
FORTRAN	The INCLUDE statement specifies a file or library module to be included at the point of the statement. You may also use a CDD record.
Assembly language	An auxiliary source file or macro library can be specified in the command line or by using a CDD record.
Pascal	The %INCLUDE directive and INHERIT attribute specify files to be included at the point of the declarations. You may also use a CDD record.
PL/I	The %INCLUDE preprocessor statement specifies a file to be inserted as source. You may also use a CDD record.
RPG II	An auxiliary source file can be specified in the command line.
SCAN	The INCLUDE FILE statement can be used to include common source from other SCAN source language modules. SCAN does not have text library or CDD support.

3.1.3 Using OpenVMS System Services

Not all OpenVMS system services are modular, according to the definitions in this manual. Procedures that call nonmodular system services are nonmodular themselves. If your procedure uses a nonmodular system service, you should list the system service in the Side Effects section of the procedure description. (For information about the procedure description, see Section 2.5.2.) For more information about particular system services and modularity, see the *OpenVMS System Services Reference Manual*.

3.1.4 Invoking Optional User Action Routines

An optional user action routine is a useful way to let the calling program gain control at a critical point in your procedure's algorithm. Success routines and error routines are the most common user action routines. Control is passed from your procedure to the optional error routine if the specified error is encountered within your procedure. To transfer control, the calling program must pass the user action routine as an argument to the called procedure. To make it easy for the calling program to pass information to its action routine, your procedure should supply an optional **user-arg** argument that the calling program can pass to its action routine. Your procedure merely copies the argument list entry of the user argument, if present, to the argument list it passes to the action routine. This achieves the same effect as up-level addressing.

3.1.4.1 Bound Procedure Value (VAX Only)

VAX

On VAX systems, the bound procedure value (DSC$K_DTYPE_BPV) is used by DEC Pascal and other languages where context of the procedure must be known. The procedure might do up-level addressing of a variable defined in a syntactically outer block and allocated in another frame. (If you use a procedure entry mask, this context is specified in the **user-arg** argument.)

For a bound procedure value passed by reference, the argument list entry contains the address of two longwords. The first longword contains the address of the procedure and the second contains the environment pointer to be loaded into R1 before the procedure is called. This environment pointer allows you to specify the context of your action routine enabling you to do up-level addressing. To provide a user action routine using the bound procedure value passed by reference, the calling sequence is as follows:

```
CALL myproc [action-routine [,user-arg]]
```

In this example, **action-routine** is a function call of the bound procedure value type that is passed by reference, and **user-arg** is unspecified.

If you want to use the bound procedure value data type to pass access to a user routine specified as a procedure entry mask, then you must pass the first longword by value and omit the second longword. Then, the user action routine would have this calling sequence:

```
status = action-routine (...[,user-arg])
```

In this example, **status** is a longword condition value that is passed by value, and **user-arg** is unspecified. Your procedure copies the 32-bit argument list entry passed by the calling program to the argument list provided to the action routine. Therefore, the calling program and its action routine can communicate using any data type, access type, passing mechanism, or OpenVMS usage. ♦

3.2 Initializing Modular Procedures

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

- Storing in static storage a value that can only be determined at run time
- Declaring an exit handler using the $DCLEXH system service
- Allocating a processwide resource once
- Opening a file the first time the procedure is called

You must perform initialization carefully to maintain modularity.

Initialization must not affect the calling program. Therefore, avoid initializing by providing an entry point that must be called before any other entry point is called. Providing an entry point that must be called first forces the calling program to provide an initialization entry point to its caller, and so forth. Also, you would have to rewrite your calling programs if you needed to substitute a procedure with an initialization call for one without an initialization call.

If your procedure uses LIB$INITIALIZE, you must preserve a modular environment that does not conflict with the environment set by any other procedure using LIB$INITIALIZE. (For more information, see *OpenVMS Programming Interfaces: Calling a System Routine*.)

Several ways to initialize a procedure are as follows:

- Initialize at compile or link time
- Use the mechanism provided by LIB$INITIALIZE to perform initialization once for each image activation
- Set a first-time flag at run time
- Initialize storage each time it is allocated at run time
- Initialize storage each time a procedure is called at run time

The use of each method is explained in the following sections. Figure 3–2 summarizes these methods.

Figure 3–2 Methods of Initializing

Initialization Needed	Method				
	Initialize at Compile/Link Time	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:		•			

ZK–3085–GE

3.2.1 Initializing Storage

For a procedure to produce predictable results, all statically and dynamically allocated areas must be initialized to known values before they are read. Initialization of dynamically allocated stack and heap data involves writing the data after each allocation and before reading it.

If your procedure has static storage, it is usually initialized to zero. In some languages, you do not need to explicitly initialize static storage. These languages will automatically initialize static storage to zero. To see if the language you are using initializes static storage implicitly, refer to your reference manual for that language.

There are three ways to explicitly initialize storage: you can use an initialization statement, test and set a first-time flag at run time, or use LIB$INITIALIZE. The method of testing and setting a first-time flag is explained in Section 3.3.4.2.

Figure 3–3 shows examples of how languages supported by the OpenVMS operating system initialize a longword, DAT, in static storage using an initialization statement.

3.2.2 Testing and Setting a First-Time Flag

To do first-time initialization, your procedure can test and then set to one a statically allocated first-time flag each time it is called. This flag is initialized to zero at compile or link time.

Setting and testing the flag with the RTL procedure LIB$BBSSI, a Branch on Bit Set and Set (BBSS) VAX instruction, or a Branch on Bit Set and Set Interlocked (VAX BBSSI) instruction, ensures that initialization is executed exactly once. (Some high-level languages provide semantics for accessing these VAX instructions: for instance, the _BBSSI built-in for C.)

However, if your implementation language does not have access to VAX instructions and the procedure is to be AST-reentrant, it must follow these steps:

1. Test the first-time flag.
2. If the first-time flag is set, initialization is complete.
3. If the first-time flag is not set, disable ASTs. Remember the previous state of AST enable, and retest the flag.
4. If the first-time flag is now set, then initialization was performed by an AST that occurred between the first test and the AST disable; enable ASTs if remembered state of ASTs was enable. Initialization is now complete.
5. If the first-time flag is not set, perform the initialization.
6. Set the flag.
7. Enable ASTs if remembered state of ASTs was enable — initialization is complete.

For additional information, see Section 3.3.

Note

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

Figure 3–3 How to Initialize Static Storage

Language	Statement	Initialized Value	Time of Initialization
Ada	X : INTEGER :=1	1	Elaboration time
BASIC	BASIC does not permit static storage within a module, only common static storage.		
BLISS	OWN DAT;	0	Compile time
	OWN DAT INITIAL(0);	0	Compile time
	OWN DAT INITIAL(100);	100	Compile time
C	static int x;	0	Compile time
	static int x = 1;	1	Compile time
	extern int x;	Defined externally[1]	Compile time
	int x;	0	Compile time
	int x = 1;	1	Compile time
	globaldef int x;	0	Compile time
	globaldef int x = 1;	1	Compile time
COBOL	01 NUM PIC	0	Compile time
	01 NUM PIC 9 VALUE 0.	0	Compile time
	01 NUM PIC 9(3) VALUE 100.	100	Compile time
DIBOL	At compile time, fields within records, commons, and/or groups are initialized to spaces or zeros (depending on data type).		
FORTRAN	INTEGER*4 DAT	0	Compile time
	INTEGER*4 DAT/0/	0	Compile time
	DATA DAT /0/	0	Compile time
	DATA DAT /100/	100	Compile time
MACRO	DAT: .BLKL 1	0	Compile time
	DAT: .LONG 0	0	Compile time
	DAT: .LONG 100	100	Compile time
PASCAL VAR	DAT :[STATIC] INTEGER;	0	Compile time
	DAT :[STATIC] INTEGER :=0;	0	Compile time
	DAT :INTEGER : = 100;	100	Compile time
PL/I	STATIC INIT(2)	2	Compile time
	EXTERNAL INIT(3)	3	Compile time
	GLOBALDEF INIT(4)	4	Compile time
	GLOBALREF INIT(5)	5	Compile time
RPG	RPG II has static storage at the module level only. Numeric variables are initialized to zero and alphanumeric variables are initialized to spaces at compile time.		
SCAN	No initialization clauses–use assignment statement.		

[1] You cannot initialize a variable declared with an external attribute.

ZK–6507–GE

Example 3–1 illustrates the use of a first-time flag in a Pascal program to allocate a resource.

Example 3–1 Pascal Program That Uses a First-Time Flag

```
{+}
{ Program to demonstrate the use of a first-time flag when allocating
{ a resource.  This technique is AST-reentrant, but is NOT multithread
{ reentrant.
{-}

PROGRAM ALLOCATE;

CONST
    VM_SIZE = 512;

VAR
    INITIALIZED : BOOLEAN := FALSE;
    VM_ADDRESS :  INTEGER := 0;
    AST_STATUS :  INTEGER := 0;
    VM_STATUS :   INTEGER := 0;
    DISABLE :     INTEGER := 0;

FUNCTION LIB$GET_VM (SIZE : INTEGER; VAR ADDR : INTEGER) : INTEGER; EXTERNAL;
FUNCTION SYS$SETAST (VAR STATUS : INTEGER) : INTEGER; EXTERNAL;

BEGIN

    {+}
    { Check the first-time flag.  If set, initialization has been
    { performed already.
    {-}

    IF NOT (INITIALIZED)
    THEN
        BEGIN

        {+}
        { Disable ASTs, and remember the previous state.
        {-}

        AST_STATUS := SYS$SETAST (DISABLE);

        {+}
        { Now, recheck the flag.  If it is now set, initialization was
        { performed by another invocation of this procedure between when
        { the flag was first tested and now.  Otherwise, initialization
        { is performed here.
        {-}

        IF NOT (INITIALIZED)
        THEN
            BEGIN

            {+}
            { Perform the initialization.
            {-}

            VM_STATUS := LIB$GET_VM (VM_SIZE, VM_ADDRESS);

            {+}
            { Set the first-time flag, indicating initialization complete.
            {-}
            INITIALIZED := TRUE;
            END;
```

(continued on next page)

Example 3–1 (Cont.) Pascal Program That Uses a First-Time Flag

```
{+}
{ Restore ASTs to the previous state.
{-}

AST_STATUS := SYS$SETAST (AST_STATUS);
END;

END.
```

3.2.3 Using LIB$INITIALIZE

One way to initialize a value at run time is by using the PSECT LIB$INITIALIZE. An example of a value that you may need to initialize at run time is a seed for a random number generator.

To use LIB$INITIALIZE to initialize a value at run time, you must do the following:

1. Write the main program.
2. Write an initialization procedure.
3. Write a MACRO or BLISS program to add the address of that initialization procedure to PSECT LIB$INITIALIZE.
4. Compile the initialization procedure, main program, and MACRO program.
5. Link the initialization procedure, main program, and MACRO program.
6. Run the main program.

Assuming that you have completed the main program, the first thing you must do is to write an initialization procedure. If, for example, you are going to use LIB$INITIALIZE to initialize a value for a random number generator, you might write an initialization procedure to set the seed equal to the current time. This would generate a different seed for each initialization because the time is constantly changing. One possible initialization procedure is shown in Example 3–2.

Once you have defined the initialization procedure, you must write the MACRO program to add the address of that initialization procedure to PSECT LIB$INITIALIZE. The format for this MACRO program is very simple, as seen in Example 3–3.

To modify this MACRO program for use in your own procedures, substitute the name of your initialization procedure for MY_INIT_ROUTINE.

Example 3–2 BASIC Initialization Procedure for LIB$INITIALIZE

```
100     !+
        ! Initialization routine.  A common piece of data, called SEED,
        ! is initialized based on the number of CPU seconds used by
        ! this process so far.
        !-
        SUB MY_INIT_ROUTINE(ONE,TWO,THREE,FOUR,FIVE,SIX)
        COMMON (MY_DATA) LONG SEED
        PRINT "Now in initialization routine."
        CURRENT_TIME = TIME(1)
        SEED = CURRENT_TIME
        END SUB
```

Example 3–3 Program to Add Address to PSECT LIB$INITIALIZE

```
;+
; Make references to external routines used.
;-
        .EXTRN  LIB$INITIALIZE
        .EXTRN  MY_INIT_ROUTINE
;+
; Make a contribution to the PSECT LIB$INITIALIZE.
;-
        .PSECT    LIB$INITIALIZE USR,GBL,NOEXE,NOWRT,LONG
        .ADDRESS  MY_INIT_ROUTINE
        .END
```

Once you have written the initialization procedure and the MACRO program to add the dispatch address to PSECT LIB$INITIALIZE, you can link and run your program. The sample program in Example 3–4 can be initialized in this manner.

Example 3–4 BASIC Main Program

```
10      !+
        ! Mainline.  The value of SEED is printed.
        ! The linker initializes this value to zero, but because
        ! LIB$INITIALIZE is used, an initialization routine is run
        ! before control is transferred
        ! here, and the value of SEED is changed to a
        ! somewhat random value.
        !-
        COMMON (MY_DATA) LONG SEED
        PRINT "Now in mainline.  The seed is initialized to: ";SEED
32767   END
```

To run LIB$INITIALIZE on the program in Example 3–4 and initialize the value of SEED at run time, enter the following commands:

```
$ BASIC MAIN
$ BASIC INIT
$ MACRO INIT_SECTION
$ LINK MAIN,INIT,LIBRARY
$ RUN MAIN
```

The following is an example of the output generated by these steps:

```
Now in initialization routine.
Now in mainline. The seed is initialized to: 4099
```

If your procedure establishes a condition handler by calling LIB$INITIALIZE before a main program, the action of this handler might conflict with other condition handlers established by other procedures before the main program.

3.3 Writing AST-Reentrant Code

This section describes coding techniques for modular procedures that use the asynchronous system trap (AST) interrupt mechanism or that permit calling programs to use it.

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

3.3.1 What Is an AST?

An asynchronous system trap (AST) is an OpenVMS mechanism for providing a software interrupt when an external event occurs. One example of this type of interrupt occurs when a user presses Ctrl/C. When the external event occurs, the OpenVMS operating system interrupts the execution of the current process and calls a procedure that you supply. This procedure is referred to as the AST handler.

Some OpenVMS system services let an external event interrupt a process. Because the interrupt occurs out of sequence with respect to process execution, the interrupt mechanism is called an asynchronous system trap. The AST interrupt transfers control to the AST handler that services the event. This AST handler can call other procedures, including library procedures.

The AST handler you provide and any procedures it calls are said to be executing at AST level. While at AST level, a process cannot be interrupted a second time at the same access mode. The process runs to completion at the AST level before the non-AST level procedure resumes.

A process is executing either at AST level or at non-AST level and thus consists of two threads of execution, one thread at each level. Keep in mind that these levels are threads of the same process and not separate processes.

When your AST handler finishes servicing the event, it returns control to its caller. The interrupted procedure continues execution from the point of interruption.

For example, you could call the Set Timer system service ($SETIMR) to specify the address of an AST-level procedure to be executed after a specified amount of time has elapsed. At the specified time, the system generates an AST interrupt by stopping the procedure that is currently executing and calling the specified AST handler.

For information about the implementing AST interrupts by system services, see the *OpenVMS System Services Reference Manual.*

3.3.2 AST-Reentrancy Versus Full-Reentrancy

A procedure is AST-reentrant if it meets the following conditions:

- It can be interrupted at any point, permitting itself or any related procedure to be called (reentered).
- It executes correctly when it continues from the point of interruption.

Do not confuse the term *AST-reentrant* with the term *fully-reentrant*. Full reentrancy refers to a more restrictive set of conditions.

In an AST-reentrant environment, the AST thread is expected to complete regardless of whether it encounters a locked resource. When the AST thread encounters a locked resource in an AST-reentrant environment, it expects to be given a new resource, or else it is expected to return an error message. It is never expected to wait for the resource that the non-AST level has locked.

In a fully-reentrant environment, all threads are treated equally when they encounter a locked resource; they wait for the resource to be freed. In a fully-reentrant environment, AST threads are not given any special treatment. The DEC Ada environment is an example of a fully-reentrant environment. In such a situation, there can be more than two threads of concurrent execution, and each thread can alternately progress toward an end.

Note

It is highly desirable that future code satisfy the more stringent requirement of being fully-reentrant. Full reentrancy is important for procedures that will be called from multithread environments, such as Ada tasks. For more information, refer to the Ada documentation.

DECthreads, the Digital multithreading run-time library, provides a portable interface for creating and controlling multiple threads of execution within the address space provided by a single process on AXP or VAX processors.

3.3.3 Writing AST-Reentrant Modular Procedures

To use AST interrupts, you must write an AST handler to take control at AST level. An AST handler can be written in any language. Because the particulars of writing an AST handler differ from one language to the next, see the reference manual for the language you are working in for more details.

In general, an AST handler must follow these guidelines:

- It must be separate from the procedure that is currently executing.
- It must not modify data or instructions used by the interrupted procedure or its callers.
- If it calls any other procedures, they must all be AST-reentrant.
- The AST handler cannot stall or use busy wait to avoid being called before the non-AST level is out of a critical section of code. Once the AST handler has begun executing, it cannot be interrupted by anything at a non-AST level. In fact, the only thing that can interrupt the AST handler is another procedure running at AST level in a more privileged access mode.

  If you attempt to use a busy wait and expect to change the condition from the non-AST level, the AST level circles the "busy wait" in an infinite loop. The process continues to loop because the non-AST level does not continue executing until the AST thread has finished and thus is never able to change the value in the "busy wait" condition.
- You cannot use the lock manager to protect a resource being accessed at non-AST level from being accessed at AST level. The lock manager is designed to lock resources between separate processes, not different threads (AST and non-AST) of the same process.

- Avoid using static storage. A procedure that does not use static storage, calls only AST-reentrant procedures, and does no up-level addressing, is automatically AST-reentrant.

3.3.4 How to Eliminate Race Conditions During Concurrent Access

When using AST interrupts, you might encounter two problems: race conditions and deadlocks. A race condition occurs when your AST handler attempts to use a nonshareable resource already in use by the non-AST thread of execution.

If you allow the AST handler to wait for the resource (for example, by waiting for an event flag to be set by the non-AST level code of the same access mode), you have caused a form of deadlock. A deadlock occurs because the non-AST level code cannot execute to free the resource until the AST-level code has finished executing. The AST level code cannot continue either, because the non-AST level code has effectively locked the resource.

A race condition occurs when you attempt to access or modify the same data in static storage by both the AST and non-AST levels of a process. For example, if an AST begins executing while the non-AST level is modifying data in static storage, that data may be left in a nonstable state while the AST handler executes. To prevent a race condition, you should allow only one thread at a time to modify data. Use atomic modify operations provided by your HLL, which correctly interlock such access.

If a procedure does not modify any static storage, then it is both AST-reentrant and fully-reentrant. Your procedure can eliminate conflict when accessing and modifying data in static storage in the following ways:

- Detecting concurrency of access to data using test and set instructions at entry to and exit from data storage. The procedure may then report an error, or retry the operation (when appropriate) if concurrency is detected.
- Keeping 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.
- Disabling AST interrupts upon entry and restore the enable state upon exiting.

3.3.4.1 Performing All Accesses in One Instruction

All data modification in static storage can be performed in a single uninterruptible instruction for some applications. However, this method applies only to the VAX MACRO assembly language, and even then does not apply to emulated instructions.

For example, you can use queue instructions to maintain a linked list in a single instruction instead of modifying the forward and backward fields of the list in several instructions. You can use a single queue instruction at the beginning of your procedure to remove one section, and another can be used at the end to insert the section back in the queue.

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

Example 3–5 VAX MACRO Program Showing Use of Queue Instructions to Perform All Accesses in a Single Instruction

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

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

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

Example 3–5 illustrates an AST-reentrant procedure that uses queue instructions to control allocation of quadword blocks.

3.3.4.2 Using Test and Set Instructions

One method of eliminating the possibility of a race condition or deadlock is to use test and set instructions to detect concurrent access. You can detect concurrent access of static storage at both AST and non-AST levels by adding the following steps to your procedures:

1. Place a Branch on Bit Set and Set (BBSS or BBSSI) instruction immediately before your procedure accesses static storage. Alternatively, use LIB$BBSSI or semantics provided by your compiler.
2. Access or modify static storage, or both.
3. Place a Branch on Bit Clear and Clear (BBCC or BBCCI) instruction immediately after your procedure has completed access to static storage. Alternatively use LIB$BBSSI or semantics provided by your compiler.

The BBSS instruction detects that a concurrency conflict is about to take place before static storage has been accessed. If the storage is being accessed by multiple processors, you must use BBSSI and BBCCI.

There are two alternate techniques for resolving concurrency conflicts detected by the BBSS and BBCC instructions:

- Use separate, statically allocated areas for storage at the AST and non-AST levels. When the BBSS instruction detects concurrency at the

[1] This example could be recoded using REMQHI and INSQHI and avoid the need for a first time flag.

beginning, use the second allocated area. Note that this technique does not work if an exception condition occurs between execution of the BBSS instruction and the BBCC instruction, or if your procedure has not established a condition handler. This is because a condition handler established by the calling program might also simultaneously call your procedure.

- Reexecute your procedure if concurrency is detected. When the BBCC instruction detects this concurrency, branch back to the beginning of your procedure and try again.

Example 3–6 illustrates the latter technique. This MACRO procedure, LIB_GET_INUM, allocates and deallocates identifying numbers.

Example 3–6 MACRO Program Showing Use of Test and Set Instructions

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

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

3.3.4.3 Keeping a Call-in-Progress Count

If the database is to be kept separate between calls, you can keep track of when your procedure is called by using a call-in-progress count. Before database access, the count is incremented and used as an index for an address table of the separate databases. You should check for a count that exceeds the table length. After the database has been accessed, the count is decremented.

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

3.3.4.4 Disabling AST Interrupts

A procedure is also considered AST-reentrant if AST interrupts are disabled while critical sections of code execute. However, this method of maintaining AST reentrancy is not recommended.

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 once the critical section of code has finished executing. However, this technique could adversely affect performance of real-time programs using AST interrupts. The $SETAST system service, which is used to enable and disable AST interrupts, is time consuming. Therefore, you should avoid disabling AST interrupts whenever you can by using the techniques described in Section 3.3.4.1 to Section 3.3.4.3.

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

Example 3–7 demonstrates how $SETAST can be used to disable ASTs and then restore the previous state of the enable.

Example 3–7 A FORTRAN Program Disabling and Restoring ASTs

```
!+
! This program demonstrates using the System
! Service SYS$SETAST to disable and then
! reenable AST interrupts.
!-
        INCLUDE '($SSDEF)'
        INTEGER*4 SYS$SETAST
!+
! Turn off ASTs and remember the previous setting.
!-
        ISTAT = SYS$SETAST (%VAL(0))
!+
! The statements in the program during whose
! execution you want ASTs disabled.
!
! If ASTs were previously enabled,
! reenable them.
!-
        IF (ISTAT .EQ. SS$_WASSET) CALL SYS$SETAST( %VAL(1))
        END
```

3.3.5 Performing Input/Output at AST Level

If your procedure performs I/O using OpenVMS RMS (RMS), you must use the following coding techniques 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 RMS error status RMS$_ACT (active) after each $CREATE or $OPEN service. This error indicates that a record operation has already started for the process-permanent file. The error does not occur for files that are not process permanent, 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) or RMS$_BUSY (structure in use) after each $GET and $PUT service. This error indicates that a record operation has already been started for the file. If the error occurs, perform a $WAIT using the same record access block (RAB). When control returns to your procedure, try the $GET or $PUT service again. Repeat this procedure until it succeeds.

- Avoid storing data in an RAB that RMS could still be accessing. You can avoid this situation by doing either of the following:
  - Allocate the RAB on the stack so the AST and non-AST level have separate RABs.
  - Allocate RAB in heap or 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 input/output (I/O that is always completed before returning control to your procedure), you can allocate the RAB in either of these ways. However, the first method is more reliable, because it does not use static storage and therefore does not become corrupted if an exception is signaled.

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

3.3.6 Condition Handling at AST Level

You should not allow an exception to propagate out of an AST handler because the exception might be caught by any procedure that is active at the time of the AST. Condition handlers for other active procedures might react as if the exception was caused by a procedure that they had called.

Another reason for not allowing exceptions to propagate out of an AST handler is that, for run-time environments that use multiple threads in a process such as Ada, it cannot be determined which stack of the threads of execution is used to deliver the AST. (The AST is delivered on the stack of whichever thread is active at the time of the AST interrupt.)

It is best to catch all exceptions in the AST handler and not allow them to propagate.

4

Testing Modular Procedures

A successful test system is one that uncovers errors. To ensure successful testing, plan how to test your procedures while you are designing them and begin testing while you are coding. You should test for the following:

- To ensure that the procedure you developed fulfills your requirements or specifications.

  Carefully test the functionality to ensure that the procedure does everything that it is supposed to do. The methods you use to test this aspect of your procedure depend upon the functions your procedure performs.

- Ensure that the procedure is modular and executes without error.

This chapter focuses on testing procedures for modularity. Modularity is especially important to procedures that will be included in a library facility. A procedure that is not modular can adversely affect the results and performance of other procedures that call it.

To ensure modularity within procedures, perform at least the following tests:

- Unit testing
- Language-independence testing
- Integration testing

This chapter discusses methods for designing and administering these types of tests. It also describes reentrancy, performance analysis, and RTL procedures for time and resource monitoring.

4.1 Unit Testing

Before you begin combining units of code (such as subprograms, subroutines, and internal procedures) to form your new procedure, it is essential to ensure that each of these units works separately. Thorough unit testing is important for the following reasons:

- Testing small units separately decreases the level of complexity within the tests.
- It is easier and faster to debug a small unit of code than it is to find an error within several units and their interfaces.
- It makes the integration stage that follows much easier if each of the separate units has been thoroughly tested and the problems corrected.
- The earlier an error is found in development, the less expensive it is to fix.

Unit testing includes the following steps:

1. Review the goals of your procedure
2. Choose test cases

3. Run the tests

The goals of your procedure are chosen at the requirements or specifications stage. As mentioned earlier, this topic is not discussed in this manual because it does not have a significant effect on modularity. However, it does have a significant effect upon whether your final product can be considered successful. If your product does not perform the functions or meet the requirements decided upon at the requirements or specifications stage, it is not a successful project. You should have at least one test for each of the requirements that your procedure was designed to fulfill.

You can use the following two types of tests:

- Black box tests
- White box tests

Black box tests assume that you know nothing about the internal workings of the procedure that you are testing. All that you are interested in is the output that you receive for given sets of input.

White box tests (also called clear box tests) are more complicated because they are designed to step through particular sections of code or algorithms internal to the procedure. They assume that you know, in great detail, the internal workings of the procedure being tested.

4.1.1 Black Box Testing

When you are performing black box testing, you are interested only in the output you receive for particular input values. Execute the procedure repetitively using input from different classes. The best way to do this is to write a command procedure or test driver program to execute the procedure a given number of times using test data that you supply. (For information about writing command procedures, see the *OpenVMS User's Manual*.)

You should execute your procedure with test cases from each of the following categories:

- Expected inputs

  These include the values that you expect your procedure to receive most of the time.

- Boundary values

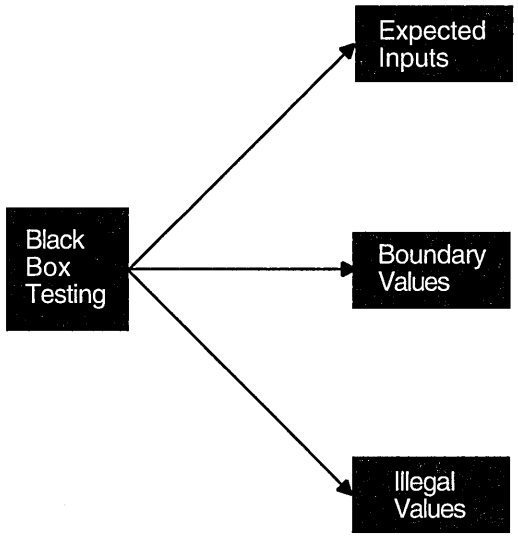
  If your procedure expects an input value from 1 to 999, use 1 and 999 as test cases to make sure that your procedure returns the expected results for the boundary cases.

- Illegal values

  Using the boundary values example, what happens if your procedure receives as input a value that is less than 1 or greater than 999? Does the user receive a useful error message? Does the procedure simply stop, or does it attempt to use values outside its limitations and simply return an incorrect answer? It is essential that you run the procedure using illegal input values to determine the answers to these questions.

Figure 4–1 summarizes the methods of black box testing.

Figure 4–1 Black Box Testing Methods



ZK–4071–GE

4.1.2 White Box Testing

When performing white box testing, unlike black box testing, you must understand the internal workings of the procedure. Keep in mind that you are testing internal workings—the specific lines of code.

To perform white box testing, do the following:

1. Test each statement

   For this step, you need to provide sets of test values that ensure that every statement in the procedure is executed at least once. This includes all statements — even those executed only when optional arguments, user-supplied arguments, subroutines, user-action routines, or specific error codes are present.

2. Test each decision

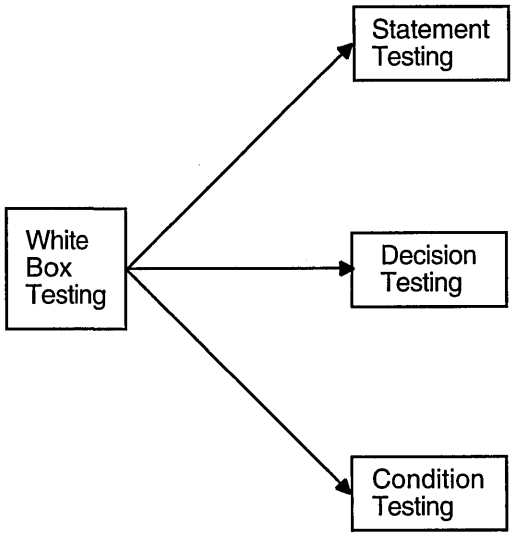
   At this step, your goal is to provide test cases that ensure that each branch of a decision is executed at least once. In the case of a standard Boolean decision, this generally requires providing two values; however, this number may be much greater in the case of compound or nested decisions.

3. Test each condition

   Condition testing requires writing test cases that ensure each condition in a decision takes all possible outcomes at least once and each point of entry to the program or subroutine is invoked at least once. Multiple test values must be supplied in cases of compound and nested loops. In testing the entry points, remember to invoke any optional routines (either internal or external), as well as error handlers. If your procedure contains a JSB entry point, that entry point should also be tested.

Figure 4–2 summarizes white box testing.

Figure 4–2 White Box Tests



ZK–4069–GE

Note that each white box test finds a specific type of error. For example, statement testing does not find an error on a negative value for a condition if the statement is given a positive input the only time it is executed. Therefore, you must perform all three white box tests.

4.2 Language-Independence Testing

For your procedures to be as useful as possible, they must be able to be called by programs in any language. Providing for language independence is essential to producing a useful procedure.

Testing for language independence is a very specific type of unit testing. It ensures that your program executes correctly regardless of the language from which it is called.

To test your procedures for language independence, write several driver programs in languages you have chosen randomly. The driver program need only contain a call to the procedure being tested.

If you do find that your procedures are not language independent, make sure that they conform to the following rules:

- All atomic data must be passed by reference and all strings must be passed by descriptor.

  Adherence to this single guideline is the most important factor in achieving language independence.

- Statements that assume a particular language environment are NOT allowed.

  For example, the statement ON ERROR GO BACK in a BASIC procedure assumes that the calling program is also written in BASIC.

4.3 Integration Testing

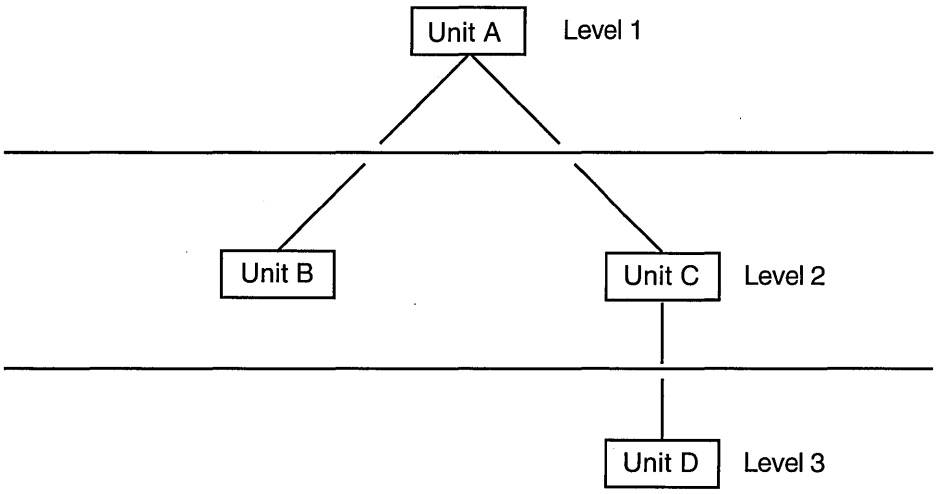
Integration testing is the next logical step following unit testing. Unit testing is designed to test each separate component. Depending on your procedure, that component might be a module, a subprogram, a subroutine, an internal procedure (fac_ ___ name), or a particularly intrinsic piece of code. Once you have determined that each unit works separately, you need to determine that the units also work together to form the complete procedure.

Integration testing can be completed by either of the two methods described in Section 4.3.1 and Section 4.3.2.

4.3.1 All at Once Approach to Integration Testing

One method of integration testing is the all at once approach. In this method, you finish all the units, link them together, and test the completed structure all at once. Use of this method is strongly discouraged, because it makes it very difficult to find the location of errors. For example, look at the organization of the units in the sample procedure shown in Figure 4–3. Assume that this procedure used the all at once approach and found an error; the procedure did not work. There is no way of knowing whether the error was in unit A, unit B, unit C, or unit D.

Figure 4–3 A Sample Procedure for Integration Testing



4.3.2 Incremental Approach to Integration Testing

The recommended approach to integration testing is called incremental testing. Incremental testing involves testing the procedure by starting with one unit and building on it one unit at a time. Each unit should always be subjected to thorough unit testing before it is included in the integration tests.

Incremental integration testing is especially useful for finding the following types of error:

- Problems with the calling interface between units (for example, inconsistent ordering of arguments between the calling and called unit)

- Incorrect assumptions about what values are returned and the units to which they are returned
- Unexpected transfer of control between units

Using the sample procedure in Figure 4–3, complete the test of unit A on level 1 before proceeding to level 2 where you test units A and B in combination. At each level you correct any errors before proceeding to the next level. When you have completed the last step, you know that the entire procedure works correctly.

Because you started at the top of the sample procedure and added units incrementally from lower levels, you were using the top-down approach to integration testing. You could just as easily have started at Level 3 and used the bottom-up approach.

As you can see from the example, there are several distinct advantages to incremental integration testing:

- It is not necessary to wait until the procedure is complete to begin integration testing.
- Debugging is simplified by incremental testing because the modules and interfaces can be tested as the system grows.
- Programming errors in the interfaces and incorrect assumptions between units are discovered at an early stage.
- Because previously tested units are retested as new units are added, the probability of discovering less obvious errors is increased substantially.

4.4 Testing for Reentrancy

It is important to test your procedures for reentrancy before placing them into a library facility. Because ASTs can occur at any time, procedures that are not AST-reentrant may exhibit unexpected behavior. In particular, an AST occurring during storage modification in a procedure that is not AST-reentrant can corrupt the contents of the procedure's storage. (For further information about AST reentrancy, see Section 3.3.)

Full-reentrancy is important to multithread tasking environments such as the environment used by Ada.

To avoid problems with reentrancy, carefully read and follow the coding guidelines described in Section 3.3.

4.4.1 Checking for AST-Reentrancy

There are two methods of checking a procedure for AST-reentrancy. You can use the OpenVMS Debugger or perform a manual desk check.

4.4.1.1 Using the Debugger to Check for AST-Reentrancy

When using the debugger to check for AST-reentrancy, do the following:

1. Create an activation of the procedure.
2. Set watchpoints on all storage used by the procedure.
3. Create a second activation of the procedure using the CALL command. Allow this second activation to run to completion. (The second activation represents the AST-level thread.)

Check to be sure that the AST-level thread of execution does not modify the storage accessed by the non-AST level thread of execution. If the AST-level thread of execution does modify any of that storage, check to ensure that it does not cause any unwanted side effects for the non-AST level thread of execution.

4. Step one instruction in the first activation.
5. Repeat Steps 3 and 4 until the end of the procedure for the first activation.

For more information about the debugger, refer to the *OpenVMS Debugger Manual.*

4.4.1.2 Using Desk Checking to Check for AST-Reentrancy

Desk checking is the term for tracing through a procedure's execution manually. Performing a desk check for AST-reentrancy consists of the following four steps:

1. Create an activation of the procedure being tested and its data using the method you normally use for manually tracing through a procedure.

   This activation represents the non-AST level of your procedure's execution.

2. Create a second activation of the procedure using the process you used above. This second activation represents the AST-level thread of your procedure's activation.

   Trace through the AST-level thread's execution to completion, one statement at a time.

   Remember to update the contents of all storage locations and variables for each instruction of the procedure.

   Check to be sure that the AST-level thread of execution does not modify the storage accessed by the non-AST level thread of execution. If the AST-level thread of execution does modify any of that storage, check to ensure that it does not cause any unwanted side effects for the non-AST level thread of execution.

3. Step through a single statement of the non-AST level thread of execution, remembering to update the contents of all storage locations.
4. Repeat steps 2 and 3 until you have stepped through every statement in the non-AST level thread of execution. (Note that every statement of the AST-level thread is stepped through in each pass through step 2.)

As you can see, what you are actually doing in the process is testing between the execution of every two statements in the procedure. The most rigorous method of applying this type of desk checking for AST-reentrancy is to step through the procedure at the assembly language level and test between each assembly language instruction.

4.4.2 Checking for Full-Reentrancy

Full-reentrancy differs from AST-reentrancy in the number of threads of execution. An AST-reentrant environment can support only two threads of execution, the AST-level thread and the non-AST level thread. Full-reentrancy is important in environments that can support many threads of execution, such as Ada.

A procedure is fully-reentrant if any number of threads of execution can execute to completion without affecting any of the other threads of execution.

Generally, a procedure that is AST-reentrant is also fully reentrant. For further information on full-reentrancy and environments supporting multiple threads of execution, refer to the documentation for DEC Ada.

4.5 Performance Analysis

All timer and resource allocation procedures should make statistics available for performance evaluation and debugging. You should code timer and resource allocation procedures with the following two entry points:

LIB_SHOW_name LIB_STAT_name

4.5.1 SHOW Entry Point

A SHOW entry point provides formatted strings containing the information you need. The calling sequence for a SHOW entry point is as follows:

LIB_SHOW_name [code [,action-routine [,user-arg]]]

code
An optional code (in the form LIB_K_code) designating the statistic you need. Define a separate code for each statistic available; the codes should be the same for the SHOW and STAT entry points. The values associated with the codes start at one for each procedure. The functional specification in the procedure's documentation should list the codes used. If the code is omitted, or zero, the procedure provides all statistics.

action-routine
The address of an action routine. This is an optional argument. If omitted, statistics are written to SYS$OUTPUT.

user-arg
An optional user argument to be passed to the action routine. If omitted, a shortened list is passed to the action routine. The **user-arg** argument, if present, is copied to the argument list passed to the action routine. That is, the argument list entry passed by the calling program is copied to the argument list entry passed to the action routine. The access type, data type, argument form, and passing mechanism can be arbitrary, as agreed between the calling program and the action routine.

The optional action routine should have the following form:

ACTION-ROUTINE (string [,user-arg])

See Section 3.1.4 for an example of the code to invoke a user action routine.

4.5.2 STAT Entry Point

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

LIB_STAT_name (code ,value)

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

value
The value of the returned statistic.

4.6 Monitoring Procedures in the Run-Time Library

The run-time library (RTL) contains several procedures for time and resource monitoring. These RTL procedures and their functions are as follows:

- LIB$SHOW_VM

  LIB$SHOW_VM is a resource monitoring procedure that returns the statistics accumulated from calls to LIB$GET_VM and LIB$FREE_VM.

  The following three statistics are returned by default:

  - Number of successful calls to LIB$GET_VM
  - Number of successful calls to LIB$FREE_VM
  - Number of bytes allocated by LIB$GET_VM but not yet deallocated by LIB$FREE_VM

    LIB$SHOW_VM returns these statistics in the formatted form, nnnn.

- LIB$STAT_VM

  LIB$STAT_VM is a resource monitoring procedure that returns to its caller one of the three statistics available from calls to LIB$GET_VM and LIB$FREE_VM. These are the same statistics that are returned by LIB$SHOW_VM. Unlike LIB$SHOW_VM, which returns the statistics in formatted form to SYS$OUTPUT, LIB$STAT_VM returns the specified statistic in a signed longword integer.

- LIB$SHOW_TIMER

  LIB$SHOW_TIMER is a time monitoring procedure that returns the times and counts accumulated since the last call to LIB$INIT_TIMER and displays them on SYS$OUTPUT. A user-supplied action routine may alter this default behavior.

  The following statistics are provided by default:

  - Elapsed real time
  - Elapsed CPU time
  - Count of buffered I/O operations
  - Count of direct I/O operations
  - Count of page faults

- LIB$STAT_TIMER

  LIB$STAT_TIMER is a time monitoring procedure that returns the same information as LIB$SHOW_TIMER. The difference is that LIB$STAT_TIMER returns the information as an unsigned longword or quadword, whereas LIB$SHOW_TIMER returns the information in the format hhhh:mm:ss:cc for times and the format nnnn for counts. In addition, LIB$STAT_TIMER returns only one of the five available statistics per call.

For more information about these time and resource monitoring procedures, see the *OpenVMS RTL Library (LIB$) Manual*.

5
Integrating Modular Procedures

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

Digital supplies several procedure libraries, such as the Run-Time Library, that support components of the OpenVMS operating system. You can use procedures in the Run-Time Library to perform frequently used operations by including calls to Run-Time Library procedures in your program. The linker automatically searches the default libraries to resolve references to Run-Time Library procedures. (For information about the procedures available in the Run-Time Library, see the *OpenVMS Programming Concepts Manual.*)

This chapter briefly describes how you can create your own procedure libraries and shareable images. For more information about creating libraries and shareable images, use the guidelines in the *OpenVMS Linker Utility Manual.*

5.1 Creating Facility Prefixes

A facility prefix is the group identifier for a set of related procedures contained in a library facility. The facility prefix appears in the procedure name of every procedure in that library facility. An example of a library facility is the Screen Management facility in the Run-Time Library. The names of all the procedures in the Screen Management facility begin with SMG; for example, SMG$ERASE_CHARS.

To create your own facility prefix, follow these steps:

1. Choose a facility prefix. This prefix can be from 1 to 27 characters in length. However, it is recommended that you choose facility prefixes between 2 and 4 characters.
2. If your facility will be generating messages, you must specify a unique facility number in the message source file. This number can range from 0 to 4095. Any number within this range and not being used by someone else on your system is acceptable. This facility number will be used by the message utility in generating the condition value for the message.

   Bit 27 (STS$V_CUST_DEF) of a condition value indicates whether that value is supplied by the user or by Digital. This bit must be 1 if the facility number is user created. For more information, see the *OpenVMS System Messages and Recovery Procedures Reference Manual.*
3. Use the facility prefix when naming all procedures within the new facility. Remember to follow the naming conventions described in Section 3.1.1.

5.2 Creating Object Module Libraries

In addition to using the system default object module libraries, you can create your own object module libraries. An object module library that you create can contain object files produced by any language compiler supported by the VMS operating system.

For more information about creating object module libraries, see the *OpenVMS Linker Utility Manual*.

5.3 Creating Shareable Image Libraries

If you have a collection of procedures you expect a number of users to use, you can group these procedures into a shareable image library. A shareable image library is similar to an object library, except that it has been prelinked so that all references between procedures in the library have already been resolved.

A shareable image has the following advantages:

- Conserves memory space

  Several processes can "share" a single copy of a shareable image rather than each process retrieving its own copy from the disk.

- Conserves disk storage space

  Programs linked to a shareable image share a single disk copy of the library code rather than each program including the code in its own executable image.

- Shortens link time

  Because the internal references in the library have already been resolved, there is less work for the linker.

- Allows for updates without relinking

  You can supply a new version of a shareable image that can automatically be used by all programs linked to it without the need for the users to relink their programs.

For more information about creating shareable image libraries, see the *OpenVMS Linker Utility Manual*.

6
Maintaining Modular Procedures

This chapter describes important aspects of maintaining modular procedures. Specifically, it covers the following topics:

- Making your procedures upwardly compatible
- Regression testing
- Adding arguments to existing routines
- Updating libraries

6.1 Making Your Procedures Upwardly Compatible

Upward compatibility is very important when maintaining procedures. If a procedure is upwardly compatible, then changes and updates to the procedure do not affect executing and using previous versions of that procedure.

For example, imagine a user-written procedure named LIB_TOTAL_BILL. The calling sequence for this procedure is as follows:

```
CALL LIB_TOTAL_BILL (sale, tax)
```

Assume that the user who wrote this procedure decided to update the procedure so that it could be used to calculate the total bill for credit-card customers. To do this, a third argument, **interest**, must be added. To be upwardly compatible, adding the argument **interest** must not conflict with the way the procedure was previously run. The new calling sequence would be as follows:

```
CALL LIB_TOTAL_BILL (sale, tax [,interest])
```

The procedure should be written so that the user can still call the procedure as it was called before, simply omitting the **interest** argument.

If, in the updated version of this procedure, the user can still follow the calling sequence of the previous versions, the procedure is said to be upwardly compatible.

To ensure that your procedures are compatible with future versions of a shareable image, see the *OpenVMS Linker Utility Manual.*

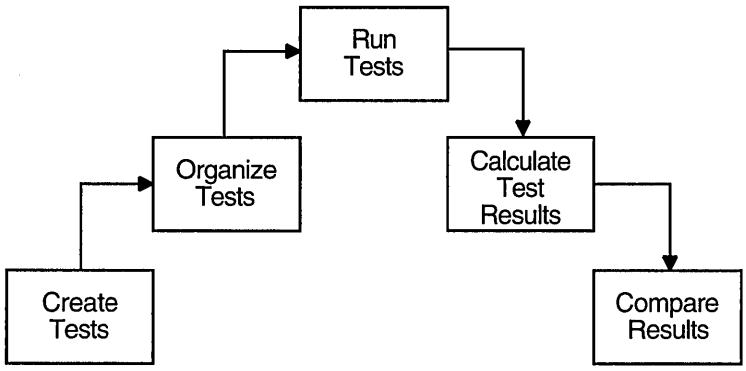
6.2 Regression Testing

Regression testing is a method of ensuring that new features added to a procedure do not affect the correct execution of previously tested features. In regression testing, you run established software tests and compare test results with expected results. If the actual results do not agree with what you expected, the software being tested may have errors. If errors do exist, the software being tested is said to have regressed.

Regression testing includes the following steps, as illustrated in Figure 6–1:

1. Create tests by writing command files to test your software.
2. Organize files to allow easy access to tests as they are needed.
3. Run tests as follows:
   - To run a single test, submit its command file to the batch queue.
   - To run multiple tests, create a command file that submits each test to the batch queue.
4. Calculate the expected test results either by hand or by using previously tested software.
5. Compare actual test results to the results you expected. If there are inconsistencies, repeat your calculation in step 4. If the inconsistency still exists, examine the changes you have made to the software to discover the error.

Figure 6–1 Regression Testing



ZK–4061–GE

It is important to write new tests and repeat the regression testing steps every time you add new functionality to the procedure. If you do not do so, the procedure may regress while the errors go undetected.

6.3 Adding Arguments to Existing Routines

During the normal course of maintenance, it sometimes becomes necessary to pass new or additional information to an existing procedure rather than create a new procedure. This new information may be passed to the procedure in one of the following two ways:

- Directly, by adding new arguments to the procedure
- Indirectly, using an argument block

6.3.1 Adding New Arguments to the Procedure

There are two rules you must follow when directly adding new arguments to a procedure:

- New arguments must be added at the end of the existing argument list.
- New arguments must be optional.

It is important that new arguments be added at the end of the existing argument list to maintain upward compatibility. If you change the order of the existing arguments by placing the new argument at the beginning or middle of the list, all applications written with the previous version of the procedure will no longer work.

Your procedure should also treat the new argument as an optional argument. If the new argument is required, applications that used the previous version of the procedure are invalidated.

Because you cannot assume that all previously written applications will be rewritten to include the procedure's new argument, the procedure must test for the argument's presence before attempting to access it. If the procedure does not verify the presence of the new argument and attempts to access that argument when it is not present, the results will be unpredictable.

The passing mechanism of the new argument must conform to the guidelines in Section 2.2.1.

6.3.2 Using Argument Blocks

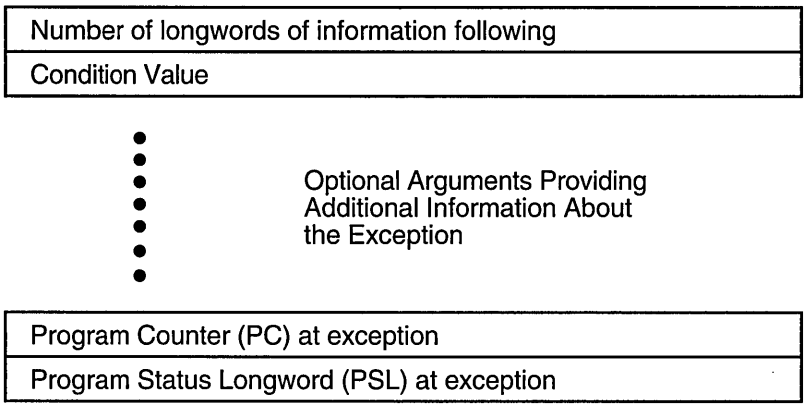
By using an argument block, you can avoid adding multiple arguments to your procedure. When an argument block is used, the calling program passes a single argument to the called procedure. This argument is the address of an argument block. The argument block is a block of information containing any information agreed on by the calling and called procedures. This information is required by the called procedure to perform its task.

The argument block itself is simply a contiguous piece of virtual memory. The information contained in the argument block can be numeric or scalar data, descriptors, bit vectors, and so on. The format is simply agreed on by the users of the procedure and its writer.

The first longword in the argument block contains the length of the block. The length can be in bytes or longwords, but it must be agreed on by both the calling program and the called process and be implemented and documented as such.

One example of an argument block is the signal argument vector used in condition handling. A condition handler is called with a signal argument vector and a mechanism argument vector. Each vector is an example of an argument block. The signal argument vector in Figure 6–2 is an example of an argument block.

Figure 6–2 One Type of Argument Block, the Signal Argument Vector



ZK–4030–GE

As you can see, the signal argument vector contains the number of longwords of actual information in its first longword. What information actually follows depends on the condition value of the signal.

Note that, if you lengthen an argument block to provide new information to a called procedure, your procedure should check the length of the argument block for validity before attempting to access the information. As with adding new arguments directly to a procedure, the calling program may have been written to pass the previous, shorter argument block. If your procedure does not check and attempts to access information past the end of the actual argument block, the results will be unpredictable.

6.4 Updating Libraries

Any time modifications or enhancements are made to modular procedures that are a part of a library, the library containing the procedures must be updated to reflect the new or changed procedures.

6.4.1 Updating Object Libraries

If the updated procedures are in an object library, the library needs to be updated so that subsequent access to that library by LINK or other commands will access the object modules for the new or changed procedures.

To update an object library, use the LIBRARY command with the REPLACE qualifier, as follows:

```
$ LIBRARY /REPLACE library-name filespec[,...]
```

In this example library-name is the name you have given the library. The default file type for library-name is OLB. The name of an object module is filespec. The default file type for filespec is OBJ.

6.4.2 Updating Shareable Images

If the updated procedures are part of a shareable image, the shareable image needs to be relinked so that it contains the new or changed versions of any updated object modules. If new procedures are added, the transfer vector (on VAX systems) or symbol vector (on AXP systems) needs to be updated and recompiled prior to relinking the shareable image. If new modules are added, the linker options file needs to be updated prior to relinking. If new procedures and new modules are added, then the transfer vector (on VAX systems) or symbol vector (on AXP systems) and the linker options file will need to be updated. If the transfer vector (on VAX systems) or symbol vector (on AXP systems) is changed, the minor identification value of the GSMATCH must be incremented by one. When this has been done, the shareable image can be relinked.

For more information about updating shareable images, see the *OpenVMS Linker Utility Manual*.

A

Summary of Modular Programming Guidelines

This appendix summarizes the modular programming guidelines that are described in this manual. References to the appropriate sections appear after each guideline. The word Optional appears before the section reference if the guideline is not required to maintain modularity.

A.1 Coding Rules

The coding rules in this section pertain to all procedures. These rules are grouped in the following categories:

- Calling interface
- Initialization
- Reporting exception conditions
- AST-reentrancy
- Resource allocation
- Format and content of Coded modules
- Upward compatibility

Detailed descriptions of the rules for each of these categories are presented in the sections that follow.

A.1.1 Calling Interface

- Calls to procedures must follow the *OpenVMS Calling Standard*. Some elements of this standard restrict procedures to a subset of the *OpenVMS Calling Standard* to increase the ability of procedures to call each other. (See *OpenVMS Programming Interfaces: Calling a System Routine*.)
- A procedure makes no assumptions about its environment other than those of this standard. In particular, to operate as specified, a procedure neither makes assumptions about, or places requirements on, the calling program.
- A procedure should not call other procedures or system services if the resulting combination violates this standard from the calling program's viewpoint. A procedure can call other procedures or system services that do not follow optional elements of this standard. However, if the resulting combination (as seen from the calling program) does not follow the optional elements, the calling procedure must indicate such nonconformance in its documentation. (See Section 3.1.3.)
- A modular procedure must provide an interface to its callers that allows the callers to follow all required elements of this standard.
- Each module should only contain a single public entry point. (Optional.)

VAX

- On VAX systems, when a procedure uses a JSB entry point, it should also provide an equivalent call entry point to maintain language independence. This is because, although JSB calling sequences may execute faster than procedure calls, an explicit JSB linkage to an external routine may not be provided in some high-level languages. (Optional. See Section 2.3.) ♦
- The order of required arguments should be the same as that of the hardware instructions, namely, read, modify, and write. Optional arguments follow in the same order. However, if a function value is large or is of type string, the first argument specifies where to store the function value, and all other arguments are shifted one position to the right. (See Section 2.2.4.)
- A procedure's caller should indicate omitted trailing optional arguments either by passing argument list entries that contain zero or by passing a shortened argument list. However, system services require trailing arguments and do not adhere to this guideline. (Optional. See Section 2.2.5.)
- String arguments should always be passed by descriptor. (See Section 4.2.)
- Procedures must not accept data from, nor return data to, their calling programs by using implicit overlaid PSECTs or implicit global data areas. All arguments accepted from or returned to the calling program must use the argument list and function value registers. (See Section 2.2.2.)
- A procedure cannot assume that the implicit outputs of procedures it calls will remain unchanged if subsequently used as implicit inputs to those procedures or to companion procedures. (See Section 2.2.2.)

VAX

- On VAX systems, position-independent references (in a module) to another PSECT must use longword relative addressing so the linker can correctly allocate the data PSECT anywhere with respect to the code PSECT no matter how many code modules are included. ♦

VAX

- On VAX systems, external references must use general-mode addressing to allow the referenced procedures to be put in a shareable image without requiring changes to the calling program. ♦
- Procedures cannot require their callers to pass dynamic string descriptors. (See Section 4.2.)
- Some procedure interface specifications retain state information from one call to the next, even though the procedures are not resource allocating. The interface specification uses one of the following techniques (in order of decreasing preference) to permit sequences of calls from independent parts of a program by either eliminating the use of static storage or overcoming its limitations:
  - The interface specification consists of a sequence of calls to a set of one or more procedures — the first procedure allocates and returns (as an output argument to the calling program) one of the following:
    - The address of heap storage
    - Another processwide identifying value

    This argument is passed to the other procedures explicitly by the calling program, and the last procedure deallocates any heap storage or processwide identifying value.
  - The procedure's caller allocates all storage and passes the address on each call.

- 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 routines during its execution. Results are retained by the procedure across calls to the action routines. (No static storage used.)
- The interface specification consists of a sequence of calls to a set of one or more procedures. The first procedure, saves the contents of any still active static storage on a push-down stack in heap storage, and the last procedure, restores the old contents of static storage. Static storage is made available for implicit arguments 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).

A.1.2 Initializing

- If a procedure requires initialization once for each image activation, it is done without the caller's knowledge by one of the following:
  - Initializing at compile time
  - Initializing at link time
  - Adding a dispatch address to PSECT LIB$INITIALIZE
  - Testing and setting a statically allocated first-time flag on each call
- A procedure must not use LIB$INITIALIZE to establish a condition handler before the main program is called if its action can conflict with that of other condition handlers established before the main program. For more information about initializing modular procedures, see Section 3.2.

A.1.3 Reporting Exception Conditions

A procedure must not print error or informational messages either directly or by calling the $PUTMSG system service. It must either return a condition value in R0 as a function value or call LIB$SIGNAL or LIB$STOP to output all messages. (LIB$SIGNAL and LIB$STOP may be called either directly or indirectly.) (See Section 2.5.)

A.1.4 AST-Reentrancy

- To be AST-reentrant, a procedure must execute correctly while allowing any procedure (including itself) to be called between any two instructions. The other procedure can be an AST-level procedure, a condition handler, or another AST-reentrant procedure. (See Section 3.3.)
- A procedure that uses no static storage and calls only AST-reentrant procedures is automatically AST-reentrant. (See Section 3.3.3.)
- If a procedure uses static storage, it must use one of the following methods to be called from AST and non-AST levels:
  - Perform access and modification of the database in a single uninterruptible instruction. This can be done only from VAX MACRO, and emulated instructions are not allowed. (See Section 3.3.4.1.)

  - Detect concurrency of database access with "test and set" instructions at each access of the database. (See Section 3.3.4.2.)
  - Keep a call-in-progress count incremented upon entry to the procedure and decremented upon return. (See Section 3.3.4.3.)
  - Disable AST interrupts on entry to the procedure and restore the state of the AST enables on return. (See Section 3.3.4.4.)
- If a procedure performs I/O from the AST level by calling RMS $GET and $PUT system services, it must check for the record stream active error status (RMS$_RSA). If this error is encountered, the procedure issues the $WAIT system service and then retries the $GET or $PUT system service. (See Section 3.3.5.)
- A procedure should not depend on AST interrupts being disabled to execute correctly if there are other coding methods available. For example, RMS completion routines are implemented via ASTs and will not work if ASTs are disabled. (See Section 3.3.)

A.1.5 Resource Allocation

- A procedure should not allocate static storage unless it is a processwide, resource-allocating procedure or unless it must retain results for implicit inputs on subsequent invocations.
- Timing procedures and resource allocation procedures should make statistics available for performance evaluation and debugging by providing the entry points fac_SHOW_name and fac_STAT_name. (Optional. See Section 4.3.)
- If a procedure uses a processwide resource, it calls the appropriate resource allocating library procedure or system service to allocate the resource to avoid conflict with allocations made to other procedures. To conserve resources, a procedure that requests resource allocation does one of the following:
  - Calls the deallocation procedure before returning to the calling program
  - Remembers the allocation in static storage and calls the deallocation procedure later
  - Passes the responsibility for deallocation back to the calling program
  - Allocates a fixed number of the resources independent of the number of times it is called (See Section 2.4 and Section 3.1.3.)

A.1.6 Format and Content of Coded Modules

- Each module must be documented with a module description. (See Section 2.5.1.)
- Each procedure must be documented with a procedure description. (See Section 2.5.2.)
- When symbol definitions are to be coordinated between more than one module (such as control blocks, procedure argument values, and completion status codes), the definitions should be centralized in a common source file. Note, however, that the modules must be written in the same language. (See Section 3.1.2.)
- Procedure entry point names, module names, and PSECT names must conform to the naming conventions. (See Section 3.1.1.2, Section 3.1.1.4, and Section 3.1.1.5.)

- Digital recommends that you also adhere to the naming conventions in choosing names for facilities and files. (Optional. See Section 3.1.1.1 and Section 3.1.1.3.)

A.1.7 Upward Compatibility

When a new version of a procedure replaces an existing library procedure, all new arguments should be added at the end of the call sequence and made optional to maintain upward compatibility. (Optional. See Section 2.2.5 and Chapter 6.)

Index

J

L

M

N

O

P

R

S

T

U

W

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

Reader's Comments

Guide to Creating OpenVMS Modular Procedures
AA–PV6AA–TK

Your comments and suggestions help us improve the quality of our publications.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

For software manuals, please indicate which version of the software you are using: _____

_____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OpenVMS Documentation
110 SPIT BROOK ROAD ZKO3–4/U08
NASHUA, NH 03062–2642

--- Do Not Tear - Fold Here ----------